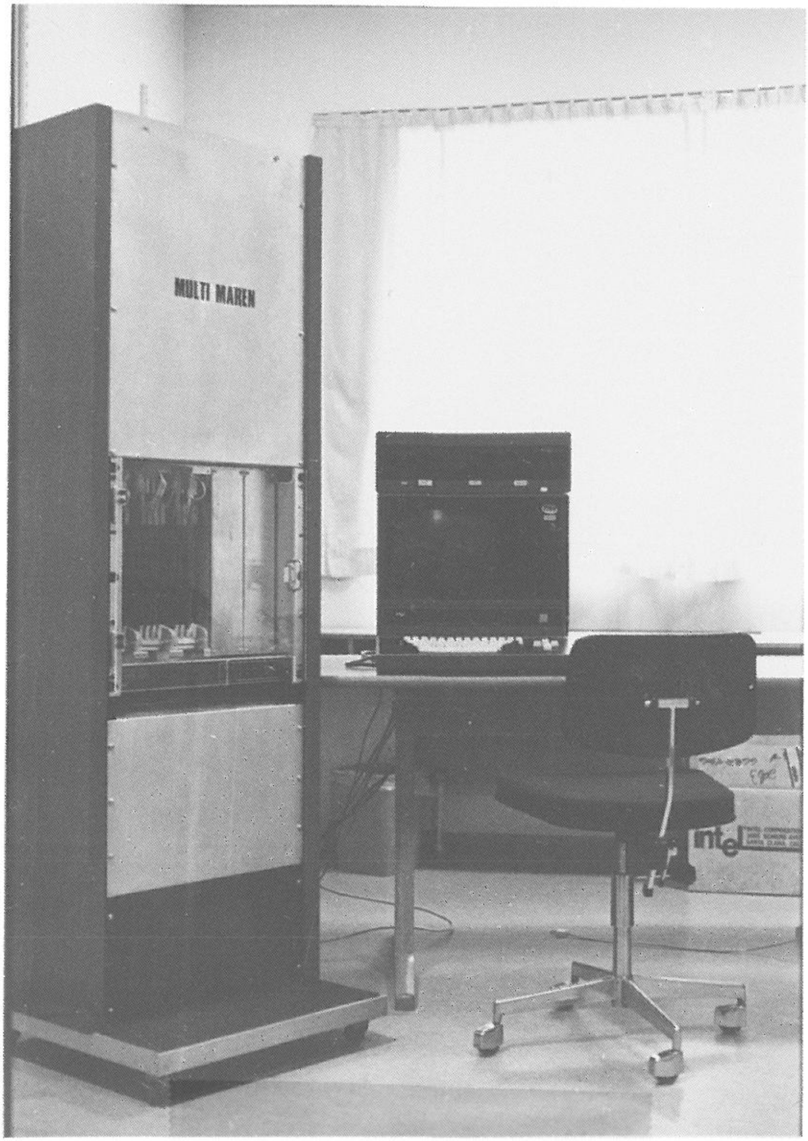
ISSN 0105-8517

EXPERIMENTS WITH A MULTIPROCESSOR

Peter Møller-Nielsen
Jørgen Staunstrup

DAIMI PB - 185
November 1984



The multiprocessor laboratory.

Table of Contents

1. Introduction

In this report we summarize four years of experience with the Multi-Maren multiprocessor laboratory.

The key component of this laboratory is a custom made multiprocessor with ten processing units. There are many interesting hardware problems in designing and building such a multiprocessor. We have, however, preferred to work with problems concerning its software. In particular whether there are algorithms that can utilize the potentially very large processing power of a multiprocessor. We have found that this is certainly the case and in the following chapters we present a few of these fast multiprocessor algorithms. On the other hand we have also found a number of examples where the gain in using a multiprocessor is very small. In all these examples the limiting factors are inherent in the algorithm, they are not hardware bottlenecks. In chapter 2 we justify this somewhat surprising claim by showing that even in multiprocessors with an order of magnitude more processors than ours it is unlikely that hardware bottlenecks will limit the performance. Except for chapter 2 this report contains very little about the hardware of multiprocessors.

During the design and implementation of the laboratory we have been confronted with many decisions. In making these decisions we have tried to be faithful to our overall goal: the study of multiprocessor algorithms. As little effort as possible has been spent on all other aspects, even though many of these might be interesting in their own right.

To avoid spending time on designing and building hardware we have used standard equipment. In many respects such hardware is not ideal. It does, for example, contain many superfluous components. However, it is unlikely that more specialized hardware would have influenced our results significantly.

We wanted to be able to program the machine in a high level language. As it was the case with the hardware, standard languages were not ideal, but on the overall probably better than designing a new language. We found Concurrent Pascal to be a useful starting point, and this language has therefore been implemented on the multiprocessor.

There were two major design criteria leading us through this implementation:

- the implementation effort should be as small as possible,

- the implementation should be as transparent as possible.

The first design criteria is rather obvious; the less effort we spent on the implementation the more we could spend studying multiprocessor algorithms. The second criteria is more subtle. In order to experimentally verify models for the dynamic behaviour of multiprocessor algorithms, it is very important to be able to distinguish properties of the particular algorithm from properties of the underlying software. This is only possible if this underlying software is transparent, e.g. there are no such things as hidden scheduling policies, or system facilities for collecting performance data. The price we pay for this high degree of transparency is that the Concurrent Pascal program which is executed on the multiprocessor, is a somewhat awkward merge of the following three rather different program modules:

- a module which implements the basic algorithm under investigation,
- a module which implements the decisions about what quantities to measure during a single execution of the algorithm, and when to collect them,
- a module which makes it possible to perform several independent executions of the basic algorithm (and related data collections) in a single execution of the complete program. The values of the parameters for the individual executions of the algorithm (e.g. the accuracy required for the quadrature algorithm discussed in chapter 5) are also controlled by this program module, and all input and output is done here. In short, this module contains the overall control of the progression of the measurements during a single program execution.

The only tool which is made available by the underlying software and hardware is the Concurrent Pascal function 'realtime' which supplies the time with a resolution of 0.01 sec. In this connection we would like to point out that the analysis of the dynamic behaviour of an algorithm is simplified a lot by all processors being identical, e.g. executing Concurrent Pascal statements with the same speed.

In chapter 8 the Concurrent Pascal implementation is described in further detail together with some of our modifications to the language.

The appearance of our multiprocessor is as a Concurrent Pascal machine which is able to execute up to ten processes simultaneously, one for each processor. It is a single user machine, so only one Concurrent Pascal program can be executed at a time. So the machine has no operating system to administer different users sharing the machine. From the above it should be obvious that we have refrained from building an operating system, because it is not absolutely necessary for our work with construction and analysis of multiprocessor algorithms, and it decreases the transparency of the software underlying the algorithms.

The only i/o facility of the multiprocessor is a serial line usually connected to a small standard development system (see fig. 1.1).

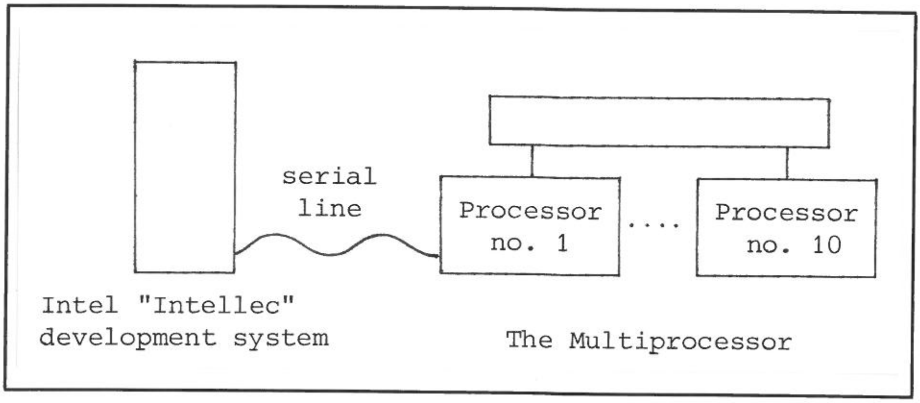


Fig. 1.1

The lack of i/o devices has also been a limitation; it has, for example, been cumbersome to run programs requiring large amounts of input data. But the advantage has been avoiding spending a long time on the tedious task of writing i/o drivers, spoolers etc.

In chapters 4, 5 and 6 we present some examples of multiprocessor algorithms together with an analysis of their behaviours. Typically these results have been obtained by a number of iterations of the following steps:

— invent or modify an algorithm to obtain a multiprocessor algorithm with ten or less independent processes,

— make a model of the behaviour of the algorithm, typically an analytic expression for the execution time of the algorithm;

— find which parameters, e.g. time spent in isolated parts of the algorithm, that will allow validation of the model;

— write a Concurrent Pascal program that implements the multiprocessor algorithm and which allows measuring the parameters of the model;

- run the Concurrent Pascal program on the multiprocessor to obtain measurements of the parameters;
- confront the measurements with the predictions made by the model.

Deviations between model and measurements lead to changes in the algorithm or model and all or some of the above steps are repeated until a reasonable model of the behaviour of an algorithm is obtained. As in all modelling, the goal is a model that provides insight which can be used in predicting properties of similar algorithms.

Since our approach has many similarities with experimental physics, it is quite natural that we have borrowed some terms from this field, terms like: laboratory, experiment, model and measurement.

This introduction has listed a number of things that we have not done, in the following chapters we present in further detail what we have done.

Acknowledgement

A large number of people have assisted us in the Multi-Maren multiprocessor laboratory. The initial Concurrent Pascal implementation was mainly done by Jens Søgaard Andersen and Kurt Søe. Later Ole Eriksen has made a number of important improvements. Many multiprocessor algorithm have been implemented and analyzed by graduate students at Aarhus University. Special thanks are due to two of our colleagues, Ole Caprani and Sven Skyum who have both had a significant influence on our work. Last, but certainly not least, we want to thank John W. Nielsen and Jan Christensen for their prompt and skillful hardware support.

The Multi-Maren multiprocessor laboratory has been built with the financial support from

»Aarhus Universitets Forskningsfond«
»Statens naturvidenskabelige Forskningsråd«
»Statens teknisk-videnskabelige Forskningsråd« and
»Aarhus Universitet«.

2. Saturation in a Multiprocessor

In this chapter we discuss the problem of saturation in a multiprocessor with a common bus architecture. It is demonstrated that it is logical constraints in the algorithm being executed which limit the number of processors that can be utilized. By taking a few simple precautions, saturation of the common bus can be avoided. This is even true for a common bus multiprocessor with several hundreds of processors.

Although this claim is based on experiments with our particular multiprocessor, it holds for a large class of multiprocessors with a similar architecture. Even if different hardware may reduce the saturation point significantly, it is still beyond the limits imposed by other factors of the applied technology, e.g. power supplies, wire lengths etc.

Multi-Maren has ten identical processors connected by a common bus to a global store (see fig. 2.1). Each processor consists of a processing unit (Intel 8086), a local store, timers and various i/o ports, all mounted on a single board. The common bus is an Intel Multibus [Intel 1979] equipped with an arbiter which resolves bus conflicts in a round robin manner. The arbiter is described in further detail in section 7.3. By using a round robin policy the arbiter can guarantee a fixed maximal waiting time.

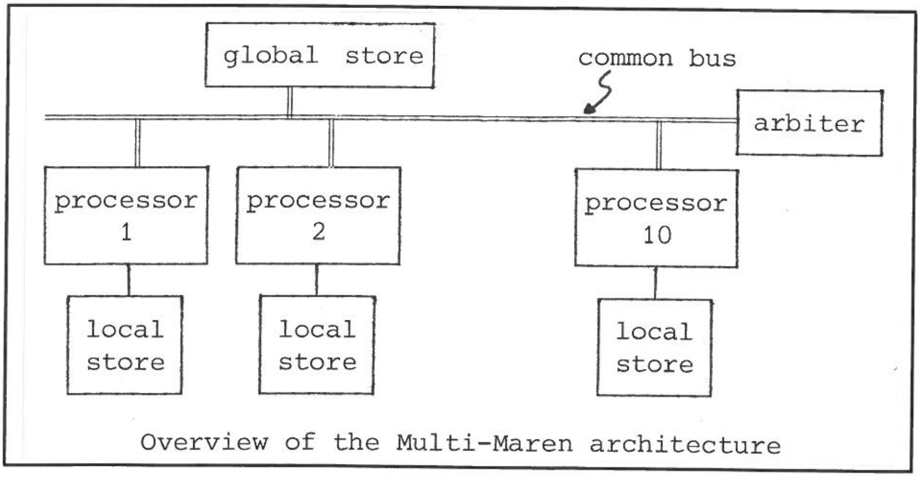


Fig. 2.1

When first building the machine it seemed obvious to us that this architecture had an inherent bottleneck, namely the common bus/global store, but pilot studies in other projects [Jones 1980] indicated that with approximately ten processors the common bus would not become a bottleneck. Furthermore, there was no other alternative which did not require us to design and build new hardware. To reduce the danger of saturating the common bus, we separated the various parts of a Concurrent Pascal program as follows:

- Shared variables: the synchronization variables that are necessary to provide mutually exclusive access to the shared variables (i.e. a critical region) must be placed in the global store. The shared variables themselves are also placed in the *global store*.

- Local variables and program code: local variables used by one process only are placed in the local store of the processor executing that process. Similarly, the code (program) describing the process is placed in the *local store*. Even when several processors execute the same code, that is identical processes or the same monitor procedure, this code is duplicated in all the local stores of those processors. This may seem an unnecessary waste of storage space, but the benefit is that all references to the code are strictly local and they do not affect the common bus. By making a static allocation of processes to processors, the amount of code that must be duplicated is reduced considerably.

Since 1981 we have experimented with implementing a number of different algorithms. The common trend in these experiments is: *the common bus is almost never used*. Typically, the bus is busy less that 1% of the time when ten processes are running. At first this was a surprise; but the explanation is simple. When instructions are placed in the local stores, all the machine cycles concerned with instruction fetch, address calculation, decoding etc. do not affect the common bus. Operand fetching is only a small part of the instruction and these operands are only rarely shared variables placed in the global store. This explanation is confirmed by the experiments described below.

Now we consider the distribution of storage references at the hardware level, i.e. the storage references generated when executing machine instructions. Viewed from a particular processor these storage references can be divided into two categories: local and global. The time it takes to execute a global storage reference is the sum of two components: G, which is the time it takes to transfer data to or from the global store, and A, which is the time it takes to gain exclusive access to the global store, i.e. the arbitration time.

```
              A                      G

    x--------------x-----------x
```

Fig. 2.2

G is fixed, but A may vary with the load on the common bus.

Usually there are many references to the local store between two references to the global store. Let the total time for such a sequence of local references be L. An execution on a particular processor may be depicted as in fig. 2.3.

```
          L       A    G       L        A     G

... x-------X---X---X-------x-----x---x ...
```

Fig. 2.3

From [Brinch Hansen 1973] it is seen that when L is fixed and the same for all processors, the number of processors that can be connected to the global store before it becomes saturated, is limited by the value

$$N^* = 1 + \frac{L}{A_0 + G}$$

A_0 is the average arbitration time when only one processor is using the common bus. In our machine $A_0 = 0.5$ μsec. If L is fixed, i.e there is a fixed time interval between accesses to the global store, all accesses to the global store will take $A_0 + G$ as long as the number of processors is less than N^*. When N grows beyond N^*, the arbitration time increases proportionnally to $(N - N^*)$. This is indicated by the solid curve in fig. 2.4.

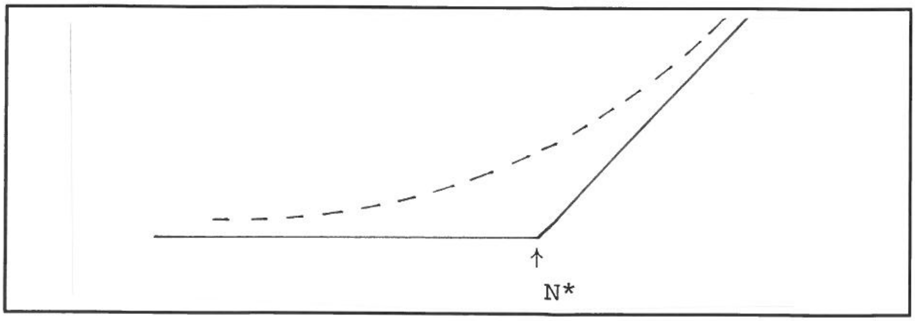


Fig. 2.4

When L is not constant, A is no longer a constant, and the average value of A will follow a pattern indicated by the dotted curve in fig. 2.4.

In order to estimate the value of N^*, we have measured the values of L, A and G under different circumstances. We obtained the following values:

	min.	*average*	*max.*
L	400	$3 \cdot 10^5$	–
A	$6 \cdot 10^{-2}$	0.5	7.5
G	1.5	1.5	1.5

(All numbers given in μsec.)

The average value of L is taken over a sample of algorithms including a root searching algorithm, a quadrature algorithm, a sorting algorithm and a pattern matching algorithm. The min. value of L is obtained by an artificial Concurrent Pascal program which references global monitor variables as frequently as possible. The average value of A is the average arbitration time observed for all the Concurrent Pascal programs that were used to form the average value of L. Due to the round robin nature of the arbiter, the max. value of A for a system with 10 processors and an arbiter like ours is $\approx (10 - 1) \times (1.5 + 0.1) \approx 14\mu sec$. (This number increases with the number of processors.) However, from the numbers in the table it is seen that we have not been able to arrange a situation in which this max. value occurs. The min. of A merely indicates, that the arbitration time can be as small as 0. The fact that average A is equal to A_0 indicates that ten processors are well below the saturation point N^*. G is fixed, as we expected. From the values in the table we find that

$$N^* = 1 + \frac{averageL}{A_0 + G} > 10^5$$

for the practical Concurrent Pascal programs we have studied.

Even for a Concurrent Pascal program that includes constructs for which we have found no practical use, N^* is rather big, namely:

$$N^* = 1 + \frac{minL}{A_0 + G} \approx 200$$

In order to estimate the impact of the most unfortunate case, i.e. when a processor constantly experiences a maximum arbitration time of 14 μsec., we may notice that a normal Concurrent Pascal process is delayed only by

$$\frac{14}{300000} \times 100\% < 10^{-3}\%$$

and a pathological program by

$$\frac{14}{400} \times 100\% \approx 3.5\%$$

which is just observable.

All these numbers are of course specific for Multi-Maren and our implementation of Concurrent Pascal. Similar numbers for other multiprocessors may differ, but even if they differ by several orders of magnitude (which we believe is very unlikely), hundreds of processors may be connected to the bus without getting hardware saturation. But the precondition is that *all code and local variables are placed in the local store* of the processor using these.

We find that the above gives us good reason to concentrate on the programming of the multiprocessor. In the next chapters we will summarize our work on such multiprocessor algorithms.

3. Problem-heap Algorithms

A primary goal of the Multi-Maren project has been to find simple and fast multiprocessor algorithms. To achieve fast execution the task to be done must be split into a number of subtasks which can be done simultaneously by different processors. Some tasks can be split into a number of independent subtasks, one for each processor, before the computation starts. When such a *static splitting* is possible it is straightforward to get a simple and fast multiprocessor algorithm. But when a static splitting cannot be made or does not give a satisfactory performance a *dynamic* algorithm must be used. We have therefore studied a number of algorithms which use a dynamic splitting of those subtasks that turn out to be hard and require much work to be completed. This chapter describes a class of such simple algorithms called *problem-heap algorithms* together with an analysis of their performance. The main characteristics of these algorithms are:

— *a number of identical and asynchronous processes* cooperate on doing a certain task. The processes share one or more data structures that represent partial results and those parts of the task that remain to be done, this is called:

— *a problem-heap*. To be in this class of algorithms the task of the algorithm must be such that it can be split into a number of subtasks called *problems*. A process takes a problem and tries to solve it. This may generate new problems which are then put in the problem-heap or the problem may be simple enough that it can be solved immediately. This class of algorithms clearly contains the »divide and conquer algorithms« [Aho, Hopcroft and Ullman 1974] , but as it will be clear below it also contains a number of other algorithms.

The major asset of the problem-heap algorithms is their simplicity; all processes are asynchronous, identical and an arbitrary number of them may be used. The task will be completed (i.e. all problems solved) no matter how few or how many processes that are used. What may vary is the performance of the algorithm i.e. how fast the task is completed. This variation is almost exclusively caused by properties of the problem-heap. It turns out that there is a small number of different sources which may lead to reduced performance. The major part of this chapter is an identification of these sources, together with an analysis of their effect.

3.1 Skeleton Algorithm

As mentioned above one characteristic of the problem-heap algorithms is that all processors execute the same process. Furthermore, there is a common skeleton of this process which may be found in all problem-heap algorithms:

```
problem-heap skeleton:

CYCLE
    take a problem from the heap;
    IF the problem is simple
        THEN
            solve the problem;
            include the result in the solution;
        ELSE
            split the problem into other problems;
            put the new problems back in the heap;
END
```

Algorithm 3.1

The primitives used in this skeleton will of course vary; as the most obvious »solve the problem« is different for each task to be performed.

Example

A well known example of a »divide and conquer algorithm« is Quicksort, which sorts a list of numbers represented in an array. In the problem-heap formulation, a problem is a consecutive sublist of the array (see fig. 3.1):

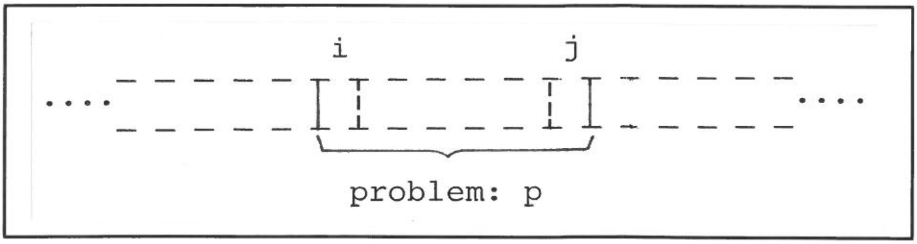


Fig. 3.1

To solve *p*, all numbers in *p* must be sorted and put back in the same index range *i..j* of the array. All processors execute the following process:

```
CYCLE
    p: = a problem from the heap;
    IF size(p) = 1
        THEN
            report p is sorted
        ELSE
            split p into p1 and p2 so that:
           p: | p1 | p2 |
            and all x in p1, and all y in p2 satisfy $x \leq y$;
            put p1 and p2 back in the heap;
END
```

Quicksort is chosen as an example because it is a well known algorithm; as it will be demonstrated later it has a rather poor performance on a multiprocessor.

3.2 The Problem-heap

As described above the problem-heap is a shared data structure which all processes may access (see fig. 3.2). It can be viewed as a source of data which supply all processes. When the problem-heap is drained the processes starve and their processing power is lost.

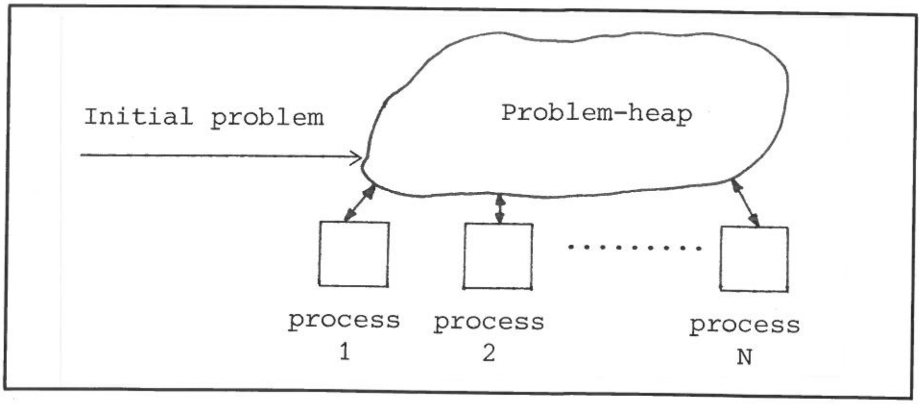


Fig. 3.2

The structure of the problem-heap is not nearly as uniform as indicated by the skeleton algorithm discussed above. Sometimes it is organized as a stack, sometimes the problems are sorted, sometimes solving one problem requires deleting other problems in the heap etc. This is all dependent on the task to be done. The heap is initialized with one problem, the *initial problem*, describing this task.

In principle the heap is global to all processes but in some algorithms it is advantageous to divide it into a number of subheaps, this is discussed in further detail in section 5.2 where the results from a number of experiments with a subdivided heap are reported.

3.3 Speed-up

Most of our performance analysis has been concerned with finding the speed-up of problem-heap algorithms. Let $T(N)$ be the time it takes to perform a certain problem-heap algorithm using N processes (each executing on a separate processor) the *speed-up* is defined as:

$$S(N) = T(1)/T(N)$$

The major advantage of this measure is that it is almost *processor and implementation independent*. Consider, for example, two machines where one is twice as fast as the other. The running times for a particular algorithm would be very different on these two machines, but the speed-up would be the same.

This definition of the speed-up looks quite obvious, but there are several pitfalls that should be avoided when using it. Therefore we clarify our use of this measure before going into further detail.

Consider a task T and the following four algorithms for solving T:

T_S: A sequential algorithm for solving T without splitting it into subproblems,

T_H: a sequential algorithm for solving T by splitting it into subproblems which are solved one at a time,

T_C: a concurrent algorithm for solving T by splitting it into subproblems some of which are solved concurrently by interleaving the computations,

T_M: a multiprocessor algorithm for solving T by splitting it into subproblems, some of which are solved simultaneously by different processors.

