Loop optimisation for parallel processing

M. Di Manzo*, A. L. Frisiani*, G. Olimpo†

The problem of loop optimisation for parallel processing is examined in this paper and a method is proposed to evidence the inherent parallelism in a FORTRAN-like loop. The method is based on the concept of interrupting tasks and imposes little restrictions on the loop structure. Parallelism is achieved by concurrent execution of the loop body for different sets of values for the indices; processors activities must be synchronised and therefore the approach is best suited for SIMD machines, even if it can be used also on other kinds of multiprocessor machines provided that the processor switching and synchronising cost is low.

(Received December 1976)

In diesem Beitrag wird die Frage von der Ring-Optimisierung für parallele Verfahren geprüft und eine Methode vorgeschlagen, um den inneren Parallelismus in einem FORTRAN- ähnlichen Ring in den Vordergrund zu stellen. Die Methode gründet sich auf den Begriff von interferieren den Aufgaben und erlegt kleine Einschränkungen auf die Ring-Struktur auf. Der Parallelismus wird durch eine beitragende Ausführung von dem Ring-Körper für verschiedene Reihen von den Indexwerten erreicht; Prozessorentätigkeiten müssen synchronisiert werden und deshalb ist der Annäherungsweg am besten für SIMD-Maschinen verfolgt, auch wenn er auch bei anderen Typen von Multiprozessorenmaschinen benutzt werden kann. wenn der Prozessorschaltung-und Synchronisationkosten niedrig sind.

1. Introduction

The interest in parallel processing is rapidly increasing and several unconventional machines have been developed in recent years for various special purpose applications (Enslow, 1974). Some new problems arise in programming this kind of computer system. In fact, when the number of processors is large (tens or hundreds of units) a good utilisation of resources necessarily implies a drastic reorganisation of programs to evidence all the computation which can be performed concurrently.

Such a reorganisation may be made by the programmer himself, if the language has suitable control structures, or by the compiler, as an optimisation of the object code derived from sequential source code but up to now there is no agreement about what may be the best solution. The execution of a program on a parallel machine is indeed sensitive to the presence or absence of a particular feature, much more than on a standard sequential machine; this would encourage one to leave more freedom of choice to the compiler. Moreover, it is stated that as computer complexity increases, programmers become less competent at optimisation (Lamport, 1975). On the other hand, semantics can often allow a lot of parallelism but that cannot be detected easily by a compiler.

In this work we will be concerned only with automatic optimisation. In a program there are two main sources of parallelism which can be detected by a compiler. The first consists of arithmetic or logic expressions, like B * C + D * E; clearly, we can concurrently execute B * C and D * E. This case has been studied by Hellerman (1966), Ramamoorthy and Gonzalez (1969) and Stone (1967), but its exploitation does not look very profitable. The second source of parallelism consists of loops. This case has recently been examined (Lamport, 1974) and is much more interesting (Erickson, 1975; Presberg and Johnson, 1975). In this paper we will be dealing with loop rewriting. Our goal is to remove some constraints which have been introduced in other similar methods (Lamport, 1974), which seem to be limiting. The approach is based on the concept of interrupting tasks, commonly used in the determinacy problem, a classical topic in Operating Systems theory.

2. The determinacy problem

If a loop is executed sequentially, the determinacy of the result is guaranteed by the fact that there is a specific order of execution. This is no longer true if the loop is executed concurrently for all the values of the indices. Therefore, to restore determinacy, we must impose some constraints on the order of execution allowing only limited concurrency for properly defined subsets of values for the indices. The determinacy problem has been solved for the general case of a system of tasks by Coffman and Denning (1973), where task and system of tasks are defined as follows:

D1. A *task* is a computational unit which can be specified only in terms of its external behaviour; its internal operations are of no concern.

D2. A *system of tasks* is a pair $S = (\tau, \lessdot)$, where τ is a set of tasks and $\lessdot$ is a precedence relation on τ.

The operations performed by a task can be defined by a mapping from a set of input values to a set of output values. Hence, two sets of memory locations, called *domain* and *range*, can be associated with each task; the domain contains the input values and the range the output values. In the following we will call D the domain and R the range. A basic definition can now be introduced:

D3. Tasks T_1 and T_2 are *non interrupting* if:

$$(T_1 \lessdot T_2) \text{ or } (T_2 \lessdot T_1) \text{ or } (D_1 \cap R_2 = D_2 \cap R_1 = R_1 \cap R_2 = \Phi)$$

It has been proved (Coffman and Denning, 1973) that a system of non interrupting tasks is determinate. Clearly if the order of execution is specified, that is $T_1 \lessdot T_2$ or $T_2 \lessdot T_1$, the result is determinate. Also, if the input of one task is not the output of the other and they do not share any output, then regardless of the order of execution they will produce the same result.

*Facolta di Ingegneria, Istituto di Elettrotecnica, Università di Genova, Viale Francesco Causa 13, 16145 Genova, Italy
†Laboratorio per le Tecnologie Didattiche, CNR, Genova

3. Basic considerations about loop rewriting

In this section we define the computational structure with which we are concerned. Basic definitions are similar to those in Lamport (1974) and are stated here for completeness. The main difference is in assumptions, which are less restrictive than those required by other similar methods (Lamport, 1974; Presberg and Johnson, 1975).

