PARALLEL PROGRAMMING RECIPE

As said before, the process of development of parallel programs can be summarized as follows:

1. Pick up a particular problem of interest;
2. Conceptualize the solution;
3. Split this solution into components to be executed simultaneously as cooperating processes;
4. Code each component;
5. Arrange components in groups;
6. Allocate to each group a separate processor of suitable type;
7. Execute simultaneously all components, noting overall run time.

EXECUTION OF PARALLEL PROGRAMS

… consists of repeated firing of new, cooperating processes.

Parallel programming languages have constructs permitting us to do that. Basically, these constructs amount to:

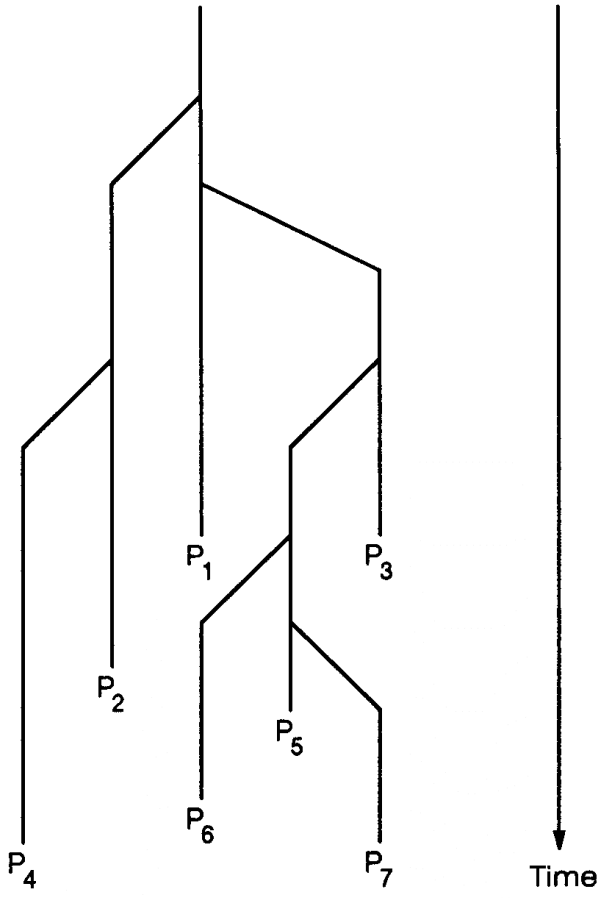
FORK (<processor_name>, PID)

Where process ID

PID ::= < <routine_id>, <instantiation_no> >

Because there may be several, different processes running the same routine for different data.

Execution of a parallel program resembles an explosion of fireworks:



ON THE COMPILER'S INABILITY TO FORECAST COMPUTATIONAL EFFORT

Can we effectively automate the job of assigning optimal processor to each new process?

No. To illustrate that, consider the following simple example:

Given the routine below, can you figure out the amount of computation needed to produce the result?

```
X, Y : integer;

Read (X);
Y := 0;
While X <> 0 loop
        X := X – 2;
        Y := Y + 1;
End loop;
Print (Y);
```

