V. 8, n.º 2 (juny 1984) pp. 87-92

PARALLEL PROGRAMMING THROUGH SCHEMES

ALBERT LLAMOSI

UNIVERSITAT POLITECNICA DE BARCELONA

Taking as a basis the underlying model of Ada language, but using a more syncretic notation, the main purpose of the present paper is to show how, given a problem in parallel programming, several solutions to it can be found systematically by direct or compound instantiation of two well characterized basic schemes which correspond to the general and somewhat dual situations of cooperation and competition.

Keywords: PARALLELISM, PROGRAM SCHEMES, PROGRAMMING METHODOLOGY.

1. INTRODUCTION.

Current research on parallel programming has been concentrated in the design of models for which a safe notation and semantics and a simple calculus could be developed. However, little attention has been paid to programming methodology, in spite of the fact that an important feature for a model is to serve as a basis for good program construction methodologies.

On the other hand, the usefulness of program schemes in sequential programming has been widely recognized and this suggests that the same approach could reasonably help to cover that lack of methods.

The main steps in providing general schemes are to elaborate a classification of the existing problems and to establish how each class can be characterized in terms of a scheme. An exhaustive treatment of examples made elsewhere /10/ has led to the conclusion that only two basic schemes cover satisfactorily extremely wide series of cases.

The design process proposed by the method presented here would consist then in decomposing the operations to be carried out by a program in those that can, or must, be executed in parallel, analyzing their interactions and formulating them in terms of one of the two proposed basic systems. This would allow to determinate the internal structure of each process following an appropriate pattern. Although obviously, once a solution has been obtained by this procedure, further transformations can be essayed in order to simplify it. In a longer, but yet unpublished work /10/, some general cases of simplification are also considered.

I have only dealt with algorithms that are intended to work forever because to propagate termination through processes properly, by the use of basic rendez-vous, usually makes the algorithms cumbersome. However, in /10/ I have proposed some notational hints that would allow to maintain the pleasant simplicity of the schemes presented below in a terminating context. But this exceeds the purpose of the present paper.

In §1 and §2 the basic schemes are presented with examples. Composition of basic schemes to deal with more complex problems is considered in §3 and finally some balance is drawn out.

2. COOPERATION.

In a general sense, all processes of a program cooperate in order to make it satisfy the required specifications. Hence a narrower notion is needed to characterize a cooperation system.

- Albert Llamosi - Facultat d'Informàtica de la Universitat Politècnica de Barcelona
Jordi Girona Salgado, 31 - Barcelona 34

- Article rebut el Març de 1984.

By this I mean the situation where there exist n (n>0) processes whose behaviour is function of the information combinedly produced by other m (m>0) processes. So the last ones cooperate in the first ones' control.

A suitable solution for structuring an algorithm in that situation can always be obtained by the introduction of a new process, let us call it C, that centralizes and distributes the produced informations.

The sending and receiving processes shall have the following simple structure:

```
 m
 ||  Sk ::
k=1
    [ s_initk;
        *[ True→  prodk(s_infk); C.senk(s_infk)]
    ]
 n
 ||  Rk ::
k=1
    [ r_initk;
        *[ True→  C.rec(r_infk); consk(r_infk)]
    ]
```

where the central process's structure is

```
C ::
[ c_init;
    m
  *[ []  pos_sk; <sendk(s_infk); inck>→ reg_sk
   k=1
     n
     []  pos_rk; <reck(r_infk); extk>→ reg_rk
    k=1
] ]
```

Decision steps should be:

a) Consideration of the information that must be sent and received (s_inf_k and r_inf_k).

b) Choice of a data structure for process C. Its purpose is to constitute at every moment a summary of the information sent such that information to be received can be obtained.

c) Determination of under which conditions sent information can be taken into consideration (pos_s_k) and under which conditions reception of information can be carried out (pos_r_k).

d) Determination of the operations that must be performed when a sent or received information is incorporated to or drawn out from the data structure. Both operations should be decomposed into the part that must be performed inside the rendez-vous area due to its dependency from parameter passing and the part that can be performed outside it (inc_k, reg_s_k, ext_k and reg_r_k respectively).

e) Establish the appropriate initialization (c_init).

EXAMPLE. A REAL-TIME SCHEDULER.

Several tasks should get into execution at fixed intervals of time. However, an operator should be able to change, through a terminal, the current time, the status of each task (activated, not activated) and change their starting times and periodicities. This problem was posed and solved in a more artificious way at /5/.

In principle there are so many processes as tasks to be controlled. Moreover, there should be a real-time-clock process and a process that maintains a dialogue with the operator.

Behaviours of tasks are function of real-time clock pulses and operator commands. So the problem can be interpreted as a cooperation between the two last processes onto the control of the tasks.