In general, we will refer to a FORTRAN-like loop of the form:

$$\begin{array}{l} \text{DO 1 } I_1 = \ell_1, u_1 \\ \vdots \\ \text{DO 1 } I_n = \ell_n, u_n \\ \quad \langle \text{loop body} \rangle \end{array} \tag{1}$$

1 CONTINUE

where ℓ_i and u_i are integer non-negative valued expressions (we could allow arbitrary integer values, but the solution would become more complicated).

We make the following assumptions on the loop body:

1. The loop body contains no transfers of control to statements outside the loop.
2. Function calls do not modify data.
3. For every subroutine call we can distinguish a set of input parameters, which are not modified by the call, and a set of output parameters, which are modified by the call.
4. For each input/output statement, the corresponding I/O stream can be unequivocally identified.
5. Every occurrence, in the loop body, of a subscripted variable is of the form $V(e_1, e_2, \ldots, e_k)$, where each e_i is a linear expression involving the loop indices $I_1, I_2, \ldots, I_n$ or constants.

Assumption (4) is needed because in rewriting the loop the execution order of the I/O statements can be changed, and the result is correct only if each input variable is assigned the same input value independently of such an execution order, and if the output values are written in the same order.

In a practical implementation, assumptions (2), (3) and (4) are hard to satisfy. It may then be convenient to assume that the loop body contains no subroutine or function calls and no I/O statements. The meaning of assumptions (2), (3) and (4) is that, for instance, a function call can be admitted in the loop body only if we are sure that it does not modify data; in the case of standard library functions such a knowledge can be reasonably assumed.

Let N denote the set of all integers, and N^n denote the set of n-tuples of integers. The set Λ is defined as the subset of N^n consisting of all values assumed by $\{I_1, I_2, \ldots, I_n\}$ during the execution of loop (1). The elements of Λ can be ordered in the usual way (Coffman and Denning, 1973; Lamport, 1974): if $P = \{p_1, p_2, \ldots, p_n\}$ and $Q = \{q_1, q_2, \ldots, q_n\}$ are two n-tuples, $P, Q \in \Lambda$, then $P < Q$ if:

(a) $p_i = q_i \ \forall\ i: 1 \leqq i \leqq j - 1$

(b) $p_j < q_j \qquad 1 \leqq j \leqq n$

Therefore, e.g. $\{2, 5, 3\} < \{3, 2, 1\} < \{3, 4, 0\}$.

Our goal is to rewrite the loop in the following form:

$$\begin{array}{l} \text{DO 1 } J_1 = \lambda_1, \mu_1 \\ \vdots \\ \text{DO 1 } J_k = \lambda_k, \mu_k \\ \text{DO 1 CONC FOR ALL } (J_{k+1}, \ldots, J_m) \in \Theta \\ \quad \langle \text{loop body} \rangle \end{array} \tag{2}$$

1 CONTINUE

where Θ is a subset of N^{m-k}. Let Ψ be the subset of N^m consisting of all the values assumed by $\{J_1, \ldots, J_m\}$ and Π the subset of N^k consisting of all the values assumed by $\{J_1, \ldots, J_k\}$ during the execution of loop (2).

The lower and upper bounds for each J_i, $1 \leqq i \leqq k$, can be a function of the outer indices, that is $\lambda_i = \lambda_i(J_1, \ldots, J_{i-1})$ and $\mu_i = \mu_i(J_1, \ldots, J_{i-1})$. Parallel computation is performed within the DO CONC FOR ALL; the loop body is executed concurrently for all the elements of Θ, but the sequential order of the statements of the loop body is preserved.

The DO CONC FOR ALL control structure was developed for the ILLIAC IV FORTRAN compiler (Millstein, 1973; Millstein and Muntz, 1975) and we regard it as a well suited tool to evidence inherent parallelism in FORTRAN-like loops. To perform the above indicated rewriting we will construct a one-to-one mapping $\gamma : \Lambda \to \Psi$ so that $\{J_1, \ldots, J_m\} = \gamma(I_1, \ldots, I_n)$. In loop (2) the ordering of execution of the loop body depends only on the first k indices $J_1, \ldots, J_k$. If we define a mapping $\pi : \Lambda \to \Pi$ so that $\{J_1, \ldots, J_k\} = \pi(I_1, \ldots, I_n)$ we can state the following rule (Lamport, 1974):

1. : given two n-tuples $P, Q \in \Lambda$, the execution of the loop body for P precedes that for Q, in the new ordering of execution, if and only if $\pi(P) < \pi(Q)$.

The specification of π is the most critical operation, because of rule (1). Moreover, looking at loop (2), we see that mapping π must also enjoy the following property:

2. : each index J_i, $1 \leqq i \leqq k$, must take on every integer value from its lower to its upper bounds.*

A mapping π which satisfies rule (2) will be called a *valid mapping*. Rule (2) is not so trivial as it may appear. Each index J_i defines a partition ξ_i on Λ; the superimposition of partitions ξ_i and ξ_j defines a new partition $\xi_{i,j}$, and so on. The partition $\xi_{1\ 2,\ldots,i-1}$ then maps a subset of Λ to each $(i - 1)$-tuple of values of $\{J_1, \ldots, J_{i-1}\}$. The original loop can be written in form (2) only if within each subset of Λ the index J_i can assume all the integer values from a lower to an upper bound.

