8 Parallel Programming Tools and Environments

8.1 Introduction

The software engineering world is entering its second major crisis in twenty years. In the sixties, advances in computer architecture enabled the development of larger and more complex programs which inevitably began to fall over with increasing frequency. The introduction and rapid acceptance of parallel computer platforms, ranging from high-performance supercomputers to networks of distributed workstations has again signalled another degree of computer architecture complexity. Programs targeted for these platforms are no less complex than before, but whereas programmers could hitherto rely on a simple and stable programming model, based on von Neumann machine principles, this is no longer the case. The possibility of infinitely variable parallel hardware topologies introduces two more dimensions of program design: hardware configuration; and program mapping. Parallel programmers are expected to navigate the 3-dimensional design space (software, hardware and mapping) - having barely rnastered the complexities of one-dimensional software design for serial computers! The response to the original software crisis was the stimulus for modern software engineering. Amongst the techniques were structured programming, formal specification, etc. It is expected that many existing techniques can be extended to encompass the new parallel paradigms. However, many assumptions made by serial software engineers no longer hold. For example, according to the traditional lifecycle models, performance engineering is a relatively low-priority activity. If it appears at all, it is at the end of the life-cycle, and then, occupies a tiny part of the designers attention. Yet in high-performance computing, an area in which parallel processing is currently being applied, clearly performance is a dominant issue. Clearly, life-cycle models for conventional serial programming are insufficient to the task of parallel software engineering.

In response to this new problem, many researchers have begun to develop novel approaches to the development of suitable tools and methodologies for parallel programming. The purpose of this chapter is to introduce the fundamental steps of parallel software engineering and to demonstrate some of the tools and programming environments available currently to assist the development process of parallel programs. It is quite difficult to give a brief but comprehensive overview of this field as it is a very active and dynamic research area without widely accepted standards.

The following sections discuss the most important phases of the parallel software development process based on the development cycle shown in Figure 8-1[1]. These are summarized as follows (see the figure).

Specification of Parallel Algorithm

The first problem a user should face is the definition of parallel activities in his/her algorithm, i.e. how to express functional or data parallel partitioning. There are basically two distinct approaches for the programmers to follow:

1. using explicit language constructs to express parallelism or
2. leaving it for intelligent compilers to exploit inherent implicit parallelism of the sequential code.

Due to the lack of space, this course is concentrated on explicit parallel languages and skips techniques of automated parallelising of sequential programs.

Mapping and Load Balancing

After specifying the parallel activities in the program the next step is the distribution of parallel tasks and data partitions in the multiprocessor space. The mapping can be controlled either by the user or by the underlying parallel programming environment. Obviously the latter case is the preferable one and even in the former case some supporting tools should be provided for the user to control mapping. In order to achieve optimal performance and speed-up with a multiple processor system the mapping function should assure well balanced loading among processors. If the load distribution of the parallel system can vary dynamically during the execution, some kind of run-time support, referred as dynamic load balancing, is needed to ensure the well balanced property.

[1] Note that this can not be viewed a full software engineering lifecycle since no such standard has been established yet.

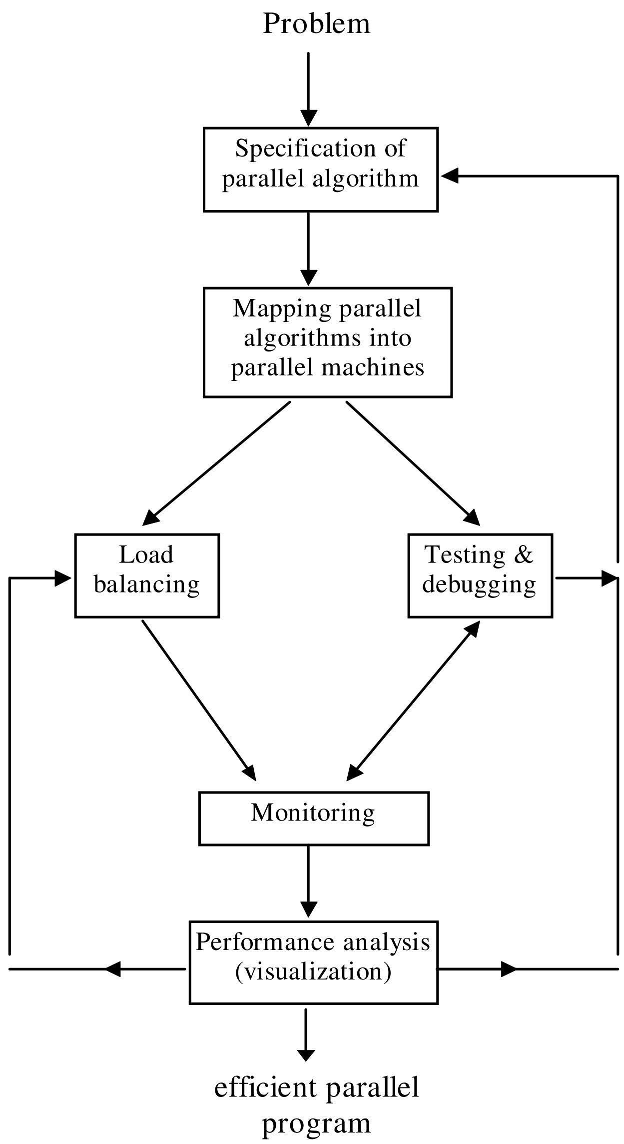


Figure 8-1: Development Cycle

Mapping and Load Balancing

After specifying the parallel activities in the program the next step is the distribution of parallel tasks and data partitions in the multiprocessor space. The mapping can be controlled either by the user or by the underlying parallel programming environment. Obviously the latter case is the preferable one and even in the former case some supporting tools should be provided for the user to control mapping. In order to achieve optimal performance and speed-up with a multiple processor system the mapping function should assure well balanced loading among processors. If the load distribution of the parallel system can vary dynamically during the execution, some kind of run-time support, referred as dynamic load balancing, is needed to ensure the well balanced property.

Debugging

Another key issue of a parallel programming methodology is the verification of parallel programs. Apart from the theoretical approaches, practical ones require systematic test generation and debugging methods. The debugging cycle is the most common methodology for finding and correcting errors in sequential programs. Cyclic debugging is effective because sequential programs are usually deterministic, i.e. they always produce the same results with the same input data. Debugging parallel programs is considerably more difficult because successive executions of the same program with the same input often result in different output and run-time behaviour.

Performance Analysis

The objective of performance analysis is to reveal performance bottlenecks of parallel algorithms and load-balancing. Performance visualization is the most common approach to cope with this stage of developing parallel programs. Usually, this is a crucial phase of parallel program development since in most cases parallelism is introduced so as to gain high performance.

Monitoring

Collection of run-time information requires some kind of monitoring during the execution of the parallel program. Monitoring is fundamental to both debugging and performance analysis of parallel applications. It can provide dynamic execution information for displaying execution states and statistical data for evaluating the performance of a program.

8.2 Specification of parallel algorithms

As the first step of the development process, the programmer must specify his/her parallel algorithm. Figure 8-2. depicts the fundamental methodologies that can be applied to define the parallel activities in the algorithm. Basically, they either can be defined by the programmer explicitly or the compiler can extracts the implicit parallelism and produces the parallel code automatically from a conventional sequential description of the problem. In the latter case we speak about parallelizing compilers.

In this course, we concentrate on parallelism expressed explicitly by the programmer as the use of parallelizing compilers is very limited currently.

In case of explicit parallelism, the programmer applies a parallel language to specify the algorithm. Similarly to the sequential languages, the parallel ones can either be procedural or declarative, object oriented or not object oriented, visual or not visual, etc.

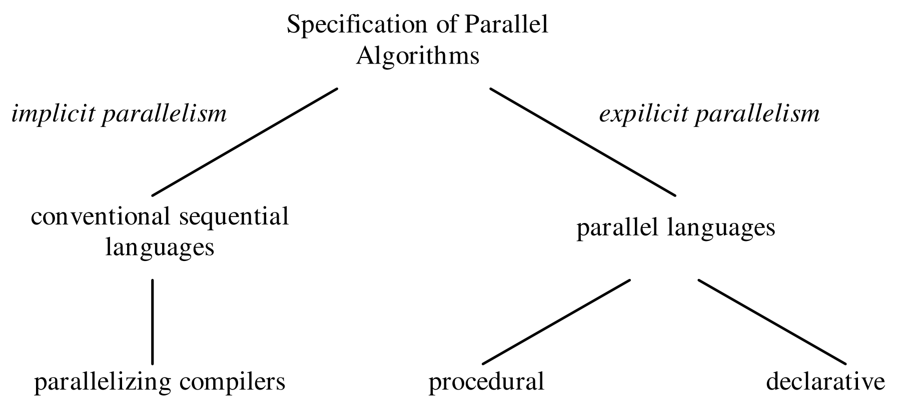


Figure 8-2. Specification Methods

In most cases, a particular parallel language is an extension of a sequential language towards handling parallelism (e.g. plenty of parallel extensions of C exist, see Chapter 1.) but there are few truely parallel language as well which do not have a sequential correspondence (e.g. OCCAM, see Chapter 4.). Pure procedural parallel languages, object oriented procedural languages and declarative parallel languages are covered in different chapters of this course. We discuss another possibility here: the visual programming approach.

A promising alternative way to textual programming. is visual programming which is easier to grasp for non-professional users of parallel computers. Several approaches have been suggested in the literature to program parallel machines based on graph languages. In the graphical environment the user can draw the parallel structure of both the algorithm (task graph) and the hardware topology. Nodes of the task graph can be either sequential programs (written in any textual language) or another task graph. This kind of visual programming environments can incorporate object-oriented design principles as well providing a user-friendly parallel programming environment.

To demonstrate some fundamental concepts, the GRAPNEL visual programming language and its supporting graphical editor is described as follows.

8.2.1 A sample visual parallel programming language: GRAPNEL

GRAPNEL (GRAphical Process's Net Language) has been designed for constructing parallel programs based on the message-passing programming paradigm. In the message passing model, a program consists of individual processes, each of them has its own sequential flow of control and local address space, that are executed in parallel and can interact only by sending and receiving messages (i.e. data) among themselves. More detailed description of message passing can be found in Chapter 5.

In GRAPNEL, a process can be either a single unit, or a member of a process group. Similarly to MPI (see Chapter 5.), a process group is an ordered collection of processes. Both processes and process groups are defined graphically as boxes. Process groups can be used in two important ways. Firstly, they can be used to specify the scope of a collective communication operation, such as multicast (i.e. a message can be received by all members of a group). Secondly, they can be used as an abstraction mechanism to support the structured design at the level of processes, i.e. processes can be put into a group to be managed together as one unit. Since process groups can be nested, they support hierarchical design of the parallel program. Communications among processes are either point-to-point or group communication. Communication always takes place via communication ports which can belong either to processes or to process groups, and which are connected by communication channels. To ensure that the form of the transmitted data (i.e. type and size) matches at both the sender and the receiver sides, each port has its own protocol. Inside the processes, send and receive operations are represented graphically, and they are joined with representatives of ports on which the communication operations are to take place. The user can define the data to be sent or received by simply listing the names of the program variables where the data should be stored or should come from.

A graphical program editor, called GRED, supporting the GRAPNEL language has been implemented as part of the GRADE programming environment. GRED runs on the top of UNIX operating system and applies the X Window system to provide a graphical interface for the user. Currently, GRED applies PVM as its low level message passing mechanism (i.e. programs constructed by GRED can run in any PVM environment) but it can be extended easily to use MPI or other message passing systems without any changes concerning the GRAPNEL language.

