ISSN 0105-8517

Problem-heap: A Paradigm for Multiprocessor Algorithms

Peter Møller-Nielsen
Jørgen Staunstrup

DAIMI PB - 200
October 1985

DATALOGISK AFDELING
Bygning 540 - Ny Munkegade - 8000 Aarhus C
tlf. (06) 12 83 55, telex 64767 aausci dk
Matematisk Institut Aarhus Universitet

Problem-heap: A Paradigm for Multiprocessor Algorithms

Peter Møller-Nielsen and Jørgen Staunstrup [1]
Computer Science Department
Aarhus University
Ny Munkegade
DK-8000 Aarhus C
Denmark

October 8, 1985

Abstract:

The problem-heap paradigm has evolved through four years of experiments with the Multi-Maren multiprocessor. Problem-heap algorithms have been formulated for a number of different tasks such as numerical problems, sorting, searching and optimization. Although these tasks are very different, the analyses of the running times of all the problem-heap algorithms are very similar. The problem-heap paradigm is illustrated by algorithms which have been implemented and analyzed using the Multi-Maren multiprocessor.

1 Introduction

A multiprocessor is capable of simultaneously executing several parts of an algorithm. Therefore, it has the potential of executing the algorithm faster than a conventional (mono-)processor. This potential can be exploited only if algorithms can be found which takes advantage of the multiprocessors ability to execute many parts simultaneously, such algorithms we call **multiprocessor algorithms**. This paper describes a particular class of such multiprocessor algorithms, called **the problem-heap algorithms**. Many different multiprocessor architectures are currently being studied, a recent overview of experimental multiprocessors is given in [Architecture 1984]. There are considerable differences between the architectures of these machines. Instead of centering on the architectural details of multiprocessors,

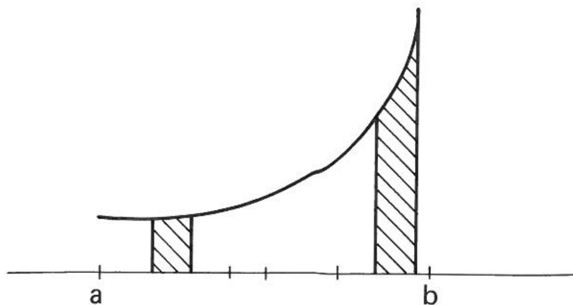
[1] This paper was written while Jørgen Staunstrup was visiting The Computer Science Department at The University of Washington, Seattle.

we want to focus on *their algorithms*. A multiprocessor algorithm is designed explicitly for a machine with many processors. Other efforts are aimed at automatically finding the potential concurrency in an algorithm. Although this approach has advantages, we find it necessary to first get more knowledge and experience with the (manual) design and analysis of multiprocessor algorithms.

In this paper we assume a multiprocessor with a number of asynchronous processors, each of which is capable of communicating with all other processors (maybe indirectly). Each processor is assumed to consist of a traditional processing unit (e.g. a microprocessor) and a local store. The processors are assumed to be almost identical, in particular they should run with roughly the same speed. For multiprocessors where this is not the case, special consideration must be given to allocating time consuming parts to the fast processors. This may complicate the algorithms considerably.

There is already a quite extensive knowledge on two classes of multiprocessor algorithms: the *vector algorithms* and the *pipeline algorithms*. Both of these classes were originaly intended for particular machine architectures (vector and pipeline processors respectively). They are, however, also very useful on an asynchronous multiprocessor. Vector algorithms simultaneously execute a particular operation on a number of different data elements (a vector). Pipeline algorithms work in the same ways as an assembly line; the algorithm is split into a number of consecutive stages, each of which does a different part of the algorithm. Data progress through these stages, and is therefore gradually transformed into the result. This gives a speed improvement only if many data sets are processed in succession.

Both the vector algorithms and the pipeline algorithms may be characterized as *static*. With vector algorithms the data is statically divided into a fixed number of vector elements, and in the pipeline algorithms there is a fixed number of stages. Although there are many examples where such static algorithms give very good results, there are also cases where they are not appropriate. Consider the simple example of computing the quadrature of a function F over some interval [a, b], see Figure 1. The quadrature can be computed by making a static division of the interval into a number of equally sized intervals, one for each processor. This may, however, be a very inefficient approach, since some of the intervals (e.g. where F rises steeply) require a lot of work to compute the quadrature with the desired accuracy, whereas other intervals (e.g. where F is flat) require little work. Hence some processors will finish very quickly and be idle, while the rest complete their



PAC [078] fig. 1.

work. When the difficulty of the parts cannot be predicted, a static division of the work is inadequate, therefore it is necessary to dynamically allocate the processing power to the parts where it is most needed.

