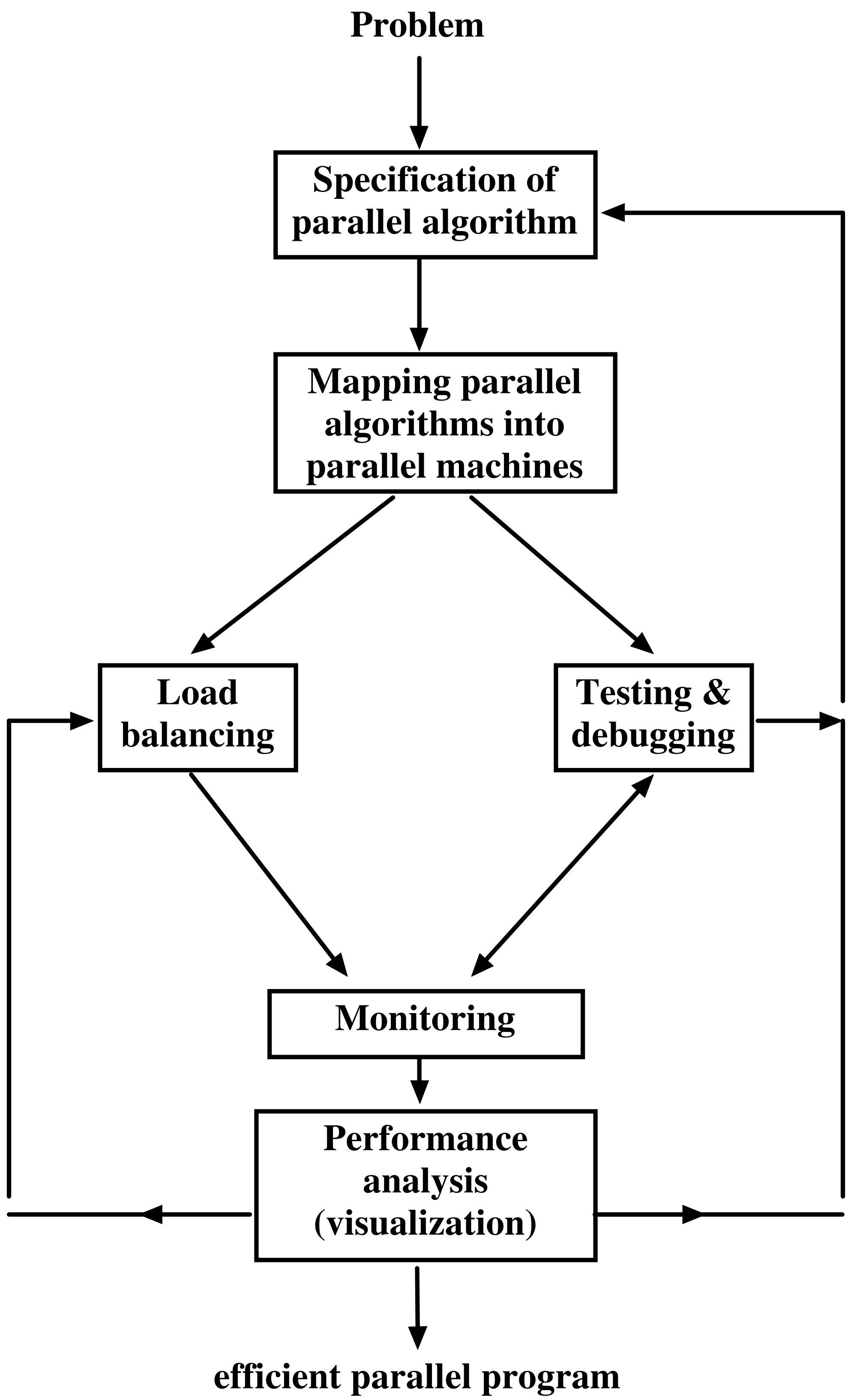
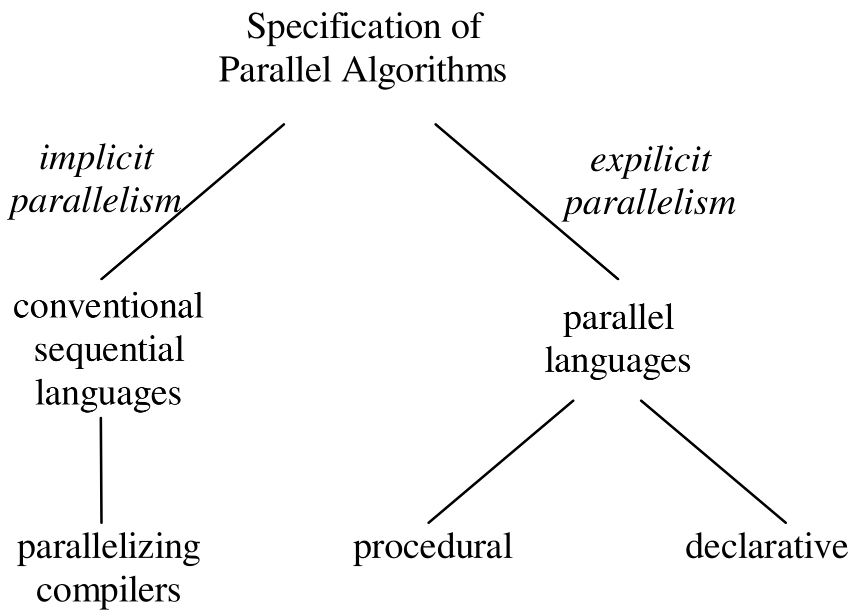
PARALLEL SOFTWARE ENGINEERING

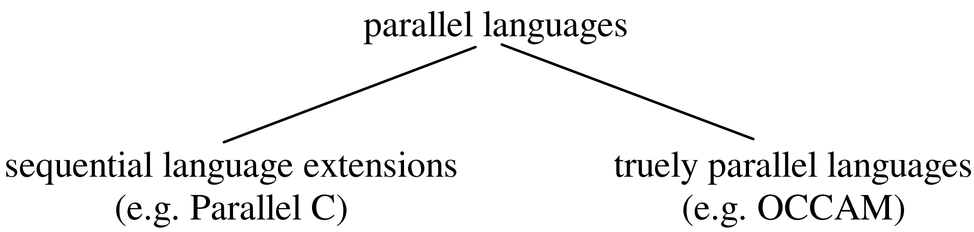


Specification of Parallel Algorithms



- **Implicit parallelism** : Parallel code is produced by the compiler from the sequential description of the problem by extracting implicit parallelism automatically ⇒ *parallelizing compilers*

- **Explicit parallelism**: Parallelism is defined explicitly by the programmer ⇒ *parallel languages*



A Visual Parallel Programming Language : GRAPNEL

(Graphical Process Net Language)

Programming Paradigm: message-passing

- parallel program consists of individual sequential processes
- processes are executed in parallel
- processes can interact only by sending and receiving messages (i.e. data) among themselves

]

Process Model:

Process

single unit — member of a group

Process

static — dynamic

Communication Model:

Communication

point-to-point (send-recieive) — group (e.g. multicast)

Hybrid approach: graphics are used together with ordinary textual language to define the program.