Usually T_H, T_C and T_M are not distinguished, since they are regarded as different ways of implementing the same abstract algorithm.

To illustrate the distinctions between T_S, T_H, T_C and T_M, consider an algorithm for finding an occurrence of a pattern in a long string (see fig. 3.3).

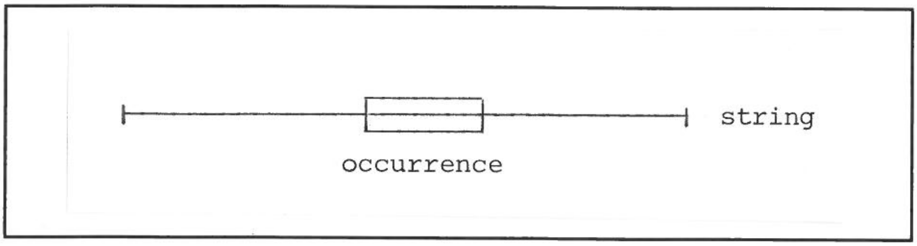
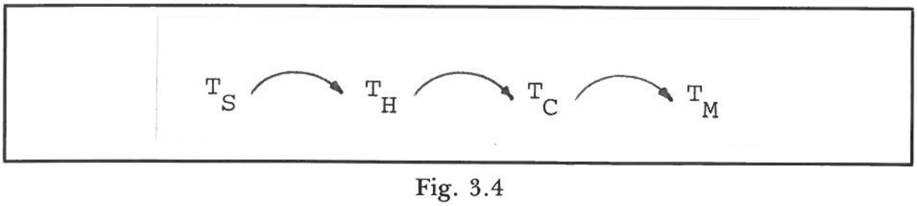


Fig. 3.3

This task could be done by an algorithm T_S which searches the whole string from the left. Alternatively the string could be divided into a number of substrings ($\approx$ subproblems). These substrings could be searched one at a time. The algorithm T_H would finish searching one substring before searching the next.

As still another alternative consider an algorithm T_C which starts searching many substrings concurrently, by switching back and forth between them. Finally, a multiprocessor algorithm T_M would search several substrings simultaneously by giving different substrings to different processors as described by the skeleton algorithm 3.1.

One way of developing a multiprocessor algorithm is by starting from a sequential algorithm T_S and then transform this into a sequential problem-heap algorithm T_H, which is then made into a concurrent algorithm T_C from which the multiprocessor algorithm T_M is obtained (see fig. 3.4). We would like to stress that this is certainly not the only method of developing multiprocessor algorithms. It is emphasized here mainly to clarify our use of the speed-up measure.



$$\begin{tikzcd} T_S \arrow[r, bend left] & T_H \arrow[r, bend left] & T_C \arrow[r, bend left] & T_M \end{tikzcd}$$

Fig. 3.4

Note, that from a single algorithm T_S it is sometimes possible to derive a whole family of similar T_H-algorithms. Take the string searching algorithm as an example. When the string is divided into substrings ($T_S \rightarrow T_H$ transformation) this opens for the possibility of choosing substrings of different length or searching the substring in another order than the left to right which is used in T_S. So for each possible size of substrings and each possible order of searching the substrings there is a distinct member of the T_H family.

Similarly the transformation from T_H to T_C might lead to a whole family of T_C-algorithms for each T_H-algorithm.

When comparing T_S and T_H, note that there are examples where either of them are fastest. By picking first the substring which contains an occurence, T_H can sometimes finish very quickly. But on the average we would expect T_H to be slightly slower than T_S due to the overhead caused by the splitting, this is discussed in further detail in section 3.6 on *iteration loss*.

When we compare T_C and T_M several phenomena can be observed. As mentioned above the problem-heap is shared by the processors and only one of them may update it at a time. This can lead to *saturation loss* for T_M which is discussed further in section 3.7.

It might also happen that the problem-heap becomes empty before the computation is finished. When this happens in T_M the processors must wait for more problems to be generated leading to *starvation loss* which is discussed in section 3.4.

The string searching sketched above is an example of a task where the amount of work done (i.e. the running time) vary radically with the order in which the subproblems are solved. If the algorithm starts by picking a substring containing an occurrence very little work has to be done, but the opposite can also happen, namely picking the substring with an occurrence as the last to be searched. When the amount of work that has to be completed vary with the order in which it is done, the algorithm is called *unstable*. Algorithms that do not have this variation are called *stable*. An example of this is an algorithm for finding *all* occurrences of a particular pattern in a string. In this case the work done by the algorithm T_S and all the algorithms in the related families T_H and T_C is almost the same.

Let us return to the definition of the speed-up:

$$S(N) = \frac{T(1)}{T(N)}$$

The term $T(N)$ does not cause any problems, it is the execution time of the T_M algorithm using N processors to solve N subproblems simultaneously. For stable (families of) algorithms there is no problem with $T(1)$, since the execution times of T_S, T_H and T_C are almost the same, it does not matter which is chosen as $T(1)$. But now consider an unstable family. In this case the execution times for T_S, T_M and T_C may vary almost arbitrarily. Depending on the choice the speed-up may be both much larger and much smaller than linear. Such anomalies have been reported by [Wilkes 1977] and [Lai 1984]. When an unstable algorithm exhibits such a speed-up larger than linear, it is because the multiprocessor algorithm makes a very fortunate choice of the order in which subproblems are solved. Another order of solving the problems might give a very small speed-up. Therefore, we think that one should *be extremely cautious in using the speed-up measure with unstable families of algorithms.*

The instability of a family of algorithm occurs because some members of the family perform more work than others to complete the task. So some of the algorithms do *superfluous work*, because it cannot be decided beforehand what work is superfluous and what is not. This can only be decided after the task has been done. The speed-up can only be used with unstable families of algorithms, if the algorithms that are compared do the same amount of work. Comparing an algorithm which does little or no superfluous work with one that does a lot is meaningless.

Finally, different input data may give different execution times for an algorithm. When this is the case, we have usually been looking for the average speed-up. This has been found as the average of the speed-ups observed by a number of different input data, let T_i be the execution time on the i-th set of input-data:

$$S_A(N) = \frac{1}{K} \sum_{1}^{K} \frac{T_i(1)}{T_i(N)}$$

Note, that this may not be the same as

$$\frac{\frac{1}{K} \Sigma\, T_i(1)}{\frac{1}{K} \Sigma\, T_i(N)}$$

i.e. the speed-up of the average running times.

To depict the execution of a particular multiprocessor algorithm, diagrams as the one shown in fig. 3.5 can be used.

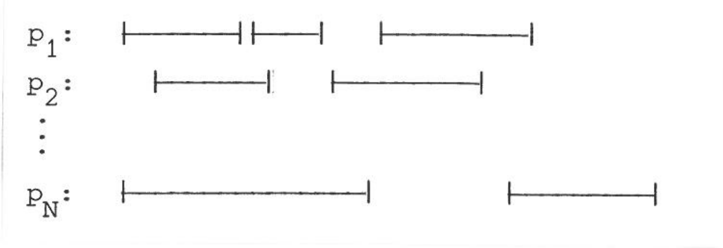


Fig. 3.5

The holes in the diagram represent loss i.e. periods where the processing power of the correponding processor is not utilized. A diagram without holes is therefore a necessary (but not sufficient) condition for achieving a linear speed-up.

In the four sections to follow we describe four different kinds of loss that we have observed in our experiments with multiprocessor algorithms. Deviations from a linear speed-up are explained as a combination of these four kinds of loss, usually one or two of them is dominant. For those algorithms that we have studied, it has always been possible to explain the observed deviations from a linear speed-up using these four kinds of loss: starvation loss, braking loss, iteration loss and saturation loss. There may of course be other kinds of loss that we have not yet identified.

3.4 Starvation Loss

The problem-heap must constantly supply all processes with problems (data). When there are too few problems in the heap, the workless processes starve and their processing power is lost. This situation is illustrated in fig. 3.6.

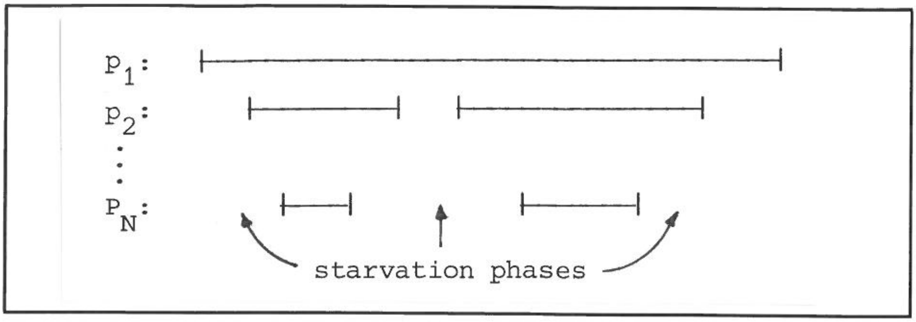


Fig. 3.6

In most problem-heap algorithms there is some starvation in the initial phase until all processes have gotten their first problem to work on. In most cases the loss caused by this is negligible but in other cases, e.g. Quicksort, the loss caused by starvation in the initial phase is the bottleneck of the algorithm. Note, that there is a distinction between the loss caused by starvation and the saturation loss caused by mutual exclusion (see section 3.7). When processes attempt to get problems there may be some loss because they need mutually exclusive access to the problem-heap. Whereas the loss caused by starvation occurs when there are *no* problems in the heap. Consider, for example, what happens while the initial problem is split in two or more subproblems. During such a splitting the problem-heap is free for all processes to inspect, so there is no saturation loss. But if the splitting takes a long time there is a considerable starvation loss. The difference is illustrated in fig. 3.7.

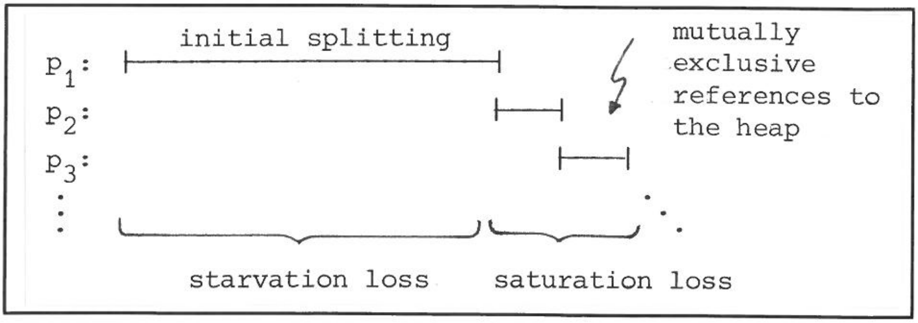


Fig. 3.7

Although starvation is often found in the initial and final phases, the phenomenon may of course occur at any point of the execution. Let T_i be the times defined by the diagram shown in fig. 3.8.

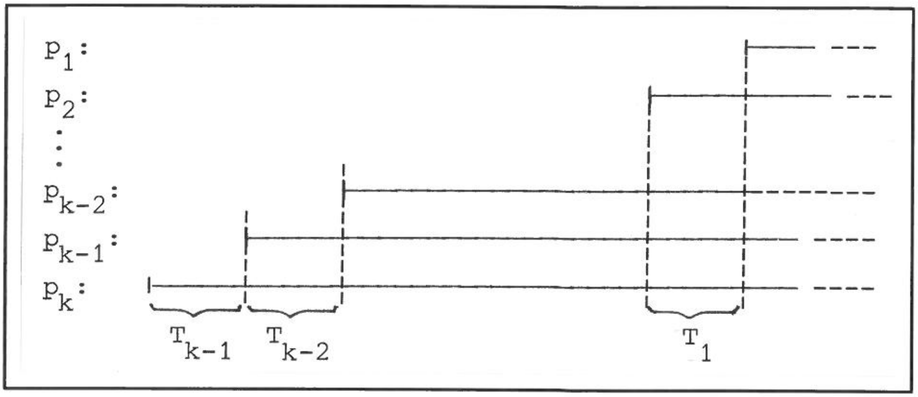


Fig. 3.8

The total loss caused by this starvation phase is:

$$S_{loss} = \sum_{i=1}^{k-1} T_i \times i$$

i.e. $k - 1$ processes each loose T_{k-1}, $k - 2$ processes each loose T_{k-2} etc. As mentioned above starvation may happen at any point of the computation because there can be several starvation phases during one execution. If this is the case each of the starvation phases cause a loss as the one described above.

The multiprocessor implementation of Quicksort exhibits a very clear example of starvation loss. The first step of the computation consists of splitting the data into two lists (problems) such that all elements in one list are less than all elements in the other. This splitting requires an inspection of all data elements, so the time to split is proportional to the number of elements, M. During this period, one processor only is working. After this two processors may be put to work on splitting the two problems into four, etc. This is illustrated by the diagram in fig. 3.9.

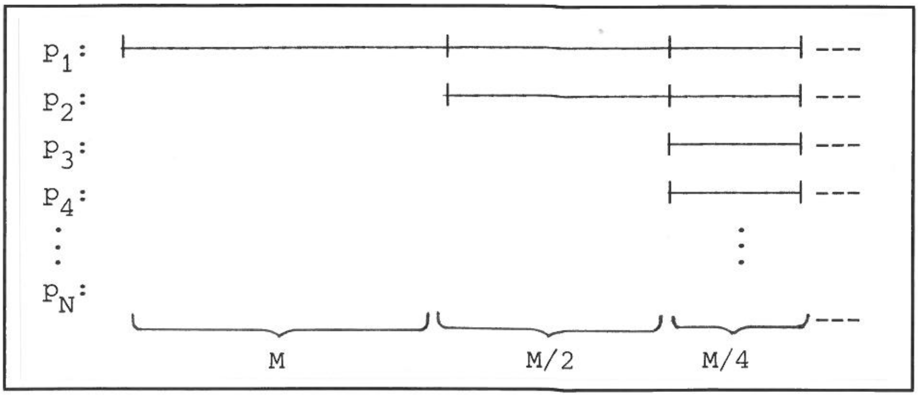


Fig. 3.9

The magnitude of the starvation loss is (assuming N is a power of two):

$$S_{loss} = \sum_{i=0}^{logN} (N - 2^i) \times \frac{M}{2^i}$$

For small values of M this is the dominant source of loss.

3.5 Braking Loss

When a process receives a problem to solve, it either solves it or splits it before consulting the problem-heap again. In any case there is a period when it works in isolation solving a subproblem; if during this period the entire task is completed, the process cannot be stopped until it again consults the problem-heap. The processing power wasted during this period is called *braking loss*.

To be more precise, consider the diagram in fig. 3.10 illustrating the final phase of a computation.

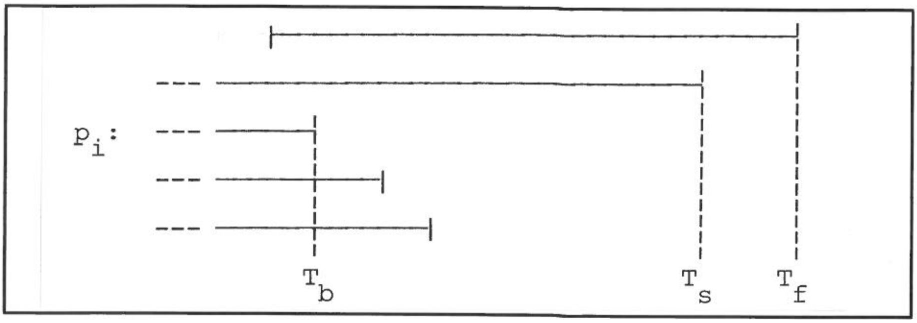


Fig. 3.10

The three instances on the diagram are defined as follows:

T_b: The first potential solution to the complete task has been found by processor p_i. Other processes *already* working on subproblems finishing later might find other improved solutions. But no processes are given new problems to solve after T_b.

T_s: The final solution is found. There might still be some processes working on subproblems not contributing to this final solution.

T_f: All processes have been stopped so none are working on solving sub-problems.

As it is clear from the examples given below, these three time instances are not always distinct. The terms *cycle time* and *access time* are frequently found in hardware specifications. The access time of a circuit is the time it takes to perform its function and the cycle time is the minimum time required between two invocations. In our terminology the access time of the algorithm is T_s and the cycle time of the algorithm is T_f.

Consider again the problem of finding the leftmost occurence of a pattern in a string (see also section 3.3). In this example T_b is the time instance where some process finds the first occurrence of the pattern. Other processes might however find other occurrences further to the left. So T_s is the time instance where an occurrence has been found and all parts of the string to the left of the occurrence have been searched without success. At time T_s there might still be processes working on (irrelevant) substrings to the right of the occurrence. T_f is the time instance when all these have stopped. If instead the task is to find not the leftmost but any occurrence of the pattern, T_b is the same as T_s, but also in this case T_s is distinct from T_f.

Braking loss can now be defined as the processing lost between T_s and T_f. This is indicated as the ///// periods in the diagram shown in fig. 3.11.

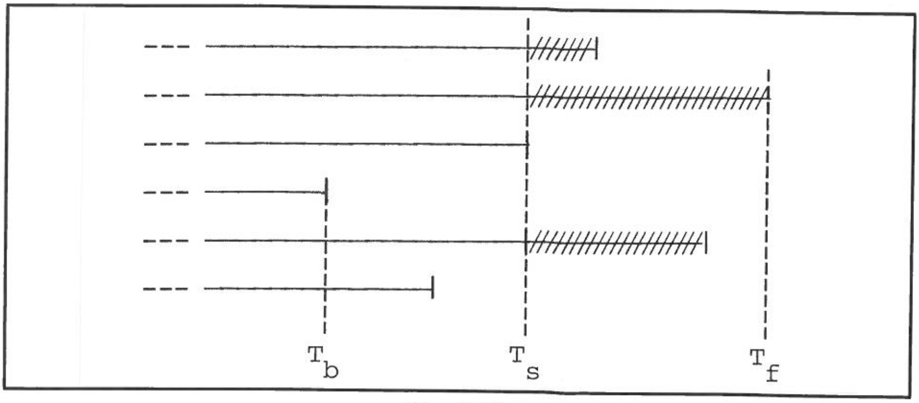


Fig. 3.11

Although braking loss is most frequently found at the end of the computation there are also examples of algorithms where the solution of one problem suddenly makes other problems in the heap superfluous. The latter happens in some algorithms where the solution of one subproblem suddenly changes the course of computation completely.

A *braking point* is a point of time where all processes should be braked either to stop completely or to get a new problem to work at from the heap. T_s is an example of a braking point. The period of time between a braking point and the next reference to the problem-heap by process i is called B_i. The loss caused by one braking point is therefore :

$$B_{loss} = \sum_i B_i$$

It is obvious that B_i and the time it takes to solve or split a problem are proportional, the exact relationship depends on the particular algorithm. Braking loss is discussed further in chapter 4.

3.5.1 On Interrupts

The loss caused by braking is analogous to the reduced response time found when an external device is not polled frequently enough. In both cases the occurrence of an event is discovered by regularly inspecting some status information (see fig. 3.12).

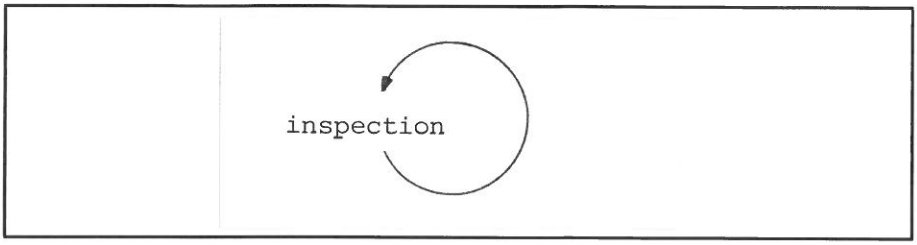


Fig. 3.12

The loss may sometimes be reduced by making the cycle shorter, i.e. inspecting the status more frequently. The most common way of making this reduction is by moving the inspection cycle to an underlying level of software or hardware. At the higher level the inspection can now be replaced by an interrupt.

Interrupts could also be a way of reducing the braking loss in problem-heap algorithms. An interrupt should force a process to consult the problem-heap immediately. But what should happen with the problem which the process works at when it is interrupted? Should it be postponed, discarded, put back in the heap or .. ? An effect similar to using an interrupt can usually be achieved by reducing the problem size.

3.6 Iteration Loss

All problem-heap algorithms are iterative (see algorithm 3.1), in each iteration a problem is either solved or split into simpler subproblems. But only in a very few cases is there an obvious limit on the problem size above which problems should be split and under which they can be solved without further splittings. By making the limit to high one may risk starvation loss as described in section 3.4. On the other hand there may be an overhead associated with splitting a problem. Solving two subproblems may require more work than solving them together as one problem. This overhead is called *iteration loss*.

Consider again the string searching example. By splitting a string, s, in two, s_1 and s_2, a little extra work is introduced since the first characters of s_2 might be inspected twice (see fig. 3.13).

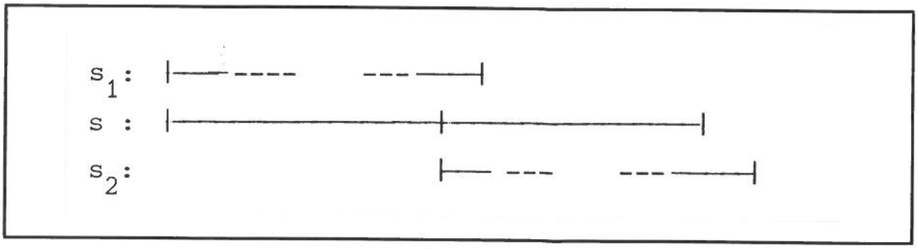


Fig. 3.13

Once to check for an occurrence towards the end of s_1 and the second time to check for an occurrence in the beginning of s_2. Obviously the iteration loss grows with the number of of iterations of the algorithm. If we assume that the iteration loss is the same amount, I, in each iteration, the total iteration loss is:

$$I_{loss} = K \times I$$

where K is the number of iterations of the algorithm. Iteration loss is discussed further in chapter 4.

3.7 Saturation Loss

The problem-heap is usually organized as a shared data structure which the processes may reference one at a time i.e. the problem-heap is a monitor in the sense of Concurrent Pascal. When two processes try to reference the problem-heap at the same time one of them must wait which causes some loss called *saturation loss*. Let $G1$ and $G2$ be the times it takes for the two processes to complete their references to the problem-heap, the diagram in fig. 3.14 illustrates the loss they can experience.

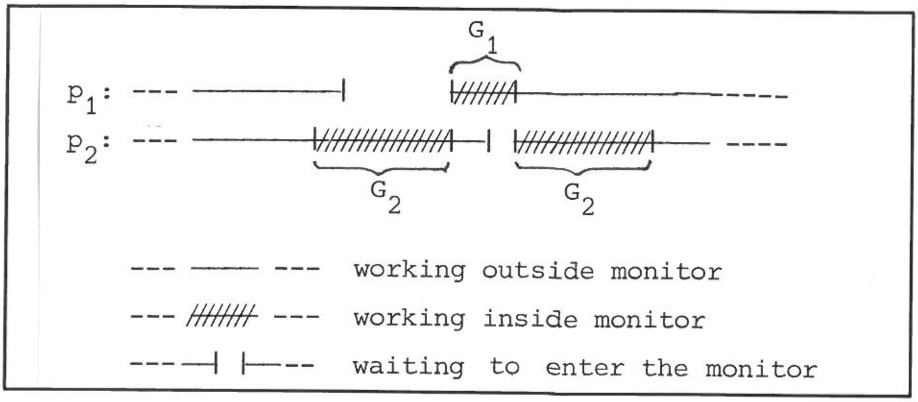


Fig. 3.14

It is obvious that this kind of loss may happen at each reference to the problem-heap i.e. once in each iteration of all the processes (see algorithm 3.1). In general the saturation loss grows with the number of iterations and processes. To analyze the saturation loss we have used traditional techniques from queueing theory. This kind of loss has been studied rather extensively in the literature. Although the emphasis there has been on hardware saturation (which has not had any significant effect in our experiments) the techniques and results can be applied to saturation loss (software) which can occur and has occurred in our experiments. Saturation loss is also referred to as software saturation, it is studied in some detail in chapter 5.

3.8 Summary

The problem-heap structure has emerged as a simple technique for programming a multiprocessor. Our experience shows that a number of algorithms follow this structure, some of them have a good performance, others have a rather poor one.

The four kinds of loss identified through our experiments give a platform for predicting and analyzing new problem-heap algorithms. There is, however, no reason to believe that this is the ultimate classification. New experiments and further work might uncover other kinds of loss or may lead to a refinement of the four proposed by us. This would certainly we welcomed.

We do not want to claim that the class of problem-heap algorithms is universal. There are other clearly distinct classes of multiprocessor algorithms e.g. pipeline algorithms where data flow through a string of processors. But there are some very nice properties of the problem-heap algorithms. They can be performed by:

- *an arbitrary number of processors*. The algorithm does not have to be rewritten when the number of processors is increased or decreased. This means that extra processing power can be added without changing the algorithm (i.e. the program text),
- *asynchronous processors*. The algorithm can be executed by processors running with different and varying speeds,
- *symmetric processors*. All processors execute the same algorithm (i.e. program text). This is much easier to handle than writing a number of different algorithms.

Note, that the pipeline algorithm does not have any of these properties, since each processor in the pipeline runs a different algorithm, and the work done by the processors must be in balance.

4. String-searching Algorithm

This chapter describes some experiments with a string-searching algorithm similar to the ones discussed in chapter 3.