GRAPNEL is a hybrid language, in the sense that it uses both graphical and textual code fragments to describe the whole parallel program. Graphics are used to depict those parts of the program which are important with respect to the parallelism but ordinary textual description can be applied to define the rest of the code (e.g. data declarations or definitions, code segments without any communication, etc.). The main purpose of the graphical of the graphical representation is to give a high level outline (or abstraction) of the parallel program where the key points are the communication actions among the processes. By GRED, textual parts of GRAPNEL applications can be written in standard C language.

In GRAPNEL programs, three hierarchical design levels are distinguished. At the top level - *Application Level* - the outline of the whole application is described graphically with respect to communication connections among the processes, while at lowest level - *Text Code Level* - the textual code fragments are given. At the middle level - *Process Level* - the send and receive operations are defined inside the code of a process graphically. These levels are described in top-down order, together with the appropriate GRED's related information, as follows.

Application Level

This is the top level of a parallel application in GRAPNEL. Processes, process groups, communication ports and connections among the processes must be defined here graphically but the functionalities (i.e. the code) of the processes are hidden at this level. In other words, processes are viewed as black boxes here and we are interested in only the message routing among them. Each port must be connected to that of another process (or process group) by a channel. If there is a channel between two processes (or groups) then they can communicate with each other during the execution of the program, otherwise, they cannot.

Fig. 3. depicts the Application Window of the GRED editor which can be used to construct the graphical code at this level of the program. Seven processes can be seen on the canvas of the window represented as rectangles. Small squares clung to them are ports connected by lines representing communication channels.

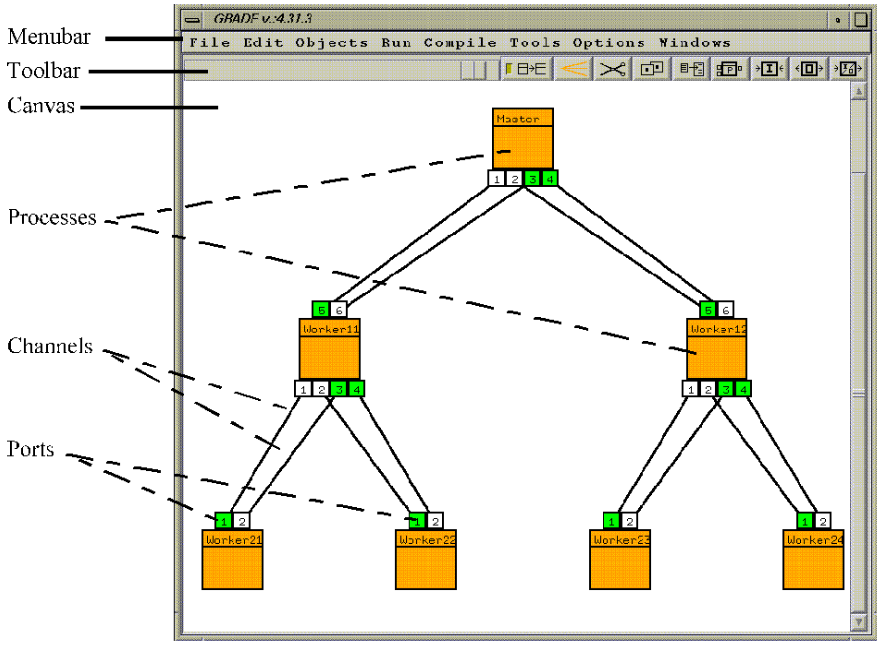


Figure 8-3: Application Window of GRED

Process Level

This level is for graphically describing the message-passing related parts of the control flow of a process. The point is that send and receive operations must appear graphically in the GRAPNEL code. This approach has two relevant advantages. Firstly, the programmers do not have to know the syntax of the underlying low-level message-passing library, i.e. he can just put an icon into the control flow of the process and simple list the variables where the data are to come from (send action) or where they are to be stored (receive action), instead of having to write all the necessary PVM function calls (e.g. different types of pack and unpack routines, buffer management, etc.). They are generated by the editor automatically based on the visual code. Secondly, in the debugging and testing phase all communication events can be animated in the graphical windows that can ease very much the identification of message-passing related bugs in the program. GRED provides graphical symbols (icons) to describe send and receive operations, loop constructs and conditional constructs. However, only those segments of the control flow must appear graphically which contain send or receive actions. The textual block symbol is defined for denoting an arbitrary large and sophisticated textual code fragment that does not contain any send or receive operation, so which can be defined textually at a lower level (i.e. text code level).

In order to support the structured design, the graphical block symbol has been introduced at the process level. A graphical block icon denotes a control flow segment that does contain communication operations therefore, the content of the graphical block is defined graphically[2]. In GRED, the so-called Process Window can be used to create the graphical code at this level. Each GRAPNEL process can have its own Process Window on the screen simultaneously. Figure 8-4. depicts a sample window of this type (leftmost window in the figure). The name of the process is shown in the title of the window. There are three static[3] icons at the top of the canvas labelled as *Global*, *Local* and *Heads*. Global data, local data and include files can be defined for the process as C text code belonging to these icons. All other graphical symbols must be defined by the user according to the task of each particular process. Ports defined in the Application Window (see the previous section) are also represented here at the left edge of the canvas. They can be connected with send or receive icons by the programmer to denote the target or the source of the message transfers.

On the right hand side of the figure, a Graphical Block Window is depicted that can be used to edit the contents of a graphical block. Graphical Block Window has the same appearance and functionalities as the Process Window except that static icons miss at the top of the canvas (i.e. data or include files can not be defined for the process in a Graphical Block Window).

---

[2] The graphical block icon can be viewed as some kind of graphical subroutine.

[3] Static means that these icons are created by the system in case of any GRAPNEL process, i.e. they always present in any Process Window.

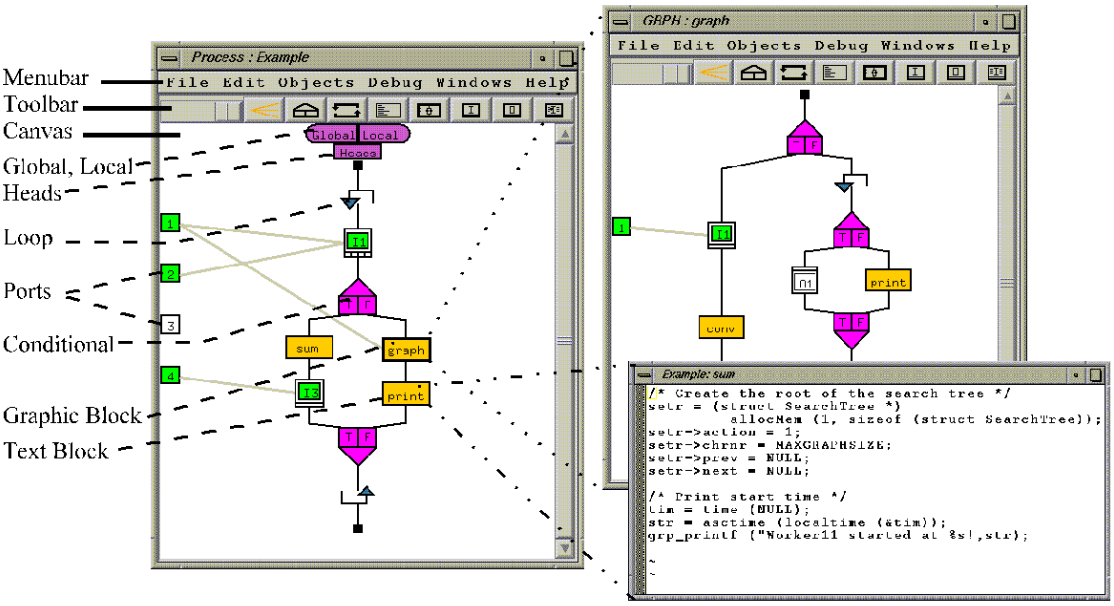


Figure 8-4:Process Level Windows of GRED

Textual Code Level

Every graphical instruction (icon) defined in the Process Window owns a piece of C text code. For example, the text code belonging to a receive icon is a simply a list of variables to store the data to be received. However, a text block icon may own an arbitrary long and sophisticated C code segment which may refer to data or functions defined in external C files or libraries. As a result, C code written earlier can be reused easily. The text editor that is invoked by GRED to edit these text parts of the program can be defined by the user through UNIX environment variables (i.e. without recompiling the system). Thus, the programmer may use his favorite text editors (e.g. emacs, vi, etc.) in GRED.

A sample text editor is also depicted in Fig. 8-4., on the right side containing the C code of the text block labelled as ``print".

8.3 Mapping and dynamic load balancing

Load balancing is one of the central problems which has to be solved to achieve a high performance from a parallel computer. Since load imbalance leads directly to processor idle times, 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. The first one, called mapping or static load balancing, can be used when the computational and communication requirements of a problem are known a priori. In this case, the problem is partitioned into tasks and the assignment task-processor is performed before the parallel application initiates its execution. By contrast, the second approach, dynamic load balancing, must be applied in situations where no a priori estimations of load distribution are possible. It is only during the actual execution that it becomes apparent how much work is being assigned to individual processors. In order to retain efficiency, the imbalance must be detected and an appropriate dynamic load balancing strategy must be devised.

8.3.1 The mapping problem

The goal of the mapping tools is to minimize the execution time of parallel programs on distributed memory machines by controlling the use of computation and communication resources. Mapping algorithms aim at maximizing the (useful) occupation of processors without increasing too much communication costs.

The tasks execution times and inter-tasks communication costs of some regular programs can be entirely determined at compile time. In this case, it is possible to perform static task allocation in advance. This is known as the *mapping* operation whose complexity is exponential in the general case. Thus, it is difficult to obtain an optimal mapping and numerous heuristic solutions have been proposed, representing different trade-offs between computation cost and quality of mapping.

8.3.1.1 Models of machines and programs

A distributed-memory parallel computer is composed of a set of nodes connected via an interconnection network. Each node includes some computation facilities and a local memory. A communication between two processors is much more time consuming than a local memory access. The MIMD model intends to map different executable codes, called tasks, onto processors. Designing a program such that only one task will be allocated on one processor of the target machine would lead to an architecture-dependent and non-scalable code. On the contrary, a too large number of tasks is difficult to manage efficiently. The granularity (size of tasks) is one important parameter for the efficiency of a parallel program.

Most parallel programs can be described using a graph formalism. In most representations, each vertex represents a task and each edge a communication link. We consider that a task can be allocated to a single processor. Any processor can make some communications and computations. We add to this basic model the computation costs of the tasks (execution time) and the amount of information communicated on the links. Often, the user cannot determine the exact values of the program parameters but can only approximate them.

In the following, we will denote: T, the set of tasks and n their number, P, the set of processors and m their number, $ex(t)$, the computation time of task t, $comm(t,t')$, the total communication time between t and t'.

8.3.1.2 Description of the problem

During mapping operation, parallel parts of the program (i.e. tasks) are distributed among the different processors. The objective considered here is to minimize the execution time of the whole program. Formally, a mapping is an application (called *alloc)* from T to P which associates to each task t an unique processor $q = alloc(t)$. The number of all possible solutions is n^m.

8.3.1.3 Quality of the solution

Most solutions of the mapping problem are based on the optimization of cost functions, denoted z. There exist in the literature many choices for z. Two opposite criteria have to be taken into account: minimization of inter-processors communications and load-balancing of computations between processors. For example, to minimize the most loaded processor is a trade-off between these two criteria:

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

This basic function does not consider that communications can be overlapped by computations, but it can be adapted by considering the maximum between computation and communication times in place of the second sum. The above cost function does not take into account the length of the exchanged messages nor the topology.