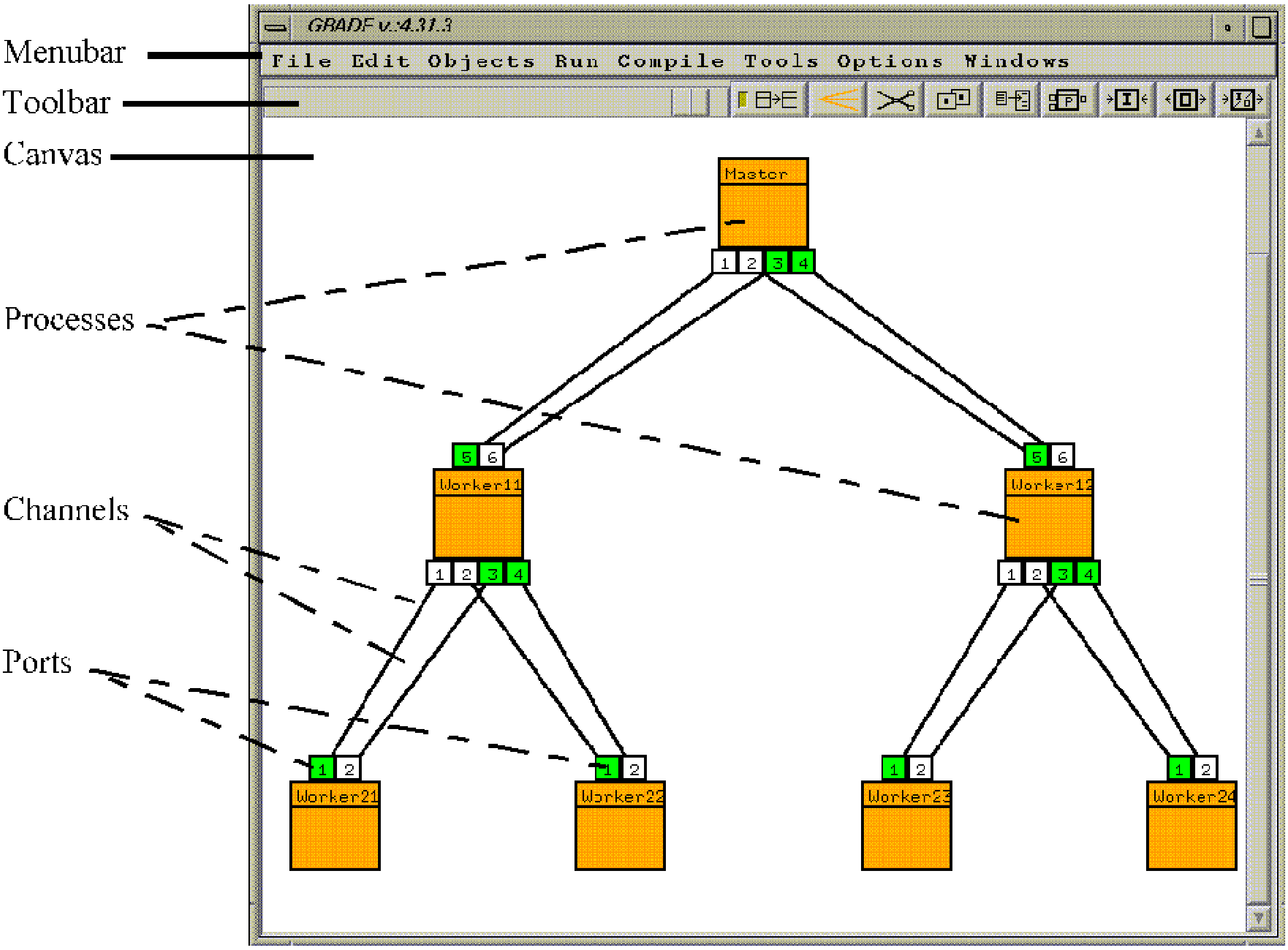
- Parallel activities (e.g. process creation, communication) are defined by graphical symbols.
- Program parts not related to parallelism can be defined in C.

GRAPNEL (Cont.)

Layered design of parallel programs: three hierarchical levels are distinguished.

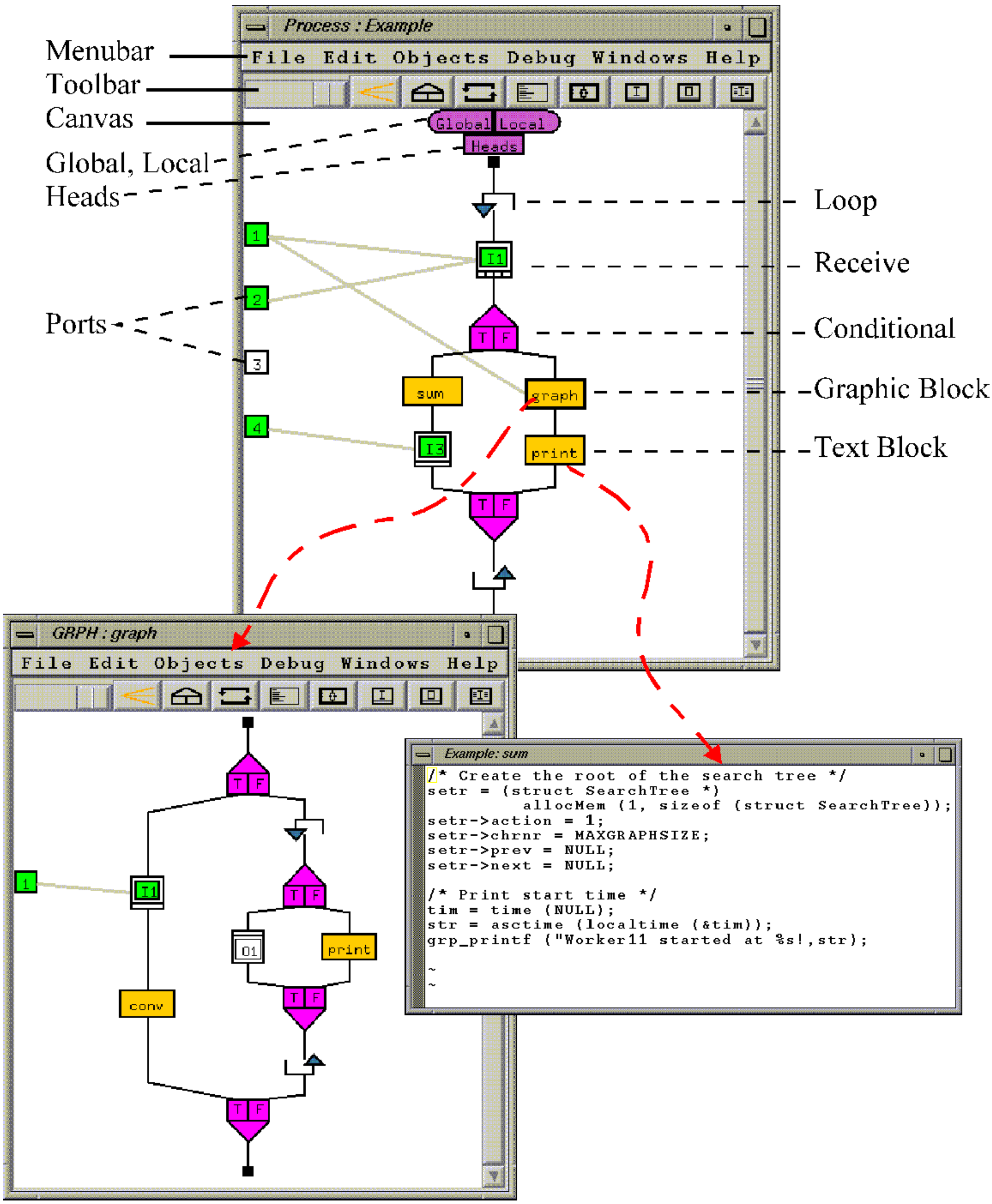
- *Application Level*: processes and their communication connections are described.
- *Process Level*: graphical outline of process's code based on the message-passing and process management activities
- *Textual Code Level*: low level semantics of the visual code elements.