Consider a string, S, of L characters from some alphabet, and a pattern, P, which is a string of K characters from the same alphabet ($K < < L$). Consider the problem of finding the leftmost occurrence of P in S, i.e. the least index i such that:

$$S(i..i + K - 1) = P(1..K)$$

Boyer and Moore [1977] have described a very fast algorithm for doing such a string-search, their algorithm is sublinear, so on the average it needs fewer than i character comparisons, to find an occurrence at index i. Here we suggest a problem-heap implementation of this algorithm which reduces the average running time further by searching a number of substrings simultaneously.

4.1 Problem-heap Formulation of String-search

The overall task to be performed is searching the string, S, which may be split into substrings (problems). Each process searches the substrings it receives using the Boyer-Moore algorithm. The details about this algorithm may be found in [Boyer and Moore 1977]. Each of the processes looks as follows:

```
CYCLE
    receive (substring);
    search (substring); »using the Boyer-Moore algorithm«
    IF match
        THEN report (position);
END
```

The substrings that remain to be searched are represented in a data structure shared by all processes called the *problem-heap*.

4.1.1 The Problem-heap

The problem-heap for the string-searching problem can be organized in a number of different ways; the one described below is quite simple, but it turns out to perform very well, therefore other more complex organizations of the problem-heap have not been tried.

The most obvious way to distribute a string of length L to N processes is by dividing it into N substrings of equal length and then let each process search through one of these as shown in fig. 4.1.

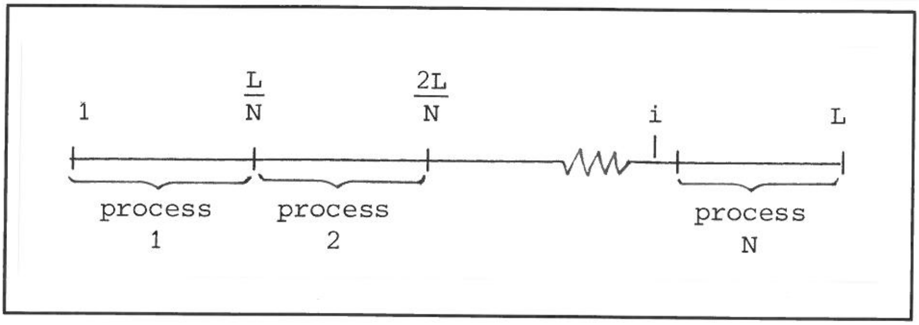


Fig. 4.1

But since the leftmost occurrence of the pattern is not known in advance, this organization is very inefficient; the work done by all the processes working to the right of i is lost. Furthermore long substrings give a large starvation and braking loss. Instead relatively short substrings are given to processes. Fig. 4.2 illustrates a typical snapshot of a search:

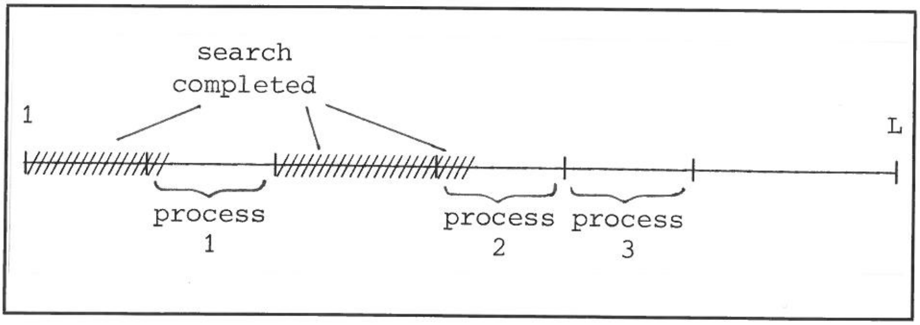


Fig. 4.2

The details of the string administration, that is keeping track of which strings have been given to which processes, can be worked out in many different ways. It turns out that the following simple administration performs very well. All substrings are of the same fixed length, D, where $D \geq K$. Processes are always allocated substrings from left to right, hence there is a point, F, up until which the string has been searched or is being searched, see fig. 4.3.

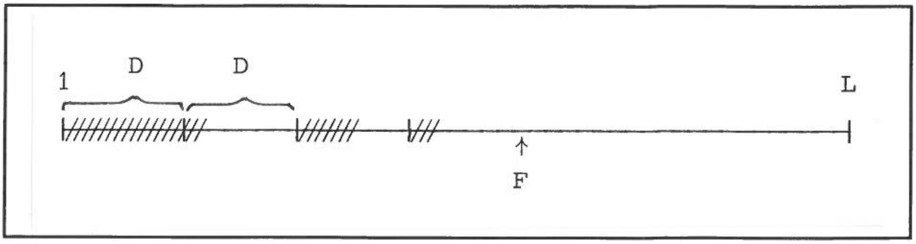


Fig. 4.3

The obvious question is of course, which *D* should be chosen? The optimal *D* depends on:

- the pattern to be found
- the position of the leftmost occurrence
- the number of processes
- the string to be searched.

Hence the optimal value of *D* cannot be computed before the search. Fortunately the running time is not sensitive to the exact choice of *D*. Fig. 4.4 shows the relationship between the measured average running time *T* and the value of *D* (for a particular choice of the above parameters):

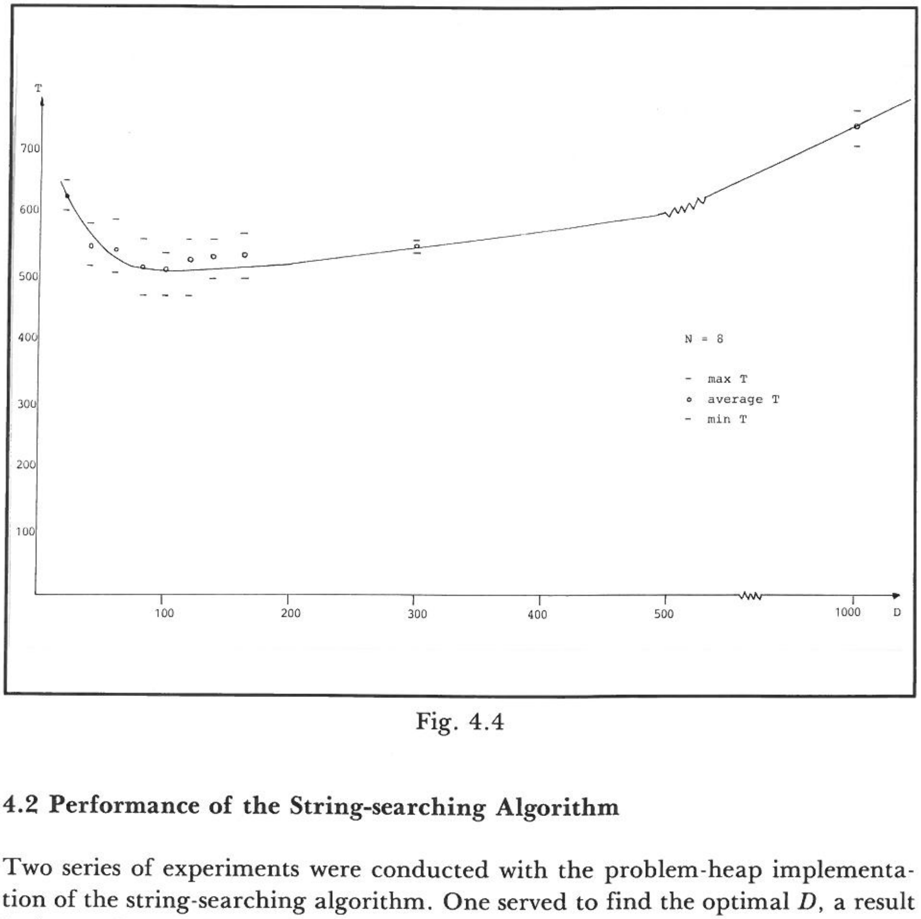


Fig. 4.4

4.2 Performance of the String-searching Algorithm

Two series of experiments were conducted with the problem-heap implementation of the string-searching algorithm. One served to find the optimal D, a result is shown above. The other served to analyze the speed-up of the algorithm.

The table shown in fig. 4.5 contains the measured speed-up for the above described implementation of the string-search. As mentioned in section 3.3 the speed-up can be defined in different ways. In the table shown above $T(1)$ denotes the execution time of the sequential algorithm where the string is not divided into substrings. $T(N)$ denotes the time to execute the multiprocessor heap algorithm with N processors, using the optimal value of D corresponding to that N. The string-searching is a little unstable, see section 3.3, so $T(N)$ includes the time to do some superfluous work.

N	Running time	Speed-up
1	3597	1.0
2	1883	1.9
4	981	3.7
8	516	7.0

Fig. 4.5

Fig. 4.5 shows that the algorithm has a good speed-up even with the very simple string administration described above. There is, however, a significant deviation from a linear speed-up. In the following section the causes for this deviation are identified. These causes are viewed as different kinds of lost processing power as discussed in chapter 3.

4.2.1 Starvation Loss and Superfluous Work

Consider a process, *M*, which finds an occurrence of the pattern; other processes may already be working on substrings to the right of *M*, see fig. 4.6.

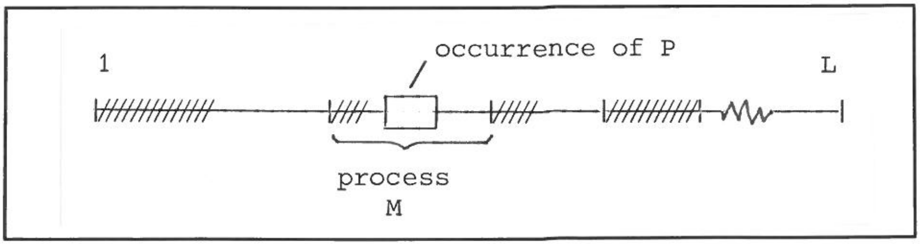


Fig. 4.6

The work done to the right of the occurrence is superfluous. Consider now the processes working to the left of *M*, these should finish the search of the substrings they are currently working on, otherwise an occurrence before the one found by *M* could be missed. The following three time instances are central to the analysis of the superfluous work and the loss:

T_1: the instance when the first substring to the right of the leftmost occurrence is given to some process.

T_2: the instance when a process finds the leftmost occurrence.

T_3: the instance when all processes have been stopped.

Those processes that finish searching a substring during $[T_1, T_2]$ are immediately put to work on a new substring to the right of the occurrence, hence each of them loses a whole period, which is the time it takes to search a substring. On the average this should be $(N - 1)/2$ of the processes. Now consider those processes that finish a search during $[T_2, T_3]$; they are not given a new substring to search, yet their processing power is lost until T_3. The diagram in fig. 4.6 illustrates this situation:

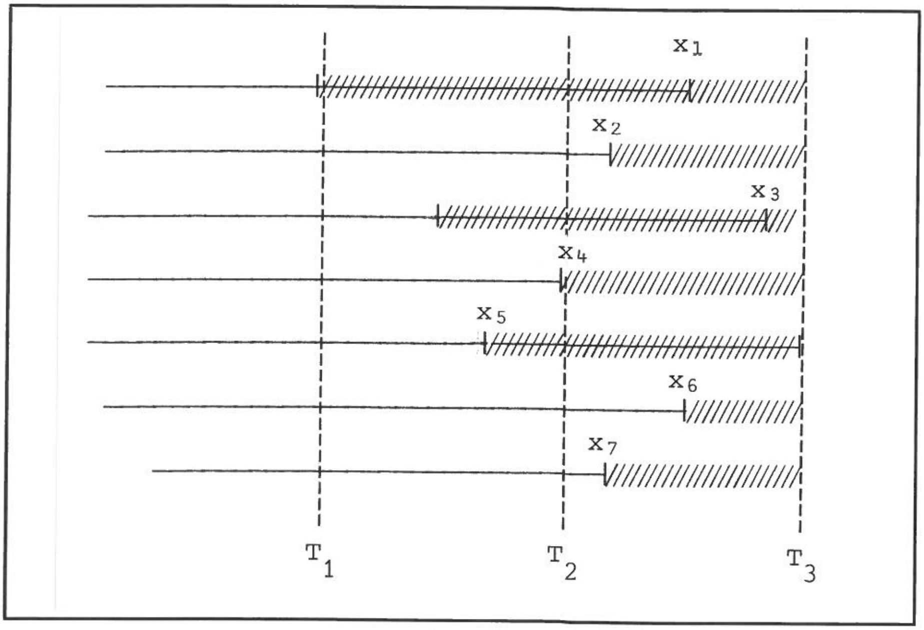


Fig. 4.7

Those sections marked by ~~//////~~ represent superfluous work and starvation loss is shown as //////.

Above it was argued that on the average the $(N - 1)/2$ processes lose a whole period, i.e. the periods marked by ~~//////~~, this amounts to $(N - 1)/2Dc$, where c is a constant. This is an estimate of the superfluous work.

To estimate the starvation loss consider the time instance T_2. The $N - 1$ remaining processes are busy searching some substring. Let the instances when they finish be $X_1, X_2, \ldots, X_n$ ($X_i = T_2$ where i is the number of the process finding the occurrence). The starvation loss is:

$$S_{loss} = \sum_{i=1}^{N} max\{X_j\} - X_i$$

To estimate the average value of S_{loss} we must find the average of $max\{X_j\} - X_i$. Let $(i_1, i_2, \ldots, i_N)$ be a permutation of $1, 2, \ldots, N$ such that:

$$X_{i_1} \leq X_{i_2} \leq \ldots \leq X_{i_N}$$

then the average we are seeking is the average:

$$X_{i_N} - X_{i_j} \qquad (0 < J < N)$$

If we assume that the X_j are uniformly distributed on $[0, Dc]$, then the average can be found using order statistics [Feller 1970]:

$$E(X_{i_N} - X_{i_J}) = (N - J)\frac{1}{N}Dc$$

Hence

$$\begin{aligned} S_{loss} &= \sum_{i=1}^{N} (N - i)\frac{1}{N} Dc \\ &= \sum_{i=1}^{N-1} i\frac{1}{N} Dc \\ &= \frac{N(N-1)}{2N} Dc \end{aligned}$$

So the total loss is:

$$\begin{aligned} B &= \text{superfluous work} + \text{starvation loss} \\ B &= \tfrac{(N-1)}{2} Dc + (\tfrac{N-1}{2})Dc \\ &= (N-1)Dc \end{aligned}$$

Some experiments were conducted to measure the actual loss. Fig. 4.8 shows the experimentally measured values against the predicted values for B.

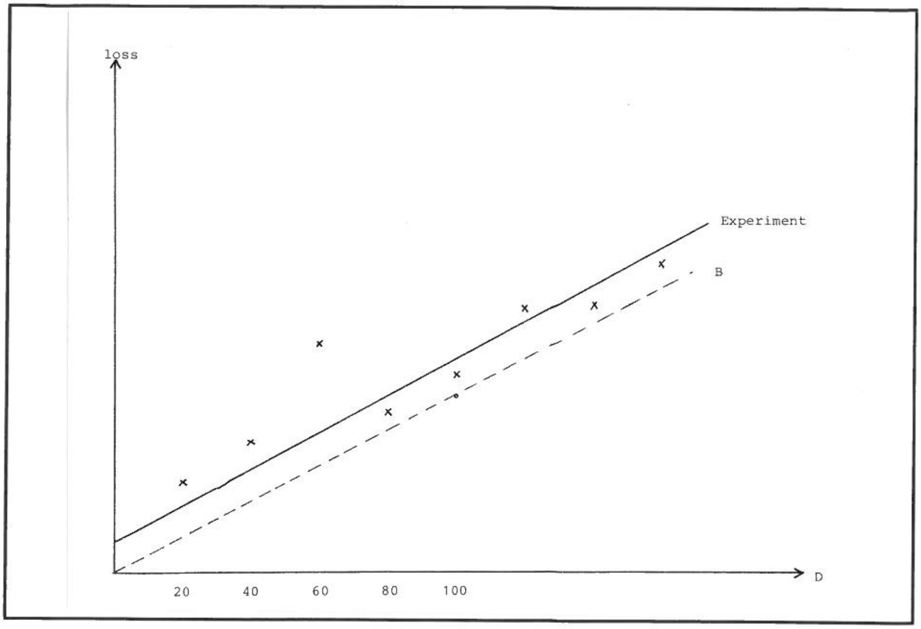


Fig. 4.8

4.2.2 Iteration Loss

For small values of D another kind of loss called *iteration loss* becomes significant. When a process finishes searching a substring and gets a new substring from the problem-heap some processing power is lost, partly because of the overhead associated with the string administration, but more significantly some momentum is lost when the jumps made through the string are cut.

The iteration loss is directly proportional to the number of iterations made by each process. Assume that the leftmost occurrence of the pattern starts at index i, then the average number of iterations is:

$$\frac{i}{D}$$

If we assume that the loss associated with each request is a constant, O, the total iteration loss is:

$$I_{loss} = O\frac{i}{D}$$

The experiments show that averaging over a large number of searches, the above is a reasonable approximation of the iteration loss, but if we consider a particular string and pattern, O is by no means constant. As mentioned above the lost momentum is the major cause for the iteration loss. Consider two consecutive values of D, d and $d + 1$. The running time of the algorithm when executed with these two values of D may be quite different. This is because the jumps through the string made by the Boyer-Moore algorithm are not the same. To see why this variation occurs, it is necessary to take a closer look at the Boyer-Moore algorithm:

The key idea in Boyer and Moore's algorithm is to compare the pattern and the string from the end of the pattern towards the start, as shown in fig. 4.9.

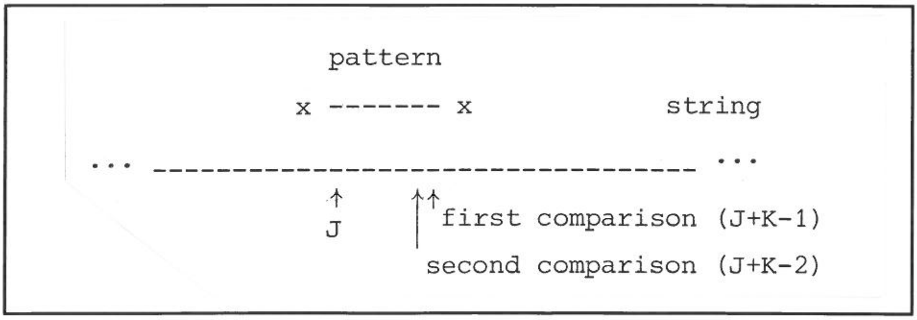


Fig. 4.9

To check for an occurrence of the pattern at index J, one compares the characters in the pattern with the string starting at index $J + K - 1$, if this matches, the characters at index $J + K - 2$ is compared with pattern $(K - 1)$ etc. If all K characters match, an occurrence of the pattern has been found, but if they do not, the pattern is moved to the right to check for an occurrence there. Because the comparison is done from the last character, the pattern can usually be moved quite far to the right. If, for example, the character at index $J + K - 1$ does not occur in the pattern, it can immediately be moved K characters to the right. In general, the algorithm makes use of all the information gained by the comparisons to move the pattern 1 to K characters to the right. The obvious algorithm which compares the characters from the start can only move the pattern one character to the right. Further details about the algorithm may be found in the references [Boyer and Moore 1977], [Galil 1979], and [Horspool 1980] . So the time it takes to search a particular substring may vary. If the pattern can be moved K characters after each comparison, the search goes very fast; as the other extreme, the pattern is sometimes only moved one character to the right, this occurs when the string is very regular with many repetitions, e.g. when the alphabet is very small [Galil 1979] .

Since it would be quite complicated to cover this variation in the expression for the iteration loss, we make the somewhat crude assumption that O is constant.

4.3 Running Time Estimation

It is claimed that B and I_{loss} explain the deviation from linear speed-up (see the table in section 4.2), i.e. that

$$T(N) \approx (T(1) + B + I_{loss})/N$$

To verify that no kind of loss has been overlooked, one may compute a predicted $T(N)$ using the above expression and compare this with the experimentally found $T(N)$. This is shown in fig. 4.10.

N	$\frac{T(1)}{N}$	$\frac{B}{N}$	$\frac{I_{loss}}{N}$	Predicted $T(N)$	Measured $T(N)$
4	899	54	25	978	981
8	450	31	25	506	516

For $N = 8, D = 100$ and $N = 4, D = 200$ is used.

Fig. 4.10

The implementation and analysis of the string-searching algorithm were done before we had clarified the distinction between the various kinds of loss made in chapter 3. In earlier descriptions [Møller-Nielsen and Staunstrup 1984] of the string-searching algorithm we used the term braking loss differently from our current use.

5. Adaptive Quadrature Algorithm

This chapter contains a summary of a series of experiments with an adaptive quadrature algorithm by Rice [Rice 1976] . The first section describes the quadrature algorithm as a problem-heap algorithm, and the performance of a version of the algorithm is analyzed using the four types of losses. This analysis shows that starvation loss is dominant, but we are very close to getting saturation loss. The second section focuses on some techniques, that can alleviate the effect of saturation loss. A version of the algorithm from the first section is used to illustrate the techniques. These techniques rather than their particular effect on the quadrature algorithm are the focal point of section 5.2.

5.1 Performance Analysis of an Adaptive Quadrature Algorithm

5.1.1 Adaptive Quadrature as a Problem-heap Algorithm

Adaptive quadrature algorithms are simple examples of problem-heap algorithms for the following reason:

Suppose that $Q(a, b)$ is an approximation to the integral of f on the interval from a to b, and $B(a, b)$ is a bound on the error of this approximation, so that:

$$| \int_a^b f - Q(a, b) | \leq B(a, b)$$

Then the solution (i.e. $Q(a, b)$ and $B(a, b)$) to the problem $\int_a^b$ can be obtained from the solutions to the two subproblems $\int_a^c f$ and $\int_c^b f$ by simply adding the solutions of the subproblems, since:

$$| \int_a^b f - (Q(a, c) + Q(c, b)) | \leq B(a, c) + B(c, b)$$

Suppose that the initial problem is to calculate $\int_0^1 f$ to the accuracy $Eps(0, 1)$, i.e. to find corresponding values $Q(0, 1)$ and $B(0, 1)$, so that $B(0, 1) \leq Eps(0, 1)$. The skeleton program for a process could then be the following:

```
CYCLE
    Take a problem (specified by the triple:
       [$a, b, Eps(a, b)$]) from the problem-heap;
    Calculate $Q(a, b)$ and $B(a, b)$;
    If $B(a, b) \leq Eps(a, b)$ then
        The problem is classified as »simple«, and $Q(a, b)$ and $B(a, b)$
        are put in the »result-accumulator«
    ELSE
    BEGIN
        Values $c, Eps(a, c)$ and $Eps(c, b)$ are chosen so that
        $Eps(a, c) + Eps(c, b) = Eps(a, b)$;
        Put the problems $[a, c, Eps(a, c)]$ and $[c, b, Eps(c, b)]$
        in the problem-heap
    END
END;
```

Algorithm 5.1

The initial problem is $[0, 1, Eps(0, 1)]$. When the problem-heap is empty and all the processes are idle, the accumulated values of Q and B form a solution to the initial problem since:

$$| \int_0^1 f - \Sigma Q(a, b) | \leq \Sigma B(a, b) \leq \Sigma Eps(a, b) = Eps(0, 1)$$

In this study, the class of functions f that can be handled, is restricted to monotonic functions with monotonic first derivatives. This restriction simplifies the calculation of B considerably. The reader is referred to [Rice 1976] for more details on this and other aspects of the actual algorithm.

5.1.2 A Model for the Performance of the Algorithm

The measurements of the total execution time of the quadrature algorithm are analyzed according to the formula:

$$T = \frac{k \ \#_f}{N_A} \qquad (5.1)$$

This formula expresses that the running time is equal to the amount of work to be done ($k \quad \#_f$), namely the time a single processor would use, divided by the average number of active processes ($\overline{N}_A$). The usefulness of the formula depends on whether simple models can be used for predicting the values of k, $\#_f$ and $\overline{N}_A$.

The quantities in the formula are defined as follows:

T:
The total running time for a given initial problem, i.e. the elapsed time from the initial problem is put into the problem-heap until the solution (to the required precision) is available in the result-accumulator. This time periode will be referred to as the *solution period.*

k:
The average running time for one cycle of a slave (in algorithm 5.1). It includes the time spent manipulating the problem-heap and waiting for access to the problem-heap.

$\#_f$:
The number of problems taken from the problem-heap and successfully completed during the solution period. $\#_f$ may be interpreted as the total amount of work done by all the processes.

$\overline{N}_A$:
The average number of active processes. A process is inactive, when it is idle, because the heap is (temporarily) empty. The average is taken over the solution period.

In the next section we give models for predicting the quantities $\#_f$, N_A and k.

Models for the quantities in the formula

k:
If the time to evaluate f is nearly the same for all points in the interval, then for $N = 1$ (one process) the running time for a cycle is almost constant. This is because a cycle includes just one evaluation of the function f and the running time for a cycle is dominated by the evaluation of Q and B. When $N > 1$, k is a function of N, $k(N)$, which includes the saturation loss, i.e. the time which is spent waiting for access to the problem-heap, because another process has exclusive access to it. The following simple model can be used to describe the saturation loss.

The execution of one process can be described on a time axis as shown in fig. 5.1.