A criterion to verify if a given mapping is valid is reasonably simple only if we restrict ourselves to linear mappings of the form:

$$J_i = \sum_{j=1}^{n} a_{ij}\, I_j, \text{ for } 1 \leqq i \leqq k \tag{3}$$

It can then be proved that mapping π is valid if all the following conditions are verified:

1. : 1. $a_{i,r_j} = 0$ for $1 \leqq j \leqq k - i$, $1 \leqq i \leqq k - 1$

   2. $a_{i,r_{k-i+1}} = 1$ for $1 \leqq i \leqq k$

   3. $b_{i\,r_j} \geqq 0$ for $1 \leqq i \leqq k$, $k + 1 \leqq j \leqq n$,

   being $b_{i,r_j} = a_{i,r_j} - \sum_{t=1}^{i-1} (a_{i,r_{k-t+1}} \cdot b_{t,r_j})$

   (for $i = 1$, $b_{i,r_j} = a_{i,r_j}$)

2. : If $d_1 \geqq d_2 \ldots \geqq d_s \ldots \geqq d_h$, $h = n - k$, is an ordering of the h-tuple $\{b_{i,r_{k+1}}, \ldots, b_{i,r_n}\}$, and $d_{s+1} = d_{s+2} \cdots = d_h = 0$, it must be:

$$d_t \leqq 1 + (u_{r_{k-i+1}} - \ell_{r_{k-i+1}}) + \sum_{j=1}^{s-t} d_{t+j} \cdot \Delta_{t+j}, \qquad \text{for } 1 \leqq t \leqq s - 1$$

$$d_t \leqq 1 + (u_{r_{k-i+1}} - \ell_{r_{k-i+1}}) \qquad \text{for } t = s.$$

where $\Delta_j = u_{r_v} - \ell_{r_v}$ if $d_j = b_{i,r_v}$ and u's and ℓ's are those defined in (2).

4. Construction of mappings π and γ

In order to apply the previously described loop rewriting, it is necessary to define mapping γ; such a definition will be based on the preliminary definition of mapping π.

*In this case and from now on in the paper, lower and upper bounds are themselves included in the set of values considered.

4.1 *Identification of the interrupt conditions*

To define mapping π we must find which precedence relations are to be imposed on any pair of distinct executions of the loop body in order to guarantee the determinacy of results. Therefore, we define tasks T_p and T_q to be the executions of the loop body for two generic n-tuples $P, Q \in \Lambda$ respectively and analyse the interference conditions between T_p and T_q, supposing they are independent. The domain and the range of each task must be carefully identified. The domain consists of:

(*a*) Variables and constants on the right side of assignment statements.

(*b*) Function parameters.

(*c*) Subroutine input parameters.

(*d*) Variables and constants in conditional branch clauses.

(*e*) Variables in output statements.

The range consists of:

(*a*) Variables on the left side of assignment statements.

(*b*) Subroutine output parameters.

(*c*) Variables in input statements.

We can allow scalar and array variables and parameters, even if array variables are usually not handled in FORTRAN. From our standpoint, the only significant distinction is between scalar or array variables or parameters, called simply *variables* in the following, and *subscripted variables*. We will also use the name *identifier* to denote a variable or the array name in a subscripted variable.

By definition D3 tasks T_p and T_q are interrupting if there is some range-on-range or range-on-domain overlapping. Constants appear only in domains and cannot be responsible for interrupting; therefore we are concerned only with variables and subscripted variables. Overlapping can be detected by the occurrence of the same identifier in the range of task T_p and in the range or domain of task T_q, or vice versa. If the overlapping is originated by a variable, the interrupt is independent of actual values of P and Q and so all tasks are mutually interrupting. This is a troublesome interrupt, because in such a case no concurrent execution of the loop body is possible. If the overlapping is originated by a subscripted variable, the interrupt depends on the actual values of P and Q. In fact, in this case the interrupt is due to a pair of subscripted variables of the form:

$$\begin{aligned} v_p &= V(e_{p,1}, e_{p,2}, \ldots, e_{p,k}) \\ v_q &= V(e_{q,1}, e_{q,2}, \ldots, e_{q,k}) \end{aligned} \tag{4}$$

where the $e_{p,i}$ and $e_{q,i}$ are linear expressions involving respectively n-tuples P and Q and V is an array name. Tasks T_p and T_q are interrupting if $v_p = v_q$, that is if P and Q are a solution of the following system:

$$\begin{cases} e_{p,1} = e_{q,1} \\ \vdots \\ e_{p,k} = e_{q,k} \end{cases} \tag{5}$$

The general solution of system (5) gives all the pairs of tasks which are interrupting because of the particular pair of occurrences of the identifier V: the mapping π must preserve the original execution ordering between every pair of interrupting tasks. A system like (5) must be solved for every pair of occurrences of all the identifiers which could be responsible for any interrupt, in order to obtain all the constraints on the definition of the mapping π. We let S_i, for $1 \leqq i \leqq h$, be one of the h systems of linear equations which give all the interrupt conditions; if the two n-tuples P and Q are a solution of system S_i, we will write $S_i(P, Q) = 0$.