Application Level:



GRAPNEL (Cont.)

Process Level:



Load Balancing

Problem: load imbalance leads to processor idle times in a distributed parallel computer ⇒ high efficiency can only be achieved if the computational load is evenly balanced among the processors.

Two kinds of load balancing schemes can be distinguished depending on the knowledge about the application behaviour:

Load Balancing Schemes

Mapping (Static Load Balancing)	**Dynamic Load Balancing**
The task-processor assignment is performed before the parallel application initiates its execution.	The task-processor assignment is performed and changed dynamically during the actual execution of the application.
⇓	⇓
Computational and communication requirements of a problem must be known a priori.	It must be applied in situations where no a priori estimations of load distribution are possible.

Mapping

Goal: to minimize the execution time of the parallel programs on distributed-memory machines by distributing parallel parts of the program (i.e. tasks) among the different processors in an "optimal" way.

Distributed parallel computer:

- a set of processors connected via an interconnection network
- each processor has a local memory
- communication between two processors is much more time consuming than a local memory access

Description of the problem: A mapping is an application (called *alloc)* from *T* (set of tasks) to *P* (set of processors) which associates to each task *t* an unique processor $q = alloc(t)$.

In general, the complexity of the mapping operation is exponential ⇒ heuristic solutions are proposed.

Quality of the mapping

Solutions of the mapping problem are based on the optimazition of a cost function, denoted *z*. Many choices have been proposed for *z*. Two opposite criteria have to be taken into account:

1. minimization of inter-process communications
2. load-balancing of computations among processors

For example, to minimize the most loaded processor is a trade-off between these two criteria:

$$z = \max_{p \in P}\Big(\sum_{t | alloc(t) = p} ex(t) + \sum_{t' | alloc(t') \neq p} comm(t,t') \Big)$$

where

ext(t) : computation time of task *t*

comm(t,t') : the total communication time between *t* and *t'*

Greedy algorithms

The mapping is done without backtracking (a choice already done can never be reconsidered). The allocation of the i^{th} task is based on a criterion depending on the mapping of the $(i\text{-}1)^{th}$ first tasks.

Modulo: The i^{th} task is allocated onto the i^{th} *modulo m* processor. Theoretically, this algorithm has the same behavior as a random mapping algorithm with a great number of tasks.

Largest Processing Time First (LPTF)**:** Tasks are first sorted by decreasing computation cost order, then allocated on the less loaded processor.

Largest Global Cost First (LGCF): Both communication and computation costs are taken into account. Tasks are first sorted according to this order, then allocated on the less globally loaded (communication and computation costs taken into account) processor.

Iterative algorithms

- They try to improve an initial solution (usually obtained by greedy algorithm).
- Usually tasks are exchanged between processors to improve locally a solution.
- Usually random perturbations are used to leave local minima of cost function.

Boillat method

At each iteration step processes move between neighbour processors to meet one of the two optimizations demands:

- minimizing communication cost
- minimizing load imbalance

The selection is based on a random number (r). Communication cost is selected with reducing probability.

```
t:=0
while (t < max_iter) do begin
    r = decrease_random(t);
      for i:=0 to max_processors do
        for k:=0 to max_task_on[i] do
            for j:=0 to max_neighbours[i] do
                if r = TRUE
                    communic(k,j);
                else
                    load_balance(k,j);
    for i:=0 to max_processors do
        for j:=0 to max_neighbours[i] do
            if r = TRUE
                do_max_communicating_move(alloc(k)=i);
            else
                do_best_improving_allowed_move(alloc(k)=i);
end
```

Heuristic move-exchange (HME)

Tasks are randomly assigned to the processors. One iteration step consists of:

1. move tasks while moves decrease the cost function
2. perform an exchange if possible and try to move again (go step 1)
3. if there are no proper moves and exchanges then append the current configuration to a queue of fixed length. If the costs in the queue are in descending order and the queue is not full then perform random moves and go step 1. If they are not then stop if the queue is full, otherwise delete all but the last element of the queue, perform random moves and go step 1.

The best configuration is always recorded.

```
best_config:=init;   /* tasks are randomly assigned to processors */
while (loop) do begin
    loop1:=TRUE;
    while (loop1) do begin
        loop2:=TRU;
        while (loop2) do
            if exist_moves_cost_decr() then
                best_move();
            else
                loop2:=FALSE;
        if exist_exchanges_cost-decr() then
            best-exchange();
        else
            loop1=FALSE;
    end
    join_queue(current);  /* the current value is appended to a queue */
    if cost(current) < cost(best_config) then
        best_config:=current;
    if queue_IS_descending_order_cost_value() then
        random_moves();
    else
        if queue_IS_full() then
            loop:=FALSE;
        else
            reset_queue(); /* all but the last item in the queue are deleted */
            random_noves();
end
```

Simulated Annealing

Describes the process of eliminating lattice defects in crystals by heating followed by slow cooling to low temperature.

In mapping a lattice defect corresponds to a mapping M with

$$CF(M) > optimal\ value$$

The system is simulated in thermal equilibrium. As parameter T (a formal analogy of temperature) is decreasing, the system is frozen to a state with the current value CF(M) low.

If the mapping M is transformed onto another mapping Mnew, then the resulting mapping is accepted if it gives the lower value of CF:

$$CF(Mnew) < CF(M)$$

Else, the new mapping is accepted with probability

$$exp\{\ (CF(M) - CF(Mnew)) / T\ \}$$

In each temperature step a pool N of mappings can be examined.

Initialization: The procedure is initialized by random generation of N mappings. Starting temperature Tmax must be chosen so that the initial percentage of succesful mappings should be about 50% =>

Parameter setting requires some experimentation !

```
best_config = init
T = Tmax
while T > Tmin do begin
   while (moves < hw*sw*Npool) do begin
      move_task
      delta = CostFnew - CostF
      if delta < 0 then
         accept_configuration()
      else begin
         r = random();
         if r < exp(-delta/T) then
            accept_configuration()
      end
      T = decrease.T()
end
```

Dynamic Load Balancing (DLB)

DLB is needed when the communication and/or computational costs of tasks can not be estimated properly before the execution of the parallel program.

The general DLB process can be divided into four phases:

1. ***Processor Load Evaluation***: A load value is estimated for each processor in the system to detect load imbalances and make load migration decisions.
2. ***Load Balancing Profitability Determination***: Potential speedup obtainable through load balancing and is weighed against the load balancing overhead to determine whether or not load balancing is profitable at that time.
3. ***Task Migration Strategy:*** Sources and destinations for task migration are determined.
4. ***Task Selection Strategy:*** Source processors select the most suitable tasks for efficient and effective load balancing and send them to the appropriate destinations.

	Application Dependent	**Distributed**
Processor Load Evaluation	YES	YES
Load Balancing Profitability Determination	NO	NO: large overhead more accurate YES: low overhead less accurate
Task Migration Strategy	NO	YES or NO
Task Selection Strategy	YES	YES

Dynamic Load Balancing
(DLB)

The main issue in any load balancing scheme is to find a *tradeoff* between *knowledge* and *overhead*.