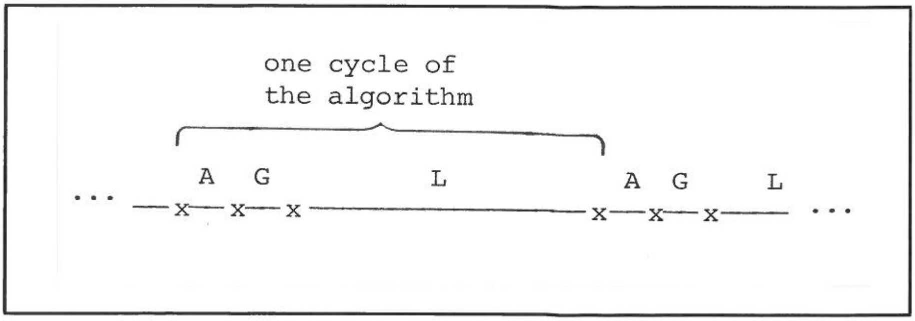


Fig. 5.1

This is similar to the model for the bus usage in chapter 2.

A: is the time spent waiting for access to the problem-heap (i.e. the critical region).

G: is the time spent manipulating the problem-heap (i.e. execution within the critical region). *G* depends only on the complexity of storing and retrieving subproblems and the complexity of accumulating the results *Q* and *B*.

L: is the time spent on local processing of a problem in the process. This part of the processing does not involve the problem-heap. *L* depends only on the complexity of *f*.

If *G* and *L* are both nearly constant for all subproblems and all processes, then *A* will stay negligible as long as the number of processes, *N*, is less than N^*:

$$N^* = 1 + L/G \tag{5.2}$$

This can be seen from fig. 5.2 (see also chapter 2).

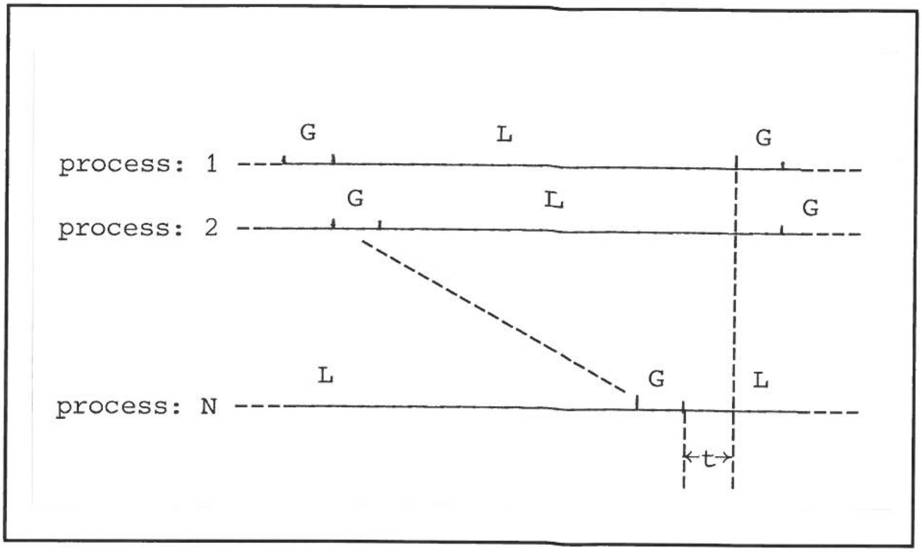


Fig. 5.2

Saturation happens, when t becomes zero, i.e. when

$$(N - 1) \times G = L$$

or

$$N = 1 + \frac{L}{G}$$

When $N > N^*$, the system is saturated and A will increase with increasing N, i.e.

$$A = (N - N^*)G$$

This implies that the speed-up is linear in N when $N < < N^*$, but constant when $N > > N^*$. So, adding more processes, in order to reduce the total running time for a given initial problem, will have no effect when the number of processes is increased beyond N^*; the problem-heap has become saturated.

On the diagram in fig. 5.3, the curve (1) shows the expected dependence of $k(N)$ on N, when G and L are the same for all subproblems. (2) shows the expected dependence when G and L are both drawn randomly and independently from negative-exponential distributions with mean-values equal to the values of G and L in (1).

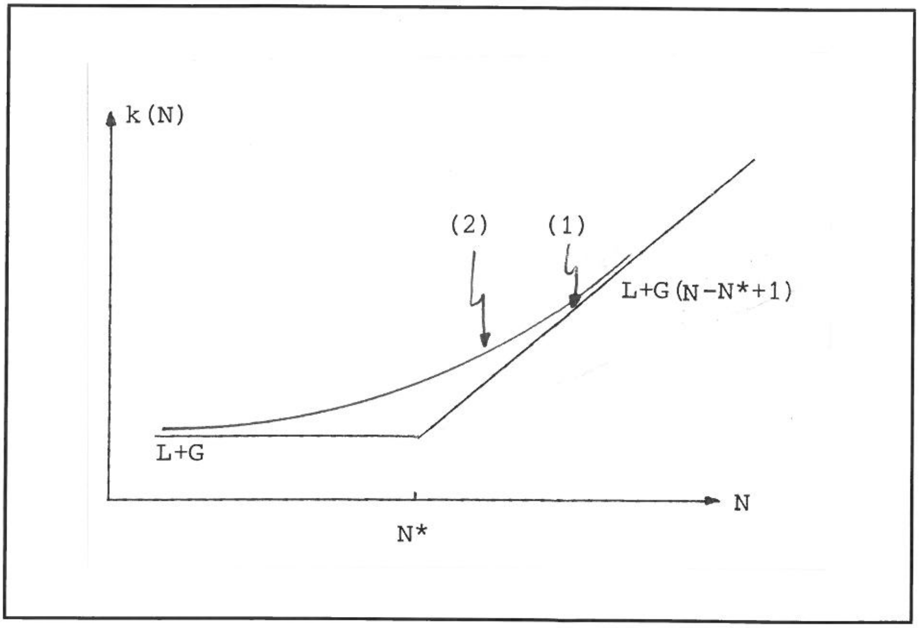


Fig. 5.3

(2) is calculated using the following queueing model shown in fig. 5.4.

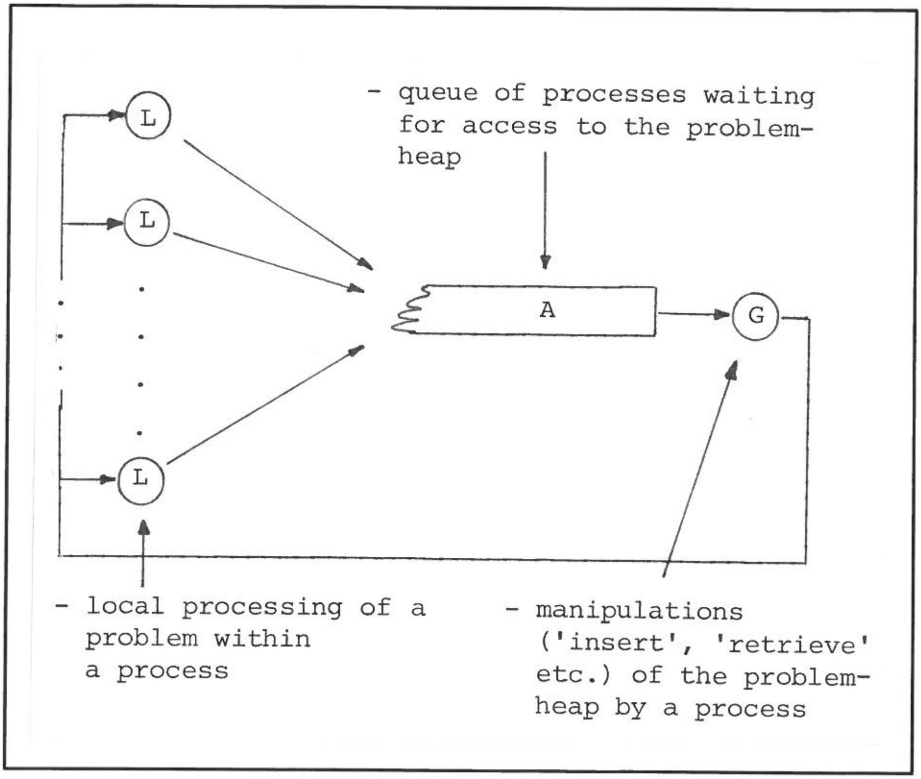

- queue of processes waiting for access to the problem-heap
L
L
L
A
G
- local processing of a problem within a process
- manipulations ('insert', 'retrieve' etc.) of the problem-heap by a process

Fig. 5.4

This model is identical to the traditional closed queueing model for a system with N terminals and a single, central processor (see [Brinch Hansen 1973 p. 217]).

In conclusion k is estimated by measuring the values of L and G using a single process. L depends on the time to evaluate Q, B and f, and G depends on the time to execute the problem-heap manipulations. The relationship between L and G determines how k is estimated when $N > 1$. If the heap is not saturated ($N << 1 + \frac{L}{G}$), k is simply $L + G$. On the other hand if the heap is saturated a simple queueing model is used to calculate k.

$\#_f$:

$\#_f$ is independent of the number of processes. It depends on the behaviour (e.g. variation) of the function f in the interval of integration and *Eps*, which is the required accuracy. For fixed f and interval, $\#_f$ and *Eps* are related as follows:

$$Eps \times (\#_f)^2 \simeq constant \qquad \text{[Rice 1976]}$$

This means that the constant can be calculated by measuring $\#_f$ for one particular value of *Eps* (probably a large value of *Eps*). For other values of *Eps* the number of function evaluations $\#_f$ can now be estimated from the expression:

$$\#_f \approx \sqrt{\frac{constant}{Eps}}$$

$\overline{N}_A$:

$\overline{N}_A$ reflects the starvation loss, which is the difference between N and $\overline{N}_A$. A model for $\overline{N}_A$ can be constructed as follows:

Assume that the only loss of processing power is due to a lack of problems during the first part of the solution period. When the initial problem is inserted into the problem heap one process becomes active. After one cycle in the active process, two processes become active, then four processes, and so on until all processes are active. The model assumes that all processes remain active until the end of the solution period. For $N = 8$, $N_A(t)$ is modelled as shown in fig. 5.5. The unit of the abscissa is the running time k for one cycle of a process. The model assumes that the variation of k from problem to problem is relatively small and that k is independent of N.

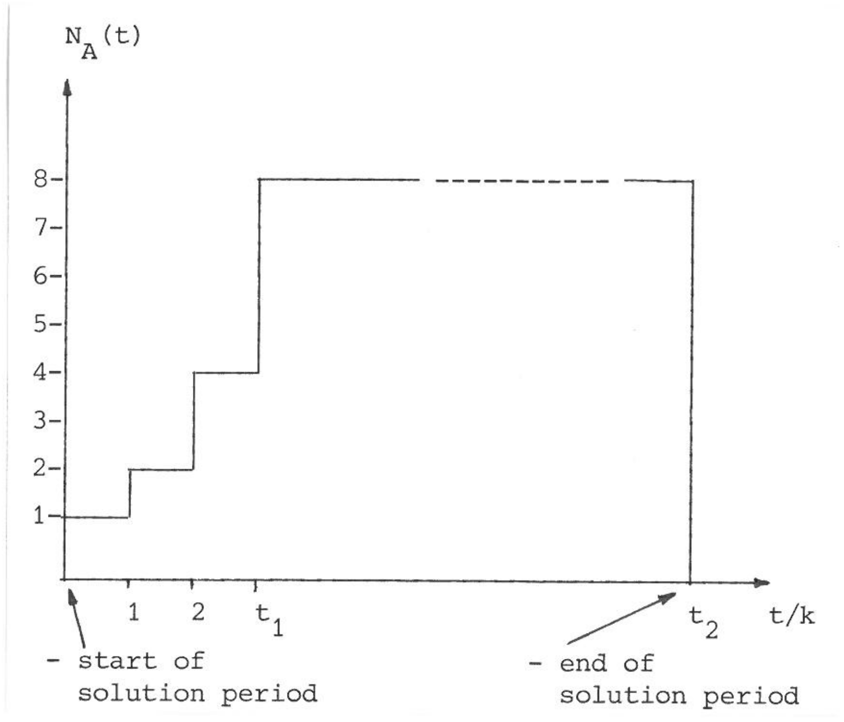


Fig. 5.5

Using this model $\overline{N}_A$ may be calculated as follows:

The area under the graph for N_A is equal to $\#_f$ so:

$$\#_f = a(t_1) + N(t_2 - t_1)$$

where $a(t_1)$ is the area under the graph from 0 to t_1.

By definition we have

$$\overline{N}_A = \frac{\#_f}{t_2}$$

t_1 and $a(t_1)$ depend only on N, and they can be read from the graph for the actual value of N.

$\overline{N}_A$ can then be calculated by the formula:

$$\overline{N}_A = \frac{N \ \#_f}{\#_f + Nt_1 - a(t_1)} \tag{5.3}$$

The reason why the model for $\overline{N}_A$ does not include any starvation towards the end of the solution period (indicated by the vertical line in the graph at t_2) is the following. Normally, a heap-algorithm terminates when the heap is empty, and all processes are idle (i.e. waiting for a problem). This adaptive quadrature algorithm uses a technique which allows a final result, that is a $Q(0, 1)$ and a corresponding $B(0, 1) \leq Eps(0, 1)$, to be obtained long before the heap is empty. The technique is that the difference between ΣB and the corresponding ΣEps can be used to decrease the required precision for the remaining non-simple problems. The application of this technique in connection with an adaptive quadrature algorithm may be found in [Rice 1976] . When this technique is applied, there is no starvation towards the end of the solution period. These two termination strategies are called termination on *empty heap* and termination on *sufficient accuracy* respectively.

Other sources of loss

When the algorithm terminates on empty heap, then there is no superfluous work but a starvation loss towards the end of the solution period. When the algorithm terminates on sufficient accuracy, then there is a small amount of superfluous work, which may be estimated as follows: At t_2 (see fig. 5.5), when a process detects that a sufficient accuracy has been reached, $N - 1$ processes are on the average half-way through a cycle and the results from these $N - 1$ cycles are not used. The amount of superfluous work therefore corresponds to an increase in $\#_f$ by $N - 1$; $(N - 1)/2$ stems from superfluous work done before t_2 and $(N - 1)/2$ stems from braking loss after t_2. The distance from t_2 (which is the running time T in formula (5.1) to the termination time of the last process can be estimated as for the string-searching algorithm in chapter 4.

The notion of iteration loss is meaningless for this adaptive quadrature algorithm, because the algorithm does not include a parameter which specifies - a priori - the boundary between simple and non-simple problems (see chapter 3). An iteration loss can be identified in the experimental results for an adaptive quadrature algorithm reported in [Grit 1983] .

5.1.3 Comparison between Measurements and the Model

Below we summarize the results from a series of experiments. A more detailed report on these experiments can be found in [Møller-Nielsen and Staunstrup 1982].

For the two values $Eps = 10^{-4}$ and $Eps = 10^{-5}$ and for the number of processes varying from 1 to 8, we measured T, $\overline{N}_A$ and $\#_f$.

$\#_f$:
$\#_f$ was constant 127 for $Eps = 10^{-4}$, and varied from 452 to 468 for $Eps = 10^{-5}$.

$\overline{N}_A$:
$\overline{N}_A$ is shown in fig. 5.6, and the measured values agree (within few percent) with the predictions based on the model described above (formula 5.3).

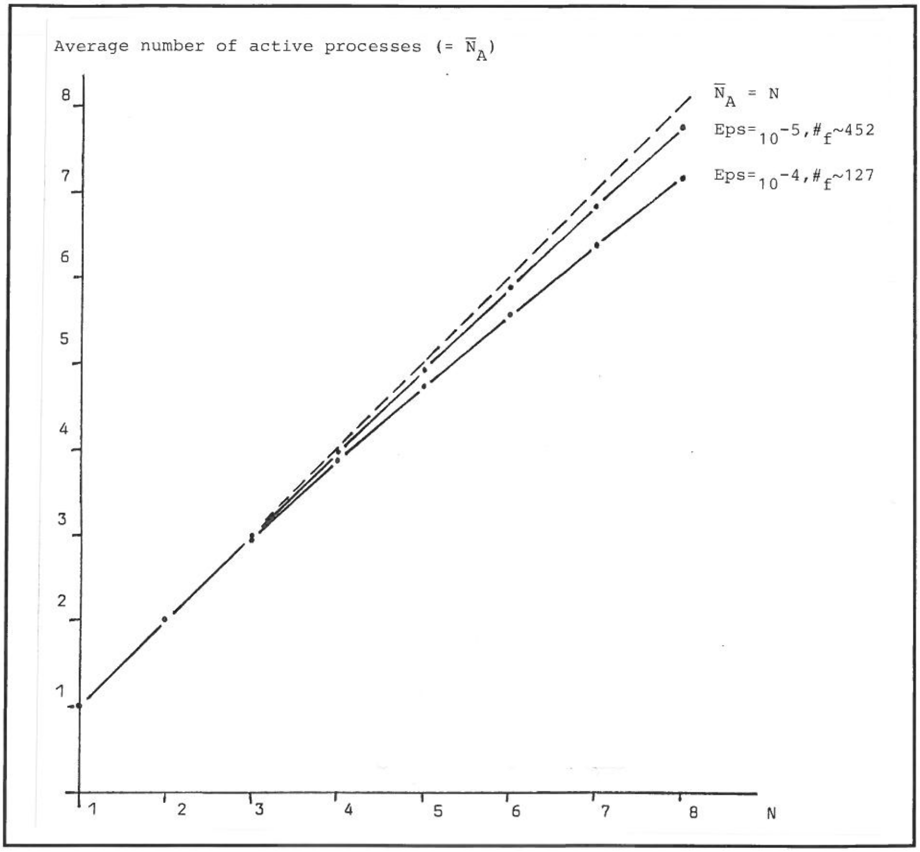


Fig. 5.6

T:
the measured values of *T* are shown in fig. 5.7 as the speed-up (i.e. $T(1)/T(N)$) for the values 1...8 of *N*.

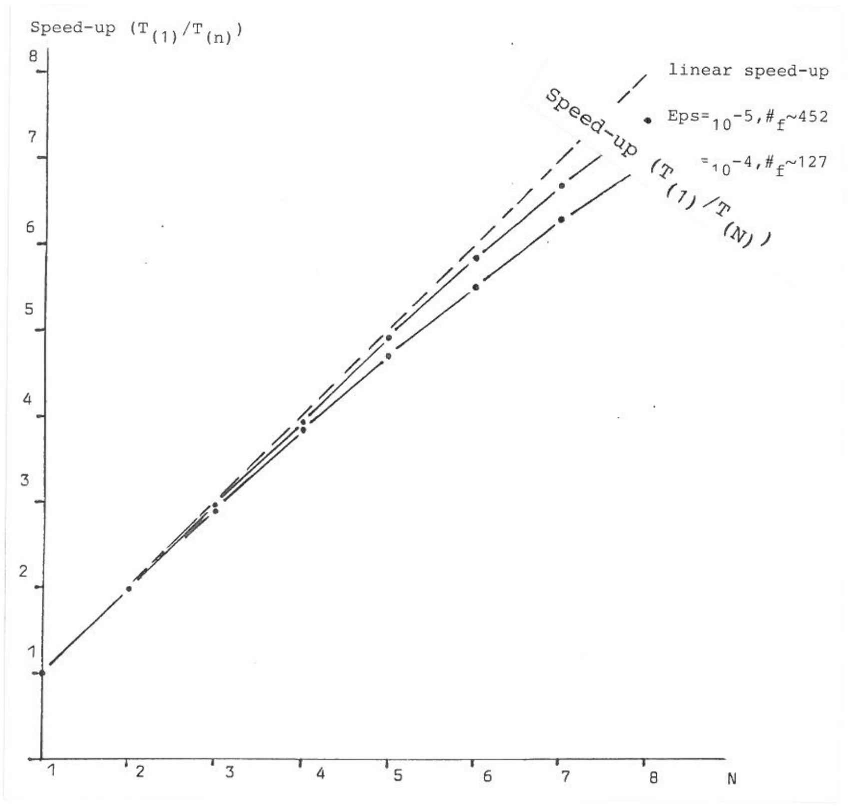


Fig. 5.7

Inserting the measured values into the formula (5.1) we find a value for k, which is independent of *Eps* and N. This implies that we expect the saturation point, N^*, to be larger than 8. The actual values for L and G were also measured. These measurements predict a value for N^* around 9. Measurements of L and G are not very accurate.

5.2 Some Techniques to Alleviate the Effect of Software Saturation

This section focuses on the problem of software saturation which can be a limiting factor, when trying to obtain linear speed-up. It presents some techniques which can be used to alleviate the effect of software saturation. All we can hope for is to postpone saturation, i.e. to increase the number of processes which can be used before the saturation becomes perceptible.

The effect of the techniques are illustrated by means of examples. For each of the presented techniques (or group of techniques), the technique is applied to the same basic algorithm, and the performance of the modified algorithm is measured and compared with the performance of the basic algorithm.

5.2.1 The Basic Algorithm

The basic algorithm is essentially the adaptive quadrature algorithm from section 5.1. Three changes have been made.

— The first change is to increase the value of G (see section 5.1) corresponding to the use of pseudo-double-precision accumulation of Q and/or B. The only reason for this change is to bring the saturation point down to around 4, so that the effect of a technique can be demonstrated with only 8 processors.

— The second change is to alter the termination algorithm so that termination happens on empty heap (see section 5.1.2). This change has been made in order to make all the techniques, discussed in this section, applicable to the basic algorithm.

— The third change is to use *busy-waiting* for medium-term scheduling. In terms of Concurrent Pascal it means that the type *queue* and the operators *delay* and *continue* are not used, neither in the basic algorithm, nor in any of the modified algorithms. This change has been made because most of the techniques presented in this section turned out to be much easier to formulate with busy-waiting than with delay/continue.

This report contains only a summary of the techniques and the measured effect on the performance of the modified algorithm. It is not obvious how the performance is influenced by the use of busy-waiting. This should be studied in more detail. Our conjecture is that for a large class of useful algorithms, performance is not impaired by the use of busy-waiting. We have reported (in section 8.9 and in [Møller-Nielsen and Staunstrup 1983]) about situations, in which the use of busy-waiting improved the performance. We believe, that the effect of busy-waiting on software saturation can be analyzed in the same way as the effect of busy-waiting on hardware saturation was analyzed in chapter 2, and with the same conclusions. But we still have too little experimental evidence.

5.2.2 The Techniques

Before the techniques are listed, we want to stress the following.

The list is probably not exhaustive. Depending on the basic algorithm some techniques are applicable and some are not, and the use of one technique may exclude the use of others. The application of some technique in order to decrease saturation loss may cause another type of loss to increase. For example: the termination algorithm described in section 5.1.2 (termination on sufficient accuracy) requires that a single problem-heap is used. If saturation is decreased by means of a technique, which splits the problem-heap into several heaps, the termination must be based on empty heaps and thereby increase the starvation loss.

There are two main groups of techniques:

(a) Change the values of L and G, so that the ratio $\frac{L}{G}$ is increased. This will increase the saturation point N^*.

(b) Split the problem-heap into several heaps and distribute the processes over the heaps in such a way, that the number of processes per heap is lower than the saturation point N^*.

5.2.2.1 Changing the ratio $\frac{L}{G}$

G can be decreased by reducing the number of accesses per subproblem to the heap. The cost is that L is increased — and often more than G is decreased. In fig. 5.8 is shown the performance of a program where this kind of technique has been applied. The basic algorithm is modified so that consecutive pairs of the operations *put-a-problem-into-heap* and *get-a-problem-from-heap* are avoided. This clearly decreases the average number of accesses per subproblem to the heap, but increases the average value of L, because of the extra manipulations of the subproblem retained in the processes.

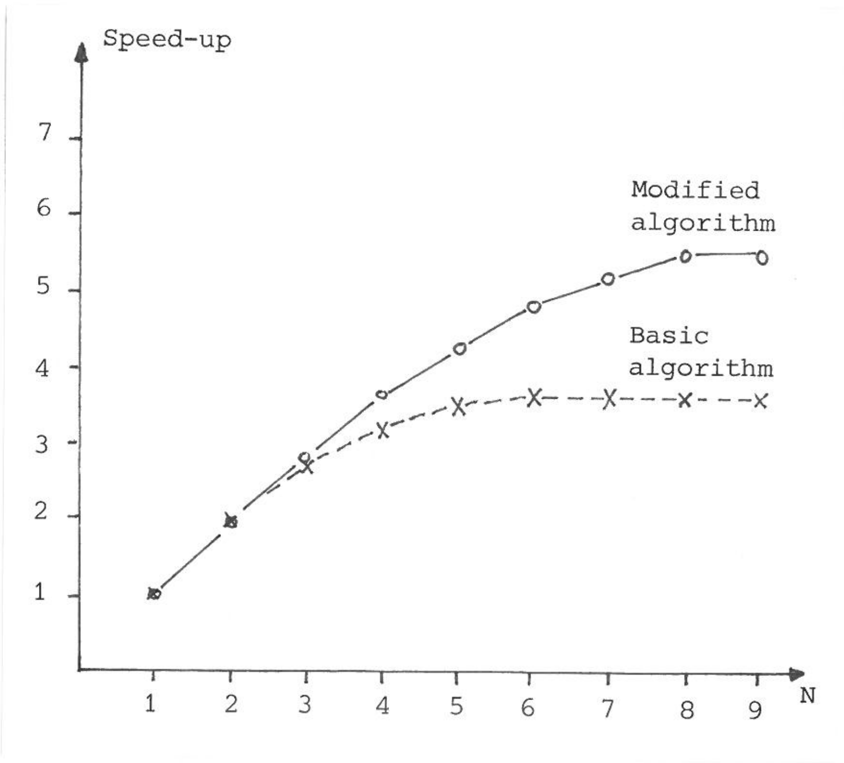


Fig. 5.8

A similar technique is to accumulate Q and B in the process and not in the problem-heap. This, however, will make it impossible to use termination on sufficient accuracy in the algorithm, and thereby increase the starvation loss.