To preserve the execution ordering for interrupting tasks we must find a partition $\zeta(\Lambda) = \{\Lambda_1, \Lambda_2, \ldots, \Lambda_v\}$ on the set Λ which enjoys the following property:

$$S_i(P, Q) \neq 0 \text{ for every pair } P, Q \in \Lambda_j, \quad \forall\, i, j : 1 \leqq i \leqq h, 1 \leqq j \leqq v .$$

Any partition on Λ which satisfies the above property is called a *suitable partition.* Since mapping π must also obey rule (1), we define an *ordered partition* as follows:

(D4) : A suitable partition $\zeta(\Lambda)$ is an *ordered partition* if, for every pair of subsets Λ_i and Λ_j, $P < Q$ for all the pairs P, Q such that $P \in \Lambda_i$, $Q \in \Lambda_j$ and $S_r(P, Q) = 0$ for every r, $1 \leqq r \leqq h$. If the above properties hold, we will write $\Lambda_i < \Lambda_j$.

Obviously, a partition on Λ is also a partition on the set of tasks. If $\zeta(\Lambda)$ is an ordered partition, all the tasks which belong to the same subset can be executed concurrently; moreover, if a task belonging to subset Λ_i must be executed before a task belonging to subset Λ_j, there is no task in Λ_i that must be executed after any task in Λ_j and so all the tasks in Λ_i can be executed before any task in Λ_j. Therefore we can give the following rule:

3. The result of the concurrent execution of loop (2) is determinate if mapping π defines an ordered partition on Λ.

The mapping π is completely defined once we know the values of k and a_{ij}, for $1 \leqq i \leqq k$, $1 \leqq j \leqq n$. It should be pointed out that the value of k is not defined *a priori*, but is derived from the computation of the coefficients a_{ij}.

4.2 *Definition of coefficients* a_{1j}

Rule (3) imposes, for every pair of n-tuples $P, Q \in \Lambda$ for which $S_t(P, Q) = 0$, that:

$$\sum_{j=1}^{n} a_{1j}(I_{1,p} - I_{1,q}) \begin{cases} \leqq 0 \text{ if } P < Q \\ \geqq 0 \text{ if } P > Q \end{cases} \tag{6}$$

where $\{I_{1,p}, I_{2,p}, \ldots, I_{n,p}\}$ is the n-tuple $P \in \Lambda$. If the equality holds, k must be greater than 1 and the correct execution ordering is preserved by some other index among the first k ones.

Writing the two inequalities (6) for each system S_t, $1 \leqq t \leqq h$, we obtain a set Ω_1 of $2h$ systems of inequalities of the following type:

$$\begin{cases} \sum_{j=1}^{n} a_{1j}(I_{j,p} - I_{j,q}) \leqq 0 \qquad (\geqq 0) \\ S_t(P, Q) = 0 \\ P < Q \qquad (P > Q) \end{cases} \tag{7}$$

Taking into account the previously defined meaning of $P < Q$ (or $P > Q$), each system of Ω_1 generates n systems of inequalities; therefore we must finally solve a set Y_1 of $2hn$ systems of inequalities like the following ($1 \leqq t \leqq h$, $1 \leqq r \leqq n$):

$$\begin{cases} \sum_{j=1}^{n} a_{1j}(I_{j,p} - I_{j,q}) \leqq 0 \quad (\geqq 0) \\ S_t(P, Q) = 0 \\ I_{1,p} = I_{1,q} \\ \vdots \\ I_{r-1,p} = I_{r-1,q} \\ I_{r,p} = I_{r,q} - K \quad (I_{r,p} = I_{r,q} + K) \end{cases} \tag{8}$$

where K is any positive integer. It can be proved that every system $R_s \in Y_1$, which has any solution,* reduces to the following inequality:

$$\sum_{v=1}^{w} c_{1,s,v}\, X_v \leqq 0 \quad (\geqq 0) \tag{9}$$

where the set $X = \{X_1, X_2, \ldots, X_w\}$ is a subset of $\{I_{1,p}, \ldots,$

*If (8) has no solutions, it simply means that a pair of n-tuples P and Q which satisfy the conditions for the first r components cannot be a solution for the system S_t; therefore such a system can be ignored.

$I_{n,p}, I_{1,q}, \ldots, I_{n,q}, K\}$ and the coefficients $\{c_{1,s,v}\}$ are weighted sums of coefficients a_{1j}. By varying s from 1 to $2hn$, we obtain a system Z_1 of inequalities which gives us all the constraints that are imposed on the n-tuple $\{a_{11}, a_{12}, \ldots, a_{1n}\}$. Recalling that the variables $X_1, X_2, \ldots, X_w$ are non-negative, a solution for (9) is actually given by every n-tuple $\{\bar{a}_{11}, \bar{a}_{12}, \ldots, \bar{a}_{1n}\}$ which solves the following system:

$$\begin{cases} c_{1,s,1} \leqq 0 \quad (\geqq 0) \\ c_{1,s,2} \leqq 0 \quad (\geqq 0) \\ \vdots \\ c_{1,s,w} \leqq 0 \quad (\geqq 0) \end{cases} \tag{10}$$

Notice that in writing (10) we have supposed that the variables X_v can take on any integer non-negative value, even if they are bounded by the pairs $\{\ell_v, u_v\}$; this hypothesis imposes some unnecessary constraints, but the solution is fairly simplified. Every solution of system Z_1 gives a possible definition of J_1.