8.3.1.4 Mapping algorithms

Many solutions can be found in the literature for solving the mapping problem. Exact algorithms give the optimal solutions but in practical cases they can not be used because of the combinatorial explosion of the number of solutions. The goal of heuristic algorithms is to give good solutions in relatively reasonable time. Two sub-classes of heuristic algorithms are introduced here: greedy algorithms which construct partially the solution and iterative algorithms whose principle is to improve an existing solution. Obviously, the cost of the mapping algorithm itself must be related to the use of the solution. The more used a given mapping, the greater the time that ought to be invested computing it.

8.3.1.5 Greedy algorithms

In a greedy algorithm, 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. Two kinds of greedy algorithms can be envisaged: either they are based on empirical methods or they come from the relaxation of classical graph theory algorithms which are optimal for some restricted cases. They are easy to implement and have a polynomial complexity. *List* *algorithms* are the most used greedy algorithms. Tasks are first sorted on a given criterion and then are mapped in this order on the processors. Some example greedy algorithms are listed as follows.

Modulo: The modulo algorithm consists in allocating the i^{th} task 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): LPTF is a heuristic whose criterion is restricted to load balancing. Tasks are first sorted by decreasing computation cost order, then allocated on the less loaded processor.

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

8.3.1.6 Iterative algorithms

All iterative algorithms try to improve an initial solution usually obtained by a greedy algorithm. Most iterative algorithms exchange tasks between processors to improve locally a solution. Most of such algorithms use random perturbations to leave local minima of the cost function and to obtain better solutions. Examples are given as follows.

The Boillat method

The algorithm is based on an iteration scheme consisting of three parts : information exchange, load balancing and communication optimization. At each iteration, processes are moved to meet optimization demands. The only possible move is the move to the direct neighbour. First the most optimal local balance of computation load for each processor is considered. Because the loading is treated superior to the communication minimization, it may happen that the load is balanced and the communication demand is not fully optimal. Therefore a process causing high local communication costs is sometimes selected to be moved to another processor in order to reduce the communication costs instead of process causing local imbalance. This selection is made with a decreasing probability, and thus the importance of communication becomes weaker. The algorithm is given in Fig. 8-5.

```
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
```

Figure 8-5: The Boillat method

The HME method

Heuristic Move-Exchange method is described as follows. The tasks are randomly assigned to the processors. This configuration is recorded as the best configuration and its cost as the best cost. While there are moves which decrease the cost function value, the move which decreases the cost function most is done. Next, if there are exchanges which decrease the cost function value, then the exchange which decreases the cost function most is performed and the algorithm starts with moves again. If there are not such moves and exchanges, the value of current configuration is appended to a queue of cost function values of length L. If the current cost function value is less than that of the best cost achieved, we record the current configuration as the best configuration and the current cost as the best cost. If the elements in the queue of the cost function values are in strictly descending order then R random moves are performed which change the configuration, afterwards the moves and exchanges are performed again. If not so and the queue is full, the algorithm quits. If not so and the queue is not full yet, all but the last element in the queue are deleted, and R random moves are performed, thus changing the configuration and returning to the moves and exchanges. The algorithm is given in Figure 8-6.

```
best_config:=init;   /* tasks are randomly assigned to processors */
while (loop) do begin
    loop1:=TRUE;
    while (loop1) do begin
        loop2:=TRUE;
        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
```

Figure 8-6: The HME method

Simulated Annealing

This is one of the best known optimization methods, often used for mapping as well. The concept of simulated annealing has been derived from physics, where it is used to describe the process of eliminating lattice defects in crystals by heating followed by slow cooling to low temperature. In our context a lattice defect corresponds to a mapping M with the value of cost function CF(M) higher than its optimal value. The current value CF(M) is formally interpreted as the energy of physical system with a structure described by the mapping M. An ensemble of such system in thermal equilibrium is simulated. As a parameter T (a formal analogy of temperature) is decreasing, the system is frozen out 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 a lower value of CF, otherwise, 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.

The annealing schedule required some experimentation. The procedure is initialized by random generation of N mappings. Starting temperature Tmax must be chosen, so that initial percentage of successful mappings with respect to all mappings should be roughly 50%. We proceed downwards in multiplicative steps amounting to a 20% decrease in T. The whole optimization process is stopped, if a current value of T becomes smaller than the value Tmin.

The outline of the simulated annealing algorithm is given in Figure 8-7.

```
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
```

Figure 8-7.: Simulated Annealing

8.3.2 Dynamic load balancing

Multiprocessor systems are very efficient at solving problems that can be partitioned into parallel tasks with uniform computation and communication patterns. However, there exists a large class of nonuniform problems with uneven and unpredictable computation and communication requirements. Dynamic load balancing (DLB) schemes are needed to efficiently solve non-uniform problems on multiprocessor systems.

8.3.2.1 General dynamic load balancing model

The load balancing process can be divided into four phases:

1. *Processor Load Evaluation*: A load value is estimated for each processor in the system. These values are used as input to the load balancer to detect load imbalances and make load migration decisions.
2. *Load Balancing Profitability Determination*: The *imbalance factor* quantifies the degree of load imbalance within a processor domain. It is used as an estimate of 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. Sources are notified of the quantity and destination of tasks for load balancing.
4. *Task Selection Strategy:* Source processors select the most suitable tasks for efficient and effective load balancing and send them to the appropriate destinations.

The first and fourth phases of the model are application dependent and purely distributed. Both of these phases can be executed independently on each individual processor. Our focus is on the Profitability Determination and Task Migration phases, the second and third phases, of the load balancing process. As the program execution evolves, the inaccuracy of the task requirement estimates leads to unbalanced load distributions. The imbalance must be detected and measured (Phase 2) and an appropriate migration strategy devised to correct the imbalance (Phase 3). These two phases may be performed in either a distributed or centralized fashion. Centralized approaches tend to be more accurate since the entire system's state information is accumulated to a single point, and a high degree of knowledge is used in the decision process. However, the accumulation of information requires synchronization which incurs an overhead and a delay. This overhead may become prohibitively large for highly parallel systems and the delay may increase to a point where the information accumulated ages and loses validity. Alternatively, distributed approaches, although less accurate since they operate with less information, incur a smaller synchronization overhead.

During the Profitability Determination Phase (triggered by a processor's load estimate or timer expiration) a decision is made as to whether or not to invoke the load balancer. The load *imbalance factor* $\Phi(t)$ is an estimate of the potential speedup obtainable through load balancing at time t. It is defined as the difference between the maximum processor loads before and after load balancing, L_{max} and L_{bal}, respectively.

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

A decision on whether or not to load balance is made based on the value of $\Phi(t)$ relative to the balancing overhead, $L_{overhead}$ required to perform the load balancing. In general, load balancing is profitable if the savings is greater than the overhead, i.e.,

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

This may be simplified by setting $L_{overhead}$ to a constant value, rather than calculating $L_{overhead}$ as a function of the system state,

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

A further simplification is to assume that once a processor's load, L_p, drops below a preset threshold, $K_{underload,}$ any balancing will improve system performance:

$$L_p < K_{underload}$$

8.3.2.2 Dynamic load balancing strategies

The main issue in any load balancing scheme is to find a trade-off between knowledge -- the accuracy of each balancing decision, and overhead -- the amount of added processing and communication incurred by the balancing process. The load balancing overhead includes the communication costs of acquiring load information and of informing processors of load migration decisions, and the processing costs of evaluating load information to determine task transfers. Typically, the more information accumulated to be used in the decision process, the more accurate the decisions become. However, other factors, such as the aging of information and the rate at which the load is changing must also be considered.

Four different DLB strategies supporting highly parallel systems are introduced as follows.

The Gradient Model (GM)

The gradient model is a demand driven approach. The basic concept is that underloaded processors inform other processors in the system of their state, and overloaded processors respond by sending a portion of their load to the nearest lightly loaded processor in the system. The resulting effect is a form of relaxation where tasks migrating through the system are guided by the proximity gradient and gravitate towards underloaded points. The scheme is based on two threshold parameters: the *.Low-Water-Mark (LWM)* and the *High-Water-Mark (HWM).* A processor's state is considered light if its load is below the LWM, heavy if above the HWM, and moderate otherwise. A node's *proximity* is defined as the shortest distance from itself to the nearest lightly loaded node in the system. All nodes are initialized with a proximity of w_{max},a constant equal to the diameter of the system. The proximity of a node is set to 0 if its state becomes light. All other nodes p with near-neighbors n_i, compute their proximity as

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

A node's proximity may not exceed w_{max},. A system is saturated, and does not require load balancing if all nodes report a proximity of w_{max},. If the proximity of a node changes it must notify its near-neighbors. Hence, the balancing process is initiated by lightly loaded processors reporting a proximity of 0. A gradient map of the proximities of underloaded processors in the system serves to route tasks between overloaded and underloaded processors.

Load balancing profitability determination is controlled by the LWM and HWM thresholds. In order for load balancing to take place, there must be at least one overloaded processor and one underloaded processor in the system. No measure of the degree of imbalance is made, only that one exists.

The proximity map is used to perform the migration phase. If a processor's state is heavy and any of its near-neighbors report a proximity less than w_{max} then it sends a unit of its load to the neighbor of lowest proximity. The fraction of load being sent by an overloaded processor can be determined as either a percentage of the initial load, or as a fixed number of tasks. Tasks are routed through the system in the direction of the nearest underloaded processors. A task continues to migrate until it reaches an underloaded processor or it reaches a node for which no neighboring nodes report a lower proximity. This scheme may perform inefficiently when either too much or too little work is sent to an underloaded processor.

Sender Initiated Diffusion (SID)

The SlD strategy is a, local, near-neighbor *diffusion* approach which employs overlapping balancing domains to achieve global balancing. The scheme is purely distributed and asynchronous. Each processor acts independently, apportioning excess load to deficient neighbors.

Balancing is performed by each processor whenever it receives a load update message from a 'neighbor indicating that the neighbor's' load, $l_i < L_{LOW}$. where L_{LOW} ; is a preset threshold. Each processor is limited to load information from within its own domain, which consists of itself and its immediate neighbors. All processors inform their near-neighbors of their load levels and update this information throughout program execution: The profitability of load balancing is determined by first computing the average load in the domain,

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

where K is the number of neighboring processors and l_k their respective loads. Next, if a processor's load exceeds the average load by a prespecified amount, $L_{threshold}$ it proceeds to implement the third phase of the load balancing process.

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,

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

Finally, the portion of processor p's excess load that is assigned to neighbor k, is defined as

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

Once the quantity of load to be migrated has been determined, the appropriate number of tasks are dispatched (note that $\delta_k > \delta_{MIN}$). Balancing continues throughout program execution whenever a processor's load exceeds the local average by more than a certain amount.

Receiver Initiated Diffusion (RID)

The RID strategy can be thought of as -the converse of the SID strategy in that it is a receiver initiated approach as opposed to a sender initiated approach. The balancing process is initiated by any processor whose load drops below a prespecified threshold (L_{LOW}). However, upon receipt of a load request, a processor will fulfil the request only up to an amount equal to half of its current load (this reduces the effect of the aging of the data upon which the request was based). In the receiver initiated approach the underloaded processors in the system take on the majority of the load balancing overhead, which can be significant when the task granularity is fine.

Hierarchical Balancing Method (HBM)

The HBM strategy organizes the multicomputer system into a hierarchy of balancing domains, thereby decentralizing the balancing process. Specific processors are designated to control the balancing operations at different levels of the hierarchy. A binary-tree hierarchical organization is illustrated in Fig. 8. In this case, processors in charge of the balancing process at a level, l_i, receive load information from both lower level, l_{i-1}, domains. Global balancing is achieved by ascending the tree and balancing the load between adjacent domains at each level in the hierarchy. This procedure is asynchronous, however, where balancing is invoked within a domain whenever an imbalance is detected by the domain's designated controller. For a binary hierarchical configuration, the size of the balancing domains double from one level to the next. The tree structure minimizes the communication overhead and can be scaled to accommodate large systems.