Load balancing overhead includes:

- comm. cost of acquiring info on load
- processing costs of evaluating profitability
- informing nodes of load migration decisions

Profitability Determination

Load *imbalance factor* $\Phi(t)$ is an estimate of the potential speedup obtainable through load balancing at time *t*:

$$\Phi(t) = L_{max} - L_{balanced}$$

where L_{max} : max processor loads before balanicing

$L_{balanced}$: max processor loads after balancing

Load balancing is profitable if:

$$\Phi(t) > L_{overhead}$$

where $L_{overhead}$: balancing overhead

Simplification: setting $L_{overhead}$ to a constant value:

$$\Phi(t) > K_{overhead}$$

Further simplification: once a node's load (L_p) drops below a preset threshold ($K_{underload}$) balancing is needed anyway:

$$L_p < K_{underload}$$

The Gradient Model (GM)

1. Based on two threshold parameters:

Load: light | moderate | heavy

LWM — Low-Water-Mark; HWM — High-Water-Mark

2. A nodes's *proximity* is the shortest distance from the node to the nearest *lightly* loaded node.
3. proximity(p) = 0 if p becomes *light*
4. All other nodes p with near-neighbours n_i compute:

$$\text{proximity}(p) = \min_i(\text{proximity}(n_i)) + 1$$

5. All nodes are initiated with:

   proximity(p) = Wmax (diameter of the system)

6. Load balancing is initiated if a node becomes *light* $\Rightarrow$ demand driven approach
7. If a processor's state is *heavy* and any of its near-neighbours report a proximity less than Wmax, it sends a load unit to its neighbour with lowest proximity.

   Tasks are routed in the direction of the nearest *light* node.

   A task continues to migrate until it reaches

   - a light node or
   - a node for which no neighbours report lower proximity
8. ***Disadvantage***: It may work inefficiently if the load unit is either too small or too large.

Sender Initiated Diffusion (SID)

- Purely distributed and asynchronous
- Overlapping balancing domains are applied to achieve global balancing.
- A balancing domain consists of a node and its immediate neighbours $\Rightarrow$ near-neighbour oriented method.
- All processors inform their near-neighbours of their load levels throughout program execution.
- Balancing is performed by each processor whenever it receives a load update message indicating that a neighbor's' load $l_i < L_{LOW}$ where L_{LOW} is a preset threshold.

Profitability determination and task migration

1. Compute the average load in the domain:

$$L_p = \frac{1}{K+1}(l_p + \sum_{k=1}^{K} l_k)$$

K: number of neighbouring processors

l_k : their respective loads

2. Task migration is initiated by a processor if its load exceeds the average load by a prespecified amount, $L_{threshold}$.
3. Task migration is performed by apportioning excess load to deficient neighbors. Each neighbor k is assigned a weight h_k according to the following formula,

$$h_k = \begin{cases} L_p - L_k, & \text{if } l_k < \overline{L_p} \\ 0 & \text{otherwise.} \end{cases}$$

These weights are summed to determine the total deficiency,

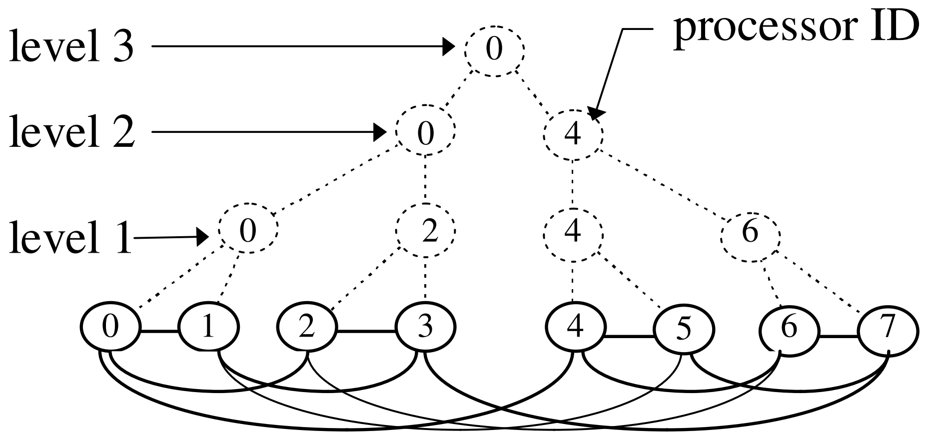
$$H_p = \sum_{k=1}^{K} h_k$$

The portion of processor p's excess load that is assigned to neighbor k:

$$\delta_k = (l_p - \overline{L}_p)\frac{h_k}{H_p}$$

Hierarchical Balancing Method (HBM)

1. Hierarchy of balancing domains ⇒ decentralized balancing scheme
2. Hierarchical organization of an 8-processors hypercube system:



3. Level l_i receives load info from both lower level l_{i-1}
4. *Global balancing* is achieved by balancing the load between **adjacent** domains at each level of the hierarchy.

⇓

asynchronous activity

5. *Profitability criterion* (evaluated at each level):

$$\left| L_L - L_R \right| > L_{threshold}$$

6. Processors within the overloaded branch transfer a designated amount of their load to their "matching" neighbor in the adjacent underloaded subtree.
7. Given an imbalance, Δi, at a level i in the hierarchy, each processor p in the overloaded subtree transfers a load amount δ_i to its corresponding neighbor q in the underloaded subtree, where

$$\delta_i = \frac{\Delta_i}{2^i}$$

Reason: at level i-1 all processors have equal loads.

The Load Update Strategy

Quality of load information depends on:

1. ***accuracy of load estimates***
    - application dependent
    - tradeoff between quality of estimate and complexity of estimation. Examples:
        - ♦ rough estimation: length of processor's queue
        - ♦ more accurate: distinguish the type of tasks in the queue
2. ***aging of information***
    - machine architecture
    - dependent load balancing strategy dependent
3. ***frequency of load update messages***

    The interval between update messages can be computed as a function of time or as a function of load level.

    - *function of load level*

        The updates may be sent at constant intervals, ΔL, of a processor's load, L_p or, the intervals may vary as a function of the load level.

        - ♦ *constant intervals*: processor p must send on the order of $L_p/\Delta L$ update messages. The error percentage increases when L_p decreases (100% when $L_p \approx \Delta L$).
        - ♦ *variable update intervals*: constant error percentage in the load info and less update messages.

            E.g.: a processor must send update messages whenever its load relative to that sent in its last update message has doubled or cut in half (.i.e. load update factor is 1/2).