4.3 *Definition of coefficients a_{ij} for $1 < i \leqq k$*

Let Z_1^* be the system of inequalities derived from Z_1 by replacing all conditions like 'less (greater) than or equal' with the stronger conditions 'less (greater) than'. A specification of J_1 is a *strong solution* for an inequality of Z_1 if the corresponding inequality of Z_1^* is satisfied.

If the chosen definition of J_1 is a strong solution for the whole system Z_1, then the value of k is 1; otherwise, the index J_2 will be required to preserve the correct execution ordering in all the cases in which the equality actually holds. Therefore J_2 can be defined by solving a set Ω_2 of $2h$ systems like the following ($1 \leqq t \leqq h$):

$$\begin{cases} \sum_{j=1}^{n} a_{1j}(I_{j,p} - I_{j,q}) = 0 \\ \sum_{j=1}^{n} a_{2j}(I_{j,p} - I_{j,q}) \leqq 0 \quad (\geqq 0) \\ S_t(P, Q) = 0 \\ P < Q \quad (P > Q) \end{cases}$$

and a generic J_r is defined by a set Ω_r of $2h$ systems like the following ($1 \leqq t \leqq h$):

$$\begin{cases} \sum_{j=1}^{n} a_{ij}(I_{j,p} - I_{j,q}) = 0 \quad \text{for every } i\text{: } 1 \leqq i \leqq r-1 \\ \sum_{j=1}^{n} a_{rj}(I_{j,p} - I_{j,q}) \leqq 0 \quad (\geqq 0) \\ S_t(P, Q) = 0 \\ P < Q \quad (P > Q) \end{cases}$$

By rewriting conditions $P < Q$ and $P > Q$, the set Ω_r generates a set of $2hn$ systems of inequalities Y_r which has the same structure of Y_1 and can be reduced to a system of inequalities Z_r having the structure of Z_1. Every coefficient $c_{r,s,v}$, in system Z_r, is the same as $c_{1,s,v}$ except that each a_{1j} is replaced by the corresponding a_{rj}; for example, if $c_{1,s,t} = 2a_{11} + 3a_{14}$, then $c_{r,s,v} = 2a_{r1} + 3a_{r4}$. Therefore system Z_2 can be derived directly from Z_1 by deleting all the inequalities which are strongly solved and by replacing each $c_{1,s,v}$ with $c_{2,s,v}$; analogously, system Z_3 can be derived by Z_2, and so on.

4.4 *Value of k and valid mappings*

Every solution of the set of systems $\{Z_1, Z_2, \ldots, Z_k\}$ gives a possible mapping π, but we are interested only in valid mappings. Verifying conditions C1 and C2 is perhaps the most complicated step in the definition of π, because it requires the knowledge of the value of k. Since this value is not known *a priori*, an iterative approach is needed: first $k = 1$ is tried out and if it does not work, we must keep increasing by 1 the value of k until eventually a valid mapping is found. Notice that at least one valid mapping exists: it is the trivial solution $J_i = I_i$, $1 \leqq i \leqq n$.

Usually it is possible to define many valid mappings, and we must choose one; it seems most reasonable to choose the mapping which minimises the number of steps in the outer nonconcurrent loops. At present we know of no general algorithm which can be used to identify such a mapping: the problem is indeed complicated because the number of steps depends on the value of k and on the minimum and maximum values of each J_i, and all these values are interrelated in a complex manner. In most cases, a simpler and satisfactory approach consists of minimising the quantity:

$$\mu_i - \lambda_i \leqq \sum_{j=1}^{n} a_{ij}(u_j - \ell_j)$$

Since the summation is extended to positive terms, the minimisation may be accomplished by choosing the smallest possible values for the a_{ij}'s by trial and error.

4.5 *Lower and upper bounds λ_i and μ_i*

To complete the definition of mapping π we must find the lower and upper bounds λ_i and μ_i, for $1 \leqq i \leqq k$. Recalling that mapping π satisfies condition (1), all bounds can be defined by rewriting each I_{r_j}, $1 \leqq j \leqq k$, as a function of

$$\{J_1, \ldots, J_k, I_{r_{k+1}}, \ldots, I_{r_n}\} \ .$$

The notation is the same used for condition (1). Therefore, the bounds are the following:

$$\begin{cases} \lambda_1 = \ell_{r_k} + \sum_{t=k+1}^{n} a_{1,r_t} \cdot \ell_{r_t} \\ \mu_1 = u_{r_k} + \sum_{t=k+1}^{n} a_{1,r_t} \cdot u_{r_t} \end{cases}$$