5.2.2.2 Splitting of the Problem-heap

A single problem-heap can be split into several heaps in a number of ways. It can either be split into heaps so that the new heaps contain different subsets of the functions in the original heap, or it can be split into a number of duplicates of the original problem-heap, all containing the same functions as the original heap. Identical problem-heaps can be arranged in different structures, depending on how they are accessed by the processes. In this section two such structures are described: a hierarchy and a ring. Splitting the problem-heap into several heaps introduces a termination problem which is not present in the basic algorithm. A solution to this problem is discussed in section 5.2.3.

Different heaps

One way to split the problem-heap is to separate independent functions into separate heaps. The execution of one cycle in a process will then generate a few calls of functions in all the heaps rather than many calls to a single heap, as shown in fig. 5.9.

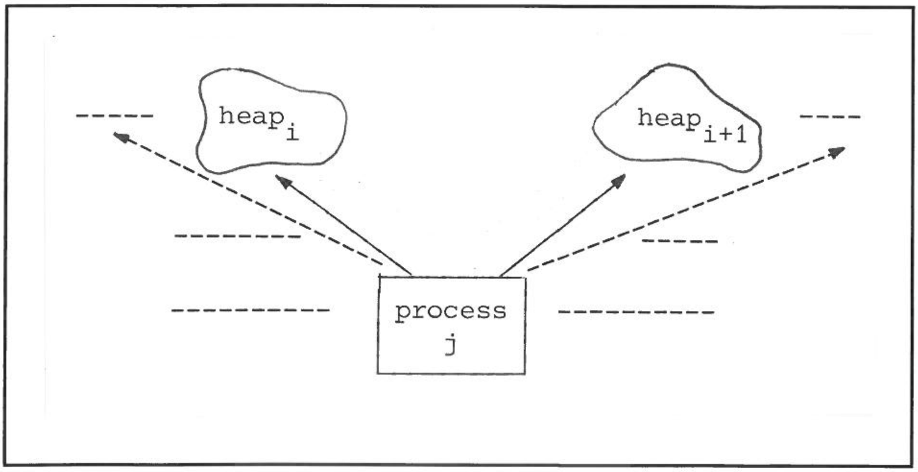


Fig. 5.9

If G_i is the time spent in $heap_i$, we expect that

$$G \simeq \Sigma_i G_i$$

and

$$N^* \approx 1 + \frac{L}{\max_i\{G_i\}}$$

This technique can be applied to the basic algorithm by introducing two heaps, one heap for storing the subproblems, and one heap for accumulating Q and B.

This is shown in fig. 5.10.

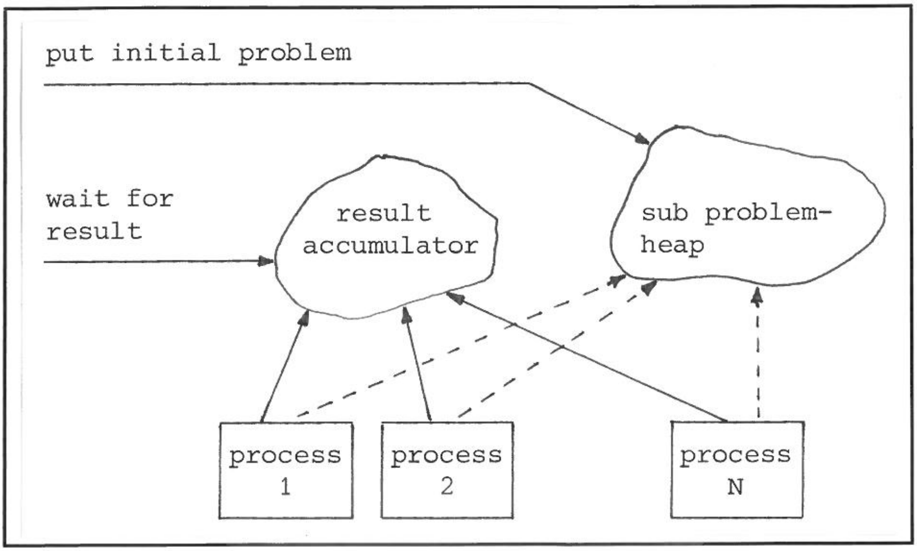


Fig. 5.10

Note, that since Q and B are still accumulated in a single heap, the termination on sufficient accuracy can be used. In fig. 5.11 is shown the performance of a program, where this kind of splitting has been applied to the basic algorithm.

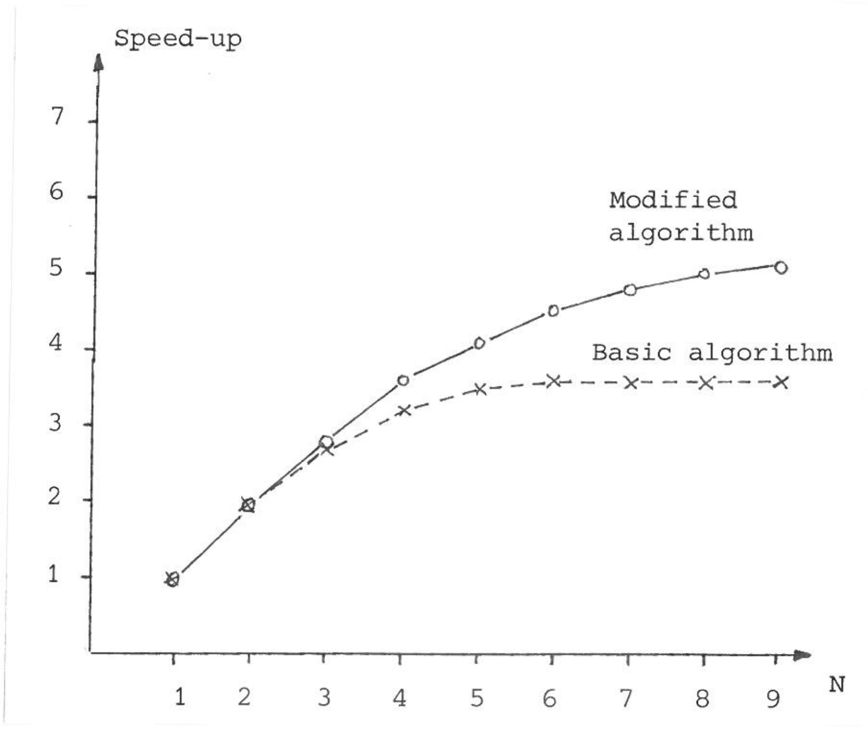

Speed-up
7
6
5
4
3
2
1
Modified algorithm
Basic algorithm
1
2
3
4
5
6
7
8
9
N

Fig. 5.11

Identical heaps

Another way to split the problem-heap is to introduce multiple copies of the problem-heap and distribute the processes over the heaps in such a way that the number of processes per heap stays below the saturation point. The performance of the structure shown in fig. 5.12 has been measured.

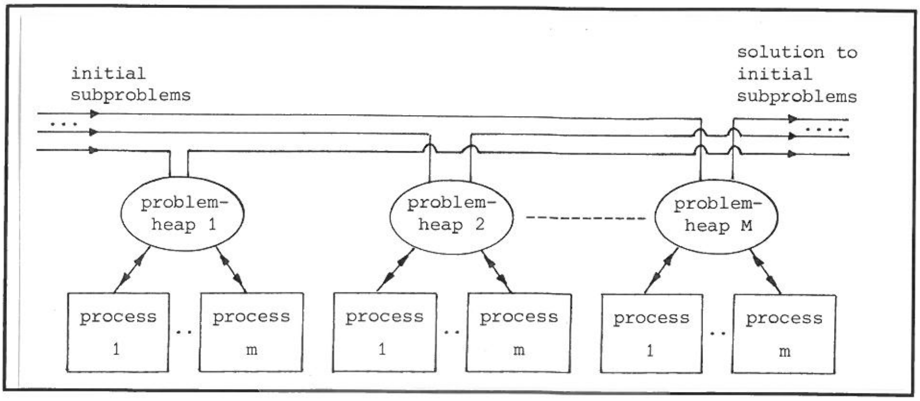
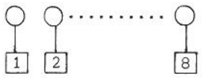
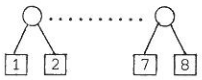
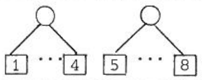
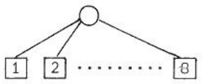


Fig. 5.12

The initial problem is divided into *M* initial subproblems and an initial sub-problem is transferred to each problem-heap. From then on, a subsystem (a problem-heap and the *m* associated processes) computes the solution to its initial subproblem independently of all other subsystems. The computation terminates when the last heap becomes empty and the associated processes become idle. All subsystems are identical, and all processors execute the same process. Note, that although the termination on sufficient accuracy can be used for each subsystem, it cannot be used to terminate the total system.

This technique has been applied to the basic algorithm, and each subsystem is identical to the basic algorithm. The table in fig. 5.13 shows some measured values for four different choices of *m* and *M* with a total of eight processes.

					speed-up	
M	m		$\max_{1 \leq i \leq M} T_i$ *)	$\min_{1 \leq i \leq M} T_i$	measured	expectation based on random variation of the size of the initial subproblems
8	1		185	13	1.8	3.0
4	2		117	16	2.9	3.7
2	4		89	23	3.9	4.1
1	8		95	95	3.6	3.6

*) T_i is the computation time for the i'th subsystem.

Fig. 5.13

The arrangements differ only with respect to the number of independent problem-heaps. The division of the initial quadrature problem into M initial sub-problems is done by dividing the interval of the quadrature into M subintervals of equal length.

The figures in the table illustrate the dilemma. For $M = 8$, processing power is lost because the initial subproblems require very different execution times to solve and therefore there is starvation towards the end of the solution period. The largest initial subproblem produces 103 subproblems (corresponding to a total execution time of 185 sec.), while the smallest initial subproblem produced only 7 subproblems. Note, that in general it is not possible to divide the initial problem into initial subproblems of equal difficulty. A comparison of the two numbers under the heading *speed-up* shows that the actual quadrature problem produces initial subproblems that are only a little more unbalanced than what would be expected if the size of the initial subproblems were drawn randomly from identical negative exponential distributions, so the actual quadrature problem is not pathological (the negative exponential distribution was chosen for simplicity). For $M = 4$ and 2 both the measured and the expected variation in the size of the initial subproblems is smaller, and since $m < N^*$ the problem-heaps are not saturated. Going from $M = 2$ to $M = 1$, saturation loss in increasing more than starvation loss is decreasing, so the resulting speed-up is decreasing. The dilemma can be summarized as follows:

— In order to avoid saturation of shared resources (e.g. a problem-heap) *decentralization should be aimed at*; therefore M should be large.

— In order to avoid starvation due to variation of the size of the initial subproblems, M should be as small as possible. That is, *centralization should be aimed at.*

Note, that the choice of termination algorithm (use of empty heap or sufficient accuracy) also depends on the degree of centralization/decentralization.

A compromise between centralization and decentralization of the problem-heap can be attempted in several ways. Below two such techniques are discussed. In both a connection is established between the subsystems. The load on these connections is balanced in order to minimize the sum of the saturation loss and the starvation loss.

A Hierarchy of Heaps
One compromise is a hierarchy of problem-heaps. This can be depicted as shown in fig. 5.14.

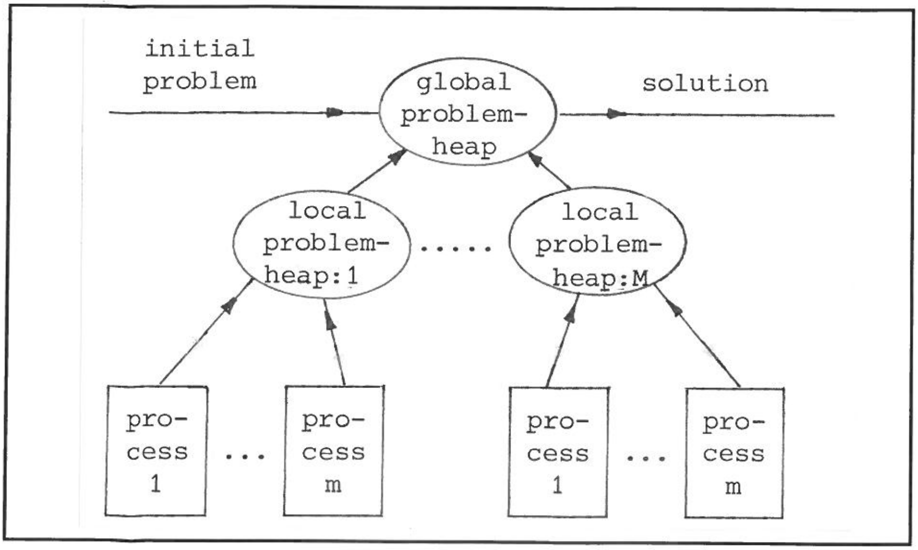


Fig. 5.14

A global problem-heap is added on top of a number of independent subsystems. Whenever the local problem-heap becomes empty, the request for a problem to solve is carried on to the global problem-heap. If the problem-heap is empty, the process waits for a problem to be inserted into the global problem-heap. This happens when a local problem-heap overflows. In this way problems can propagate from one local problem-heap to another, and no problem-heap (local or global) is saturated. However, processor power can still be lost, because a local problem-heap can contain problems while processors, associated with other local problem-heaps, are idle.

In fig. 5.15 is shown the performance of a program, which is obtained from the basic program by applying a hierarchy of problem-heaps. The local problem-heaps are slightly different from the global problem-heap which is identical to the problem-heap of the basic program, except for the termination algorithm, which is discussed in section 5.2.3.

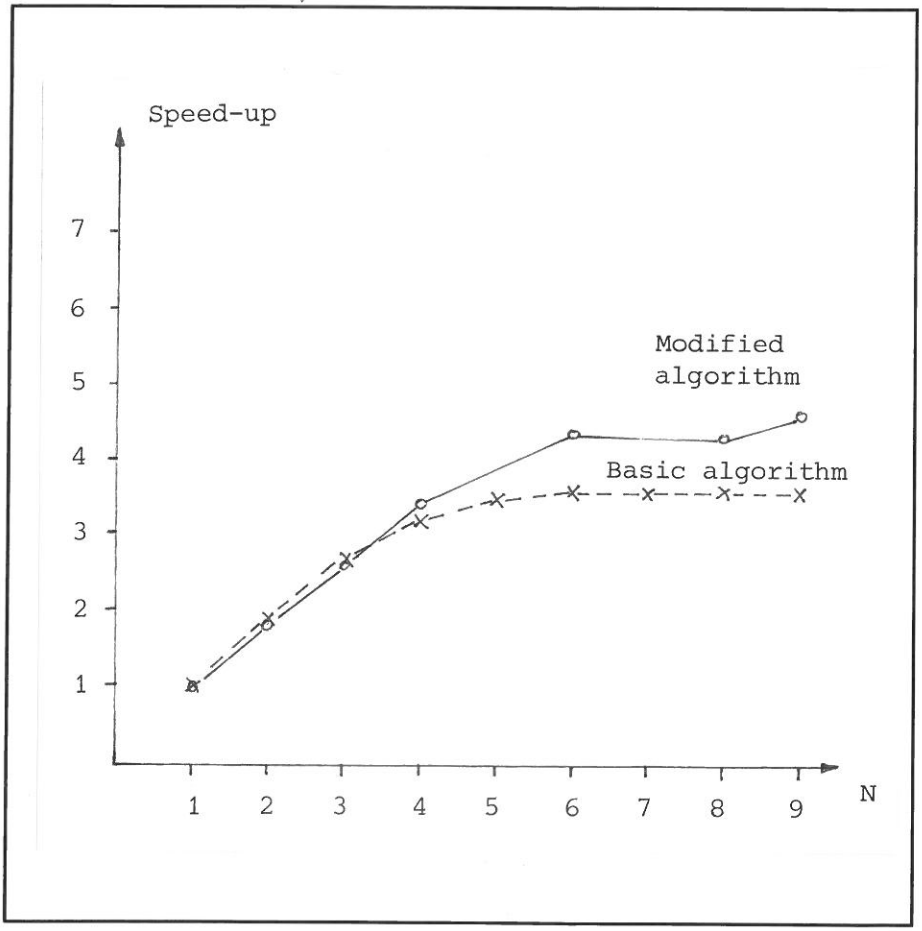


Fig. 5.15

A drawback of this technique is that the performance of the resulting program depends on the proper choice of a number of parameters like: *M*, *m*, the size of the local problem-heaps, the criteria used when selecting a problem from a heap etc. This proper choice depends on the basic algorithm to which the technique is applied, e.g. on the behaviour of the function *f*. Another compromise between centralization and decentralization is the ring of heaps described in the next section.

A Ring of Heaps

Another compromise is a ring structure of subsystems. This can be depicted as shown in fig. 5.16.

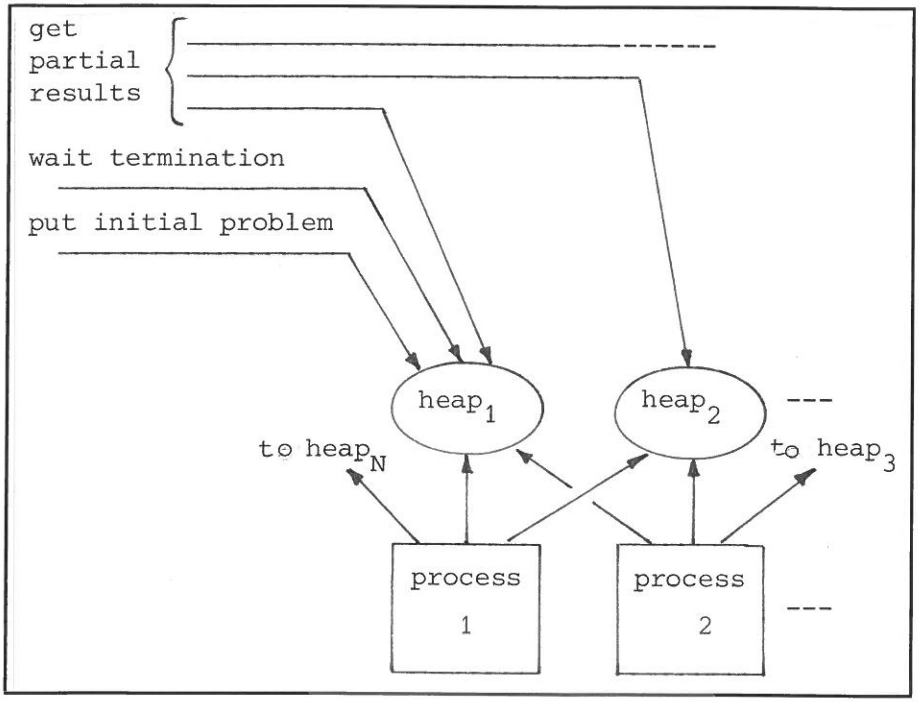


Fig. 5.16

All processes have access to all problem-heaps. Process no. *i* accesses only problem-heap no. *i* when this heap overflows or becomes empty. Then a circular search is started visiting the other $n - 1$ problem-heaps until a non-full (respectively non-empty) heap is found. The operation store (respectively retrieve) is then executed on the heap which was found by the search. Operation then continues on problem-heap no. *i* until another exceptional situation arises. The algorithm starts by putting the initial problem into one of the heaps, and it terminates when all heaps are empty and all processes are searching. The termination algorithm is discussed in section 5.2.3. After termination, partial results can be obtained from the heaps and summed to form the solution to the initial problem.

In fig. 5.17 is shown the performance of a program, which is obtained from the basic program, by connecting identical subsystems into a ring structure as described above.

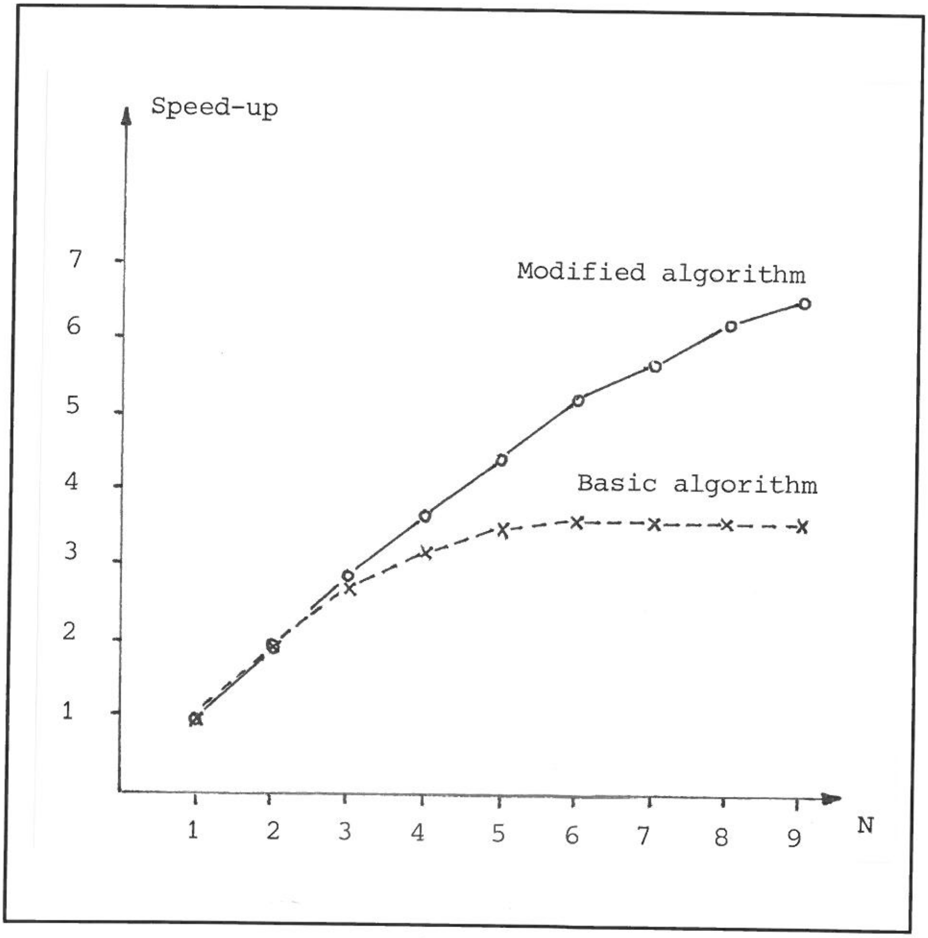


Fig. 5.17

This technique does not have the drawback of the preceeding one. There are no parameters to adjust.

5.2.3 A Termination Algorithm

In all the algorithms presented in section 5.2.2.2 there is a termination problem; that is to decide when a satisfactory solution to the initial problem has been found, and when this decision has been made to make it known to all processes. Having a single central problem-heap greatly simplifies the termination problem. In this section is described a general termination algorithm.

This algorithm is based on a termination detection algorithm for distributed computations by Dijkstra et al. [Dijkstra 1983] . They consider *N* machines, each of which is either active or passive. Only active machines send so-called *messages* to other machines; message transmission is considered instantaneous. After having received a message, a machine is active; the receipt of a message is the only mechanism that triggers for a passive machine its transition to activity. For each machine, the transition from active to passive state may occur spontaneously. From the above it follows that the state in which all machines are passive is stable: the distributed computation with which the messages are associated is said to have terminated.

These terms can be interpreted as follows: A process in a heap algorithm is either in the state active or in the state passive, it is passive when it is not processing any subproblem, and the heaps to which it has access (i.e. in which it either stores or retrieves subproblems) are all empty. The heap algorithm terminates when all processes are passive. A message is identified with a subproblem. To *send a message* means to store a subproblem in some problem-heap, and to receive a message means to retrieve a subproblem from some problem-heap. To apply the algorithm by Dijkstra et al., we must alleviate the requirement that message transmission is instantaneous. For the sake of the proof of the algorithm, it is only necessary to require that a process goes passive only when all the messages it has sent have been received by some other process. This requirement is fulfilled by the above interpretation of message transmission provided that the set of heaps from which we try to retrieve subproblems include the set of heaps in which we may store subproblems. Note, that it is not necessary to ensure that all the heaps, to which a process has access, are simultaneously empty, before the process turns passive. It is sufficient to visit the heaps one by one, and make sure that all subproblems stored by the process have been retrieved by some other process.

This algorithm has been implemented as a general termination module in Concurrent Pascal. Since the termination algorithm assumes that the processors are organized in a ring, such a communication structure is added if it is not already present. The numbers in fig. 5.15 and 5.17 were obtained using this module.

5.3 Summary

In this chapter we have described a number of experiments with an adaptive quadrature algorithm. First a model for the behaviour of this algorithm was described and verified. An important part of this model is an analysis of the saturation loss. This saturation loss occurs because processes need mutually exclusive access to one shared data structure (the problem-heap). Please note the distinction between this saturation loss caused by the algorithm and the hardware saturation discussed in chapter 2.

The last part of this chapter described a number of techniques for postponing the saturation point. As illustrated by the reported experiments the saturation point is postponed, but each technique has some other drawback for example increasing the starvation loss. This is an indication of the problems in algorithms with decentralized control, sometimes called distributed algorithms. In chapter 8 we describe a different way of postponing the saturation loss by alleviating the requirement of mutually exclusive access to the shared data structures.

6. Summary of Other Experiments

Chapters 4 and 5 contained a thorough description of two algorithms and the experiments we have undertaken to analyze them. A number of other algorithms have been implemented and analyzed in a similar way. In this chapter we give a summary of the most important results obtained, the details about these algorithms may be found in the referenced reports.