            The frequency of messages will increase as the processor loads decrease ⇒ accuracy of load info does not degrade.
    - *function of time*: time intervals need to be adjusted for different applications

DEBUGGING SEQUENTAL PROGRAMS

Deterministic behaviour

$\Downarrow$

applicable techniques

trace by log file	breakpoints	step-by step execution

DEBUGGING CONCURRENT AND PARALLEL PROGRAMS IN MULTIPROCESSOR ENVIRONMENT

PROBLEMS;

1. Concurrent programs are nondeterministic

$\Downarrow$

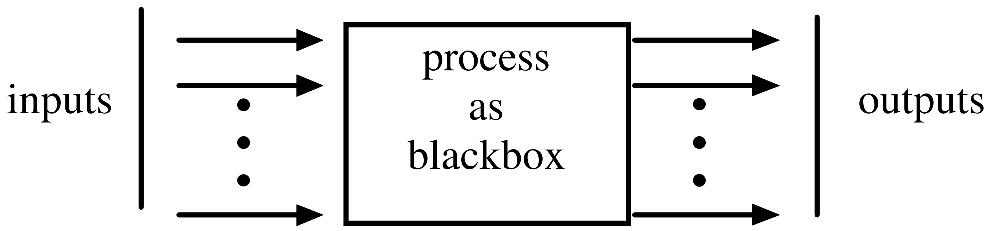
breakpoints, step-by-step execution are not applicable

2. Observibility in multiprocessor environment

a./ report messages from different processors are mixed in time

b./ correct log messages can be lost due to the error of a processor within the communication chain

THE BASIC OF REPLAY TECHNIQUE



If the - value and
- order
of the inputs are the same at each run, then the process will work deterministically, i.e. generates:
- the same values
- in the same order
at each run.

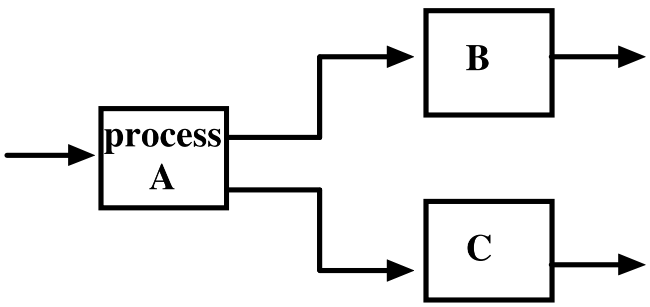
$\Downarrow$

Reproducible behaviour

Execution Phase
Collecting data

ReplayPhase
Replay based on collected data

Reducing data to be collected



If the outputs of A are inputs of B and C, there is no need to collect values (data) on this arcs, since they will be reproduced by A during the Replay Phase.

REPLAY TECHNIQUE APPROACHES

Implementation based

support is provided by modifying the language implementation (i.e. compiler, run-time system, operating system)

Language based

The program is transformed into a new form in the same language containing all necessary auxiliary code for data collection and replay.

Information that must be collected for replay **depends on the particular language** (i.e. depends on **synchronisation** and **communication constructs).**

Example: Language based replay for monitors

The types of synchronization events that occur during an execution of a concurrent program consisting of processes and monitors include

- the start of execution of a monitor procedure by a process,
- the completion of execution of a monitor procedure by a process,
- the execution of a *Wait(condition)* operation in a monitor procedure,
- the execution of a *Signal(condition)* operation in a monitor procedure.

Thus, the execution of such a program can be characterized as a sequence of synchronization events called *M* sequence:

((C1,H1,D1,N1),(C2,H2,D2,N2),...)

where Ci: the calling process
Hi: type of the event
Di: the called monitor procedure
Ni: the condition variable (in case of *Wait* and *Signal*)

The execution of a monitor based program can be determined by the M sequence and the input data.

Simplification of *M sequence* ⇒ *SM sequence*:

(C1,C2,...)

As the sequence of synchronization events inside a monitor can be determined from the sequence in which processes enter the monitor and the values of the parameters of these monitor calls ⇒ the **SM sequence and the input data** determines the execution of a monitor based program.

Example: Language based replay for monitors (Cont.)

original monitor program

```
monitor: buffer

procedure put (item: char)
begin
  if (count > size) wait (not_full)
    ⋮
end;

procedure get (varitem:char)
begin
   if(count=0) wait (not_empty);
    ⋮
end;

begin
  ⋮
end;
```

transformed monitor program

```
monitor: buffer

procedure put (item: char)
begin
   release_m_permit;
   if(count>size) wait(not_full);
   ⋮
end;

procedure get (varitem:char)
begin
   release_m_permit;
   if(count=0) wait( not_empty);
   ⋮
end;

begin
  ⋮
end;
```

original call sequence

```
process producer

begin
  ⋮
  put (item);
  ⋮
  put (item);
end;
```

transformed call sequence

```
process producer
const id = 1;
begin
   ⋮
   request_m_permit (id);
   put (item);
   ⋮
   request_m_permit (id);
   put (item);
end;
```

Example: Language based replay for monitors (Cont.)

```
process consumer

begin
   :
   get ( item);
   :

end;
```

```
processor consumer
const id = xx;
begin
   :
   request_m_permit (id);
   get ( item);
   :

end;
```

```
monitor control;

      var sm_seq : array [ 1...MAX_Length ] of integer;
      process_queue : array [ 1...MAX_ID ] of condition;
      index : integer;
      sm_length : integer (*actual length of SM-sequence*)
procedure request_m_permit (ID : integer );
begin
      if (ID<>sm_seq [index] wait (process_queue [ID])
end;
procedure release_m_permit;
begin
      index ++;
      if(index<=sm_length) signal (process_queue[sm_seq[index]])
end;

begin
      index = 1;
      read (sm_length);
      (*read in the elements of array sm_seq[1...sm_ length]*)
end;
```

Possible SM sequences:

[1, 1. 2]
[1, 2, 2]
[2, 1. 1]

should be generated by a systematic test program

Example: Implementation based replay for message-passing

synchronous message-passing communication **primitives**

$\Downarrow$

replay can be implemented by forcing on each process p_i the same order of message receptions.

$\Downarrow$

trace information for p_i, recorded during execution phase, is thus simply a **sequence of message identifiers**.

non-blocking primitives $\Rightarrow$ situation is **more complicated**

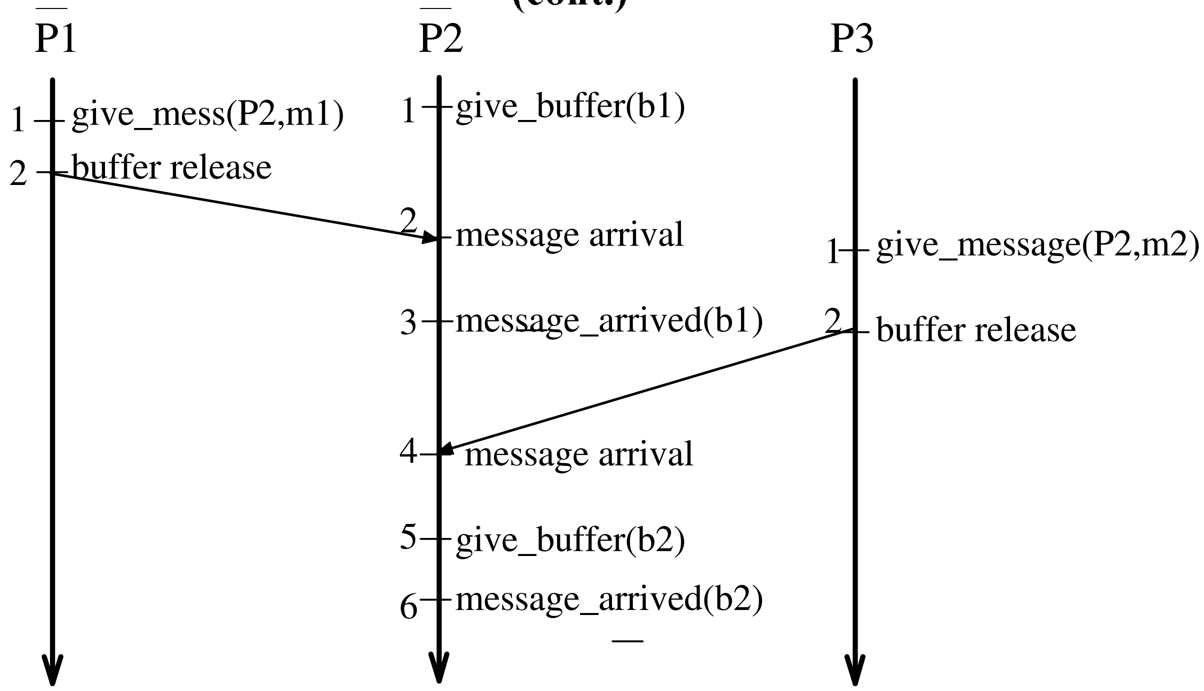
Non-blocking primitives are used to overlap communication and computation:

1. **give_buffer(buf):** gives a buffer (in the process space) to the operating system**,** into which a message is to be stored;
2. **message_arrived(buf):** returns true if the process buffer contains a message;
3. **give_mess(dest,buf):** gives to the operating system the destination process and the buffer containing a message to send;
4. **message_sent(buf):** returns true if the buffer can be reused (i.e. the message has already been sent by the message passing system).

Typical use of such primitives:

process P1	process P2
. . .	**. . .**
give_message(P2, buf1);	**give_buffer(buf2);**
. . .	**. . .**
if message_sent(buf1) then	**if message_arrived(buf2) then**
"buf1 can be reused"	**"access to buf2"**
else	**else**
. . .	**. . .**

Example: Implementation based replay for message-passing (cont.)



Initial Execution

a./ **Sensitive events (communication primitives)**: no recorded event, only increment Counter

```
give_mess(Pi , Mj):       inc(Counter)
                          call $GIVE_MESS(dest, buf);
                               /*effective system call*/
give_buffer(bi):          inc(counter);
                          call $GIVE_BUFFER(buf);
message_arrived(bi):      inc(counter);
                          call $MESSAGE_ARRIVED(buf);
```

b./ **Implicit events (message arrival and buffer release):** increment event counter + record event

```
message_arrival:          inc(Counter)
                          SaveInLogFile (Counter, sender_proc,
                                         ARRIVAL);
buffer_release:           inc(Counter);
                          SaveInLogFile (Counter, dest_proc,
                                         DEPARTURE)
```

⇓

P1's trace	P2's trace	P3's trace
(2, P2, DEPARTURE)	(2, P1, ARRIVAL)	(5, P2, DEPARTURE)
	(4, P3, ARRIVAL)	

Implementation based replay for message-passing (cont.)

Replay Execution Phase

a./ **Sensitive events**

The replay is forced (when sensitive events are executed) by calling the procedure **before_sensitive_event**.

For example:

```
function message_arrived (buf: buffer type) : boolean;
begin
        inc (counter);
        if MODE = REPLAY then
                before_sensitive_event;
        end if;
        return $MESSAGE_ARRIVED (buf);
                                        /* effective system call */
end
procedure before_sensitive _event;
        /* only called during execution replay */
begin
        while Counter = next_recorded_event.FIELD_1 do
                if  next_recorded_event.FIELD_3 = ARRIVAL
                then
                        wait until a message sent by
                        "next_recorded_event.FIELD.2" is
                        available, and put it into the first empty buffer
                        specified by the process;
                else    /*DEPARTURE*/
                        sent the first available message destinated to
                        "next_recorded_event.FIELD_2";
                end if
                next_recorded_event: = getFromLogFile( );
                inc(counter);
        end while
end
```

b./ **Implicit event** is allowed to take place only if it corresponds to the next awaited event of the process. Otherwise, the arrived message or the buffer release information are put into a temporary wait queue, to be handled later on.

Performance Analysis

Basic steps of the performance analysis process:

1. ***data collection***: data about program performance are obtained from an executing program.

   Three basic data collection techniques can be distinguished:

   a) ***Profiles*** record the amount of time spent in different parts of the program. Gathered automatically with sampling technique.

   E.g.: they can help to identify **program parts** that are taking the most time and that hence **require further investigations**.

   amount of data to be collected

   b) ***Counters*** record either frequencies of events or cumulative times. The insertion of counters may require some programmer intervention.

   E.g.: they can record **number of procedure calls**, total **number/volume of messages**.

   c) ***Event traces*** record each occurrence of various specified events, thus typically producing a large amount of data. Traces can be produced either automatically or with programmer intervention.

   E.g.: they can be used to determine casual **relationships between communications**, to localize **idle times and hot spots**. A trace **can be postprocessed** to obtain profile or count info or higher order statistics.

1. ***data transformation***: reducing total data volume, computing mean values or other higher-order statistics, etc.

2. ***data visualization***: performance data are inherently multi-dimensional consisting of execution times, communication costs, and so on, for multiple program components, on different processors => visualisation is needed to explore it.

   Data can often be reduced to one, two or three dimensions and displayed using various histograms techniques, e.g.:

   - ***Gantt chart*** *is* a horizontal bar chart in which each bar represents the status (computing, communicating, or idling) of each processor as a function of time. It can highlight unexpected dependencies between program components.

   - ***Space-time diagram*** is a Gantt chart augmented by drawing lines to connect corresponding send and receive events on different processors. It can make it easier to infer temporal dependencies.

Example visualization tool: **Paragraph**

Paragraph is a portable trace analysis and visualization package developed at Oak Ridge National Laboratory for message-passing programs.

Trace file:

- Originally was designed to analyze traces generated by the PICL message-passing library.
- PICL can generate traces automatically.
- Traces are time-stamped events for every communication operation (by default).

Displays:

- Large number of detailed or synthetic views.
- *Processor utilization* displays: allow the user to distinguish time spent computing, communicating, and idling. They can be used to identify load imbalances and communication hot spots (e.g.: Gantt chart).
- *Communication displays:* present more detailed information on communication volumes/patterns and on causal relationships.

  For example, the communication matrix display indicates, by different colors, the communication volume between all pairs of processes.