$$\begin{cases} \lambda_i = \ell_{r_{k-i+1}} + \sum_{t=1}^{i-1} (d_{i,r_{k-t+1}} \cdot J_{r_t}) + \\ \qquad + \sum_{t=k+1}^{n} b_{i,r_t} \cdot \ell_{r_t} \ . \\ \qquad\qquad\qquad\qquad \text{for } 2 \leqq i \leqq k \\ \mu_i = u_{r_{k-i+1}} + \sum_{t=1}^{i-1} (d_{i,r_{k-t+1}} \cdot J_{r_t}) + \\ \qquad + \sum_{t=k+1}^{n} b_{i,r_t} \cdot u_{r_t} \end{cases}$$

where

$$d_{i,r_j} = \begin{cases} a_{i,r_j} - \sum_{t=2-i}^{j-k-1} a_{i,r_{k+t}} \cdot d_{1-t,r_j} & \text{for } i \geqq 3 - j + k \\ a_{i,r_j} & \text{for } i = 2 - j + k \ . \end{cases}$$

4.6 *Completion of the definition of mapping γ*

The only requirement for γ is that it be a one-to-one mapping. This goal may be simply achieved by imposing $m = n$ and $J_i = I_{r_i}$ for $k + 1 \leqq i \leqq n$. In fact, given a k-tuple $K \in \Pi$, each value of the $(n-k)$-tuple $\{I_{r_{k+1}}, \ldots, I_{r_n}\}$ identifies one and only one n-tuple $P \in \Lambda$. A complete example along this line is presented in Section 5.

However, there is no real need to have $m = n$; moreover, in some cases parallel execution is possible only if $m > n$. Let us consider, for instance, the following loop:

```
  DO 1 I = ℓ, u
  A(I) = A(I + H)                                    (11)
1 CONTINUE
```

where H is a constant. The set Ω_1 is formed by two elements:

$$1. \begin{cases} a_1(I_p - I_q) \leqq 0 \\ I_p = I_q - H \\ I_p < I_q \end{cases} \qquad 2. \begin{cases} a_1(I_p - I_q) \geqq 0 \\ I_p = I_q + H \\ I_p > I_q \ . \end{cases}$$

System Z_1 consists only of the inequality $Ha_1 \geqq 0$. If we give to a_1 any positive integer value, we obtain a loop similar to loop (11) which does not allow any parallel execution. Nevertheless, if T_i is the task associated to the execution of the loop body for $I = i$, it can be easily verified, by inspection of loop

(11), that the only interference is between the pairs (T_i, T_{i+H}); so, if $H = 2$, we can execute task T_2 concurrently to task T_3, task T_4 concurrently to task T_5, and so on.

On the other hand, so far we have supposed the coefficients a_{ij} to be integer valued for convenience. This restriction is not necessary, since only the expression $a_1(I_p - I_q)$ must be integer valued; so we can impose $Ha_1 = 1$, that is by (3) $J_1 = \frac{1}{H} I$. Of course, J_1 must be integer valued, so the correct result is $J_1 = \left\lfloor \frac{1}{H} I \right\rfloor$ or $J_1 = \left\lceil \frac{1}{H} I \right\rceil$. Consequently, the bounds are

$$\lambda_1 = \left\lfloor \frac{1}{H} \ell \right\rfloor \left(\text{or } \lambda_1 = \left\lceil \frac{1}{H} \ell \right\rceil \right)$$

and

$$\mu_1 = \left\lfloor \frac{1}{H} u \right\rfloor \left(\text{or } \mu_1 = \left\lceil \frac{1}{H} u \right\rceil \right)$$

Assuming

$$J_1 = \left\lfloor \frac{1}{H} I \right\rfloor$$

and $J_2 = I$, loop (11) can be rewritten in the form:

```
DO 1 J1 = λ1, μ1
DO 1 CONC FOR ALL J2/
                    (H * J1 ≦ J2 ≦ H * (J1 + 1) − 1)
A(J2) = A(J2 + H)
1 CONTINUE
```

The use of non-integer coefficients has a serious drawback in the consequent difficulty of checking the validity of mapping π and therefore we do not consider this possibility in the general case; however, in a simple but common case like loop (11) the validity is not a problem.

To complete the rewriting of the loop, set Θ must be found; if $m = n$ this can be done simply by defining the inverse mapping $\gamma^{-1} : \Psi \to \Lambda$ and recalling that every $I_i = f_i(J_1, J_2, \ldots, J_n)$ must always be bounded by ℓ_i and u_i.