In this paper, we describe a class of dynamic multiprocessor algorithms called the *problem-heap algorithms*. This class has evolved through our experiments with the Multi-Maren multiprocessor. A number of multiprocessor algorithms has been implemented on this machine. Many of these algorithms exhibit a very good performance, but some do not. In the latter cases, we have always found that *shortcomings in the algorithms and not hardware bottlenecks is what limits performance*. The experimental results given in this paper all stem from experiments on Multi-Maren.

2 Running time analysis

Most of our running time analyses have concentrated on finding the speed-ups of problem-heap algorithms. Let $T(N)$ be the time it takes to perform a certain algorithm using N processors (the interpretation of $T(1)$ is discussed further below). The speed-up is defined as follows:

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

This measure has been studied in a number of other multiprocessor projects and although it is quite simple, it has two nice properties which are essential to our analyses:

- it is *processor and implementation independent*. Our results are not very sensitive to the particular processor and programming language we are using.

- it directly answers the key question of our work: is the processing power being utilized?

A speed-up which is proportional to N is called *optimal.* This is an upper bound on the achievable speed-up, since any algorithm which can be executed in time T on a N-processor machine, can be executed in time $N \times T$ on a mono-processor. This is of course only true if the amount of work done by the two machines is the same. Consider, for example, an algorithm for finding any occurrence of a pattern in a string (if there are several occurrences any one of them will do). When solving this problem, the amount of work done can vary radically. Since any occurrence is a solution, some executions will pick a substring containing an occurrence on the first try and hence do very little work. Most executions will not be so fortunate, and therefore need to search a large part of the string before an occurrence is found. For this algorithm one can obtain almost arbitrarily large and arbitrarily small speed-ups. Such results have been reported by [Wilkes 1977] and [Lai and Sahni 1984]. To avoid these anomalies the speed-up analysis should be done on the basis of executions, where the amount of work done using N processors and using one processor is the same, i.e. for the searching problem, the same parts of the string should be searched.

Another subtle point in the definition of speed-up is the quantity $T(1)$. Usually the running time of a multiprocessor algorithm when executed on a mono-processor $T(1)^+$ is *not* the same as the running time $T(1)$ of a sequential algorithm doing the same task. $T(1)^+$ is larger because of various kinds of overhead. In the definition of speed-up used above, the running time of the sequential algorithm, $T(1)$, is used.

When an algorithm does not achieve optimal speed-up, it is because some processing power is lost. There can be two reasons for this: one is that the algorithm does not manage to keep all processors busy with useful work, this is a deficiency of the algorithm, let S_{loss} be the total time lost because of this. The other possibility is hardware phenomenon such as bottlenecks or communication delays, let H_{loss} denote the total time lost because of this. These two should be distinguished and analyzed separately.

$$T(N) = \frac{T(1) + S_{loss} + H_{loss}}{N}$$

On the multiprocessor used for our experiments, *no significant hardware bottlenecks have been encountered.* It is a 10 processor machine with a common bus architecture. The major reason no hardware bottlenecks have

been observed is that all programs (code) is kept in a store local to each processor [Møller-Nielsen and Staunstrup 1983]. Therefore H_{loss} is ignored in the rest of this paper.

In a number of cases we have found a loss due to the algorithm, i.e. a significant S_{loss}. By analyzing this in further detail, we have found that S_{loss} can be broken into a small number of different components: *starvation loss, braking loss, separation loss,* and *saturation loss*. Deviations from an optimal speed-up are explained as a combination of these four kinds of loss, usually only one or two of them is dominant.

There may of course be other sources of loss that we have not yet identified. But even if one or two more sources are added to the list, it is still quite short. This is important for analyzing the performance of problem-heap algorithms, which we suggest is done by estimating each of the four kinds of loss. It is our experience that this analysis is almost the same for each new problem-heap algorithm. What may differ is the magnitude of each of the sources and thereby its significance for the overall performance of that algorithm.

Other classifications of loss has been attempted, e.g. by explaining the deviation from optimal speed-up as communication or synchronization overhead[Oleinick 1982]. With such a coarse identification of the loss, it can be very difficult to understand whether it is caused by the algorithm, the hardware, or the systems software. By making a more thorough distinction between different types of software loss we try to shed some light on the behavior of the algorithm.