6.1 Quicksort/Mergesort

The Quicksort algorithm for internal sorting of an array is one of the best known and nicest examples of a divide and conquer algorithm. As mentioned in section 3.1, it is straightforward to implement this algorithm as a problem-heap algorithm. The table in fig. 6.1 shows the speed-up achieved by one such implementation:

N	Speed-up
1	1
2	1.75
4	2.75
8	3.64

Fig. 6.1

The basic Quicksort algorithm can be varied in a number of ways [Sedgewick 1978] . Such variations can influence the speed-up shown above. But no major improvement can be obtained, since the dominant source of loss is starvation loss during the initial phase of the sorting.

The first step of the computation consists of splitting the data into two lists (problems) such that all elements in one list are less than all elements in the other. This splitting requires an inspection of all data elements, so the time to split is proportional to the number of elements, K. During this period, one processor only is working. After this two processors may be put to work on splitting the two problems into four, etc. This is illustrated by the time diagram in fig. 6.2.

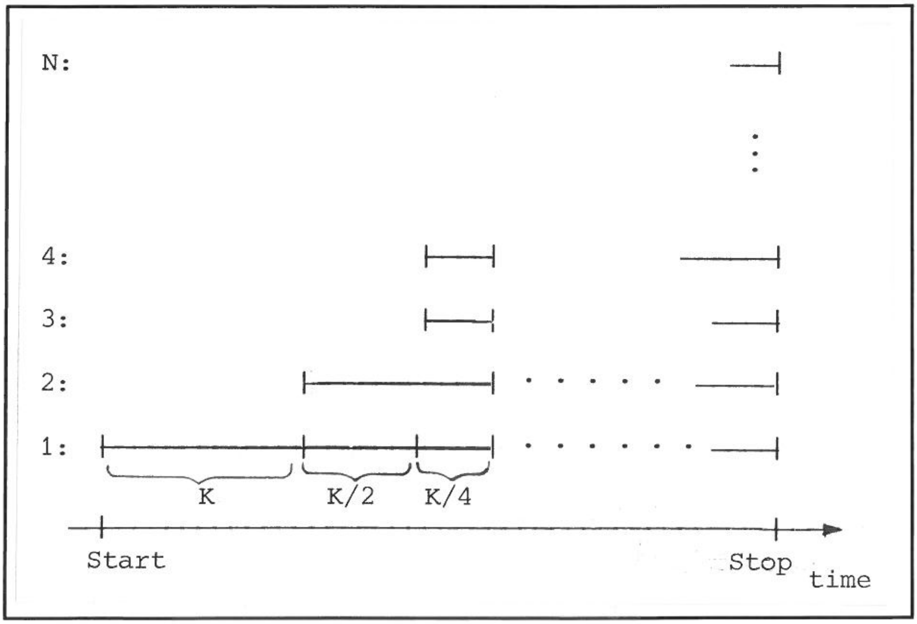


Fig. 6.2

As usual the solid lines represent useful work contributing to the solution, and the empty space represents loss. In this case clearly starvation loss.

The magnitude of the starvation loss is (assuming N is a power of two):

$$S_{loss} = \sum_{j=0}^{(\log N)-1} (N - 2^j) \times \frac{K}{2^j}$$

For very large values of K, this becomes small compared with the total execution time: $K \log K$. But these values of K are far beyond the size of our internal store (16K data elements). So we have not performed any experiments to uncover the effect of other kinds of loss for large K. Results similar to the ones reported above have been obtained on *Cm** [Jones 1980] .

Mergesort is another well known sorting algorithm, it works by repeated merging of two sorted sublists into a new longer sorted sublist. Each iteration increases the length of the sorted sublists. During the final phase of the algorithm, two sorted sublists are merged into one sorted list containing all the original data elements.

The mergesort algorithm can be viewed as doing the inverse of Quicksort. The latter creates smaller and smaller problems, whereas the former works by merging two smaller problems into one larger problem.

Consider first a problem-heap implementation of mergesort where the processes merge lists independently, i.e. does not cooperate on a particular merge. In this case the time diagram (fig. 6.3) is the inverse of the time diagram for Quicksort (see fig. 6.2).

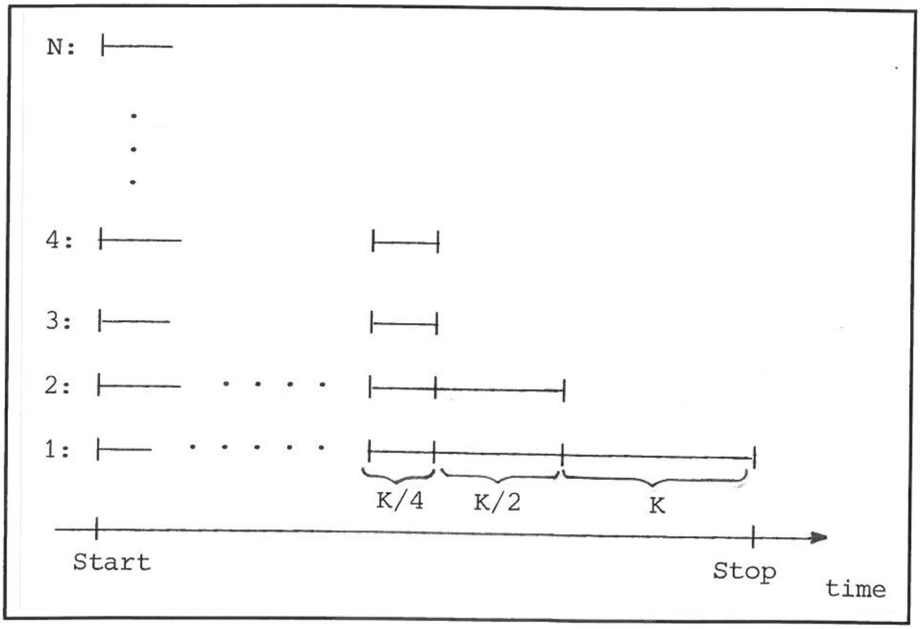


Fig. 6.3

We have not done any experiments with the mergesort algorithm but the above shows that a behaviour similar to Quicksort can be expected.

A major problem with the above sketched implementation of mergesort is the starvation loss in the final stages. One could hope to reduce this loss if several processors could cooperate on a particular merge. There are reasons to believe that this can be done efficiently. One such merging algorithm [Valiant 1975] has been implemented, it showed an almost linear speed-up. We have, however, not tried to implement and analyze mergesort utilizing such a merging algorithm.

6.2 Root Searching

Let H be a real continuous function defined on the closed interval $[a, b]$ and assume that $H(a) \times H(b) \leq 0$, i.e. that H has at least one root in $[a, b]$. We have analyzed algorithms for finding a root of H under the additional assumption that evaluating H is very time consuming.

We consider the class of *partitioning algorithms*, which are iterative algorithms where each iteration reduces the current interval containing the root. Let $x \in [a, b]$; based on the function value, $H(x)$, it can be decided which of the intervals $[a, x]$ or $[x, b]$ that contains the root, this interval becomes the new current interval (see fig. 6.4).

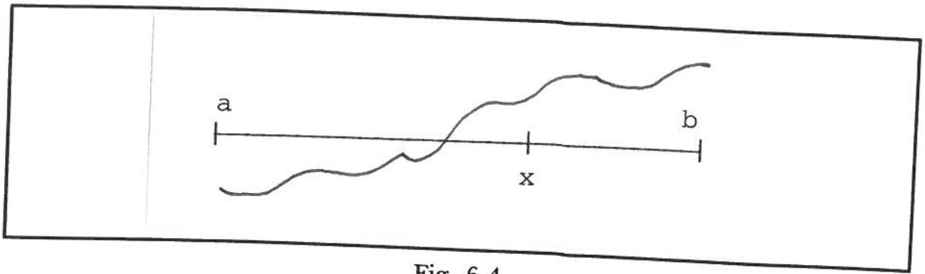


Fig. 6.4

The well known bisection algorithm, where x is the middle of $[a, b]$, is an example of a partitioning algorithm. We have only analyzed partitioning algorithms because they are simple and guarantee convergence without other assumptions on the function than continuity.

Let T_H be the time it takes to evaluate H. When T_H dominates other quantities the running time, B_T, for the bisection algorithm is

$$B_T \approx T_H \times log_2 \frac{b-a}{eps}$$

where *eps* is the absolute accuracy with which the root is obtained. This running time can be reduced by letting N processes evaluate H at different points $x_1, x_2, \ldots, x_N$ concurrently (see fig. 6.5).

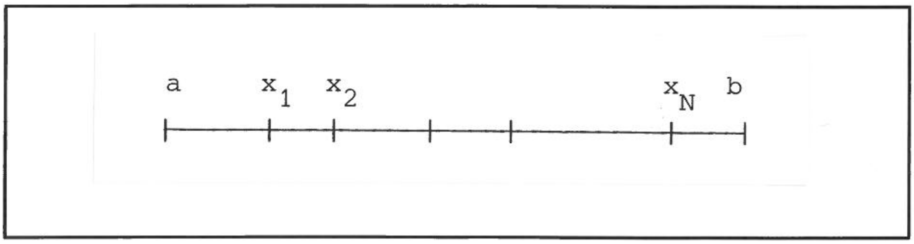


Fig. 6.5

When a process, p_i, finishes its evaluation of $H(x_i)$, it can be decided which of the intervals $[a, x_i]$ or $[x_i, b]$ contains the root. This information must be communicated to the other processes. Based on its own results and those received from others, a process now selects a new point, x, in the interval and computes $H(x)$. The running time of the algorithm depends on how the points $x_1, x_2, \ldots, x_N$ are placed in the interval. Intuitively, the N points should be spread out as evenly as possible, to reduce the worst case running time. Although this is very easy to achieve initially by dividing the interval into $N + 1$ intervals of length $\frac{b-a}{N+1}$, it is difficult to maintain such an even spreading after a few iterations. In [Eriksen and Staunstrup 1983] there is a detailed description of how this can be achieved.

The root searching algorithm shows an application of a useful heuristic for designing multiprocessor algorithms called *worst case reduction*. Consider a sequential algorithm with a fluctuating running time (different input data gives very different running times) (see fig. 6.6).

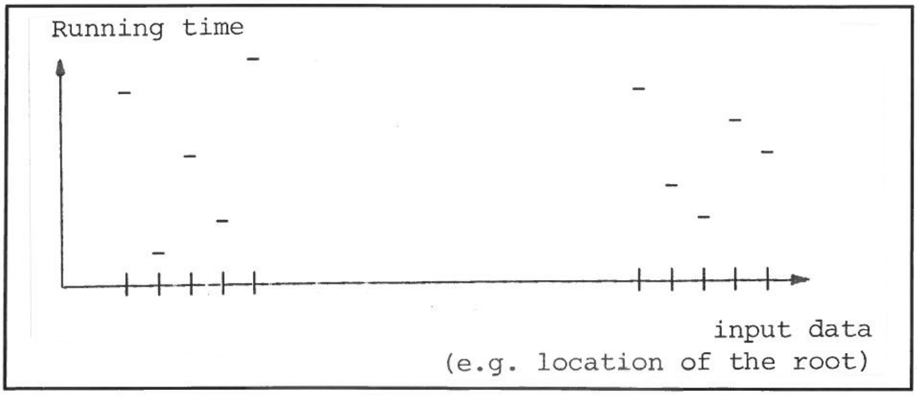


Fig. 6.6

Such a behaviour is for example exhibited by a partitioning algorithm where the current interval is partitioned into intervals of different lengths. Low running times are obtained when the root happens to lie in the short interval and conversely. This can be utilized in a multiprocessor algorithm where several different partitionings are tried in parallel. Intuitively each of the peaks in the above diagram is covered by a process effectively pulling them downwards. This will improve the worst case running time of the algorithm, but it may not improve the average case. The next section contains another application of the worst case reduction heuristic.

Our analysis and implementation of the root searching algorithm shows that it gives a very poor speed-up (see fig. 6.7).

N	Speed-up
1	1
2	1.4
4	1.9
8	2.4

Fig. 6.7

The major kind of loss in this algorithm is an extreme amount of braking loss, not only towards the end of the computation, but in almost every iteration because work done on intervals which turn out not to contain the root is superfluous. A process working on such an interval should therefore be stopped and directed towards the current interval.

6.3 Optimization by Branch and Bound

Branch and bound is a well known algorithm for solving discrete optimization problems [Little 1963] . The algorithm is based on traversing a tree spanning all possible solutions in such a way that some branches of the tree are eliminated before a complete traversal has been made. This family of algorithms is very unstable, for some data almost the whole tree needs to be traversed whereas for other data only a small part needs to be traversed.

It is simple to formulate a problem-heap algorithm for the branch and bound principle. The problem-heap consists of nodes of the tree indicating subtrees that should be traversed. The details of this implementation may be found in [Andersen 1983] . The major result of this implementation is that those data having a very long running time (i.e. where most of the tree should be traversed) give a very good speed-up whereas other data having a short running time exhibit no speed-up. The net effect is that the fluctuations in running times are smoothened out (see fig. 6.8).

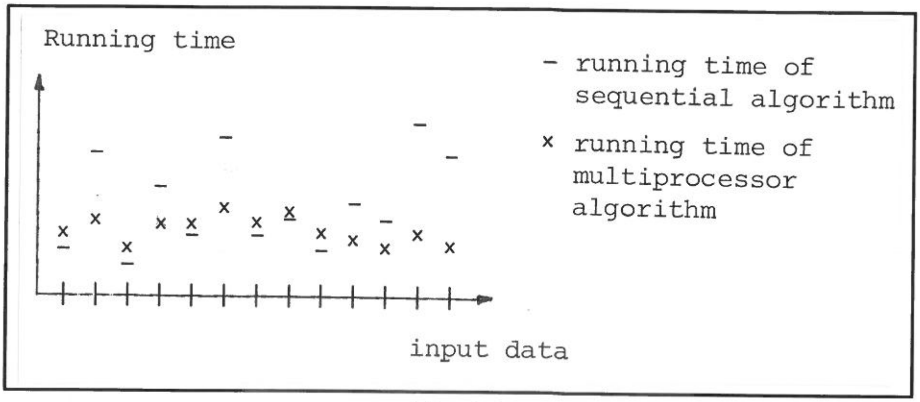


Fig. 6.8

This is another example of the *worst case reduction* principle mentioned in section 6.2. The extra processing power provided by a multiprocessor reduces the worst case running time. This may be important even if the best or average case running times are not improved.

7. The Multi-Maren Machine Architecture

This chapter describes the Multi-Maren hardware. The machine is built out of standard components, allowing a wide range of adjustments and combinations, and the machine has been shaped to our needs by selecting from the available options.

7.1 Overview

Multi-Maren consists of 10 almost identical processors connected to a global store by a common bus as shown on fig. 7.1.

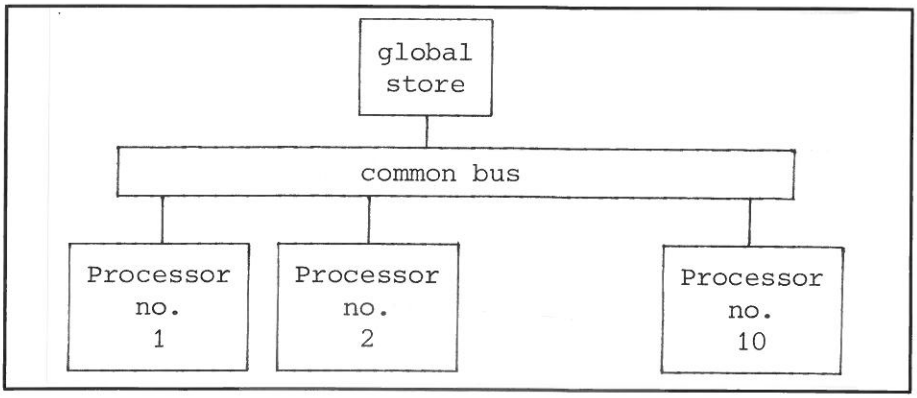


Fig. 7.1

The processors are Intel iSBC 86/12A (one is an MBC 86/12)) boards each of which contains a 16-bit processing unit (Intel 8086), a local store and some I/O interfaces. The common bus is an Intel Multibus equipped with a specially constructed arbiter (see section 7.3). The global store is an Intel iSBC 032 memory board. Each of these is described in further detail below. The processor, global store and arbiter boards are mounted in an Intel iCS80 chassis. The chassis is mounted together with power supplies, connectors for I/O and a cooling system, in a rack as shown on the front page photography.

7.2 The Processor

The processors are Intel iSBC 86/12A boards [Intel 1978a] . These boards can be adapted to a specific application by a number of wire wraps and switches.

This section describes how the boards have been adapted to our use.

The following parts of the board are currently used: The 8086 microprocessor, the RAM store, the EPROM store, the timers and the serial I/O port.

The Intel 8086 is a 16-bit microprocessor, this means that operands typically are 16 bit quantities. It is, however, possible to address 8 bit quantities (bytes). The total address space is 1 Mbyte. The Intel 8086 is described in further detail in [Intel 1979] .

7.2.1 The RAM Store

The RAM store on a board can be accessed directly by the processing unit on that board, and by any other processor in the system via the Multibus. All accesses to the RAM store are made through a so-called »dual port« which is under the control of either the processing unit on the same board or the Multibus, see fig. 7.2. Since the RAM store is placed on the same board as the processing unit, it is called the local store. (Section 7.4 describes a store which is global to all processors.)

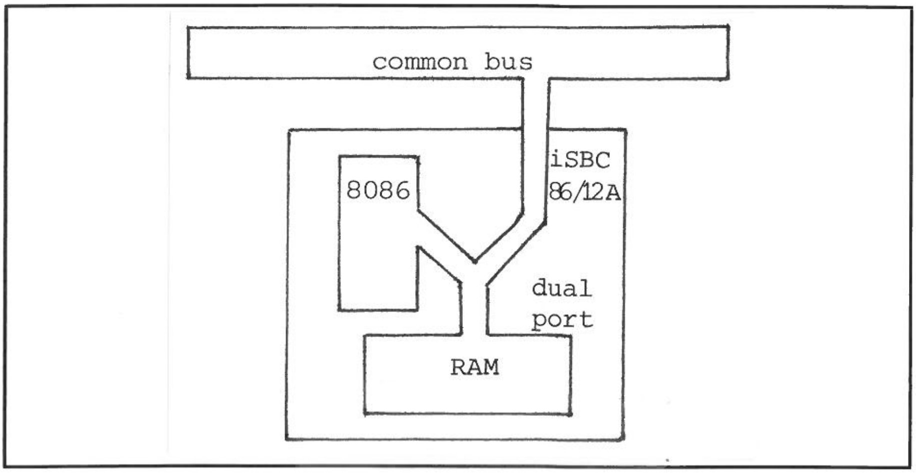


Fig. 7.2

There are 32K bytes of RAM store on each processor board. These are accessed locally by the (hexadecimal) addresses: 00000-07FFF. However, when accessed from the Multibus the addresses of all the RAM stores are different. This makes it possible to distinguish between the stores on all the boards. The (hexadecimal) addresses of the local stores, when accessed from the Multibus, are:

10000-1FFFF	store on board 1
20000-27FFF	store on board 2
.	
.	
.	
A0000-A7FFF	store on board 10

So all RAM store is accessible by all processors. *Note*, that the addresses of a RAM store are different when accessed locally and from the Multibus.

7.2.2 The EPROM Store

Each board has 8K bytes of EPROM store. This store is only accessible by the on-board processing unit. It is accessible through the addresses: FE000-FFFFF. When the power is turned on or when the system is reset, each processor starts executing the instruction placed in address FFFF0 of the EPROM store. A program (called the »inspector«) is placed at this address. This program initializes the board and makes it possible to bootstrap other programs (see section 8.7).

7.2.3 The Timers

Each of the boards has two 16 bit counters which can be used as timers by the processing unit on the same board. The two counters are called »TMR0« and »TRM1«. TMR0 is decremented with a frequency of 1.23 Mhz, i.e. every 0.8 microsecond. TMR1 is decremented every time TMR0 gets to 0. The values of TMR0 and TMR1 can be changed and read by I/O instructions. The timers are used to implement the Concurrent Pascal function called »realtime« (see chapter 1).

7.2.4 Input/Output

Each board has a serial I/O port and a number of parallel ports. These ports can be used by the processing unit on the board through its I/O instructions. The serial I/O port is used to implement a character oriented I/O device (called a »typedevice« [Brinch Hansen 1977]) for each Concurrent Pascal process. The process executing on processor no. 1 (the initial process) is connected to a terminal (see chapter 1). The remaining I/O ports are normally unconnected.

average waiting time for the bus when nobody else requests the bus:	500	nsec.
maximal waiting time for bus:		
nobody else requests the bus:	900	nsec.
all ten processors request the bus:	14	μsec.

7.4 The Global Store

The global store is an Intel iSBC 032 memory board. It contains 32K bytes and the (hexadecimal) addresses of the global store are: B0000-B7FFF. Locations in the global store can only be accessed through the Multibus. This property, combined with the possibility of locking the Multibus for a number of consecutive bus cycles (see section 7.3), provides the indivisible exchange operation on which the synchronization is based (see section 8.3). Synchronization variables must be placed in the global store. A memory location in a processor cannot be used, since the on-board processing unit cannot be locked out for more than one bus cycle.

8. The Concurrent Pascal Implementation

The purpose of the multiprocessor laboratory is to construct and analyze algorithms. Therefore we needed a high level language as a programming tool. A preliminary analysis of existing languages [Søe and Søgaard 1981] showed that none of them were ideal for our purposes. On the other hand we thought it would be a mistake to try to design a new language. First of all this would postpone the algorithmic experiments that was our primary interest; secondly there was no reason to believe that we could make a perfect language design before having a considerable experience with multiprocessor algorithms. Concurrent Pascal [Brinch Hansen 1977] was found to be a reasonable compromise because we had easy access to a good implementation. Since the language is not an issue in our laboratory we have taken the liberty to make changes to the language whenever our experiments showed a need to do so. These changes and their justification are explained below.

It was very important for us to finish the implementation of Concurrent Pascal as quickly as possible, so we could get on with the algorithmic experiments. The absolute speed of the implementation was only of secondary importance, since most of our analyses are independent of the absolute speed of the processors.

The central decision in the Concurrent Pascal implementation is to allocate one processor to each process. This leads to a very simple implementation which has several advantages: (listed in order of priority)

1. Analyzing a program running on a complex implementation can be very hard, because it is difficult to distinguish the properties of the implementation from those of the program. The experience from the Cm* multiprocessor at Carnegie Mellon University clearly confirms this [Jones 1980].

2. It can be very hard to convince oneself of the correctness of a complex implementation. Testing is extremely difficult when processors run simultaneously.

3. A simple implementation can be completed rather quickly.

Many of the details of the implementation reported below can be explained by the decision to allocate one processor for each process.

As described in Chapter 7, all processors share a global store, which can only be accessed through a common bus, the Multibus. This bus can become a bottleneck if all processors are using it frequently. Before and during the construction of the multiprocessor we were very aware of this danger. An effort was therefore made to reduce the use of the Multibus as much as possible.

8.1 Program Structure

A Concurrent Pascal program consists of a number of definitions (constants and types) and an initial process where all processes and shared data structures are declared and initialized.

The following is a typical structure of a Concurrent Pascal program:

```
CONST
        c     =   ... ;
TYPE
        ml    =   MONITOR
                  ...
                  END;
        cl    =   CLASS
                  ...
                  END;
        p2    =   PROCESS
                  ...
                  END;
        p3    =   PROCESS
                  ...
                  END;
        .
        .
        .
        m2    =   MONITOR
                  ...
                  END;
        p10   =   PROCESS
                  ...
                  END;
```

```
»initial process«
VAR
   v: ...;
BEGIN
   INIT v(   );
   ...
END.
```

The store references of one processor executing a Concurrent Pascal process can be divided into the following three categories:

- a) references to code,
- b) references to local data,
- c) references to shared data.