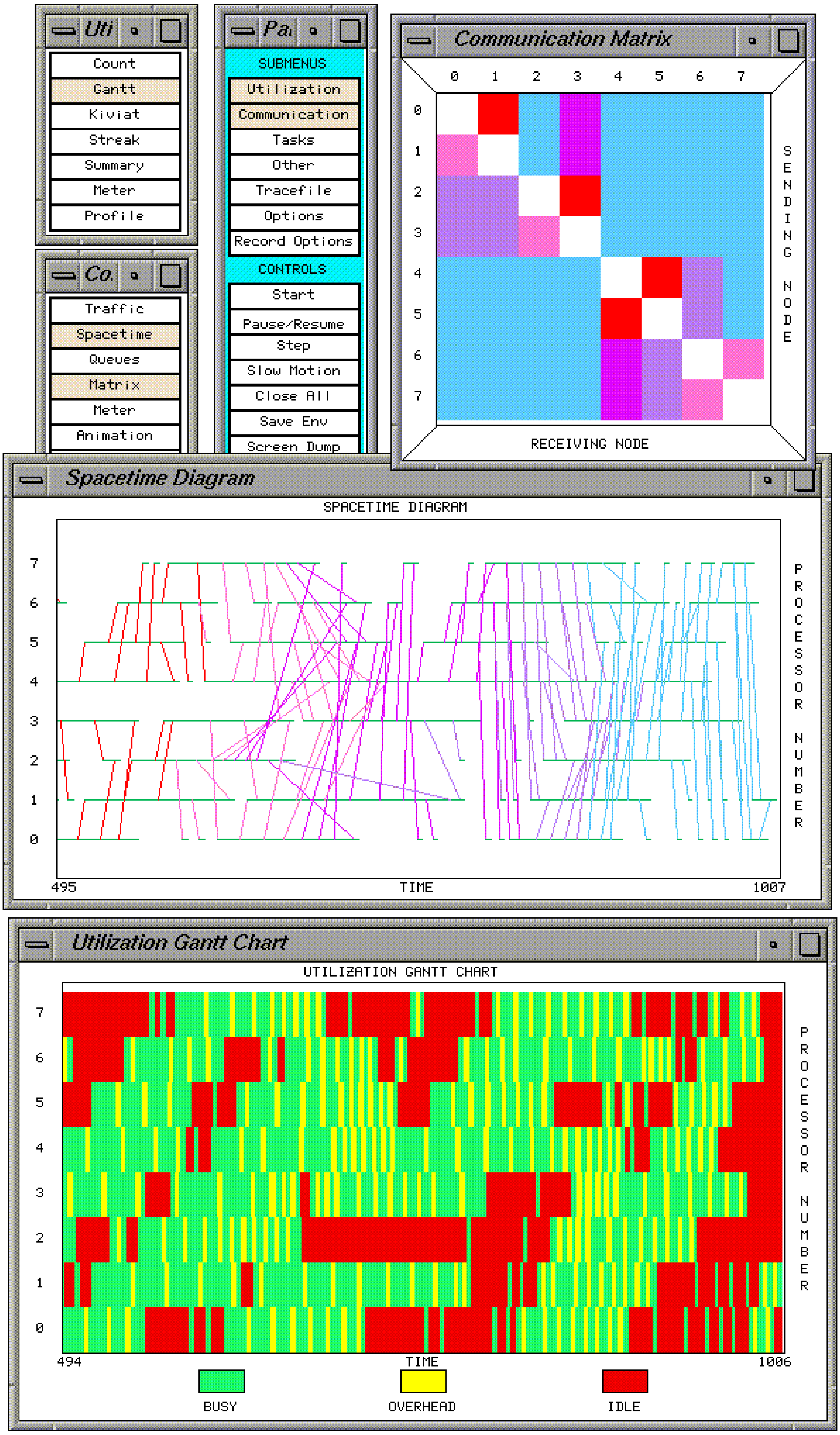
Advantages:

- PICL trace format is *de facto* standard.
- Large number of particularly intuitive displays.
- Widely distributed, pioneer visualization tool.

Disadvantages:

- The relationship between performance data and program source is not always clear.
- No scroll and zoom mechanisms are provided to focus on particular parts of display windows.

Example visualization tool: **Paragraph** (cont.)



Example visualization tool: **Prove**

Prove is a performance visualization tool developed at KFKI-MSZKI Research Institute of the Hungarian Academy of Sciences. The tool is part of the GRADE integrated program development environment for message passing programs.

Trace file:

- Prove can analyze event traces generated by the Tape/PVM monitoring system.
- Traces can be generated automatically.
- Tape/PVM trace format is similar to that of PICL but the event records contain source code info as well.

Displays:

- *Gantt chart*
- *Space-time diagram*
- *Communication displays*: show the volume and distribution of communication among processes or hosts (processors).

Advantages:

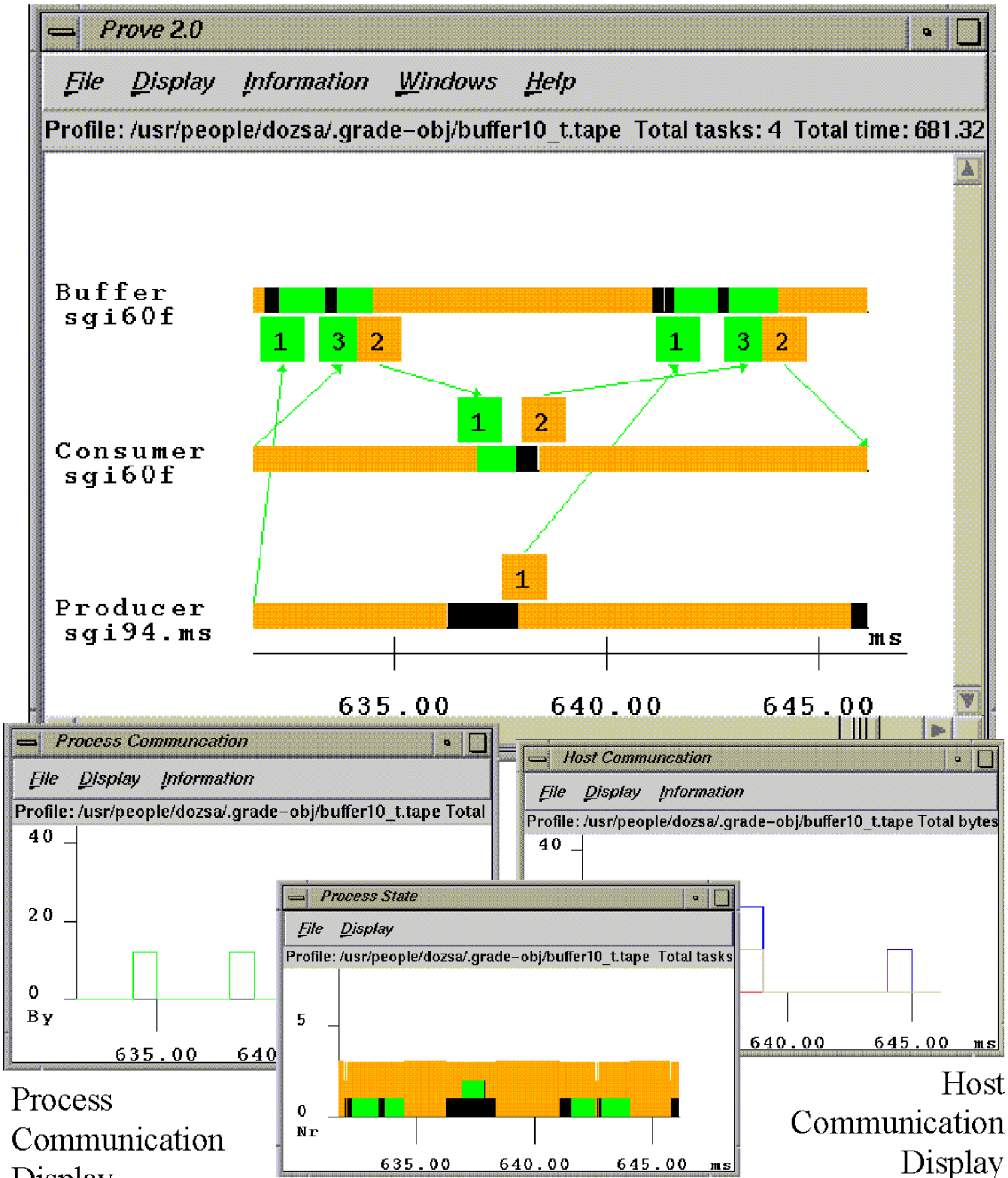
- Zoom and scroll mechanisms are provided to investigate any parts of the views in details.
- Particular processes and/or communication paths can be filtered out by the user interactively.
- Tight integration with the graphical program editor of the GRADE environment $\Rightarrow$ the correspondence between the source code and the events visualized in the displays are always straightforward.