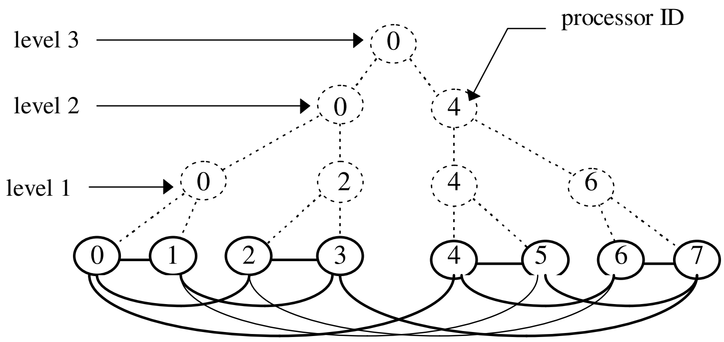


Figure 8-8: Hierarchical organization of an eight-processor system with hypercube interconnections. The processor IDs at intermediate nodes in the tree represent those processors delegated to manage the balancing of corresponding lower-level domains.

The hierarchical balancing scheme functions asynchronously. The balancing process is triggered at different levels in the hierarchy by the receipt of load update messages indicating an imbalance between lower level domains. All load levels are initialized with each processor sending its load information up the tree. Subtree load information is computed at intermediate nodes and propagated to the root. Load imbalances at different levels of the hierarchy are detected at intermediate nodes. If the load imbalance between two lower level domains is greater than a prespecified amount, $L_{threshold}$ then one domain is considered overloaded and the other underloaded. The absolute value of the difference between the left domain, L_L, and the right domain load, L_R, is compared to $L_{threshold}$:

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

Task migration is controlled by intermediate nodes which, upon detecting an imbalance, notify all processors belonging to the overloaded subtree of the amount of the imbalance and at what level it occurs. Processors within the overloaded branch transfer a designated amount of their load to their "matching" neighbor in the adjacent underloaded subtree. For each processor in the left subtree at a given level, there is a corresponding processor in the right subtree,: and visa-versa. Given a hypercube interconnection scheme, these processors are directly linked to one another.

In theory, the hierarchical balancing strategy guarantees that having reached a level h in the balancing hierarchy, all processors belonging to the left, level *h-1,* subtree have equal loads and all processors within the right, level *h-1,* subtree have equal loads. In practice, this may not quite be true due to the aging of information as well as the inaccuracy of load estimates, but, so long as the load is not fluctuating too rapidly (δ(task) >> balancing delay), the assumption is valid. Hence, as the balancing process moves up the tree to higher levels in the hierarchy, in order to correct imbalances between adjacent subtrees, all processors within an overloaded subtree may be directed to transfer an equivalent amount of load to their "matching" neighbor in the underloaded subtree. This simplifies the balancing process.

Given an imbalance, Δ_i, at a level $_i$ in the hierarchy, each processor p in the overloaded subtree is directed to transfer a load amount δ_i to its corresponding neighbor q in the underloaded subtree, where

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

The hierarchical scheme distributes the load balancing responsibilities to all processors in the system. It is effective for balancing local load imbalances as well as excessive global imbalances. Different imbalance thresholds can be specified at different levels of the tree. This can limit balancing domains to independent subtrees if load imbalances between subtrees do not exceed a preset threshold. Distant load transfers may then only be instituted to satisfy severe global imbalances.

8.3.2.3 The load update strategy

An important mechanism in most dynamic load balancing scheme is the load update strategy. Many dynamic load balancing strategies, like the one discussed here, make load balancing decisions based on the load levels of a subset of processors in the system. This subset may include anywhere from a single neighbor to all processors in the system. Although the degree of knowledge may vary from one strategy to another, the quality of information governs the intelligence of the load balancing decisions. The quality of information depends on three primary factors: 1) the accuracy of processor load estimates, 2) the aging of information due to the communication latency of the interconnection network and the destination of load information, and finally, 3) the frequency of the load update messages. The first factor is application dependent and may involve a trade-off between the quality of the estimate and the complexity of the estimation process. For example, a processor's queue length is a rough and simple load measure, a possibly more accurate yet more complex load measure might distinguish between the types of tasks within a processor's queue. The second factor is dependent on machine architecture and the load balancing strategy. The third factor, regarding the frequency of update messages, is the focus of the remainder of this section.

The interval between update messages may be computed as a function of time or as a function of the load level. If computed as a function of time, the time intervals most likely need to be adjusted for different applications.

Alternatively, intervals between updates can be determined by a change in the load level, independent of the application. The updates may be sent at constant intervals, ΔL, of a processor's load, L_p. In this case, processor p must send on the order of $L_p/\Delta L$ update messages. This strategy would contribute an error margin of $\pm \Delta L$ to the accuracy of the load evaluation. Hence, smaller values of ΔL yield more accurate information, but also increase the frequency of update messages. Note also, however, that while Lp is large ($L_p >> \Delta L$) the percentage of error in a neighbor's load information, AL/L_p, is very small, but the error percentage increases when L_p decreases (100% when $Lp \approx \Delta L$).

Using a variable update interval, that is computed as a function of the load level, yields a constant error percentage in the load information and decreases the number of required update messages. Let the variable u be defined as *the load update factor,* such that updates are sent whenever a load, L_p, increases to $(l\ /u)L_p$ or drops to uLp. If, for example, u = 1/2 then, a processor must send update messages whenever its load relative to that sent in its last update message has doubled; or is cut in half. Processor p will send on the order of $\log_u(L_p)$ update messages. The maximum error margin of load information will be 1/u of the processor load L_p. The frequency of update messages will increase as the processor loads decrease and the possibility of a processor becoming idle increases. Hence, the accuracy of load information does not degrade as it becomes more critical to the load balancing process.

The load update factor expresses a trade-off between the quality of load information and the overhead to achieve this. The larger the update factor is, the shorter the update interval becomes, and consequently, the more current or accurate the load information.

8.4 Debugging of parallel programs

Debugging sequential programs is a well-understood task that draws on tools and techniques developed over many years. One early technique was to record snapshots of the entire program state, often consisting of many pages of hexadecimal digits, for perusal by the programmer. Debugging was a programmer-intensive operation, since there were few tools for analyzing the program state. Over time this approach was replaced by interactive debuggers, which allow the programmer to examine arbitrary details of the program state during execution. Debugging became more computation intensive, since the computer was used to reproduce execution sequences with successively greater detail. As a result, the most common methodology used today to debug sequential programs is cyclic: the program is executed until an error manifests itself, the programmer postulates a set of underlying causes for the error, trace statements or additional breakpoints are inserted to gather more information about the causes of the error, and the program is reexecuted. This technique is effective because sequential programs are usually deterministic. That is, for a fixed input, each execution of a program will always follow the same execution path and produce the same results.

Debugging parallel programs is considerable more difficult because parallel programs are often not deterministic. Since parallel programs do not fully specify all possible execution sequences, the execution behavior of a parallel program in response to a fixed input may be indeterminate, with the results depending on a particular resolution of race conditions existing among processes. Therefore, cyclic debugging techniques for error isolation are not guaranteed to work because successive execution of the same parallel program may not produce the same results. We are left with two options for debugging parallel programs:

1. we can either take snapshots of the program state during execution for later examination or
2. we can provide a mechanism that guarantees reproducible behavior of parallel programs.

Since the snapshot techniques are not cyclic, the user must collect enough information during one-shot execution to diagnose any error that might arise. As a result, the amount of information to be gathered tend to be voluminous. By contrast, the second approach allows reliable use of well-known cyclic debugging techniques. In this course, we concentrate only on the second approach.

8.4.1 Replay technique

When debugging a sequential program, one can usually guarantee reproducible program execution by supplying the same input each time the program is executed. Successive executions with the same input produce the same behavior because sequential programs tend to be deterministic. The same is true of the individual processes in a parallel program[4]. If each process is supplied the same input values (corresponding to the contents of messages received or the values of shared memory locations referenced) in the same order during successive executions, it will produce the same behavior each time. In particular, each process will produce the same output values in the same order. Each of those output values may then serve as an input value for some other process. Therefore, debugging can be done in two phases:

1. *execution* phase: data relevant to produce deterministic replay of the parallel program are collected,
2. *replay* phase: replay the program, as many times as the programmer wants to, based on data collected in execution phase

By ensuring that each process sees the same input values at every step of execution, all processes will exhibit the same execution behavior during both the execution phase and replay.

There are basically two different approaches for supporting replay of parallel programs:

1. *implementation*-based approach: support is provided by modifying the language's implementation[5],
2. *language*-based approach: the parallel program is transformed into a different form in the same language, which new form contains all auxiliary code necessary for deterministic replay.

The information necessary to be collected in execution phase depends on the particular parallel language, i.e. depends on the synchronisation and communication constructs provided by the language. Two possible solution for replay mechanism is introduced as follows. The first one supports the concept of monitors which are high-level synchronisation constructs usable on shared memory machines, while the second supports message passing based programs running typically in distributed environments.

8.4.1.1 Language-based replay for monitors

A monitor encapsulates both the shared data and the procedures that manipulate it. We assume that every shared variable is inside a monitor and that processes delayed on a condition variable are

[4] Assuming that processes do not contain non-deterministic statements.

[5] The implementation of a parallel language usually has three components: the compiler, run-time system and operating system.

awakened in first-in, first-out order. 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 by a process, and
- the execution of a Signal(condition) operation in a monitor procedure by a process.

Thus, the execution of such a program can be characterized as a sequence of such synchronization events, called a monitor sequence (or M sequence), denoted by

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

where (C*i*,H*i*,D*i*,N*i*) denotes the *i*th synchronization event in the sequence, for values of *i* greater than 0. C*i* is the calling process, H*i* the type of this synchronization event, D*i* the called monitor procedure, and N*i* the condition variable if this event is a Wait(condition) or Signal(condition). It is easy to show that the result of an execution of a monitor-based program with a particular input can be determined by the program, the input, and the M sequence of this execution.

A simple monitor sequence, called an SM sequence, is denoted by

(C1,C2,...)

where C*i*, for values of *i* greater than 0, denotes the process making the *i*th monitor entry. An M sequence or an SM sequence is feasible for a program with a particular input if it can be exercised by some execution of the program with that input. An M sequence always corresponds to a unique SM sequence, but not vica versa.

Although the format of an SM sequence is much simpler than that of an M sequence, the result of an execution of a program with a particular input can be determined by the program, the input, and the SM sequence of this execution. This is true because the sequence of synchronization events inside a monitor can be determined from the monitor, the sequence in which processes enter the monitor (via monitor calls), and the values of the parameters of these monitor calls. Thus, SM sequences provide sufficient information for deterministic replay of the execution.

A partial order based SM sequence can be defined for an entire program uses monitors M_1 through M_n . A partial-ordering-based SM sequence of the program with a particular input is defined as (Q1,Q2,...,Q*n*) where Q*i* ($0 < i <= n$) is a sequence of synchronization events involving monitor M*i*. This partially ordered SM sequence can be replayed during an execution of the program with the same input by replaying Q*i* ($0 < i <= n$) for each monitor M*i*.

To illustrate the language-based approach of replay technique for monitors, a simple example program with non-deterministic behaviuor is presented first.

8.4.1.1.1 Example of non-deterministic behavior

Consider the monitor-based concurrent program P_C_Buffer in Figure 8-9., which solves the bounded-buffer problem. In P_C_Buffer, the Buffer_Control monitor controls producers and consumers that deposit and withdraw items, respectively, to and from a buffer of size two. The Producer process reads in three characters and calls the Deposit procedure three times to deposit these characters. The Consumer process calls the withdraw procedure three times to withdraw the characters deposited by Producer and prints out these characters.