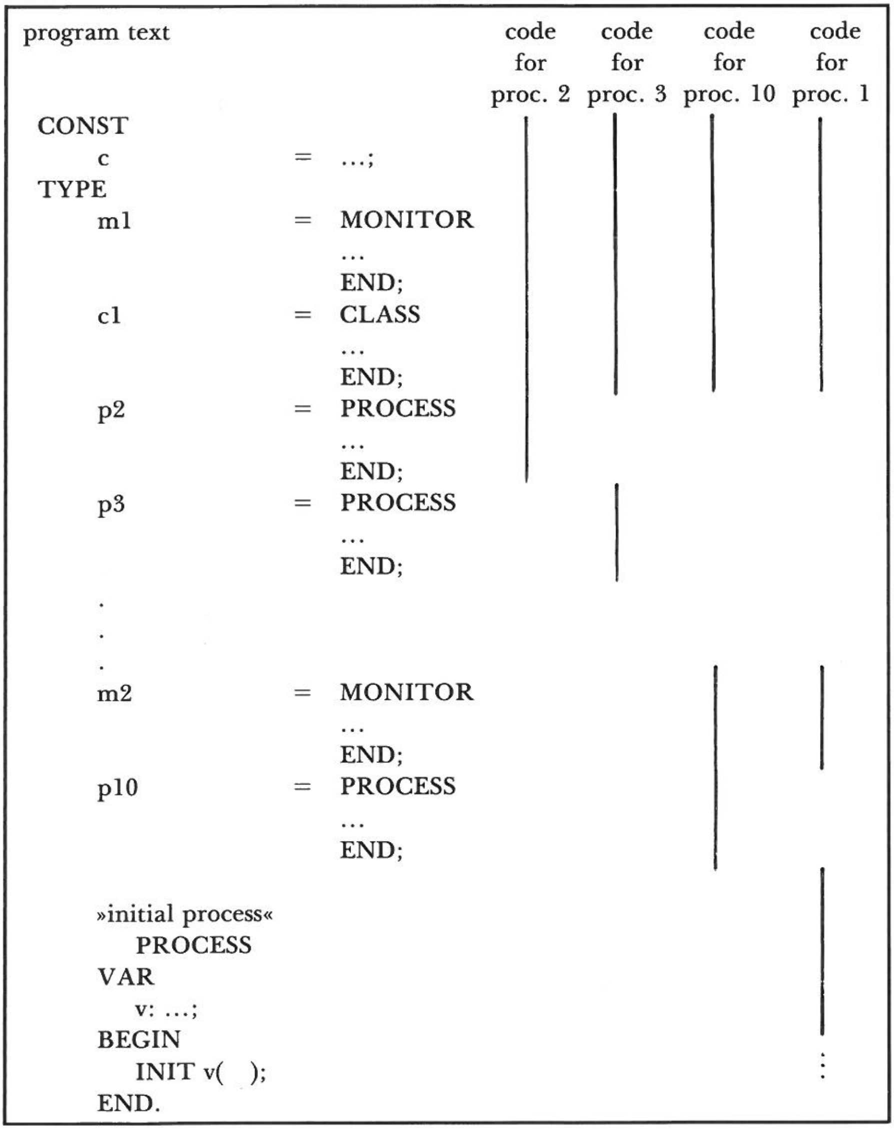
The local data (process and routine variables) are used by one process only. Whereas the shared data are used by more than one process.

Therefore it was decided to place only shared data (global variables and associated synchronization variables) in the global store. The distinction between local and global monitor variables is explained by the following skeleton of a monitor:

```
TYPE
   m =
MONITOR(s: ...);
   VAR
      g1: ...;        g2: ...;        global monitor variables
      ...

   PROCEDURE ENTRY e(p: ...);
      VAR
         l1: ...;        local routine variables
         l2: ...;
      BEGIN
         Se;             routine body
      END;
      ...
BEGIN
   Sg;                monitor body
END.
```

A consequence of this decision is that all code for the monitor body and the routine bodies is duplicated in each of the processors. Each has its own copy to avoid using the Multibus for fetching instructions. By making a static allocation of processes to processors the amount of duplicated code is reduced considerably. Fig. 8.1 shows how the compiler distributes the code of a typical Concurrent Pascal program:



```
CONST
    c           =  ...;
TYPE
    m1          =  MONITOR
                   ...
                   END;
    c1          =  CLASS
                   ...
                   END;
    p2          =  PROCESS
                   ...
                   END;
    p3          =  PROCESS
                   ...
                   END;
    .
    .
    .
    m2          =  MONITOR
                   ...
                   END;
    p10         =  PROCESS
                   ...
                   END;

    »initial process«
       PROCESS
    VAR
       v: ...;
    BEGIN
       INIT v(   );
    END.
```

Fig. 8.1

In addition to the code, the local store of each processor contains the global variables of the process together with the parameters and local variables of the routines (including monitor routines).

8.2 The Interpreter

Concurrent Pascal is implemented by means of an interpreter. The interpreter executes C-code instructions as generated by the compiler. The C-code is described in more detail by Hartmann [1977] . It has been necessary to modify a few of the C-code instructions to make it possible to utilize the full 20-bit address range of the hardware.

The interpreter has been programmed in PL/M-86, and the data type »real« is handled by the standard real emulator [Intel 1978b] . In a pilot study this emulator has been replaced by the 8087 numerical co-processor. This improved the speed of the Concurrent Pascal real operators by a factor of 20. As a side effect it freed the 8K of store used by the emulator. Exchanging the real emulator in software with the co-processor hardware has no influence on other parts of the interpreter. So far we have not found it worthwhile to invest in the co-processor hardware, again because the absolute speed of the machine has low priority. Similarly, implementing Concurrent Pascal by an interpreter written in PL/M-86, the absolute speed of the machine has been reduced significantly (at least by an order of magnitude). The advantage on the other hand is the short time it took to do the implementation (see chapter 9), and the ease with which we can do modifications.

8.3 Synchronization

There are two levels of synchronization in Concurrent Pascal. The lowest level, called the *short-term synchronization,* ensures that at most one process at a time is executing one of the monitor routines of a particular monitor. This is implemented by a so-called *gate*. The gate has two states: open and closed. When it is closed some process is executing a monitor routine.

The second level of synchronization, called *medium-term synchronization,* is used to express the logical synchronization constraints of the program, e.g. that no more elements can be put into a buffer which is full. To express the medium-term synchronization, Concurrent Pascal has a predefined type *queue*, with three associated operations *delay, continue* and *empty*.

The implementation of gates and queues is based on the fact that the machine provides an indivisible exchange operation, :=:, which exchanges the contents of a variable located in a local store with a variable contained in the global store.

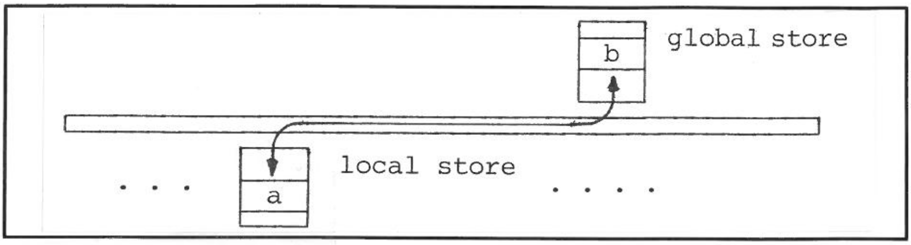


Fig. 8.2

a: = :b exchanges the contents of a and b, nobody else can use the global store during this exchange (see fig. 8.2).

8.4 Short-term synchronization

A correct solution must satisfy the following two requirements:

(1) at any time at most one process may execute monitor routines of a particular monitor (*mutual exclusion*),

(2) if no process is using a monitor, a request to enter a monitor routine must be granted (*responsiveness*).

Note, that requirement 2, responsiveness, is weaker than the fairness requirement made in the Concurrent Pascal report [Brinch Hansen 1977] . This is discussed in section 8.5.

Consider the following implementation, which satisfies (1) and (2). Entry to a monitor routine is done by calling *entermon*, and exit from the monitor is done by calling *exitmon*.

```
TYPE
   gate-state = (open, closed);

   gate =
   CLASS
      VAR
         g: gate-state;   »allocated in the global store!«

      PROCEDURE ENTRY entermon;
         VAR
            help: gate-state;   »allocated in a local store«
      BEGIN
         help: = closed;
         REPEAT
            g: = :help;
         UNTIL help = open;
      END;

      PROCEDURE ENTRY exitmon;
      BEGIN
         g: = open;
      END

   BEGIN
      g: = open;
   END;
```

Synchronization algorithm 8.1.

This is a well known synchronization algorithm which fulfills the two requirements stated above. Let $help_i$ be the local variable in the i-th processor. Then at most one of the variables $g, help_1, help_2, \ldots, help_{10}$ can have the value *open*. Therefore mutual exclusion is ensured. The responsiveness of the algorithm is based on the fact that there is a maximal waiting time for the common bus and hence also on the execution of the exchange operation. Therefore if the gate is open, this value will be received by the processor that wants to enter a monitor routine.

At first we rejected this synchronization algorithm because we feared that the busy-waiting loop would put a very heavy load on the common bus. While a process is waiting, it constantly performs exchange operations locking the bus temporarily to inspect the gate.

The load on the shared bus can be decreased by prolonging the cycle time of the busy-waiting loop.

```
REPEAT
   wait;
   g: = :help;
UNTIL help = open;
```

Ideally the wait should not burden the common bus and it should last exactly until the gate becomes open, no longer and no shorter. However, the correctness of the solution is not affected by making the wait shorter than this ideal. There are many ways of implementing the wait. We used an implementation where the waiting ceases when a gate changes from closed to open. The operation *exitmon* becomes:

```
PROCEDURE ENTRY exitmon;
BEGIN
   g: = open;
   indicate-state-change;
END;
```

Synchronization algorithm 8.2.

For almost a year we used algorithm 8.2, the details are described in [Møller-Nielsen and Staunstrup 1982] . But in the summer of 1982 we made an experiment to compare the two algorithms. The tables in fig. 8.3 give a summary of the results from these experiments. The tables show the influence of a number of experiments with a varying number of processes trying to enter a closed monitor, i.e. executing the busy-waiting loop. Horizontally the number of processes doing this is shown. Two different situations were analyzed, in the first the influence on a process trying to enter a monitor is shown. Table 1 gives the time to execute an empty monitor routine (begin end;). Table 2 gives the time it takes to execute a monitor routine with many references to the shared store.

no. of processes doing 'busy-wait':	0	2	4	7
algorithm 8.2	1.0	1.0	1.0	1.0
algorithm 8.1	0.71	0.71	0.71	0.73

Table 1

no. of processes doing 'busy-wait':	0	2	4	7
algorithm 8.2	1.0	1.0	1.0	1.0
algorithm 8.1	1.0	1.0	1.01	1.03

Table 2

Fig. 8.3

From these tables it seems that algorithm 8.1 should be preferred. The large differences in Table 1 are mainly caused by the time it takes to execute »indicate-state-change« in algorithm 8.2, this is not needed in algorithm 8.1.

The explanation of the numbers in both Table 1 and Table 2 was given in chapter 2: when all code and local variables are placed in the local stores, there are very few storage references to the global store. Even though there are few global references, there is of course a limit to how many processors that can be doing the busy loop before the bus becomes saturated. But consider what happens up to and at the saturation point. The processes doing the busy-waiting loop are slowed down, but that does not matter since they are not doing anything useful. The execution time of a monitor routine ($\approx$ Table 2) would slowly increase up to the saturation; but since the global store references are infrequent ($< 1\%$, see chapter 2), even a long delay of those is insignificant.

It is essential that the bus arbiter is constructed so that there is a maximal waiting time before the bus is granted. If this is not the case, a reference to the global store could be postponed indefinitely. This point is discussed further in section 8.5 on fairness.

The conclusion of the discussion above is that there are considerable benefits in choosing algorithm 8.1 while the drawbacks are insignificant. Therefore we are currently using algorithm 8.1 (with the busy-waiting loop).

8.5 Fairness

Usually it is required that the short-term synchronization algorithm is *fair*, i.e. that all processes that wish to enter a monitor are allowed to do so within a finite period. Requirement (2), responsiveness, stated in section 8.4 is, however, much weaker. There are several reasons why we gave up the stronger fairness requirement to the short-term synchronization. Firstly, there is not a common agreement on how to define fairness precisely [Queille and Sifakis 1983] . Secondly, if fairness is an important assumption for a multiprocessor algorithm, it must be because there are processes waiting to enter a monitor most of the time, hence it must be a rather inefficient algorithm. Finally, in almost all our programs all the processes are identical (see chapter 3). In such symmetric programs it is irrelevant which of the identical processes are served first.

It must be emphasized that these arguments are made for the short-term synchronization only. Above and underneath this level are other levels of synchronization where other arguments may apply. In particular underneath the short-term synchronization level is the bus synchronization level or bus arbitration as it was called in section 7.3. As it is pointed out several times in this report (e.g. in chapter 2), it is crucial that the bus arbiter can guarantee a maximal waiting time for the bus. This is a much stronger requirement than responsiveness.

8.6 Medium-term Synchronization

A process may await a signal in a monitor by calling »delay« on a variable of type »queue«. This releases the exclusive access to the monitor so that another process may enter a monitor routine. When another process calls »continue« on the same queue variable the delayed process can go on. But since at most one process may be executing monitor routines of a particular monitor, it must be decided in which order the two processes should finish their monitor routines. Of the many suggestions to resolve this, consider the following alternatives:

(a) the delayed process finishes first,

(b) the delayed process finishes last,

(c) avoid the issue by requiring continue to be the last statement of the routine.

Alternative c) is the one chosen in Concurrent Pascal and therefore the alternative we chose initially. Using this alternative it is, however, not possible to have more than one continue (dynamically) in a monitor routine. It turned out that almost all our programs had a monitor where one process should start a number of other processes. This is very awkward to express using at most one continue per routine. Therefore, our implementation has now been changed to alternative b). Although b) is slightly more complicated to implement than a), we chose it because it was more convenient to express our algorithms using alternative b). This was done in the summer of 1982. Section 8.9 describes another of the changes to Concurrent Pascal which we have experimented with.

8.7 Initialization

This section describes how a Concurrent Pascal program is loaded and started. When the power is turned on or when the machine is reset, a program (placed in the EPROM store) called the *inspector* is started. The inspector is similar to the *iSBC 957 debugger* supplied by Intel [Intel 1978c]. Among other facilities, this program can load and start the Concurrent Pascal loader. The inspector and the Concurrent Pascal loader implements the two levels of virtual machines shown on fig. 8.4.

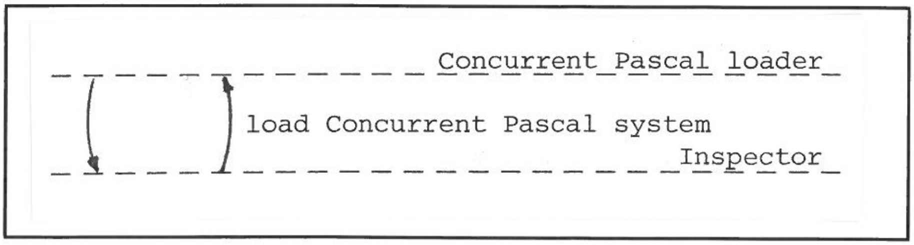


Fig. 8.4

When the Concurrent Pascal loader is running, all processors have received their copy of the Concurrent Pascal interpreter and kernel, but they have not yet received the c-code (the program to be executed). This is loaded and distributed by the Concurrent Pascal loader. This adds a new virtual machine level shown on fig. 8.5.

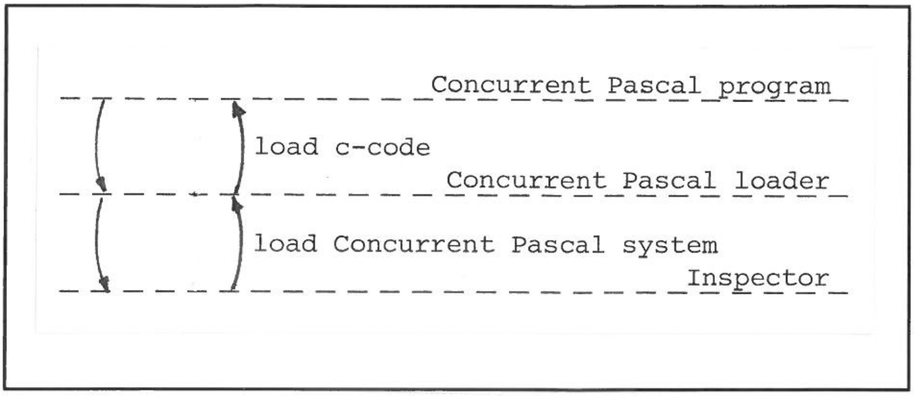


Fig. 8.5

The Concurrent Pascal loader is a PL/M-86 program running on processor number 1. It receives the c-code from the *Intellec* development system and distributes it to the relevant processors. When the processors have received their part of the c-code, they start executing it, but only until they reach the c-code instructions corresponding to the first begin in the process body (the c-code instruction *beginproc*). At this instruction they are stopped until explicitly started by an *init-statement* in the initial process (see fig. 8.6).

```
initial process        process 2  ...  process 10

var                        .                .
  p2 : ...                 .                .
  ...
  p10: ...                 .                .
                           .                .
begin                      .
  init p10; -----------------------> begin
  ...                      .
  init p2;  -------> begin
  ...
end.
```

Fig. 8.6

The states of all the processors are described in a table which has an entry for each process. An entry has one of the following values:

not started:	The interpreter has not yet been started.
wait initproc:	Waiting for init-statement.
running:	Executing c-code instructions.
terminated:	The processor has executed the last »end« of its process body.
delayed:	Delayed in a queue.
error xxxxx:	An exception has occurred, xxxxx indicates which.

The contents of this table is displayed by processor 1 either when the INTR button on the front panel is depressed or when the initial process terminates. Depressing INTR brings the system back on the loader level where a new program may be loaded and started.

Immediately after starting a Concurrent Pascal program, the contents of all the processors' local stores are as shown in fig. 8.7.

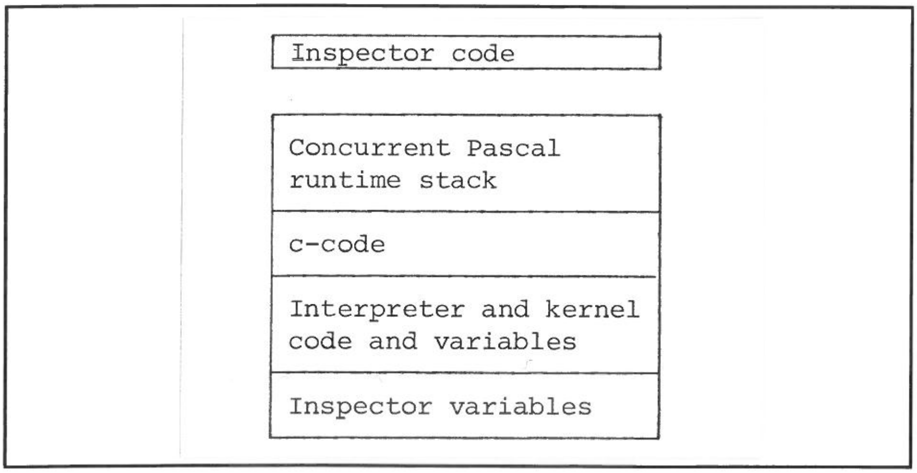

Inspector code
Concurrent Pascal runtime stack
c-code
Interpreter and kernel code and variables
Inspector variables

Fig. 8.7

The contents of the global store is shown in fig. 8.8.

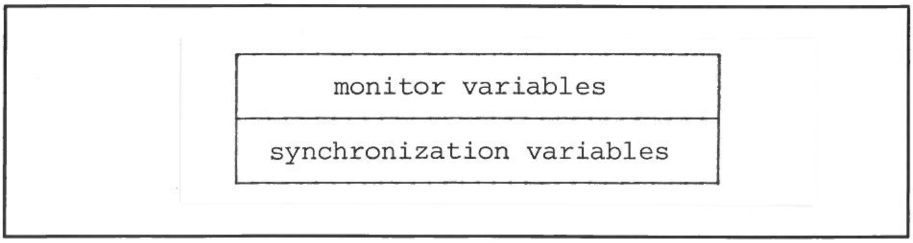


Fig. 8.8

8.8 Input/Output

There is only one I/O device available for each processor, a terminal. Each processor has its own I/O port which is a standard RS 232 connection. Therefore, up to ten terminals may be connected and used independently. Usually only one terminal is connected (to processor 1, i.e. the initial process), this terminal is actually an Intel »Intellec« development system mentioned in section 8.7 (see fig. 8.9).

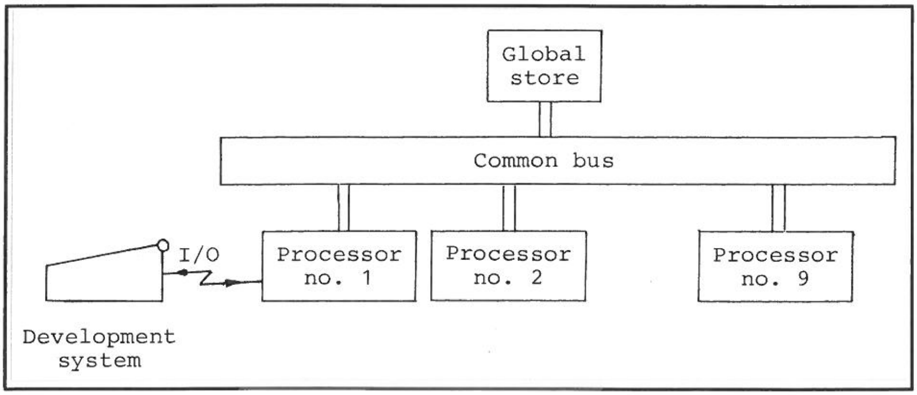


Fig. 8.9 The normal I/O connection.

This very limited I/O facility has been tolerable for our algorithmic experiments described in chapters 4, 5 and 6. It is of course completely inadequate for a multiprocessor running some service function: an operating system, a compiler or a database. In the spring of 1983 we conducted a number of experiments and investigations on how to provide adequate I/O facilities.

This was centered around the device monitor concept [Ravn 1980] . The experiments pointed out some practical problems e.g. uniform addressing of I/O ports. But more importantly they demonstrated the shortcomings of the decision to allocate one processor to each process, since this requires dedication of a complete processor to each input device. If this becomes too costly, it is necessary to multiplex a processor between several processes.

8.9 Common Class

This section contains a summary of the experiments we have made with a modified version of Concurrent Pascal which allows simultaneous access of processes to shared data. An example is the »Readers-Writers« kind of access control to a shared table. »Readers-Writers« cannot be programmed in Concurrent Pascal. The shared data must be encapsulated in a monitor, and this implies that all accesses to the shared data are treated as »Writers«.

In order to make it possible to measure how the performance of a multiprocessor algorithm changes when the access control to shared data is refined, we decided to add a suitable construct to the current Concurrent Pascal implementation on our multiprocessor system. The actual implementation of such a construct was based on the following considerations. The added construct should be seen as a tool by means of which we could experimentally study the effect of different refinements of access control to shared data. It should not be seen as a proposal for the syntax and semantics of a new construct to be included in the language Concurrent Pascal. It is our firm belief that such proposals should be postponed until the subject has matured through the study of the effect of different forms of access control refinement in programming complexity and performance. Such a study is best conducted by programming and executing a wide variety of algorithms with different refinement of access control. For this reason, we wanted to add a construct which implied as little change as possible to the Concurrent Pascal compiler and the c-code interpreter.

8.9.1 The Concept of a Shared Class

It would be possible to express »Readers-Writers« (and other strategies for controlling access to shared data) if the monitor concept in Concurrent Pascal was substituted by the *shared class*. A shared class has the following form:

```
sh =
SHARED CLASS ( < parameters > );
   VAR
      STATE    $V_s$:    ...;     »access to a procedure entry is granted
                                  based on the variables $V_s$.«
      DATA     $V_d$:    ...;     »the shared data.«
   PROCEDURE ENTRY $P_i$( < parameters > );
      ENTRY IF $B_i(V_s)$ THEN entry_statement$_i(V_s)$;

         »$B_i(V_s)$ is a boolean expression of the variables $V_s$, and entry_state-
         ment$_i$ is a statement, which manipulates the variables $V_s$. The
         subscript $i$ indicates that this boolean expression and statement is
         particular for the procedure entry $P_i$.«

        BEGIN »The body of the procedure entry $P_i$.«
           $S_i(V_d)$ »The manipulation of the shared data.«
        END
      EXIT exit_statement$_i(V_s)$;

BEGIN
   »Initialization of $V_s$ and $V_d$.«
END;
```

The meaning of the constructs in a shared class is the following. When a process calls a procedure entry in a shared class, access to the body of the entry is not granted based on a mutual exclusion principle, as it is for the monitor, but access is granted only if the boolean expression B_i (V_s) evaluates to true. Then the statement *entry_statement$_i$* is executed, and after that the execution of the statement $S_i(V_d)$ is initiated. The evaluation of B_i and the execution of the statement *entry_statement$_i$* are done indivisibly, and excluding execution of other entry- and exit-statements of the same shared class. When the execution of the statement S_i is completed, the statement *exit_statement$_i$* is executed in an indivisible manner, and excluding execution of other entry- and exit-statements of the same shared class. Entry- and exit-statements may only refer to the state-variables (i.e. V_s) and the bodies may only refer to the data-variables (i.e. V_d).

Using this concept of a shared class, »Readers-Writers« can be programmed as follows:

```
r_w =
SHARED CLASS;
   VAR
      STATE   n_readers,             »The number of processes executing Sr
                                     in the Read entry.«
              n_writers : integer;   »The number of processes executing Sw
                                     in the Write entry.«
      DATA    t:table;               »The shared table on which the Readers
                                     and the Writers are operating.«
   PROCEDURE ENTRY Read(...);
      ENTRY IF n_writers = 0
            THEN n_readers := n_readers+1;
         BEGIN
            Sr(t)
         END
      EXIT n_readers := n_readers-1;

   PROCEDURE ENTRY Write(...);
      ENTRY IF (n_readers + n_writers) = 0
            THEN n_writers := n_writers+1;
         BEGIN
            Sw(t)
         END
      EXIT n_writers := n_writers-1;

BEGIN
   n_readers := 0;     n_writers := 0;
   »Initialization of 't'.«
END;
```

A monitor can be programmed in a similar way using a single boolean (a »gate«) as V_s. The other extreme, which allows unrestricted access to a set of shared data, can be obtained by leaving the statements entry_statement and exit_statement empty and by using the constant true for the expression B.

8.9.2 The Concept of a Common Class

To implement the shared class proposed above a rather extensive change in compiler and interpreter is necessary. Instead we chose to keep the monitor construct unchanged and add a construct called a *common class*. A common class is a shared class with unrestricted access. From an implementation point of view, a common class is a monitor to which the gate is always left open. For this reason a common class is translated as a monitor except at one point; the c-codes ENTER CLASS, EXIT CLASS, BEGIN CLASS, END CLASS and INIT CLASS are generated instead of the c-codes ENTER MON, EXIT MON, BEGIN MON, END MON and INIT MON (these c-codes are explained in [Brinch Hansen 1977]). This means that no new c-code has to be added to the c-code machine, and the c-code interpreter (and kernel) can be left unchanged. The changes to the compiler are few and straightforward.