Disadvantage:

- Small number of different displays (comparing to Paragraph).

Example visualization tool: **Prove** (Cont.)

Space-time Diagram



Process Communication Display

Gantt Chart

Host Communication Display

MONITORING

Collection of execution information of a program (i.e. *monitoring*) is fundamental to both *debugging* and *performance analysis*.

MONITORING

by hardware	**by softwarehooks**	**by hw/sw**
difficult to implement	easy to implement	difficult to implement
no intrusion	high intrusion	low intrusion

<u>Software hooks:</u>

Process P

event E;

transformed
⇒

Process P

extract_trace_
info
event E;

results in probe effect

<u>Probe effect</u>

program behaviour is affected by the time spent on monitoring instructions

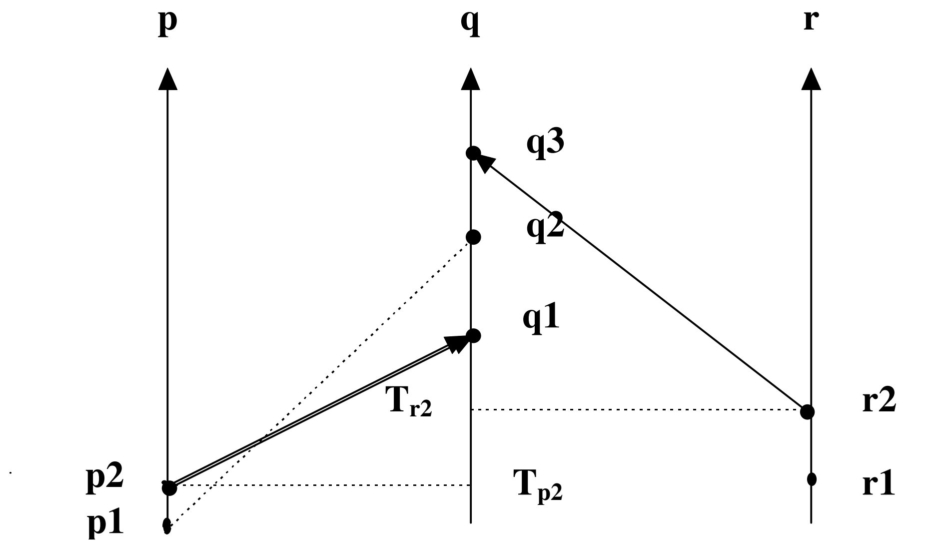
⇓

monitored program runs differently from unmonitored one

Objective: To reduce probe-effect as much as possible.
(In practice probe-effect cannot be reduced to zero.)

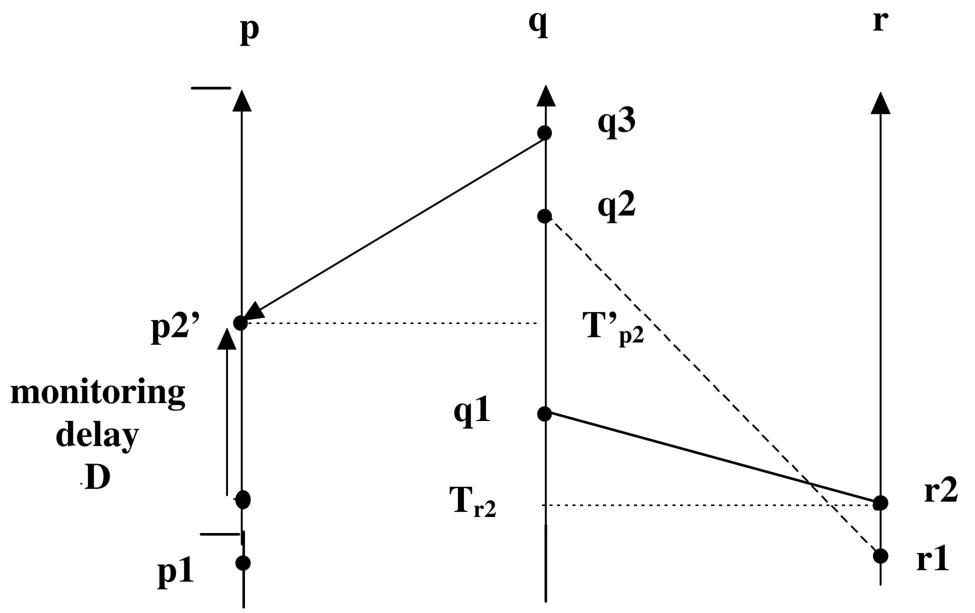
PROBE EFFECT

1. Without monitoring



p1 → q2   *AND*   *r1* || q2

2. With monitoring



r1 → q2   *and*   *p1* || q2

To avoid this probe-effect the execution of processes q and r must also be <u>artificially delayed</u> for the amount of time D.

A possible solution: <u>Logical Clock Approach</u>

THE LOGICAL CLOCK APPROACH

Features

1. Distributed Concept: a logical clock monitor is applied on each processor

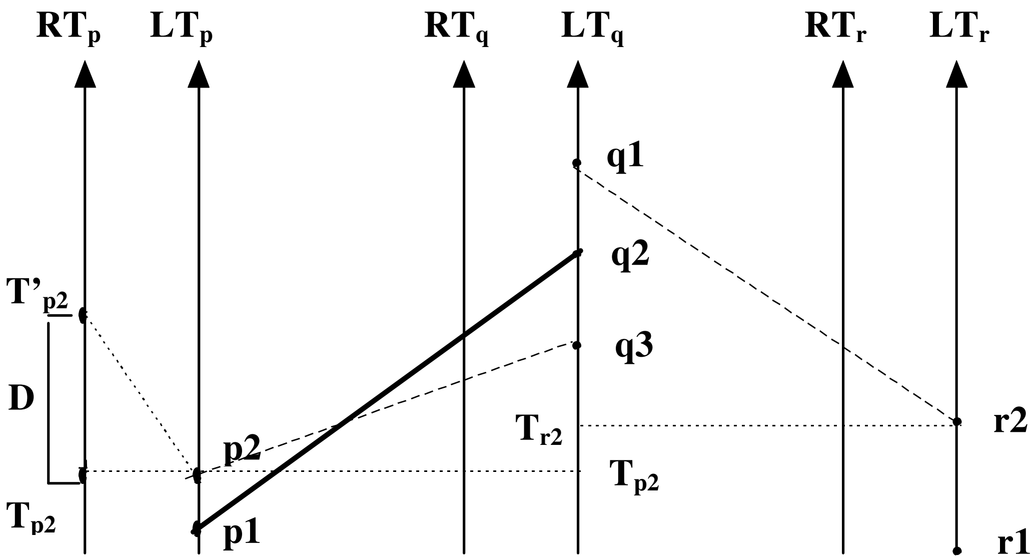
2. The logical clock is stopped during monitoring actions

   ⇓

   The logical clock shows the time that is equivalent with real-time at execution without monitoring

3. communication ordering unchanged
   - sender processes <u>time-stamp</u> messages by their <u>logical clock value</u>
   - receiver process accepts the message <u>with lowest time-stamp</u>

Example:



p1→q2 **r1 || q2**

Consequence: Although, interactive monitoring may slow down the program's execution by several orders of magnitude, there is very little disturbance of the execution time if it is measured using logical time