EXECUTION HISTORY

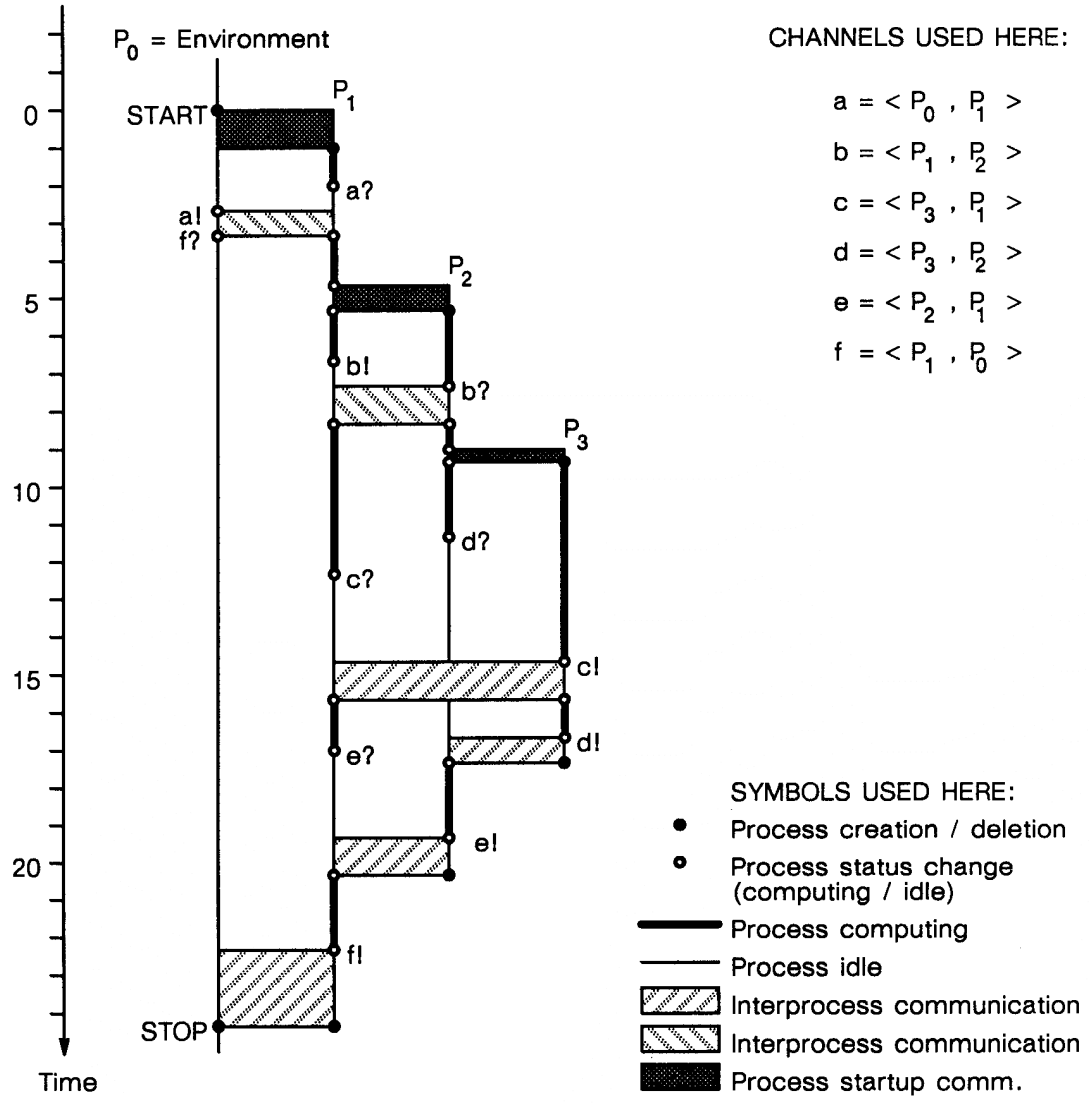
But: we can we effectively record execution histories of programs, and then deduce how fast would these programs run on different computers.

Let's define:

Channel: unbuffered, directional means of communication between exactly two processes: one called sender, the other called receiver, or simply put:

Channel: an ordered pair of two processes <**S, R**>

We could record program execution histories, viz.:



RECORDING EXECUTION HISTORIES

Data structures needed – using pseudo-Pascal:

```
Type PROCESS_STATUS is (  BORN,
                          AWAITS_INPUT,
                          AWAITS_OUTPUT,
                          RESUMES,
                          DIES  );

Type EXECUTION_EVENT is
    Record
        PID :    PROCESS_ID;
        STATUS:  PROCESS_STATUS;
        CHID :   CHANNEL_ID;
        TIME :   CLOCK_TIME;
    End record;

Type EXECUTION_HISTORY is file of EXECUTION_EVENT;
```

Using these data structures, we could write a program that would write an **EXECUTION_HISTORY** file for a given application. This file would contain histories of several, typical runs of this application, on a single-CPU machine.