Programs written by means of the shared class construct can be systematically transcribed into a program which uses only monitors and common class. A shared class is represented by a pair consisting of a common class and a monitor, to which the common class has access. The common class contains the shared data (i.e. V_d) and the operations $S_i(V_d)$. The monitor encapsulates a transcription of the entry and the exit statements of the shared class. The shared class named *sh* is transcribed into the following pair.

```
sh_mon =
MONITOR;
  VAR  $V_s$ : ...;
        w: waiting_room;              »a pool of queue-variables« .
        w_adm: waiting_room_adm;  »administration of the pool«.

  PROCEDURE ENTRY entry_to_$P_i$;
    BEGIN
      WHILE not($B_i$($V_s$)) DO
        delay(w(.w_adm.vacant.));
      entry_statement$_i$($V_s$);
      WHILE not(w_adm.empty) DO
        continue(w(.w_adm.occupied.))
    END;

  PROCEDURE ENTRY exit_from_$P_i$;
    BEGIN
      exit_statement$_i$($V_s$);
      WHILE not(w_adm.empty) DO
        continue(w(.w_adm.occupied.))
    END;
    BEGIN
       INIT($V_s$)
    END;
```

The pair *w* and *w_adm* implements a pool of queue-variables. The operations on the pool are:

empty:	= true, if the pool is empty, i.e. if all queue-variables in the pool are empty.
vacant:	the identity (e.g. an index in an array of queue-variables) of a queue-variable in the pool which is empty.
occupied:	the identity in the pool of a queue-variable which is not empty.

Note, that our implementation of the continue-statement deviates from normal Concurrent Pascal [Brinch Hansen 1977]. An execution of a continue-statement does not enforce an exit from the procedure entry (see section 8.6). Note also, that *sh_mon* contains two procedure entries (*entry_to_*P_i and *exit_from_*P_i) for each procedure entry (P_i) in the shared class. The common class associated with *sh_mon* looks as follows:

```
sh =
COMMON CLASS(...;s:sh_mon);
   VAR $V_d$: ... ;

   PROCEDURE ENTRY $P_i$(.....);
      BEGIN
         s.entry_to_$P_i$;
         $S_i(V_d)$;
         s.exit_from_$P_i$
      END;

      BEGIN
         init($V_d$)
      END;
```

It should be noted that local variables in the procedure entry of a common class are allocated anew (and locally to the calling process) for each new invocation. Only the global variables (i.e. V_d) are allocated globally to the processes, and in just one copy.

8.9.3 An Example

This section summarizes experiments with three programs performing the same task. The experiments are reported in detail in [Møller-Nielsen and Staunstrup 1983] . The programs are identical with the exception that the control of access to a shared table is programmed in three different stages of refinement. The first program uses a monitor to encapsulate the table. The second program uses a single shared class, which is transcribed into a pair consisting of a monitor and a common class as described above. The third program uses a further refinement of the access control. This refinement cannot be programmed by means of a single shared class.

All programs solve the following problem. The entries of a table are updated according to a set of transactions. The table has a fixed number of entries. Each entry has a key which identifies the entry and a field which contains some information, which is associated with the key. The set of transactions contains only two different kinds of transactions, a WRITE and a READ. A WRITE looks up a key in the table and updates the information field of the entry. Once the entry has been found, the updating must be done indivisibly, i.e. it cannot be mixed with other READs or WRITEs. A READ looks up a key in the table and reads the information field. The reading of the information can be done simultaneously with other READs (but not other WRITEs) of the same information field. Operations on different entries can be done simultaneously. The sequence in which the transactions are carried out is immaterial. The transactions are divided evenly and randomly between a number of identical processes. The problem is solved when all processes have executed all the transactions that were allocated to them. The structure of the program can be depicted as shown in fig. 8.10.

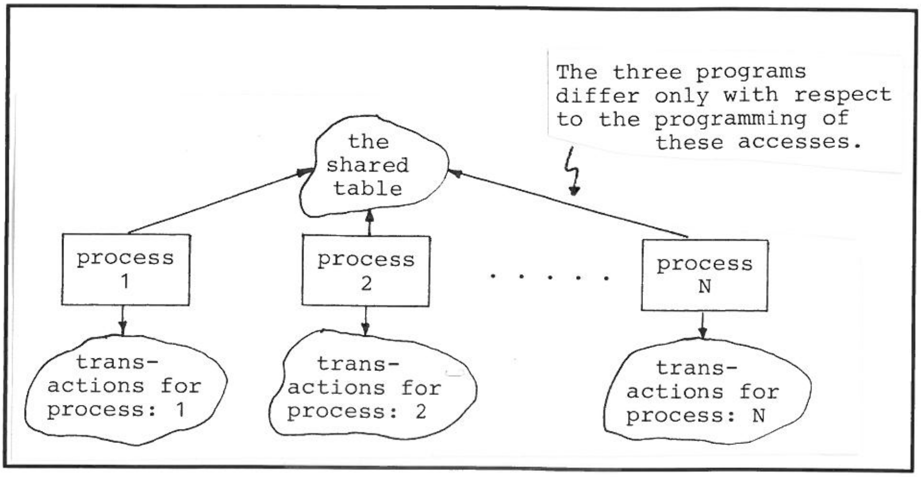

The three programs differ only with respect to the programming of these accesses.
the shared table
process 1
process 2
. . . . . .
process N
trans- actions for process: 1
trans- actions for process: 2
trans- actions for process: N

Fig. 8.10

The shared table and the control of access to it differs in the three programs:

Program 1
The shared table is encapsulated in a single monitor.

Program 2
The shared table is encapsulated in a single shared class, which is transcribed into a monitor and a common class.

Program 3
This program resembles program 2, except that the entries of the common class is programmed in such a way that the search for the proper entry is allowed to take place simultaneously with other searches and readings and writings of information fields. Details of the three programs may be found in [Møller-Nielsen and Staunstrup 1983].

The three programs have been executed on our multiprocessor system in order to measure the changes in performance as access control was refined from program 1 to program 3. The measurements are displayed in terms of the speed-up $S(N)$ in fig. 8.11. $S(N)$ depends on parameters such as the quotient between the time spent retrieving a transaction and the time spent searching the table for a single key, the fraction of WRITEs among the transactions etc. The graphs below show — for a particular choice of these parameters — the speed-up for the three programs and for different numbers of processes. The measured values illustrate how the saturation point of the shared object (i.e. the table) is moved to higher values by refining the access control. Refinement of access control can thus be added to the set of tools which can be used to postpone software saturation (see section 5.2).

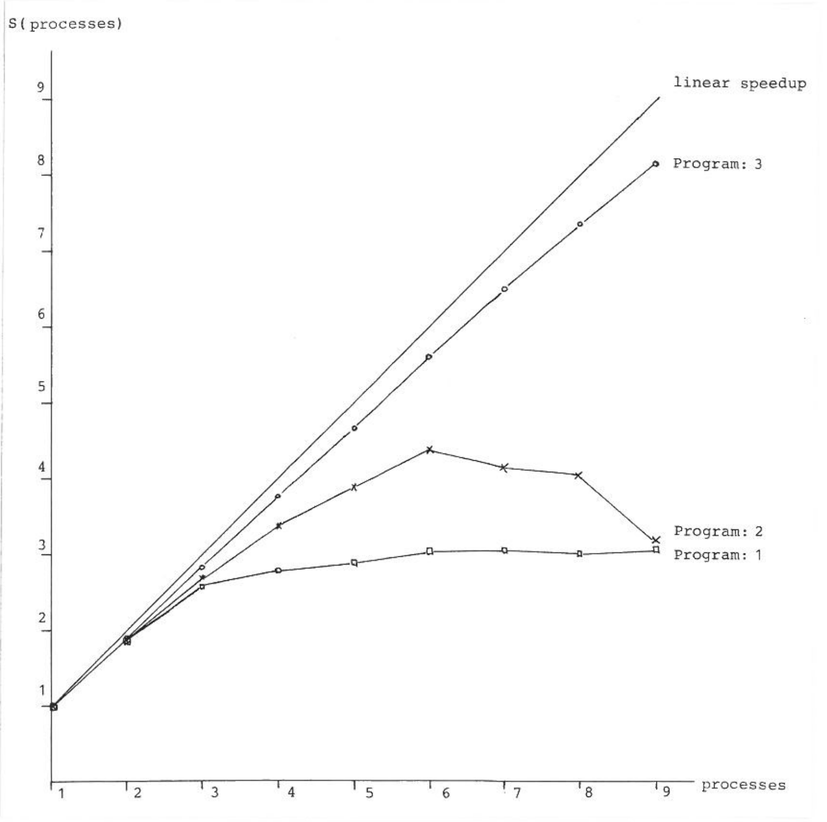


Fig. 8.11

8.9.4 Busy-Waiting Revisited

In section 8.4 we discussed *busy-waiting* as a way of implementing the short-term synchronization. Our conclusion was that the simplest implementation was also the most efficient. This same discussion could be made for the medium-term synchronization: Why not use busy-waiting?

Below is sketched a transcription of a shared class which is quite different from the transcription in section 8.9.2. The delay and continue-statements are substituted by a busy-waiting loop as follows.

```
sh_mon =
MONITOR
    VAR $V_s$: ...;
    FUNCTION ENTRY entry_to_$P_i$: boolean;
        BEGIN
            IF $B_i(V_s)$
                THEN
                    BEGIN
                        entry_statement$_i(V_s)$;
                        entry_to_$P_i$ := true
                    END
                ELSE
                    entry_to_$P_i$ := false
        END;

    PROCEDURE ENTRY exit_from_$P_i$;
        BEGIN
            exit_statement$_i(V_s)$
        END;
BEGIN
    INIT($V_s$)
END;
.
.
.
sh =
COMMON CLASS(...;s:sh_mon);
    VAR $V_d$: ...;

    PROCEDURE ENTRY $P_i(...)$;
        BEGIN
            WHILE not (s.entry_to_$P_i$) DO;
            $S_i(V_d)$;
            s.exit_from _$P_i$
        END
    BEGIN
        INIT($V_d$)
    END;
```

There is no doubt that this is simpler than the transcription in section 8.9.2. The question is whether it is too inefficient? The *sh_mon* will be accessed very frequently. But as it was the case on the short-term level, the fraction of the total running time spent using the shared resource (here the *sh_mon*) is very small. Therefore even a considerable extension of the calls to this monitor is not noticeable. The graph in fig. 8.12 shows the speed-up of program 2 (transcribed as in section 8.9.2) and program 2 transcribed using busy-waiting.

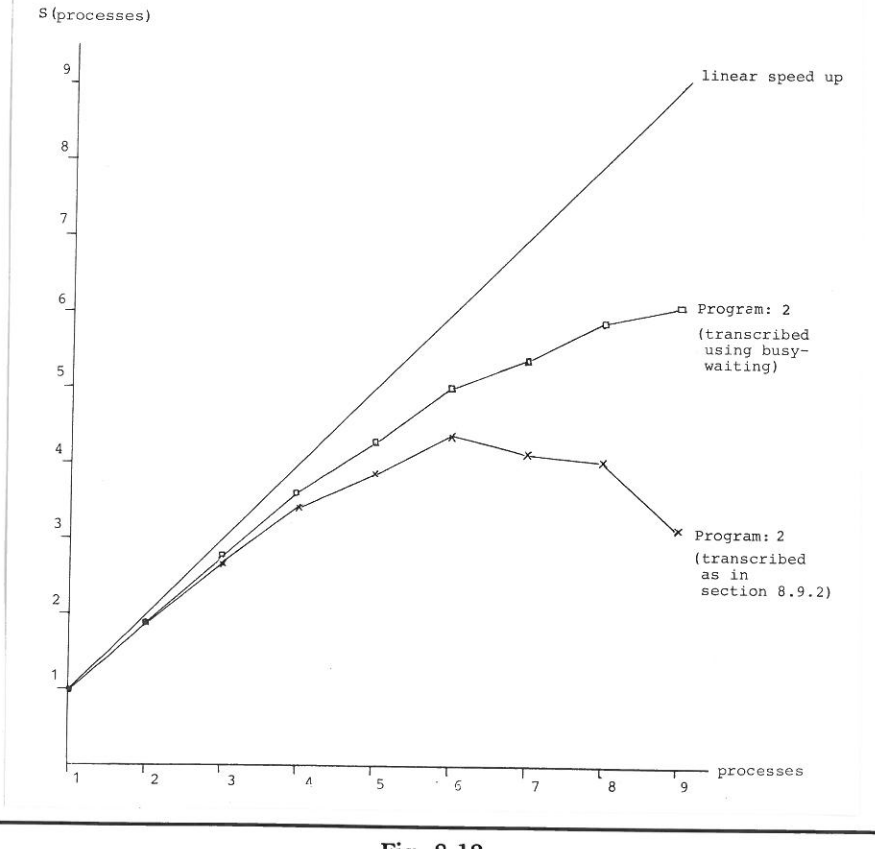


Fig. 8.12

The transcription which uses busy-waiting has the best speed-up. This may be explained as follows. The transcription which uses busy-waiting is inherently faster than transcriptions using delay and continue-statements. An adverse effect is caused by the heavy traffic on the entries of the monitor called *sh_mon*. This traffic will saturate the *sh_mon* for a relatively small number of processes. However, if the time spent executing an entry in the *sh_mon* is small compared to the total time spent in an entry of the common class called *sh*, the relative loss is small for the processes that are doing useful work. Whether a process loses its power executing a busy-waiting loop or is delayed in a queue-variable is immaterial when a static allocation of processors to processes is used. Please note the similarities between the discussions in sections 2.1, 8.4 and this section.

9. History

In January 1980 a pilot study was initiated. A small number of microprocessors were investigated with regard to

- price,
- capacity (primarily cycle time and storage),
- connection possibilities (e.g. common store, bus etc.),
- connection of peripherals,
- suitability for supporting a Pascal like language (Concurrent Pascal, Modula, or Platon)

under the limitation that all hardware had to be commercially available (see chapter 1). This pilot study and its results are described in [Søe and Søgaard 1981] . The study provided us with a knowledge of the available hardware and its capabilities.

During the spring of 1980 the purpose of the laboratory was formulated more precisely: *experiments with implementation and analysis of algorithms utilizing concurrency*. Having this purpose in mind the major design goals were laid down. This led us to choose a hardware system with a common bus structure and to choose Concurrent Pascal as the programming language (see chapter 1). These decisions were made during the summer of 1980. At that point we asked all relevant manufacturers to demonstrate to us a system with:

- a 16-bit processor board connected to
- a 16-bit wide bus (which should not be affected by local computations on the processor boards) and
- some kind of connection to either a main frame host or a cheap development system.

It turned out that the only manufacturer which was able to fulfil these requirements was Intel. Since there were no important deficiencies with the Intel system, we chose the Intel 8086 as the cornerstone of the multiprocessor. On September 18, 1980 the following equipment was delivered:

- two Intel iSBC 86/12A processor boards,
- an Intel iCS 80 chassis with power supply,
- an Intellec 230 development system.

During the fall, work was started on the Concurrent Pascal implementation and a communication channel between the Computer Science Department's PDP-10 and the Intellec system was established. The communication channel was established by writing a program which made the Intellec appear as a terminal to the PDP-10. Using this, files could be transmitted between the development system and the PDP-10 file system (the speed was 2400 baud).

In December 1980 the basic control structures and arithmetic operations of Concurrent Pascal were implemented, so that an initial process without classes, monitors or other processes could be executed. Two months later, all the modifications in the compiler and interpreter due to the distribution of object code were completed. Then followed the implementation of the synchronization primitives and the initialization of a program with several processes and monitors. Independently of this the type real was implemented.

The entire implementation was finished around May 1, 1981. At that point all language constructs (with the very few exceptions mentioned in [Staunstrup 1982] were implemented, but still the system had only two processors.

The initial configuration of the system with only two processors was sufficient to test the Concurrent Pascal implementation, but it was clearly not sufficient for the real purpose of the project: »experiments with multiprocessor algorithms«. When we became confident that the Concurrent Pascal implementation would work, we ordered seven more processors (iSBC Intel 86/12A boards) and a 32K byte memory board. These were delivered in May and June. This expansion required two hardware changes:

(1) the bus arbitration was changed (see section 7.3);

(2) the power supply was changed together with some mechanical rearrangements.

These hardware changes were completed in June 1981. During the late spring and summer, processor boards were gradually added to the system. This revealed a few hardware problems (malfunctioning CPU's and EPROM sockets) and some shortcomings in the user interface. After correcting these, the entire system became available to students and faculty in August 1981.

During the implementation period parts of the system were used for preliminary experiments. In the spring of 1981 the first graduate course using the system was offered. This was an introduction to the system and to PL/M-86. Each participant wrote a multiprocessor algorithm in PL/M-86 and made some speed measurements on this program.

Since 1981 we have given a number of graduate courses, in which the students have used the Multi-Maren multiprocessor. The exact contents of these courses have varied but they always include small projects where the student run various experiments on the Multi-Maren machine. Below is a list of these courses with the subjects of the student projects. All the written reports from these projects are in Danish.

Spring 1981: **Multiprocessor Algorithms**
P. Møller-Nielsen and J. Staunstrup

Projects:
A problem-heap algorithm for numerical quadrature
E. Knudsen and J.O. Graulund
A parallel line-clipping algorithm
K. Nielsen and E. Sandvad
Comparison of message-passing and monitor communication
T. Hansen and L. Dahl

Fall 1981: **Multiprocessor Algorithms**
P. Møller-Nielsen and J. Staunstrup

Projects:
Quicksort
E. Frederiksen, M. Hagner, K. Hansen and L. Karlsen
Fixpoint calculation
E. Bobach and C. Johansen
Knights' tour
A. Andersen and G. Kjær-Nielsen
Rootsearching using interval-arithmetic
S. Top
A four-processor root-searching algorithm
O. Eriksen

Spring 1982: **Software for a Multiprocessor**
P. Møller-Nielsen and J. Staunstrup

Projects:
Alternative primitives for medium-term synchronization
K.G. Hansen, P.J. Jensen and B. Munk
Distribution of storage references
B. Poulsen and L. Geisler
Fair synchronization
K.B. Nielsen and K.I. Grøn
Tuning the c-code interpreter
P.B. Hansen
Adding interval arithmetic
B.G. Thams and F. Terpling
Tools for measuring running and waiting times
K. Gregersen and P.T. Jensen

Spring 1983: **I/O on a Multiprocessor**
P. Møller-Nielsen and O. Caprani

Projects:
A communication system between a graphical workstation and Multi-Maren
C. Nørgaard
Implementing device monitors in the Multi-Maren c-code
V. Tøttrup Jensen, H.M. Kristensen, P.H. Larsen and P.F. Thøstesen
Implementing device monitors in the Concurrent Pascal compiler
I. Bohlbro, P.H. Kristiansen and M. Schwartzbach
Connecting a disc to Multi-Maren
J. Katborg, J.V. Ludvigsen and C.R. Pedersen
Extending Concurrent Pascal on Multi-Maren to include several processes on a single processor
K.B. Munkholm, F. Barrett, J. Stausgaard and G. Andersen

Fall 1983: **Multiprocessor Algorithms, in particular the problem-heap paradigm**

No written projects.

Spring 1984: **Improving the I/O on Multi-Maren**
Peter Møller-Nielsen and O. Caprani

Projects:
Tuning the c-code interpreter which includes device monitors
B. Kristensen, L. Bendix, L. Christensen and S. Hvidbjerg
Storage allocation problems in Multi-Maren
K.E. Johansen, U.L. Jørgensen, H. Lauritsen and S.E. Nielsen

Master's Theses

Design and Implementation of the Multi-Maren Multiprocessor. May 1982.
Kurt Søe and Jens S. Andersen
(Advisors: P. Møller-Nielsen and J. Staunstrup).

Comparison of Communication Primitives on a Multiprocessor. October 1982.
Torben Hansen
(Advisor: J. Staunstrup).

Concurrent Evaluation of the Eigenvalues of a Real, Symmetric and Tridiagonal Matrix. November 1982. (In English.)
Kell Häuser
(Advisors: J. Staunstrup and O. Caprani).

Multiprograms for Branch and Bound Algorithms. January 1983.
Axel Andersen
(Advisor: P. Møller-Nielsen).

A Multiprocessor Database-machine. June 1983.
Ole V. Johansen and Jens O. Jespersen
(Advisor: J. Staunstrup.)

A Parallel Implementation of a Functional Language. January 1983.
C. Paulsen
(Advisor: Neil Jones.)

Connecting a Multiprocessor to a Local Network (ETHERNET). April 1984.
K. Glassau Hansen
(Advisors: O. Caprani and P. Møller-Nielsen.)

10. References

[Aho, Hopcroft and Ullman 1974]

The Design and Analysis of Computer Algorithms, A.V. Aho, J.E. Hopcroft and J.D. Ullman, Addison-Wesley, 1974.

[Andersen 1983]

Multiprogrammer til »Branch and Bound« algoritmer, A. Andersen, Computer Science Department, Aarhus University, January 1983.

[Boyer and Moore 1977]

A Fast String Searching Algorithm, R.S. Boyer and J.S. Moore, Communications of the ACM, vol. 20, no. 10, 1977.

[Brinch Hansen 1973]

Operating Systems Principles, P. Brinch Hansen, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1973.

[Brinch Hansen 1977]

The Architecture of Concurrent Programs, P. Brinch Hansen, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1977.

[Dijkstra 1983]

Derivation of a Termination Algorithm for Distributed Computation, E.W. Dijkstra and C.S. Scholten, Information Processing Letters, Vol. 16, 1983.

[Eriksen and Staunstrup 1983]

Concurrent Algorithms for Root Searching, O. Eriksen and J. Staunstrup, Acta Informatica, Vol. 18, pp. 361-376, 1983.

[Galil 1979]

On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm, Communications of the ACM, Vol. 22, No. 9, 1979.

[Grit 1983]
Programming Divide and Conquer on a Multiprocessor, D.H. Grit and J.R. McGraw, Lawrence Livermore National Laboratory, University of California, May 1983.

[Hartmann 1977]
A Concurrent Pascal Compiler for a Minicomputer, A.C. Hartmann, *Lecture Notes in Computer Science, Vol. 50*, Springer Verlag, 1977.

[Horspool 1980]
Practical Fast Searching in Strings, R.N. Horspool, Software Practice and Experience, Vol. 10, 1980.

[Intel 1978a]
iSBC 86/12 Single Board Computer Hardware Reference Manual, Intel Corporation 1978.

[Intel 1978b]
PL/M-86 Programming Manual for 8080/8085-based Development System, Intel Corporation 1978.

[Intel 1978c]
iSBC 957 Intellec – iSBC 86/12 Interface and Execution Package, User's Guide, Intel Corporation 1978.

[Intel 1979]
The 8086 Family User's Manual, Intel Corporation 1979.

[Jones 1980]
The Cm Multiprocessor Project: A Research Review*, A.K. Jones and E.F. Gehringer (editors), Computer Science Department, Carnegie-Mellon University, July 1980.

[Lai 1984]
Anomalies in Parallel Branch and Bound Algorithms, T.H. Lai and S. Sahni, Communications of the ACM, Vol. 27, No. 6, 1984.

[Little 1963]

An Algorithm for the Travelling Salesman Problem, J.D.C. Little et al., Operations Research 11, 1963.

[Møller-Nielsen and Staunstrup 1982]

Early Experience from a Multi-processor Project, P. Møller-Nielsen and J. Staunstrup, Computer Science Department, Aarhus University, DAIMI PB-142, January 1982.

[Møller-Nielsen and Staunstrup 1983]

COMMON CLASS - a tool for programming the access to shared data, P. Møller-Nielsen and J. Staunstrup, Computer Science Department, Aarhus University, DAIMI PB-170, December 1983.

[Møller-Nielsen and Staunstrup 1984]

Experiments with a Fast String Searching Algorithm, P. Møller-Nielsen and J. Staunstrup, Inf. Proc. Letters 18, March 1984.

[Queille and Sifakis 1983]

Fairness and related properties in transition systems - A temporal logic to deal with fairness, J.P. Queille and J. Sifakis, Acta Informatica, Vol. 19, Fasc. 3, 1983.

[Ravn 1980]

Device Monitors, A.P. Ravn, IEEE Transactions on Software Engineering, Vol. 6, No. 1, 1980.

[Rice 1976]

Parallel Algorithms for Adaptive Quadrature III, Program Correctness, J.R. Rice, ACM Transactions on Mathematical Software, Vol. 2, No. 1, March 1976.

[Sedgewick 1978]

Implementing Quicksort Programs, R. Sedgewick, Communications of the ACM, Vol. 21, No. 10, 1978.

[Staunstrup 1982]

Concurrent Pascal at DAIMI, User's Guide, J. Staunstrup, Computer Science Department, Aarhus University, August 1982.

[Søe and Søgaard 1981]
Undersøgelse af 16-bits mikroprocessorkort, K. Søe and J. Søgaard Andersen, Computer Science Department, Aarhus University, October 1981.

[Valiant 1975]
Parallelism in Comparison Problems, L.G. Valiant, SIAM Journal on Computing, Vol. 4, No. 3, 1975.

[Wilkes 1977]
Beyond Today's Computers, M. Wilkes, IFIP-77, North-Holland, 1977.