5. An example of loop rewriting

As a simple example of the application of the proposed method, let us consider the following FORTRAN loop:

```
DO 1 I1 = ℓ1, u1
DO 1 I2 = ℓ2, u2                                   (12)
DO 1 I3 = ℓ3, u3
A(I1, I2, I3) = A(I1, I3, I2) + A(I2, I1, I3)
1 CONTINUE
```

Interruption is described by two systems S_1 and S_2:

$$S_1: \begin{cases} I_{1,p} - I_{1,q} = 0 \\ I_{2,p} - I_{3,q} = 0 \\ I_{3,p} - I_{2,q} = 0 \end{cases} \qquad S_2: \begin{cases} I_{1,p} - I_{2,q} = 0 \\ I_{2,p} - I_{1,q} = 0 \\ I_{3,p} - I_{3,q} = 0 \end{cases}$$

The set Ω_1 is made up of four systems of inequalities:

$$\omega_1: \begin{cases} \sum_{j=1}^{3} a_{1j}(I_{j,p} - I_{j,q}) \leqq 0 \\ S_1(P, Q) = 0 \\ P < Q \end{cases}$$

$$\omega_2: \begin{cases} \sum_{j=1}^{3} a_{1j}(I_{j,p} - I_{j,q}) \geqq 0 \\ S_1(P, Q) = 0 \\ P > Q \end{cases}$$

$$\omega_3: \begin{cases} \sum_{j=1}^{3} a_{1j}(I_{j,p} - I_{j,q}) \leqq 0 \\ S_2(P, Q) = 0 \\ P < Q \end{cases}$$

$$\omega_4: \begin{cases} \sum_{j=1}^{3} a_{1j}(I_{j,p} - I_{j,q}) \geqq 0 \\ S_2(P, Q) = 0 \\ P > Q \end{cases}$$

The system Z_1 is:

$$\begin{cases} -(a_{12} - a_{13})\,K \leqq 0 & \text{from } \omega_1 \\ (a_{12} - a_{13})\,K \geqq 0 & \text{from } \omega_2 \\ -(a_{11} - a_{12})\,K \leqq 0 & \text{from } \omega_3 \\ (a_{11} - a_{12})\,K \geqq 0 & \text{from } \omega_4 \end{cases}$$

Any set of values which satisfies the conditions $a_{12} > a_{13}$ and $a_{11} > a_{12}$ is a strong solution. A possible mapping π is therefore:

$$\pi : J_1 = 2I_1 + I_2$$

with $k = 1$. Conditions (1) and (2) are satisfied, so the mapping is valid. The mapping γ is:

$$\gamma: \begin{cases} J_1 = 2I_1 + I_2 \\ J_2 = I_1 \\ J_3 = I_3 \end{cases}$$

Loop (12) can be rewritten as follows:

```
DO 1 J1 = 2ℓ1 + ℓ2, 2u1 + u2
DO 1 CONC FOR ALL (J2, J3)/(ℓ1 ≦ J2 ≦ u1,
                   ℓ3 ≦ J3 ≦ u3, ℓ2 ≦ J1 − 2J2 ≦ u2)
A(J2, J1 − 2J2, J3) = A(J2, J3, J1 − 2J2) +
                                A(J1 − 2J2, J2, J3)
1 CONTINUE
```

6. Conclusions

We have presented a method for loop rewriting based on the concept of non-interrupting tasks. Assumptions are not very restrictive, compared with similar methods and a large set of common loop structures can be optimised for parallel processing. The required computation is not simple because a certain number of system inequalities must be solved; this can slow down dramatically the compiler in the case of large, highly nested loops, but the compilation time is less of a problem as the optimised code will be used many times. Possible developments of this work could consist of an analysis of mutually invariant sets of statements which would allow more parallelism by removing the assumption that each processor executes the whole loop body.

References

BERNSTEIN, A. J. (1966). Analysis of programs for parallel processing, *IEEE TC*, Vol. 15, pp. 757-762.

COFFMAN, E. G. and DENNING, P. J. (1973). *Operating Systems theory*, Prentice-Hall.

ENSLOW, P. H. (Editor). (1974). *Multiprocessors and parallel processing*, John Wiley.

ERICKSON, D. B. (1975). Array processing on array processors, *ACM SIGPLAN NOTICES*, Vol. 10, pp. 17-24.

GRIES, D. (1971). *Compiler construction for digital computers*, John Wiley.

HELLERMAN, H. (1966). Parallel processing of algebraic expressions, *IEEE TC*, Vol. 15, pp. 82-91.

LAMPORT, L. (1974). The parallel execution of DO loops, *CACM*, Vol. 17, pp. 83-93.

LAMPORT, L. (1975). On programming parallel computers, *ACM SIGPLAN NOTICES*, Vol. 10, pp. 25-33.

MILLSTEIN, R. E. (1973). Control structures in ILLIAC IV FORTRAN, *CACM*, Vol. 16, pp. 621-627.

MILLSTEIN, R. E. and MUNTZ, C. A. (1975). The ILLIAC IV FORTRAN compiler, *ACM SIGPLAN NOTICES*, Vol. 10, pp. 1-8.

NEEL, D. and AMIRCHAHY, M. (1975). Removal of invariant statements from program loops by semantic attribute method, *ACM SIGPLAN NOTICES*, Vol. 10, pp. 87-96.

PRESBERG, D. L. and JOHNSON, N. W. (1975). The Paralyzer: IVTRAN's parallelism analyzer and synthetizer, *ACM SIGPLAN NOTICES*, Vol. 10, pp. 9-16.