Now, for this application, we could attack a refined

PROBLEM OF OPTIMAL PROCESSOR ALLOCATION

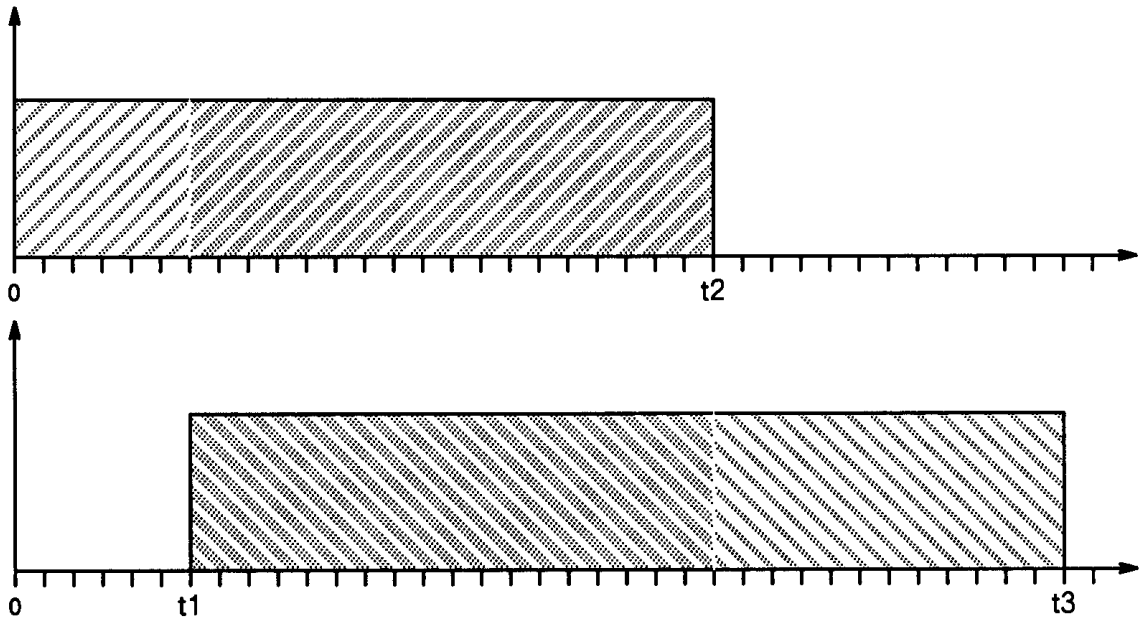
1. Write an processor allocation program that would read an **EXECUTION_HISTORY** file for a given application and recommend the optimal processor allocation of N processors, where 0 < N <= P processes, and

2. How would you allocate processors to your program?

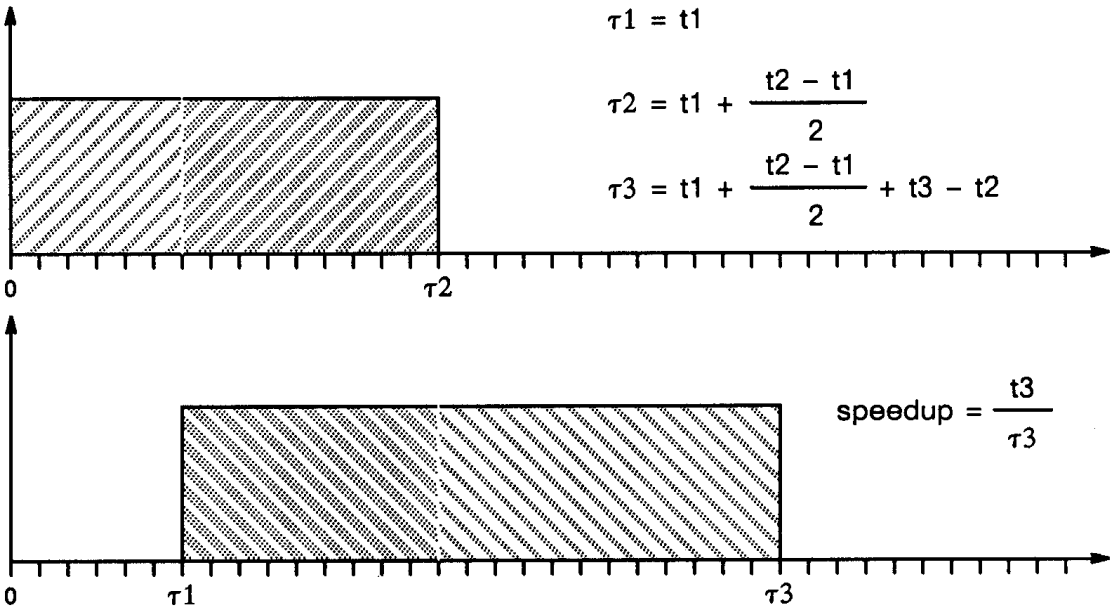
OPTIMAL PROCESSOR ALLOCATION

It is possible to calculate (ie. extrapolate or simulate) the execution time of a program, if we changed the number of processors. Example:

Two processes sharing one processor:



The same processes running on dedicated processors

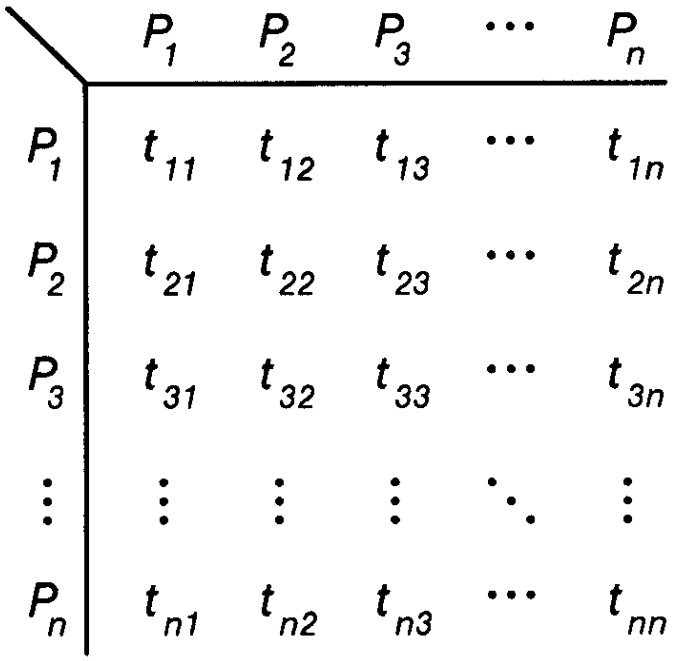


ANALYSIS OF EXECUTION PATTERNS

By scanning execution patterns of a given application, one could compute, among others:

1. Max. number of useful processors
2. Maximally parallelized execution pattern
3. Maximum speedup attainable
4. Process activation matrix (showing probabilities, times, or correlations)
5. Process lingering matrix (showing times or probabilities)
6. Process activation pattern

NOTE: Process matrices have the form:



	P_1	P_2	P_3	⋯	P_n
P_1	t_{11}	t_{12}	t_{13}	⋯	t_{1n}
P_2	t_{21}	t_{22}	t_{23}	⋯	t_{2n}
P_3	t_{31}	t_{32}	t_{33}	⋯	t_{3n}
⋮	⋮	⋮	⋮	⋱	⋮
P_n	t_{n1}	t_{n2}	t_{n3}	⋯	t_{nn}

NOTE: Elements of the process activation matrix have the following property:

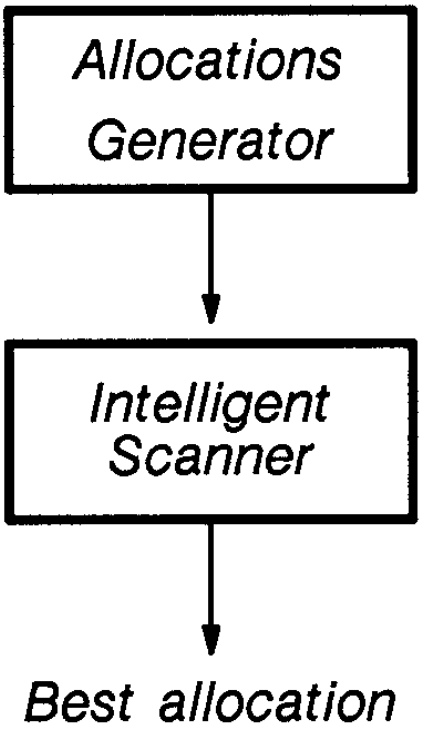
$$t_{ij} = t_{ji} \quad \text{and} \quad 0 <= t_{ij} <= \min (t_{ii}, t_{jj})$$

NOTE: A process activation pattern is a list of process activation records, each record having the form:

$$< P_k, P_l, P_m, \ldots , P_n, t >$$

SEARCH FOR OPTIMAL PROCESSOR ALLOCATION: A NAÏVE SEQUENTIAL APPROACH

The naïve strategy is to scan sequentially the execution patterns of every possible allocations, viz.:

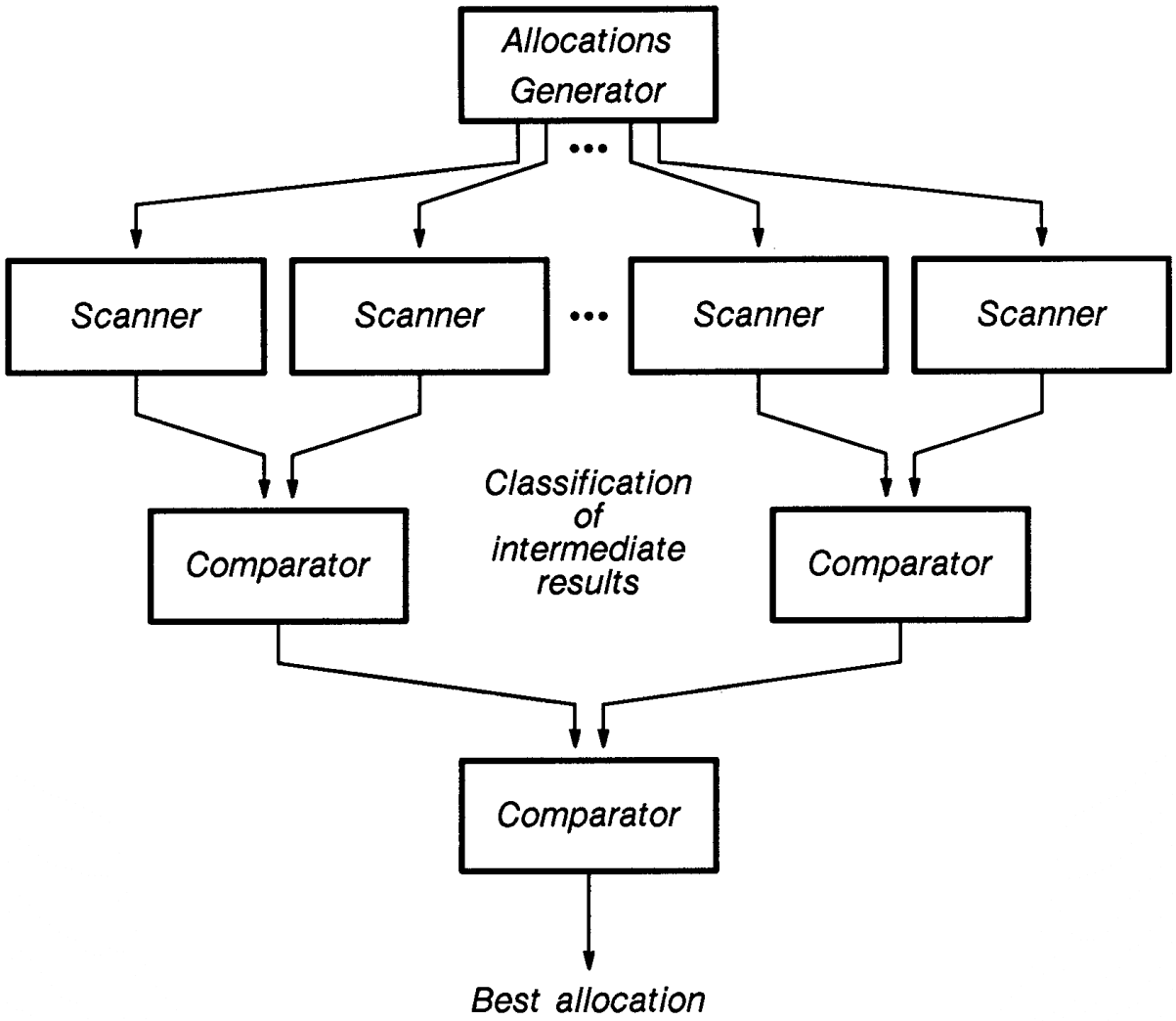


The intelligent scanner:

1. Remembers and updates the results of the best allocation found so far;
2. Abandons the evaluation of the current allocation:
    - At once – if the allocation is found not feasible;
    - While scanning – if the scan deadline cannot be met.