3 Problem-heap 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. Consider, for example, the task of finding *all* occurrences of a particular pattern in a string. This task can be done by splitting the string into a number of equally sized substrings and then letting each processor search one of these. When such a *static splitting* is possible, it is straightforward to get a simple multiprocessor algorithm. But when a static splitting is not possible, or when it does not give a satisfactory performance, a *dynamic* splitting must be considered. The quadrature algorithm mentioned in the introduction is an example, where a dynamic

splitting is necessary. Further examples are given below.

The problem-heap algorithms described in this paper is a simple class of such dynamic algorithms. The main characteristics of these algorithms are:

- *a number of identical and asynchronous processes* cooperate on doing a certain task. These processes share one or more data structures that represent partial results and those parts of the task that yet remain, this data structure is called:

- *a problem-heap* describing the remaining subtasks called *problems*. A process takes a problem from the problem-heap and tries to solve it. This may generate new problems which are then put back in the problem-heap, or the problem may be simple enough that it can be solved immediately.

One 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 is done (all problems solved) no matter how few or how many processes that are used. What may vary is the speed with which the task is done. Another major asset is that we have found a recurring pattern in the analysis of the performance of problem-heap algorithms by estimating the magnitude of the four sources of loss mentioned above.

3.1 Skeleton algorithm

All processors of a problem-heap algorithm execute the same process. Furthermore, there is a common skeleton of this process which is found in all problem-heap algorithms:

```
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
```

The primitives used in the skeleton will of course vary.

3 PAC 0782

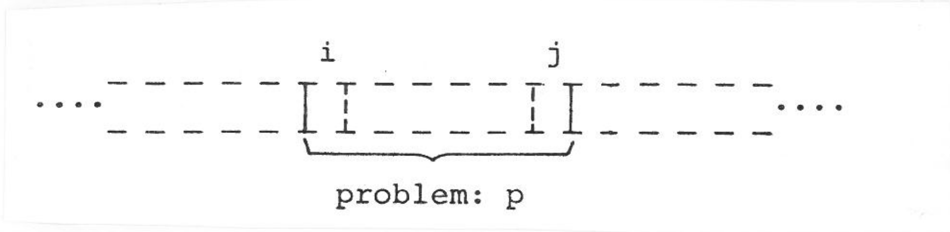


Figure 2: Snapshot of array during a Quicksort

75)

3.2 Example: Quicksort

A well known example of a so-called divide and conquer algorithm is Quicksort [Hoare 1962], which sorts a list of numbers represented in an array. Often Quicksort is presented as a recursive procedure; but as all other divide and conquer algorithms it can easily be formulated as a problem-heap algorithm. The following algorithm describes the process executed by all processors:

```
CYCLE
  p:= a problem from the heap
  IF size(p) = 1
     THEN report p is sorted
     ELSE split p into p1 and p2 such that: p = p1 p2  and
                   all x in p1 are smaller than all y in p2
          put p1 and p2 in the heap
END
```

In this formulation, a problem, p, is an index range $i..j$ of the array, see Figure 2. To solve p, all numbers in p must be sorted and put back in the same index range $i..j$ of the array. Quicksort is chosen as an example because it is a well known algorithm; it actually has a rather poor performance on a multiprocessor, see section 4.1.

3.3 The problem-heap

The problem-heap is a data structure accessible by all processes. It can be viewed as a source of data which supply all processes in the problem-heap algorithm with work. When the problem-heap is drained the processors starve and their processing power is lost. The structure of the problem-heap is not nearly as uniform as was the case with the processes. 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 all depends on the task to be solved. The heap is initialized with one problem, the *initial problem* describing the task to be done.

The divide and conquer approach is an obvious way of breaking a task into subtasks, which can then be done by the independent processors of a multiprocessor. Several such algorithms were implemented on Cm*[Jones and Gehringer 1980]. More recently the Crystal multicomputer [DeWitt, Finkel and Solomon 1984] has been used to experiment with problem-heap algorithms where the heap is distributed over many processors [Finkel and Manber 1985]. The problem-heap is similar to the so-called ask for monitors[Lusk and Overbeek 1983].

4 Software loss

In this section we describe the four sources of loss that has been identified in our experiments.

4.1 Starvation loss

The problem-heap must constantly supply all processes with problems. When there are too few problems in the heap the workless processes starve and processing power is lost, this is illustrated in Figure 3. In most problem-heap algorithms there is some starvation in the initial phase until all processes have gotten their first problem to work on. Usually 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.

Although starvation is usually found in the beginning and in the end of a computation, it may of course happen at any point in between. Let T_i be the times defined by the diagram shown in Figure 3. The loss caused by this starvation phase is:

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

So k-1 processes each lose T_{k-1}, k-2 processes each lose 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.