During an execution of P_C_Buffer, the relative progress of the Producer and Consumer processes is unpredictable, so the sequence in which these two processes enter the Buffer Control monitor is also unpredictable. You can denote such a sequence, called a simple monitor sequence, with the sequence of names of the entering processes. Simple monitor sequences that might be exercised during executions of P_C_Buffer include:

- (Producer, Consumer, Producer, Consumer, Producer, Consumer),
- (Producer, Producer, Consumer, Consumer, Producer, Consumer), and
- (Consumer, Producer, Consumer, Producer, Consumer, Producer).

```
monitor: BUFFER_CONTROL;
  const SIZE = 2;
  var    NOT_FULL: condition;
         NOT_EMPTY condition;
         BUFFER: array [0..SIZE-l] of
character;

procedure DEPOSIT(ITEM: character);
begin
  if COUNT > SIZE then wait(NOT_FULL);
  BUFFER[IN] := ITEM;
  IN := (IN+l) mod SIZE;
  COUNT := COUNT + 1;
  signal(NOT_EMPTY)
end; (* DEPOSIT *)

procedure WlTHDRAW(var ITEM:
character);
begin
  if COUNT = O then wait(NOT_EMPTY);
  ITEM := BUFFER[OUT];
  OUT := (OUT+1) mod SIZE;
  COUNT := COUNT -1;
  signal(NOT_FULL)
end; (* W1THDRAW *)

begin
  COUNT := 0, IN := 0; OUT := 0
end; (* BUFFER_CONTROL *)

process: PRODUCER;
  var ITEM: character;
begin;
  READ(ITEM);
  DEPOSIT(ITEM);
  READ(ITEM);
  DEPOSIT(ITEM);
  READ(ITEM);
  DEPOSIT(ITEM)
end; (* PRODUCER *)

process: CONSUMER;
  var ITEM: character;
begin
  WlTHDRAW(ITEM);
  WRITE(ITEM);
  WlTHDRAW(ITEM);
  WRITE(ITEM);
  WlTHDRAW(ITEM);
  WRITE(ITEM);
end; (* CONSUMER *)
```

Figure 8-9: The monitor-based program, P_C_Buffer, for solving the bounded-buffer problem.

During an execution of P_C._Buffer that exercises either of the first two simple monitor sequences, the order in which the Producer and Consumer processes enter the Buffer_Control monitor is exactly the order in which they leave the monitor. But during an execution that exercises the third simple monitor sequence, the Consumer process is blocked by the statement Wait(Not_Empty) each time it enters the Buffer_Control monitor. Then Producer enters the monitor, deposits an item, unblocks Consumer using Signal (Not_Empty), and leaves the monitor. Consumer then resumes execution, with draws the item, and leaves the monitor.

P_C_Buffer has an error that allows three consecutive completions of Deposit operations on a buffer of size two. But this error cannot be detected by an execution of P_C_Buffer (with any valid input) that exercises the simple monitor sequences above, since such an execution always produces a correct result In fact, P_C_Buffer might be executed many times without detecting this error.

This error can be detected by an execution of P_C_Buffer with the input ('A','B','C') only if this execution exercises another simple monitor sequence: (Producer, Producer, Producer, Consumer, Consumer, Consumer). The output of this execution is ('C','B','C'), not the expected output ('A','B','C'). The incorrect output ('C','B','C') indicates the existence of an error, but it is difficult to locate the error's cause from just reading the output.

A common debugging approach is to insert additional statements in P_C_Buffer and then execute the program again with input (‘A’,’B’,’C’) to collect debugging information for the erroneous execution. However, there is no guarantee that this fourth simple monitor sequence will be exercised by executing P_C_Buffer with the input (‘A’,’B’,’C’). After you have located the error and modified the program, you must execute P_C_Buffer again with the input (‘A’,’B’,’C’) to verify that the error has indeed been corrected. and that the correction has introduced no new errors. You can verify the correction only if an execution of P_C_Buffer with input (‘A’,’B’,’C’) exercises this fourth simple monitor sequence, since only it caused error to reveal itself. These problems can be solved by forcing deterministic execution during debugging and regression testing, according to given synchronisation sequences.

8.4.1.1.2 Transforming monitor based programs

A language-based synchronization-sequence replay tool for a concurrent language transforms a concurrent program written in that language into another program written in the same language to control the execution of synchronization events in the original program[6]. If monitors are used as synchronization constructs then the monitor-based program is program is transformed into a new monitor-based version that controls the sequence of monitor entries.

Each process in the transformed program declares a constant denoted by ID, that is assigned a unique integer value used as the process's identifier. If a global procedure or function in the original program contains a synchronization operation, a parameter is added to the procedure or function in the transformed program to pass the identifier of the calling process.

```
monitor: CONTROL;
  const  MAX_LENGTH = ...;        (* maximum length of SM sequence *)
         MAX_ID = ...;            (* maximum number of processes *)
  var    SM_SEQ: array[1..MAX_LENGTH] of integer;
         PROCESS_QUEUE: array[l..MAX_ID] of condition;
         SM_LENGTH: integer;      (* actual length of SM sequence *)
         INDEX: integer;

procedure REQUEST_M_PERMIT(ID: integer);
begin
  if ID <> SM SEQ[INDEX] then wait(PROCESS_QUEUE[ID])
 end; (* REQUEST_M_PERMIT *)

procedure RELEASE_M_PERMIT;
begin
  INDEX := INDEX+ 1;
  if INDEX <= SM_LENGTH then signal(PROCESS_QUEUE[SM_SEQ[INDEXl])
end; (* RELEASE_M_PERMIT *)

begin
  INDEX := 1;
  READ(SM_LENGTH); . . . (* READ IN THE ELEMENTS OF ARRAY SM_SEQ[1 ..SM_LENGTH] *)
end; (* CONTROL *)
```

Figure 8-10: The Control monitor.

In the transformed program, processes make requests for a permit called M_Permit. The sequence in which M_Permit is given to processes is based on the SM sequence. The program requests (releases) M_Permit in the form of a procedure call to the Request_M_Permit (or Release_M_Permit) procedure. Each monitor-call statement to the monitor in question, denoted by M_PROC(...), is modified as follows:

```
Request_M_Permit(ID);
M_PROC(...);
```

Each external procedure in the monitor is also modified. The original external procedures look like:

```
procedure M_PROC(...);
begin
...
end; (* M_PROC *)
```

The modification needed is simple: The following line is added after the Begin statement:

```
Release_M_Permit;
```

[6] The control is based on a synchronisation sequence that has been recorded during the initial execution phase of the original program.

where Request_M_Permit and Release_M_Permit are procedures in a monitor, called Control (shown in Figure 8-10.) inserted into the transformed program. Before each monitor-call statement, a process first calls the Request_M_Permit procedure and passes its ID. For the *j*th monitor entry (for all values of *j* greater than 0) in the SM sequence, only the process P*k,* where *k* is SM_Seq[*j*], may exit Request_M_Permit If a monitor call is out of sequence, the process is delayed inside Request_M_Permit. The process identifier determines the condition variable, Process_Queue[ID], on which the process is delayed. (We assume that the implementation allows arrays of condition variables.)

A process that is permitted to exit Request_M_Permit and enter the monitor must then call the Release_M_Permit procedure. Release_M_Permit first increments the variable Index to allow the next monitor entry in the SM sequence. Because the process that is to make the next monitor entry may have been delayed in Request_M_Permit, Release_M_Permit performs a signal operation.

```
monitor: BUFFER_CONTROL;
  const  SIZE = 2;
  var    NOT_FULL: condition;
         NOT_EMPTY: condition;
         BUFFER: array [0..SIZE-I] of character;
         COUNT, IN, OUT: integer;

procedure DEPOSIT([ITEM: character);
begin
  RELEASE_M_PERMIT;              (* REPLAY *)
  if COUNT > SIZE then wait(NOT_FULL);
  BUFFER[IN]:= ITEM;
  IN := (IN+I) mod SIZE;
  COUNT := COUNT + 1;
  signal(NOT_EMPTY)
end; (* DEPOSIT *)

procedure W1THDRAW(var ITEM: character);
begin;
  RELEASE_M_PERMIT;              (* REPLAY *)
  if COUNT = O then wait(NOT_EMPTY);
  ITEM := BUFFER[OUT];
  OUT := (OUT+1) mod SIZE;
  COUNT := COUNT - 1;
  signal(NOT_FULL)
end; (* W1THDRAW *)

begin
  COUNT := 0, IN := 0; OUT := 0
end; (* BUFFER_CONTROL *)

monitor: CONTROL;                (* REPLAY *)
  ...(* see Figure 8-9 *)        (* REPLAY *)
end; (* CONTROL *)               (* REPLAY *)
```

```
process: PRODUCER;
  const ID = 1;                  (* REPLAY *)
  var ITEM: character;
begin
  READ(ITEM);
  REQUEST_M_PERMIT(ID);          (* REPLAY *)
  DEPOSIT(ITEM);
  READ(ITEM);
  REQUEST_M_PERMIT(ID);          (* REPLAY *)
  DEPOSIT(ITEM);
  READ(ITEM);
  REQUEST_M_PERMIT(ID);          (* REPLAY *)
  DEPOSIT(ITEM);
end; (* PRODUCER *)

process: CONSUMER;
  const ID = 2;                  (* REPLAY *)
  var ITEM: character;
begin
  REQUEST_M_PERMIT(ID);          (* REPLAY *)
  W1THDRAW(ITEM);
  WRITE(ITEM):
  REQUEST_M_PERMIT(ID);          (* REPLAY *)
  W1THDRAW(ITEM);
  WRITE(ITEM):
  REQUEST_M_PERMIT(ID);          (* REPLAY *)
  W1THDRAW(ITEM);
  WRITE(ITEM):
end; (* CONSUMER *)
```

Figure 8-11: The P_C_Buffer program from Figure 8-9. transformed for deterministic replay

Figure 8-11. shows the program P_C_Buffer from Figure 8-9. transformed for deterministic execution debugging. Statements that have been inserted or transformed are marked with the comment (* Replay *).

8.4.1.2 Implementation-based replay for the message passing model

In the case of parallel programs using only synchronous message passing communication primitives (the well-known *send-receive* primitives[7]), replay can be implemented by forcing on each

[7]See Chapter 4. and 5. for detailed information about the message passing programming model.

process p_i the same order of message receptions. The trace information for pi, recorded during execution phase, is thus simply a sequence of message identifiers. Driving of the replay is more complicated if non-blocking primitives have to be considered. Non-blocking primitives are used to overlap communication and computation and may be defined, for instance, as follows (primitives (1) and (2) are related to receptions, whereas primitives (3) and (4) are related to message sending):

1. **give_buffer(buf):** gives a buffer (in the process space) to the operating system**,** into which a message can be stored. If a message is already available, it is immediately transferred from the system buffer into the process buffer. If not, the next incoming message for the process will be directly copied into the process buffer;
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).

Such primitives are typically used as shown in Figure 8-12.

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
. . .	. . .

Figure 8-12: Programming style with non-blocking primitives.

The replay technique presented here distinguishes between *sensitive events* and *implicit events*:

- **sensitive events** correspond to communication related events, as for example send(), receive(), give message(), message_arrived().
- **implicit events** correspond to events that modify the process state, but are not executed by the process itself (the implicit events are the result of sensitive events executed by the other processes). The arrival of a message and the buffer release (in the case of the sending of a message) are implicit events.

The idea of the technique is to number on each process sensitive and implicit events (see Fig. 13.), and to force the same ordering of these events during replay. Information for driving the replay is recorded only when implicit events occur, and is composed of two parts:

1. the sequence number of the event;
2. an identification of the event. Assuming FIFO communication channels, an implicit event corresponding to the arrival of a message can be identified by the pair (*sender_prss*, *ARRIVAL*). An implicit event corresponding to the sending of a message can be identified by the pair (*destination_prss*, *DEPARTURE*).