Hence, steps to be taken should be

a) Operator process sends commands. Real-time clock sends signals. Task processes receive signals. Their structure shall be:

```
OP :: *[ True → read_command (c); C.scom(c)]
RTC :: *[ True → wait_next_second; C.tic]
 n
 ||  T(i):: *[True → C.conti; opi]
i=1
```

b) The central process should contain three informations about each task: if it is activated, its next starting time and its period. Moreover it should keep track of time.

```
type status = record  active: boolean;
                      start : integer;
                      period: integer
              end

var t: integer;
    s: array [1..n]  of status;
```

c) Commands and clock tics should always be accepted. Continuation signals may only

be received if the corresponding task is active and its starting time has elapsed.

d) Operations to apply when receiving a command or a time pulse are quite obvious in this case. On the other hand, reception of a continuation signal should have as a result the increase of starting time in the corresponding period.

e) Initially all tasks are inactive.

```
c ::
[i:=0; *[i<n → i:=i+1; s(i).active:=False]

  *[<scom(c); c1:=c> → (t,s):=f(s,c1)

   ▯ <tic>  → t:=t⊕1

   ▯(k=1,n) s(k).active ∧ (t⊖s(k).start); <cont_k>
       s(k).start := s(k).start⊕s(k).period
]  ]
```

3. COMPETITION.

A system of competition is one where there are m(m>1) processes that use n(n>0) resources, and an arbitrary interleaving of operations upon them can produce undesirable effects. Thus some kind of restrictions should be introduced.

As in the case of cooperation, easy solutions can be obtained by adding a new central process that centralizes the information about the state of the resources in conflict.

User processes have undefined structure, but their access to resources must be parenthesized as usual.First they ask permission, and that takes generally two steps: one to notice their purposes of use and a second one to wait for acceptance and receive information about the resources conceded. After use, release must also be noticed.

```
‖(k=1,n) U_k :: [...; C.not(purp); C.req_id_k (conc);
                 use_k;
                 C.rel (conc); ...
               ]
```

Whereas the central process structure is

```
c :: [c_init;
      *[<not(purp); inc_p>  → reg_not
       ▯ pos_req;<req_1(conc);ext>  → reg_req
       ▯ <rel(conc); inc_c>  → reg_rel
      ]  ]
```

Decision steps should be

a) Consideration of how to represent access purposes, concession and process identification (purp, conc, id_k).

b) Choice of a data structure for process C. Its function should be to constitute at every moment a register of the state of the resources controlled by C and the noticed purposes not yet satisfied.

c) Determination of under which conditions and identities a request can be conceded (pos_req,1).

d) Determination of the operations that should be performed when a notification of purposes or abandon is received, or a request has been conceded. All those operations should be decomposed into the part that must be performed inside the rendez-vous area due to its dependency from parameter passing, and the part that can be performed outside it (inc_p, reg_not, ext, reg_req, inc_c, reg_rel).

e) To establish the appropriate initialization (c_init).

EXAMPLE. A DISK SCHEDULER.

On a moving head disk, the time taken to access certain track increases monotonically with the distance between the current and target positions of the head. So a minimum access time is obtained on the average if processes that are waiting to access, in mutual exclussion, do it in order of shortest displacements. Since several processes that repeatedly access at one extreme could delay indefinitely other processes that are waiting for access at the other extreme, some additional criteria should be introduced to grant fairness. A simple one is to minimize the frequency of change of direction of movement of the heads. At every moment the head is

kept moving in a given direction and the next process to access shall be the one that waits for access at the closest track in the current direction. Only if there is no such request, the direction changes. This problem was originally posed at /8/ and solved there by monitors with priorities.

a) Access purposes are trivially track numbers. Concession of information is nothing in this case, as the user process knows everything about the resource. Identification of processes could be a special type, but since different processes that try to access the same track are indistinguishable from the viewpoint of the problem it seems natural to identify processes with this number. So user processes will appear as

$$\|_{k=1}^{m} U_k :: [\ldots; C.not(t); C.req_t; access\_at\_track\_t; C.rel; \ldots]$$

b) The information needed by the central process is the current direction (+1 or -1), the status of the disk (busy or not), the current head's position (p) a counter of processes waiting at every track and the next expected position.

```
var  busy : boolean;
     d,p,next,i: integer;
     w: array [0.. maxtrack] of integer
```

c) Access can be conceded if the disk is not busy and process's identification equals the value of the integer variable next.

d) In case of notification of access, the corresponding counter should be increased and both the expected next track and current direction should be reevaluated. In case of access, the busy state of disk and current head's position must be recorded and, again, the expected next track and eventually the current direction should be reevaluated. Indeed the corresponding waiting counter must be decreased.

e) Initially the head's position is zero and the current direction is in ascending order of tracks. No process is waiting.