3 PAC 078 3

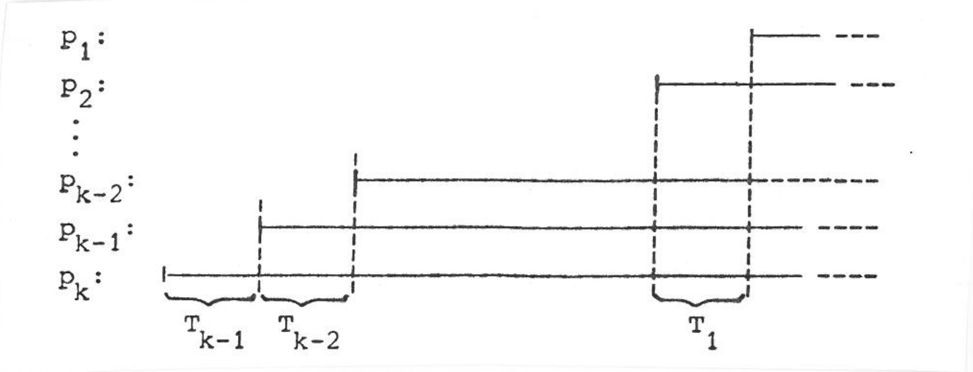


Figure 3: Starvation phase

751

The problem-heap implementation of Quicksort exhibits a very clear example of starvation loss. The first step of a Quicksort consists of splitting the data into two (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 in the list, M. During this period one processor only is working. After this, two can get to work on splitting the two problems into four, etc., see Figure 4. The magnitude of the starvation loss is (assuming N is a power of two):

$$St_{loss} = \sum_{j=0}^{logN-1} (N - 2^j) \times \frac{M}{2^j}$$

For moderate values of M, this is the dominant source of loss. Even if the array to be sorted is copied back and forth between the local stores, the loss caused by this copying is negligible compared with the starvation loss. The speed-up we have observed is shown below. If the expression given for St_{loss} above was exact we would expect:

$$T(N) = \frac{T(1) + St_{loss}}{N}$$

which may be rewritten as:

$$N = S(N) + \frac{St_{loss}}{T(N)}$$

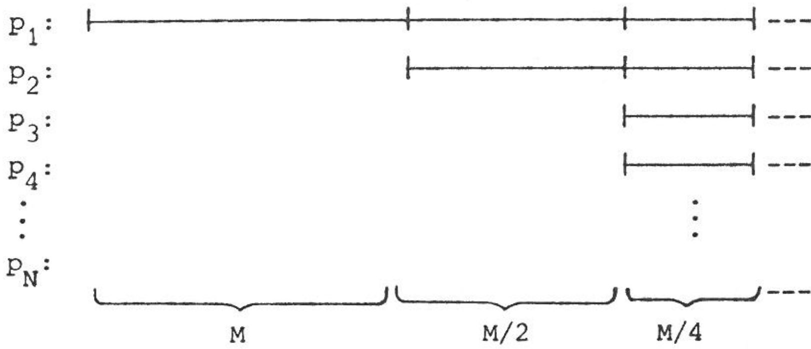


Figure 4: Starvation in Quicksort

3PAC 078 4

Our measurements of $\frac{St_{loss}}{T(N)}$ has been very crude, so this number is given with one significant digit only in the table below. The numbers stem from an experiment where 1000 integers placed in the common store were sorted.

N	$S(N)$	$\frac{St_{loss}}{T(N)}$
2	1.8	0.2
4	2.8	1
8	3.6	4

This table confirms that the starvation loss explains the speed-up observed for Quicksort. The Quicksort algorithm has also been studied on the $Cm^{\star}$ [Jones 1980] with essentially the same conclusions.

4.2 Braking loss

When a process receives a problem to solve, it either solves it completely or splits it before consulting the problem-heap again. In both cases there is a period when it works in isolation on a particular subproblem (solving or splitting); if during this period the entire task is completed, the process is not stopped until it again consults the problem-heap. Hence the process does some superfluous work; the loss caused by this is called *braking loss* .