This information is sufficient, since during replay sensitive events will always be executed in the same sequence by the process. Thus during replay we only have to ensure that:

- when executing a sensitive event, all preceding implicit events have occurred;
- an implicit event can take place only when all preceding (sensitive and implicit) events have occurred.

By knowing from the trace the sequence number of the implicit events, sensitive events can be delayed during the replay: a sensitive event will only be executed once all preceding implicit events have occurred. Consider a pair *(e_id, n)* in the process trace, where *e_id* is the id of an implicit event *e*

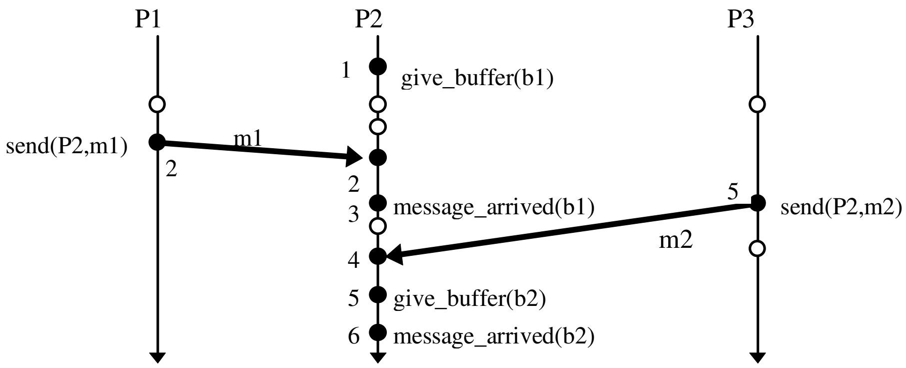


Figure 8-13: One possible execution with sensitive and implicit events numbered for each process.

and n the corresponding event number. Event e will be allowed to take place during replay only when the replay event counter is equal to n, which guarantees that all events preceding e have occurred.

As an example, consider the execution illustrated in Fig. 8-13. and the corresponding control information recorded for the three processes (Fig. 8-14.). Arrival of message m_1 will have to occur after execution of primitive *give_buffer(bufl)* and before execution of primitive *message_arrived(bufl),* whereas m_2 will have to arrive between events *3* and *5*.

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

Figure 8-14: Recorded control information for the execution of Fig. 13.

We now present how trace recording and driving of the replay are integrated into the communication primitives. During the initial execution, the only special treatment when executing a sensitive event is to increment a counter called *Prss_Counter.* When an implicit event occurs (message arrival or buffer release, handled by the operating system), control information is recorded in the process trace, as described in Fig. 8-15.

```
procedure message_arrival(sender_prss: process_type);
begin
  inc(Prss_Counter);
  saveInLogFile(Prss_Counter, sender_prss, ARRIVAL);
end message_arrival;
```

Figure 8-15: Trace recording during the initial execution.

The replay is controlled when executing sensitive events. Before executing a sensitive event *e,* existence of implicit events preceding *e* is checked. If such events exist, occurrences of these implicit events is forced (in the correct sequence). This is done by procedure *before_sensitive_event* of Fig. 8-16., which is supposed to be called before each sensitive event occurrence during the replay (see Figure 8-17.). An implicit event is allowed to take place (i.e. *Prss_Counter* incremented and trace consumed) only if it corresponds to the next awaited event of the corresponding process. Otherwise, the arrived message or the buffer release information are put into a temporary wait queue, to be handled later on.

The presented replay technique for message-passing primitives has been implemented on an iPSC/2 parallel computer. The major part of collecting information of the inital execution phase and the control of the replay has been implemented by modifying the routines of the communication library. In order to track (during initial execution) and control (during replay) the arrival of messages and buffer releases (i.e. implicit events), modifications at the operating system level were also necessary.

```
procedure before_sensitive_event;
  (* only called during execution replay *)
begin
  while Prss_Counter = next_recorded_event'FIELD_1 do
        (* The fields of a recoreded item are named FIELD_1, FIELD_2, and FIELD_3 *)
        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  (* next_recorded_event'FIELD_3 = DEPARTURE *)
            send the first available message destinated to "next_recorded_event'FIELD_2";
        endif;
        next_recorded_event := getFromLogFile();
        inc (Prss_Counter);
  end while;
end before_sensitive_event;
```

Figure 8-16: Driving of the replay when executing a sensitive event.

```
function message_arrived (buf: buffer_type) : boolean;
begin
  inc(Prss_Counter);
  if MODE = REPLAY then
        before_sensitive_event;
  endif:
  return $MESSAGE_ARRIVED(buf);      (* effective system call *)
end message_arrived;
```

Figure 8-17: Replay integrated into the communication primitives.

8.5 Performance analysis

We distinguish three basic steps in the performance analysis process: data collection, data transformation, and data visualization. *Data collection* is the process by which data about program performance are obtained from an executing program. Data are normally collected in a file, either during or after execution, although in some situations they may be presented to the user in real time. Three basic data collection techniques can be distinguished:

- *Profiles* record the amount of time spent in different parts of the program. This information, though minimal, is often invaluable for highlighting performance problems. Profiles typically are gathered automatically.
- *Counters* record either frequencies of events or cumulative times. The insertion of counters may require some programmer intervention.
- *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.

The raw data produced by profiles, counters, or traces are rarely in the form required to answer performance questions. Hence, *data transformations* are applied, often with the goal of reducing total data volume. Transformations can be used to determine mean values or other higher-order statistics or to extract profile and counter data from traces. For example, a profile recording the time spent in each subroutine on each processor might be transformed to determine the mean time spent in each subroutine on each processor, and the standard deviation from this mean. Similarly, a trace can be processed to produce a histogram giving the distribution of message sizes. Each of the various performance tools described in subsequent sections incorporates some set of built-in transformations; more specialized transformation can also be coded by the programmer.

Parallel performance data are inherently multidimensional, consisting of execution times, communication costs, and so on, for multiple program components, on different processors, and for different problem sizes. Although data reduction techniques can be used in some situations to compress performance data to scalar values, it is often necessary to be able to explore the raw multidimensional data. As is well known in computational science and engineering, this process can benefit enormously from the use of *data visualization* techniques. Both conventional and more specialized display techniques can be applied to performance data.

A wide variety of data collection, transformation, and visualization tools are available. When selecting a tool for a particular task, the following issues should be considered:

1. *Accuracy.* In general, performance data obtained using sampling techniques are less accurate than data obtained by using counters or timers. In the case of timers, the accuracy of the clock must be taken into account.
2. *Simplicity.* The best tools in many circumstances are those that collect data automatically, with little or no programmer intervention, and that provide convenient analysis capabilities.
3. *Flexibility. A* flexible tool can be extended easily to collect additional performance data or to provide different views of the same data. Flexibility and simplicity are often opposing requirements.
4. *Intrusiveness.* Unless a computer provides hardware support, performance data collection inevitably introduces some overhead. We need to be aware of this overhead and account for it when analyzing data.
5. *Abstraction.* A good performance tool allows data to be examined at a level of abstraction appropriate for the programming model of the parallel program. For example, when analyzing an execution trace from a message-passing program, we probably wish to see individual messages, particularly if they can be related to send and receive statements in the source program. However, this presentation is probably *not* appropriate when studying a data-parallel program, even if compilation generates a message-passing program. Instead, we would like to see communication costs related to data-parallel program statements.

8.5.1 Data collection

Next, we examine in more detail the techniques used to collect performance data. We consider in turn profiling, counters, and event traces, focusing in each case on the principles involved. Finally, two example tools are described: Paragraph and Prove.

8.5.1.1 Profiles

The concept of a profile should be familiar from sequential computing. Typically, a profile shows the amount of time spent in different program components. This information is often obtained by sampling techniques, which are simple but not necessarily highly accurate. The value of the program counter is determined at fixed intervals and used to construct a histogram of execution frequencies. These frequencies are then combined with compiler symbol table information to estimate the amount of time spent in different parts of a program. This profile data may be collected on a per-processor basis and may be able to identify idle time and communication time as well as execution time.

Profiles have two important advantages. They can be obtained automatically, at relatively low cost, and they can provide a high-level view of program behavior that allows the programmer to identify problematic program components without generating huge amounts of data. (In general, the amount of data associated with a profile is both small and independent of execution time.) Therefore, a profile should be the first technique considered when seeking to understand the performance of a parallel program.

A profile can be used in numerous ways. For example, a single profile on a moderate number of processors can help identify the program components that are taking the most time and that hence may require further investigation. Similarly, profiles performed for a range of processor counts and problem sizes can identify components that do not scale.

Profiles also have limitations. In particular, they do not incorporate temporal aspects of program execution. For example, consider a program in which every processor sends to each other processor in turn. If all processors send to processor 0, then to processor 1, and so on, overall performance may be poor. This behavior would not be revealed in a profile, as every processor would be shown to communicate the same amount of data.

Profilers are available on most parallel computers but vary widely in their functionality and sophistication. The most basic do little more than collect sequential profile data on each processor; the most sophisticated provide various mechanisms for reducing this data, displaying it, and relating it to source code. Because efficient profiling requires the assistance of a compiler and runtime system, most profiling tools are vendor supplied and machine specific.

8.5.1.2 Counters

As its name suggests, a counter is a storage location that can be incremented each time a specified event occurs. Counters can be used to record the number of procedure calls, total number of messages, total message volume, or the number of messages sent between each pair of processors. Counts may be generated by compiler-generated code, by code incorporated in communication libraries, or by user-inserted calls to counter routines.

Counters complement profilers by providing information that is not easily obtainable using sampling techniques. For example, they can provide the total number and volume of messages, information that can be combined with communication time data from a profile to determine the efficiency of communication operations.

A useful variant of a counter is an *interval timer,* a timer used to determine the length of time spent executing a particular piece of code. This information can be accumulated in a counter to provide an accurate determination of the total time spent executing that program component. A disadvantage of interval timers is that the logic required to obtain a timer value can be expensive.

8.5.1.3 Traces

An execution trace is the most detailed and low-level approach to performance data collection. Trace-based systems typically generate log files containing time-stamped event records representing significant occurrences in a program's execution, such as calling a procedure or sending a message. Trace records may include information such as the type of event and the procedure name or destination task, and can be generated either automatically or under programmer control. Table 8-1. shows an example of trace records.

Trace-based approaches support a particularly broad study of program behavior. They can be used to examine causal relationships between communications, to localize sources of idle time, and to identify temporary hot spots. For example, an execution trace could be used to determine that all processors are sending to the same processor at the same time. An execution trace can also be post processed to obtain profile, count, and interval timer information; to compute higher-order statistics such as the means and variances of these values; and to obtain other data such as mean message queue length in a message-passing system.

Trace Record	*Description*
11 0 15553 2 0 1078	Timer data
6 0 2237 2 1 1 8	Receive
11 0 2237 2 0 1078	Timer Data
11 0 2500 2 0 1341	Timer Data
4 0 3186 2 3 2 12	Send
11 0 3186 2 0 1341	Timer Data
11 0 3502 2 0 1656	Send
11 0 3672 2 0 1656	Timer Data

Table 8-1. Trace records generated by the Portable Instrumented Communication Library. The various records contain information regarding the type of event, the processor number involved, a time stamp, and other information. Clearly, these records are not meant to be interpreted by humans.