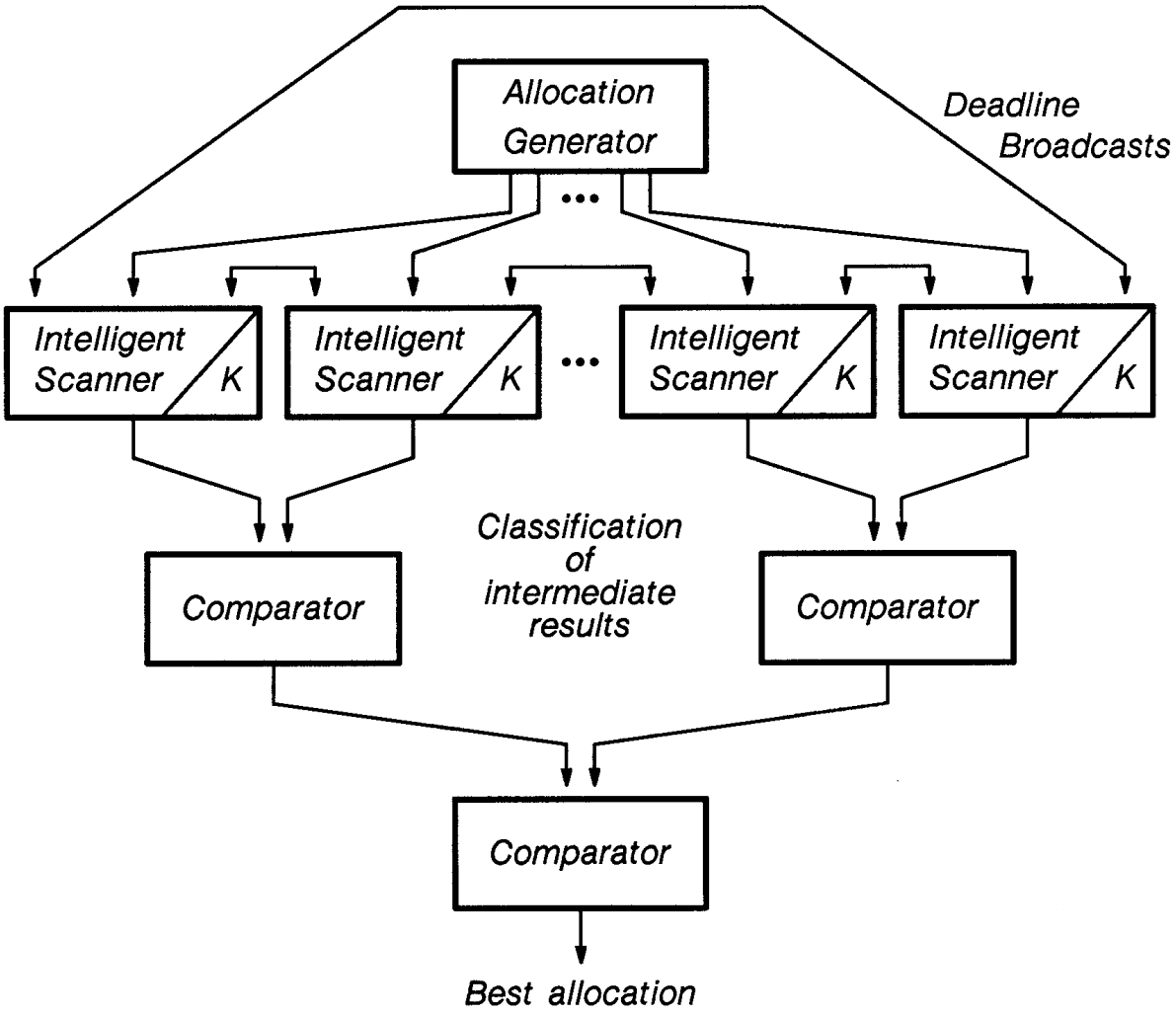
SEARCH FOR OPTIMAL PROCESSOR ALLOCATION: A NAÏVE PARALLEL APPROACH

The naïve strategy is to scan in parallel the execution patterns of every possible allocations, viz.:



SEARCH FOR OPTIMAL PROCESSOR ALLOCATION: A BIOLOGICALLY INSPIRED APPROACH

The strategy is to scan in parallel the execution patterns of allocation groups, using intelligent scanners of concurrent scanning capacity K, viz.:



EQUIPMENT OPTIMIZATION POSSIBILITIES:
Manipulate the number of intelligent scanners S working in parallel and their concurrent scanning capacity K so that the product of the total search time times S is minimized.

IMPROVED PROCESSOR ALLOCATION RECIPE

1. Pick up a particular program of interest;

2. Split this program (differently) into components to be executed simultaneously as cooperating processes;

3. Code each component;

4. (Re)arrange components into groups;

5. (Re)allocate a separate processor of suitable type to each group;

6. Execute all processes, noting overall run time.

7. If the execution time is satisfactory
   then exit successfully
   else go to (5)
   unless all allocations have been covered, in which case go to (4)
   unless all arrangements have been covered, in which case go to (2)
   unless all possible splits have been covered, in which case
   exit unsuccessfully.