C ::

```
[busy:= False; p:=0; d:=1; next:=0;
 *[next ≤ maxtrack → w(next):=0; next:=next+1]
 *[<not(t); w(t):= w(t) + 1> → evaluate_next
  [] ¬busy; <req_next> → p:=next; w(p):=w(p) - 1;
                         busy:= True;
                         evaluate_next
   [] <rel> → busy:= False
 ] ]
```

Where evaluate_next states for

```
evaluate_next ::
[ next: =p;
  [d > 0 → limit:=maxtrack [] d < 0 → limit:=0]
  *[w(next) = 0 ∧ next ≠ limit → next:=next + d]
  [ w(next) ≠ 0 → skip
  [] w(next) = 0 →
       d:= -d; next:=p;
       [d > 0 → limit:=maxtrack [] d < 0 → limit:=0]
        *[ w(next)=0 ∧ next≠limit → next:=next + d ]
 ]]  ]
```

Use of sentinel techniques can obviously simplify this algorithm, and use of additional redundant counters increase its efficiency.

Total fairness can be granted initializing next at p+d instead of p at the beginning of searches, but then a poorer performance can be expected. More subtle alternatives are also feasible.

4. COMPOSITION OF BASIC SCHEMES.

Simple schemes can sometimes be inadequate to solve complex problems. But nothing prevents composing them, in the sense that the existence of undefined operations in schemes allows the possibility that parts of them act as operations that belong to other systems or some of their operations, or processes themselves can be further decomposed in new parallel systems.

In a cooperation system, for example, production of some information can involve access at shared resources or reception of information from other cooperation systems. Similarly, consumption of information can involve sending of information to the central process of any other system, or even the same: as a particular case, a system can be compound with itself.

Classical pipe-line systems would be an exam-

ple of composition of cooperating systems. Sometimes cyclical structures can appear. That would be the case of a spheric integration (for weather prediction calculus, for example) that is carried out by a frame of processes that integrate locally. Each node is then a cooperation system where the behaviour of a process is function of the results produced by each of its neighbours.

5. CONCLUDING REMARKS.

The method proposed allows to reason parallel programming problems following well established patterns. The order proposed for the decision steps is obviously only orientative. Many of the decisions to be taken are very interrelated and sometimes the discovery of difficulties leads to reconsider previous decisions. However, the order proposed is oriented to minimize this kind of backtracking. Conscience of decision steps is anyway an important element to discover different solutions systematically.

Indeed, several particular cases of the general schemes could have been considered. For example, for the cooperation situations where m=1 or n=1 or both, there exist simplifications that become safe transformation rules when the basic schemes are included in a wider framework.

As the success obtained introducing undergraduate students into parallel programming through the method presented here seems to show, the conceptual simplicity of the basic schemes provides an orientative element at design level in the sense that the programmer knows in advance what he has to look for. Furthermore, the regularity that appears in the structure of the solutions obtained increases readability. And, as it has been pointed out in /12/, regularity of programming habits increases programs portability as well as programmers portability.

Last, but not least, I have the confidence that restriction to the schemes presented here could allow simpler and practical techniques for reasoning and grant total correctness. Current research is being done in this direction.

6. REFERENCES.

/1/ ADA,: The Programming Language Ada Reference Manual , LNCS 105, Springer Verlag, Berlin, 1981.

/2/ BERT, D.: "La Programmation générique. Construction de logiciel, spécification algébrique et verification", Univ. de Grenoble, IMAG, 1979.

/3/ BERT, D.:"Generic Programming: A tool for designing universal operators. Application to program algebra", RR nº336, IMAG, Univ. de Grenoble, 1982.

/4/ BROOKES, S.D., HOARE, C.A.R. and ROSCOE, A.W.: "A Theory of Communicating Sequential Processes", Technical Report PRG-16, Oxford Computing Laboratory, Programming Research Group, 1981.

/5/ BRINCH HANSEN P.: The Architecture of Concurrent Programs , Prentice-Hall, Englewood Cliffs, N.J., 1977.

/6/ GOGUEN, J.A., MESEGUER, J. and PLAISTED, D.: "Programming with Parameterized Abstract Objects in OBJ", in Ferrari, D., Bolognani, M. and Goguen, J. (eds.), Theory and Practice of Software Technology, North-Holland, 1982,pp.163-193.

/7/ GOGUEN, J.A.: "Parameterized Programming" to appear in Perlis, A., Workshop on Reusability in Programming. Proceedings.

/8/ HOARE, C.A.R.: "Monitors: an Operating System Structuring Concept", Comm. ACM, XVII, 10 (Oct. 1974) pp. 549-557.

/9/ HOARE, C.A.R.: "Communicating Sequential Processes", Comm. ACM, XXI, 8 (Aug.1978), pp. 666-677.

/10/LLAMOSI, A.: "Programació paral.lela usant esquemes: vers un mètode sistemàtic de construcció de programes concurrents, doctoral thesis, F.I.B., Barcelona 1982.

/11/MILNER, R.: A Calculus for Communicating Systems ,Springer Verlag, LNCS 92, Berlin 1980.

/12/ SCHOLL, P.-Cl.: "Vers une programmation systématique étude de quelques méthodes et outils", thèse d'état, USMG-INP, Grenoble, 1979.