The disadvantages of trace-based approaches stem primarily from the huge volume of data that can be generated. Particularly when a program is executing on large numbers of processors, it is easy to generate tens, hundreds, or even thousands of megabytes of data. (For example, if a 20-byte record is logged for every message on a 128-processor system, then assuming messages are sent at the rate of one every 10 milliseconds, trace data will be generated at 256 kilobytes per second, or about 1 gigabyte per hour.) This large data volume has three unwelcome consequences. First, the logging of this data tends to perturb performance, thereby leading to what is called the *probe effect* in which the measuring of performance data changes their characteristics. Second, the sheer volume of data makes post processing difficult. Frequently, sophisticated analysis is required to extract relevant information. Third, the programmer, in order to combat the problems caused by volume, may have to spend considerable effort tuning the data collection process so that only relevant events are recorded while the phenomenon of interest is retained. Tracing then becomes a labor-intensive process. For these reasons, tracing should be used with care and only if other data collection techniques are not available or do not provide sufficient information.

Many parallel programming tools provide some automatic tracing capabilities, for example by generating a trace record for every message generated or received. These capabilities are invoked by linking with a specialized version of a communication library and/or by a runtime flag. Mechanisms for generating user-defined events may also be provided.

In principle, event traces can be interpreted in various ways by using different tools. A stumbling block here is a lack of standards for event log records. One proposed standard is the Pablo Self Describing Data Format (SDDF) designed at the University of Illinois. As illustrated in Figure 8-18., this associates an integer event type with a record description that specifies a type and name for each field.

```
#105:
// "description" "Procedure Exit Trace Record"
"Procedure Exit Trace" {
  // "Time" "Timestamp"
  int "Timestamp"
  // "ID" "Event ID"
  int "Event Identifier"
  // "Node" "Processor Number"
  int "Processor Number"
  // "Procedure" "Procedure Index"
  int "Procedure Index"
  // "Inclusive Duration" "Inclusive Procedure Duration"
  int "Inclusive Duration"
  // "Exclusive Duration" "Exclusive Procedure Duration"
  int "Exclusive Duration"
};;
```

*Figure 8-18: An example of the Pablo Self Describing Data Format. The data record "*Procedure Exit Trace*" has an event type of 105 and six data fields, all integers.*

8.5.1.4 Summary of data collection tools

A broad spectrum of data collection mechanisms can be used to obtain information about parallel program performance. In general, those requiring the least programmer intervention are also the least intrusive and provide the highest-level, least-detailed view of program behavior; those providing greater detail are progressively more intrusive and demand more programmer effort. Hence, for maximum programmer efficiency, the process of collecting and interpreting performance data should proceed in a staged manner, as follows.

1. Use profile and count information to obtain any parameter values needed to complete performance models.

2. Measure execution times for a range of processor counts and problem sizes, and compare these results with values predicted by performance models.

3. If observed and modeled performance differ significantly, use profile and count information to verify the basic assumptions underlying your model. For example, check that message counts and volumes match your predictions, and check for load imbalances and replicated computation.

4. If there are still unexplained aspects of program performance, incorporate simple tracing (or enable automatic tracing capabilities), and study performance on a few processors. Increase the number of processors as your understanding improves.

Of course, the actual path followed to obtain performance data will also depend on the functionality provided in a particular parallel programming system.

8.5.2 Data transformation and visualisation

Data transformation and visualization tools transform raw data collected during program execution to yield data and images more easily understood by the programmer. In this section, we provide a general discussion of transformation and display techniques, indicating which are useful for which purposes. In the next section, we present examples of specific tools and describe specific transformations and display formats.

8.5.2.1 Profile and counts

A typical profile provides information about the time spent in each procedure on each processor, the number of times each procedure is called, the number of messages generated on each processor, the volume of these messages, and so forth. Data reduction techniques can be used to reduce this multidimensional data to a smaller number of dimensions, and various forms of display can be used to visualize both the original and the reduced data.

Zero-dimensional (scalar) data are of course trivial to display, consisting of a single number: total computation time, total number of messages, mean message size, and so forth. However, numbers of this sort provide relatively little insight into program behavior. For example, we may notice that total communication volume is greater than expected. This observation may stimulate us to ask several questions. Is the additional communication concentrated in a subset of the processors? Is it concentrated in a single procedure? In which phase of the computation does it occur? More data are required if we are to answer these questions.

The histogram is often a convenient display format for one-dimensional data. If the number of processors is large, the size of a histogram can be reduced by binding, in which case histogram bars represent the number of processors (or procedures or whatever) that have computation time in a specified range. Two-dimensional data can be displayed using color and a two-dimensional matrix.

8.5.2.2 Traces

Trace data can often be reduced to one, two, or three dimensions and then displayed using the histogram techniques described in previous Section.. For example, we can plot communication volume or efficiency as a function of time, or plot histograms of trace values. Other forms of display can provide more detailed views of temporal dependencies between different processors and program components by sacrificing scalability and abstraction for detail. We describe just two displays of this sort; others are illustrated in later sections.

The *Gantt chart is* a horizontal bar chart in which each bar represents the status of each processor as a function of time. Bars can simply represent status (computing, communicating, or idling) and/or indicate the program component or procedure that is executing on each processor at a particular time. A Gantt chart can highlight unexpected dependencies between program components. Note that dependencies inferred from these sorts of displays are valid only if the computer and performance tool that we are using ensure that times recorded for events occurring on different processors are consistent.

If we augment a Gantt chart by drawing lines to connect corresponding send and receive events on different processors, we obtain a *space-time diagram.* A space-time diagram can make it easier to infer temporal dependencies, because it is often possible to identify the specific communication event for which a processor is waiting and hence idle.

8.5.3 Tools

Next, we describe briefly two example performance tools, explaining how each is used to collect and display performance data. While various tools exhibit important differences, there are also many similarities, and frequently our choice of tool will be driven more by availability than by the features provided.

8.5.3.1 Paragraph

Paragraph is a portable trace analysis and visualization package developed at Oak Ridge National Laboratory for message-passing programs. It was originally developed to analyze traces generated by a message-passing library called the Portable Instrumented Communication Library (PICL) but can in principle be used to examine any trace that complies to its format. Like many message-passing systems, PICL can be instructed to generate execution traces automatically, without programmer intervention.

Paragraph is an interactive tool. Having specified a trace file, the user instructs Paragraph to construct various displays concerning processor utilization, communication, and the like. The trace files consumed by Paragraph include, by default, time-stamped events for every communication operation performed by a parallel program. Paragraph performs on-the-fly data reduction to generate the required images. Users also can record events that log the start and end of user-defined "tasks".

Paragraph's *processor utilization* displays allow the user to distinguish time spent computing, communicating, and idling. Communication time represents time spent in system communication routines, while idle time represents time spent waiting for messages. These displays can be used to identify load imbalances and code components that suffer from excessive communication and idle time costs. In the space-time diagram, the color of the lines representing communications indicates the size of the message being transferred.

Communication displays can be used both to obtain more detailed information on communication volumes and communication patterns and to study causal relationships, for example between communication patterns and idle time. For instance, the communication matrix display indicates, by different colors, the communication volume between all pairs of processes.

A disadvantage of Paragraph is that the relationship between performance data and program source is not always clear. This problem can be overcome in part by explicitly logging events that record the start and end of "tasks" corresponding to different phases of a program's execution. Paragraph provides task Gantt and task histogram displays to examine this information. Paragraph displays are particularly intuitive, although the inability to scroll within display windows can be frustrating.

8.5.3.2 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. Prove can analyse event traces generated by the Tape/PVM monitoring system[8]. The instrumentation of the parallel program is done automatically by the programming environment but it can be customized by the programmer as well.

Comparing to Paragraph, Prove has relatively small number of displays which include space time diagram, Gantt chart and some communication displays to show the volume and distribution of the communication among processes or hosts (i.e. processors). However, zoom and scroll mechanism are provided that allows the user to examine any part of the graphs even if they are rather crowded because of the large amount of various events. The user allowed also to concentrate on different parts of the parallel program by selecting interactively particular processes or communication paths among processes to be presented in the different displays.

A strength of Prove is its tight integration with the graphical program editor of GRADE that allows the user to relate communication events with source code. This integration is based on the trace format of Tape/PVM which is close to that of PICL but every Tape/PVM's event also contains source code information. For example, the user can click on a line representing a communication in the space-time diagram to identify the corresponding communication operation in the source code. Moreover, the reverse direction is supported as well, e.g.. the user can click on a communication statement in the GRAPNEL code to highlight the next (or previous) event generated by that particular statement in the space time diagram.

8.6 Monitoring

Collection of execution information of a program, referred as *monitoring*, is fundamental to both debugging and performance analysis. Unlike the monitoring of a sequential program, where the program behaviour is generally not affected by the amount of time between any two successive instructions, monitoring a parallel program will unavoidably have effects on the execution of that program. This is referred to as the *probe-effect*: because of multiple threads of control and the existence of contention or non-determinism, any attempt to observe the behaviour of the system may modify the condition of such contention or nondeterminism and change that behaviour. Obviously, the effect on the program's execution should be kept as small as possible, so that the information obtained by the monitor does not confuse the programmer's analysis. Otherwise, the very act of monitoring could mask the execution behaviour that should be observed. Thus, we define:

---

[8] The Tape/PVM tool has been developed at LMC-IMAG, Grenoble, France, to generate event traces of PVM applications.

transparency is being achieved by a monitor when the events that constitute a monitored program's execution are identical both in the presence and absence of the monitor.

The degree to which the probe-effect is intrusive will depend on the monitoring approach adopted. There are some monitoring methods, relying on extensive hardware support, which do not affect, in a noticeable way, the behaviour of the programs being observed. However, such hardware support can be an expensive addition to the machine and is not always feasible. The alternative is to use a software approach, which obtains information using software only hooks. To reduce the intrusiveness of this method, a fast call is normally necessary, i.e. as few operations as possible must be executed for each action which extracts information. Consequently, for both hardware and software approaches, the analysis of a program is usually two-folded: a monitoring phase followed by a post-mortem analysis. During monitoring, information about the program's execution is recorded only. In the post-mortem phase, debugging or performance analysis can be conducted either by analysing the information collected or by replaying the program's execution according to this information .

In the next section, we present the *logical clock* monitoring approach, which aims to minimize the amount of intrusion, thus achieving a high transparency of monitoring parallel programs.

8.6.1 The logical clock approach

A universal principle is that the future cannot influence the past. This implies that if an event has some effect on another event, then the former can never occur after the latter, i.e. the cause must always precede the effect. This can be very easily captured in a sequential program: because of the single time reference, events can be ordered according to the time of their occurrence. However, it is not an easy task to capture the cause and effect of a parallel program executed on a parallel computer which has no global reference clock. Since the local clock of each processor may start at a different time and advance at a different pace, it is possible that the local time at which a cause event occurs is behind the local time at which the effect event happens!

In the discussion throughout this section, a distributed memory MIMD architecture is assumed, in which each processor has its own clock. It is also assumed that a parallel program running on this kind of architecture is a collection of processes, each of which consists of a sequence of events and which can only communicate with other processes by message-passing mechanisms. An event is defined as the execution of a single instruction or a set of instructions. The communication can be synchronous or asynchronous. A communication is said to be completed after the message has been transferred from one process to another. In such an environment, transparent monitoring can be obtained by preserving the communication ordering on each process.

Generally, the execution of a parallel program can be pictorially represented by the *space-time* diagram. The execution of a process is represented by a vertical line where the occurrence of events is shown in time order from the bottom to the top (see Figure 8-19.). A directed line between the vertical lines represents a communication (from the sending event to the receiving event).

If the execution of a process in a parallel program is delayed by the monitoring activities (for instance, a communication that is delayed because it is probed by the monitor), the communication sequence may be changed, thus the cause and effect relation in the original program may be lost. For example, in Figure 8-19., originally process *Q* receives a message from process *P* first, then from process *R*, illustrated as arrows with dashed lines in the figure. Thus, event *p1* could be the cause of event *q2*. Because of the delay introduced by the monitor, process *P* might become ready to communicate at a later time[9] T'_{p2} than that without monitoring T_{p2}, so that $T_{p2} < T_{r2} < T'_{p2}$. The monitoring delay on *P* is $D = T'_{p2} - T_{p2}$. Therefore, because of $T_{r2} < T'_{p2}$, process *Q* will receive the message from process R first, instead of process *P*. This is illustrated as arrows with solid lines in Figure 8-19. The original cause and effect is lost, event *r1* rather than event *p1* may become the cause of event *q2*.