RAMAMOORTHY, C. V. and GONZALEZ, M. J. (1969). A survey of techniques for recognizing parallel processable streams in computer programs, *Proc. AFIPS* 1969 *FJCC*.
SCHNECK, P. B. (1972). Automatic recognition of parallel vector operation in a higher level language, *Proc. ACM* 1972 *National Conference*.
SCHNECK, P. B. (1975). Movement of implicit parallel and vector expressions out of program loops, *ACM SIGPLAN NOTICES*, Vol. 10, pp. 103-106.
STONE, H. S. (1967). One-pass compilation of arithmetic expressions for a parallel processor, *CACM*, Vol. 10, pp. 220-223.

Book reviews

Microcomputer Problem Solving Using Pascal by K. L. Bowles, 1977; 563 pages. (*Springer-Verlag*, $9·80)

This book is yet another contender for the 'introduction to problem solving and computer programming' market and, like other recent books, is based on PASCAL. The book is derived from courses presented at the University of Calfornia at San Diego (UCSD). UCSD have implemented a stand-alone single user PASCAL system which is interpreter based and runs on a variety of microcomputers, for example LSI-11, Z80 and Intel 8080 based systems. The book is based on this PASCAL system, and is intended for students with both mathematical and non-mathematical backgrounds. The problem examples are of a non-numeric nature, concentrating on graphical and string manipulation examples. UCSD have extended PASCAL with built-in functions and procedures to suit these applications.

For those students who have access to a UCSD system with graphics hardware, the book could be useful. However, I do not feel that I can recommend the book for students using other PASCAL systems. There are other books on the same topic which seem to be much more system independent and therefore more useful to a wider audience.

P. A. LEE (Newcastle upon Tyne)

The Design of Well Structured and Correct Programs by S. Alagic and M. A. Arbib, 1978; 292 pages. (*Springer-Verlag*, $14·00)

I am impressed–this book is a welcome change from the usual programming text. The authors' aim is to teach top-down program development using Hoare invariance methods; they succeed. The reader will not only pick up many good programming habits, but also learn some PASCAL along the way. For programmers solving the class of problems for which this methodology is suitable (i.e. most) this book is a must. The authors assume the reader has done some computer programming, but everything else is introduced carefully, logically and well.

The book is well illustrated throughout by numerous examples and exercises (minor criticism—no solutions) and the motivation for the next step in the argument is always given. The examples used cover a wide range of applications and are all taken from the literature but the proofs are usually new. This means that the reader will often be familar with the problem (gcd, file merging, 8 queens, etc.) and can concentrate on the method of solution. I found this advantageous as the text is quite concise and can best be digested in small bites.

The first three chapters cover the basic ground work and introduce top-down design, PASCAL structured statements and their proof rules and PASCAL data structures. In Chapter 4 these are brought together in a number of case studies where particular solutions are developed. This chapter is called 'Programs and proofs' but due to the size of the examples used it would probably have been better titled 'Routines and proofs'. Having seen how to develop reliable components the authors turn, in Chapter 5, to the very important task of their interconnection. Procedures, functions and block structure are dealt with in this chapter whilst Chapter 6 deals with recursive algorithms *and* data structures. Had the book stopped at this point, it would have been highly recommended, but the best is yet to come; Chapter 7 covers the final structuring statement **goto**! Here much new material is introduced and I find it the most sane discussion on the goto problem which I have ever read. The book closes with appendices on PASCAL syntax, summaries of proof rules, a glossary and a good bibliography.

Like many good ideas Hoare's preconditions and postconditions are, once they have been pointed out, brillantly simple. Invariance, I would claim, is one of the research topics which seems to offer most help to the practising programmer. This text should be read by anyone who is, or who wishes to be, a professional programmer. Good simple ideas like these should be brought to the attention of a much wider public and this book is a good first step in this direction. The book meets its aims and one cannot criticise it for things it does not try to do but perhaps we can now look forward to similar texts based on FORTRAN and COBOL. If we continue in this way, work which starts from a theoretical standpoint can impact upon and improve our professional practice.

D. SIMPSON (Sheffield)

Artificial Intelligence, edited by A. Bundy, 1978; 253 pages. (Edinburgh University Press, £5·00)

This is a collection of lecture notes from the course Artificial Intelligence 2 given at the University of Edinburgh by the staff of the Department of Artificial Intelligence. The course surveys all those areas of artificial intelligence which must now be considered classical and are under the headings Problem solving, Natural languages, Question answering and inference, Visual perception, and Learning. It was aimed at second and third year undergraduates from disciplines other than computing, but according to the teaching notes students were expected to spend three hours a week at an interactive computer terminal. There must be some doubt whether it is reasonable to mix the initial teaching of computer usage with consideration of the ultimate power of machines, and in an afternote the authors seem to accept this criticism.

There are four different authors, all well known in the field, and different sections inevitably show a different style of presentation. However, these are *notes*, terse and pithy, but not lengthy explanations. They are not suitable for self-study by a novice, and would be best appreciated by an intending teacher of a similar kind of course. The general approach is to stimulate the students to find out for themselves and constant use of the computer together with extensive reading in a variety of topics is indicated. It would seem to make heavy demands on an undergraduate's time and mental agility, in a way which is currently not fashionable, and the authors say that later courses had less practical work and that students were divided into groups according to programming ability. One thing has been achieved; the case for this kind of course within a general curriculum has been made firmly.

J. J. FLORENTIN (London)