In order to achieve transparency, it is necessary to maintain the same communication sequence as that which exists without monitoring. It is clear that in order to compensate for the delay on process *P*, the vertical lines of processes *Q* and *R* must also be stretched so that *Q* can once again receive the message from process *P* prior to that from *R*. In other words, the execution of processes *Q* and *R* must also be artificially delayed for the amount of time *D*. This is shown in Figure 8-20.

[9]For convenience of description, process *Q*'s real time clock RT is used as the reference for the times of event occurrences.

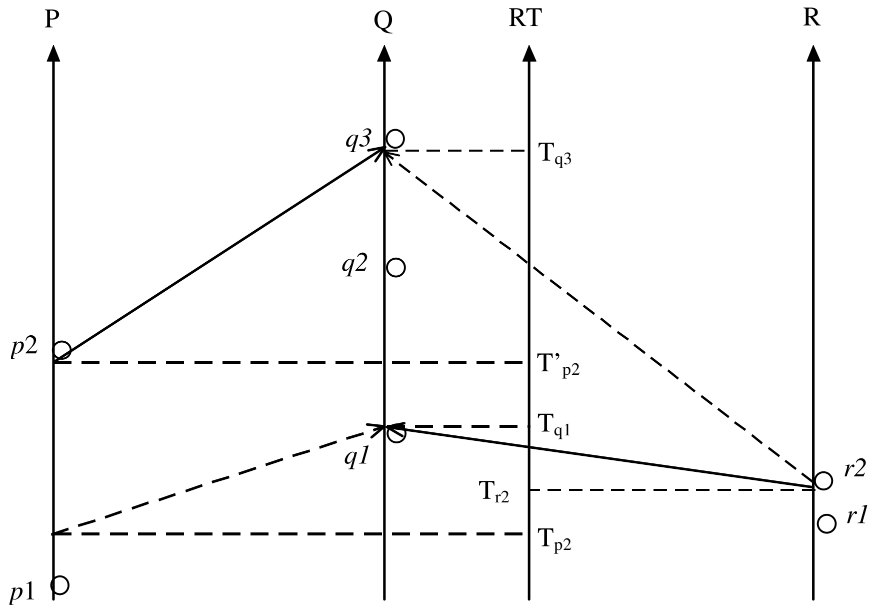
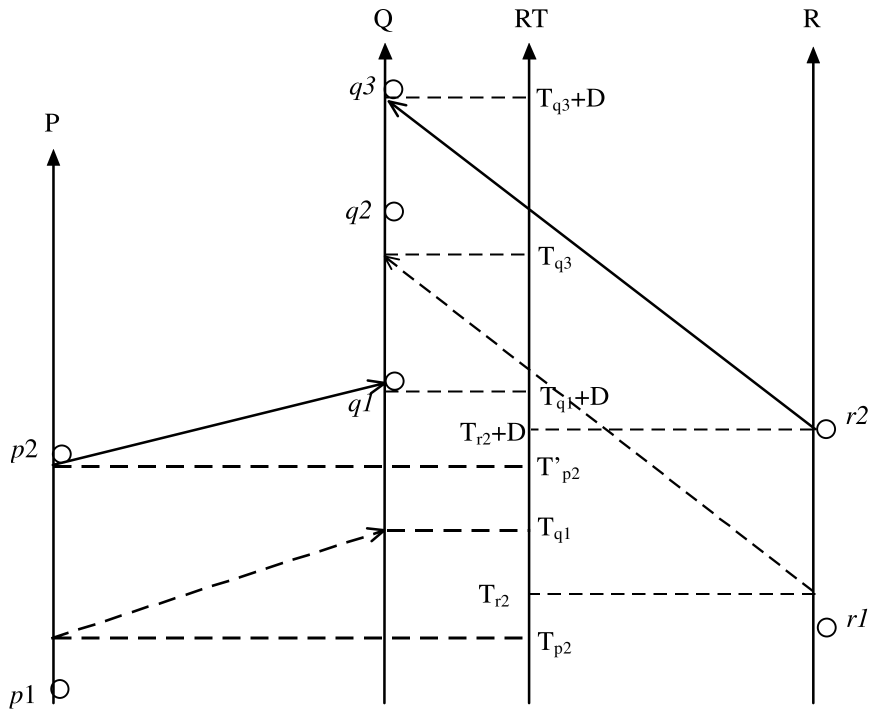


Figure 8-19: Effect of monitoring delay.



$$D = T'_{p2} - T_{p2}$$

Figure 8-20: Suspension of the execution.

A possible solution of hiding the effect of monitoring delay is to introduce the concept of *logical clock* which stops when the monitoring operation is being conducted. The motivation for introducing logical clocks into monitoring is to have each process believe that it is executing in real time. In other words, the process will perceive that all events, both internal and external, are occurring at the same logical time with monitoring as they would do in real time without monitoring, so that all events in a process can happen in the same order whether the process is running without or with the monitor. As a result, transparency in monitoring is achieved. The concept of logical clock is a fully distributed approach, i.e., a distributed monitoring system may be adopted with a logical clock monitor on each processor of the parallel architecture.

8.6.1.1 Intuitive description of the logical clock approach

The monitor is designed as a program that interacts with the monitored processes in order to get information about process state and inter-process communication. This interaction can be implemented either by hardware (e.g. hardware trap and interrupt) or by software (e.g. code insertion) hooks, which are tightly integrated with certain events in the monitored processes. Thus, the interaction provides the monitor with the capability of inspecting interesting events and extracting the information about what is happening in the monitored parallel program.

During monitoring, a logical clock is maintained for each process, which reflects the real-time behaviour of that process when it runs without monitoring. Moreover, the logical clock renders the monitoring delay invisible in that it stops whenever the process's execution is interrupted by the monitoring activities, e.g. upon breakpoint or during information extraction. Therefore, although the real-time execution of a process is slowed down by the monitoring activities, it is not changed as measured using logical time. Each process is now executing according to logical time, reading its own logical clock that is unrelated to the logical clock of any other process.

In order to keep the communication ordering unchanged, it is not enough only to introduce the logical clocks. The monitor also has to control the occurrence of inter-process communication so as to preserve the original communication ordering. To achieve this, the monitored process has to inform the monitor (by monitoring hooks), with a timestamped message, every time it is ready to communicate, then wait for the monitor to permit that communication. The monitor makes the decision as to which inter-process communication should happen next based on logical time, rather than real time. It delays the occurrence of a communication if it is aware that there is another possible candidate process for that communication which is running in an earlier logical time. In this way, the communication is prevented from occurring either too early in logical time or too late in real time. The monitor uses the following principle in permitting the communication:

For each process, what should happen at real time T without the monitor now happens at logical time T with the monitor.

8.6.1.2 Algorithm

A monitoring hook behaves like a breakpoint, which can be inserted into the program no matter where the user attempts to extract information. An event cannot occur until the monitoring hook which reports the occurrence of that event completes its action. The action of a monitoring hook is said to be completed when the monitor resumes the execution of the process from which information is extracted. Without loss of generality, the monitoring hook which extracts for an event E is defined as:

send request and extract information;
wait for permission;
E;

To achieve transparency, we need to preserve the communication ordering on each process. Thus, for non-communication events, the monitor will simply adjust the corresponding logical clock to hide the monitoring delay. However, for communication events, the monitor also needs to impose control on their occurrence, as described below.

When a parallel process is running together with the monitor, the logical clock of the process P_i is defined as

$$t = L_i(T)$$

where t is the logical time and T is the real time. In the following, *Hstart(E)* is used to refer to the real time at which the information extraction for event E starts, and *Hend(E) is* used to refer to the real time at which the occurrence of the event E is permitted. The monitoring algorithm, therefore, can be informally described as follows:

1. Each process P_i, has a logical clock L_i which is initially set equal to the real time T_0 at which P_i starts to execute, i.e. $L_i(T_0) = T_0$.
2. If event E of process P_i is not a communication event, then permit the occurrence of E immediately after completing the information extraction. The logical clock of P_i is reset to $L_i(Hstart(E))$, i.e. $L_i(Hend(E)) = L_i(Hstart(E))$.
3. If event E of process P_i is a communication event, and its occurrence only depends on one process P_j, then:

   3.1. if P_j is not ready to communicate with P_i, then postpone the permission of E until case 3.2;

   3.2. if P_j is ready to communicate with P_i, and the corresponding communication event is E' (if E is a sending event, then E' is a receiving event and vice versa), then permit the communication, and reset their logical clocks according to the following:

   3.2.1. if the communication is synchronous, then reset the logical clocks of P_i and P_j to

$$\max(L_i(H\ start(E)), L_j(H\ start(E')));$$

   3.2.2. if the communication is asynchronous, the logical clocks are reset only when the monitor detects the message has arrived: suppose E is the receiving event and E' is the sending event, and the logical time of message arriving is LA_j, then P_i's logical clock is reset to

$$\max(LA_j, L_i(Hstart(E)))$$

   and Pj's logical clock is reset to

$$L_j(Hstart(E')).$$

4. If event E of process P_i is a receiving event, and its occurrence depends on the execution of other processes P_1, P_2,...,P_k, in which P_{r1}, P_{r2},...,P_{rm} are ready to send messages to P_i, while P_{b1}, P_{b2},...,P_{bn} are not (where m + n = k, and for asynchronous communication it is assumed there is no monitoring delay introduced on message transmission, and LA_{ri} (i = 1,...,m) corresponds to the logical arriving time of the message from process P_{ri}) then

   4.1. for synchronous communication,

   if $\min(L_{r1}, L_{r2},...,L_{rm}) > \min(L_{b1}, L_{b2},...,L_{bn})$ OR $m = 0$,

   or for asynchronous communication,

   if $\min(LA_{r1}, LA_{r2},...,LA_{rm}) > \min(L_{b1}, L_{b2},...,L_{bn})$ OR $m = 0$,

   then postpone the permission of E until case 4.2;

   4.2. for synchronous communication,

   if $\min(L_{r1}, L_{r2},...,L_{rm}) <= \min(L_{b1}, L_{b2},...,L_{bn})$ AND $m \neq 0$,

   or for asynchronous communication,

   if $\min(LA_{r1}, LA_{r2},...,LA_{rm}) > \min(L_{b1}, L_{b2},...,L_{bn})$ AND $m \neq 0$, then permit the communication between P_i and P_{rs} which satisfies

$$L_{rs} = \min(L_{r1}, L_{r2},...,L_{rm})$$

for synchronous communication
or

$$LA_{rs} = \min(LA_{r1}, LA_{r2},...,LA_{rm})$$

for asynchronous communication

and then reset the logical clocks as in case 3.2.

8.6.1.3 Implementation issues

In the logical clock approach, the monitor not only acts as a collector of information from the monitored parallel program, but also needs to have control over its execution. It requires the following:

- the ability to keep a *logical clock* for each parallel process;
- the ability to get an accurate *timing* of any monitoring activity;
- the ability to trap *inter-process communication*;
- the ability to control the occurrence of an *inter-process communication*.

These capabilities could be directly provided by modifying the machine architecture and integrating it with special monitoring hardware. However, the provision of hardware support is expensive and heavily depends on a particular machine architecture. Software approach, i.e. the program is modified by inserting the necessary hooks for the monitor to carry out the above functions at runtime, voids the need for special modification of the underlying machine architecture, and therefore enhances the portability of the monitor.