75) 3 PAC 078 5

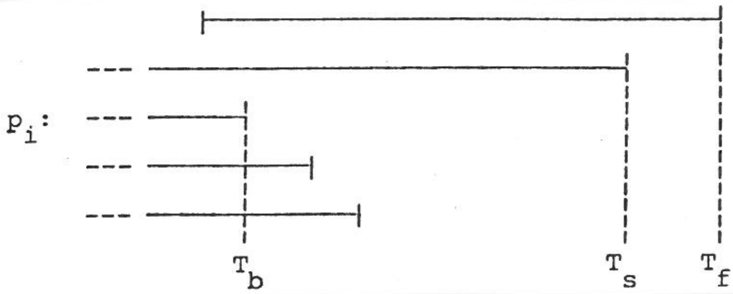


Figure 5: Time diagram showing braking loss

To be more precise, consider the diagram in Figure 5 illustrating the final phase of a computation. The three time instances introduced in Figure 5 are defined as follows:

T_b: The first potential solution to the complete task is found by process p_i. Other processes working on other subproblems finishing later might find other improved solutions. But no process is given any new subproblems 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 are stopped, so no more subproblems are solved.

These three time instances are not always distinct. Consider the problem of finding the *leftmost* occurrence of a pattern in a string. 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 farther to the left. So T_s is the time instance where an occurrence is 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, while T_s is still distinct from T_f.

Braking loss is defined as the processing power lost between T_s and T_f. This is indicated by the crossed periods in the time diagram shown in Figure 6. Although braking loss is most frequently found at the end of the execution of an algorithm, there are also examples of algorithms where the solution of one problem suddenly makes other problems in the heap superfluous.

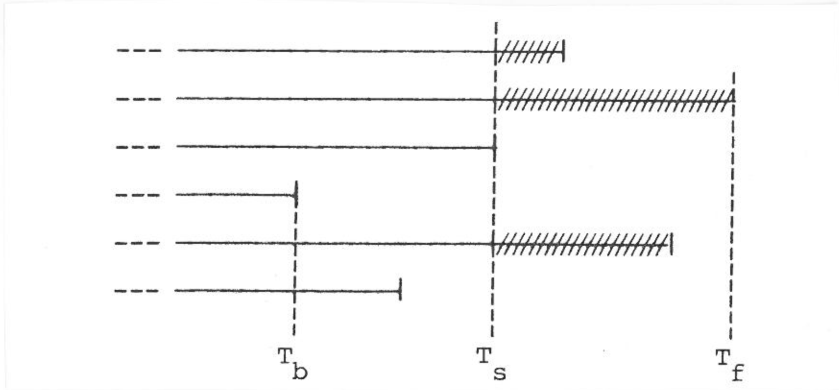


Figure 6: Time diagram showing braking loss in string search

3 PAC 0786 75

We have observed this in a root-searching algorithm based on the same principle as the well-known bisection algorithm. When one process locates an interval containing the root, all other problems (intervals) in the heap becomes irrelevant. All processes should brake and turn their attention to the interval now known to contain the root.

Consider again the string searching algorithm where the leftmost occurrence of a pattern must be found. Figure 6 illustrates the braking phase of this algorithm. To estimate the magnitude of the braking loss, the hatched periods must be estimated. We have done this both analytically and experimentally, the analytical estimate is based on a simple application of order statistics [Feller 1970], for further details please see [Møller-Nielsen and Staunstrup 1984]. The experimental results are shown in section 4.3.

4.2.1 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. 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. We have not tried to use interrupts in any of our

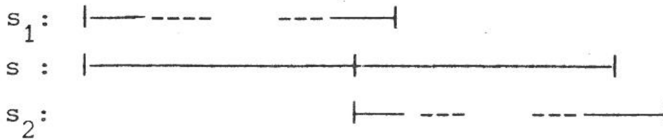


Figure 7: Separation loss in string searching algorithm

3
75

PAC 078 7

algorithms, since this would complicate them significantly.

4.3 Separation loss

All problem-heap algorithms are iterative (see the skeleton in section 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, this size is sometimes referred to as the *granularity* of the problem. With a larger granularity one may risk starvation loss as described in section 4.1, see also section 5. On the other hand, there is an overhead associated with splitting a problem. Solving two subproblems separately may require more work than solving them together as one problem. This overhead is called *separation loss* .

Consider again the task of finding the leftmost occurrence of a pattern in a string. By splitting the 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 Figure 7. 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. So, the separation loss is the extra work introduced by breaking the task into subproblems. A small fraction of this may be overhead associated with administering the heap, but the significant part is the extra work introduced, e.g. the characters that have to be inspected twice.

The separation loss grows with the number of iterations of the algorithm. When the loss in each iteration is the same; it is quite easy to estimate the total separation loss, but when this is not the case, it may be more difficult. The algorithm for finding the leftmost occurrence of a pattern in a string has some separation loss which we have estimated both analytically

and experimentally. It can be expressed as follows:

$$Sep_{loss} = i \times \frac{C}{D}$$

C is a constant which depends on details of the implementation, D is the number of characters in a substring(problem), and i is the position of the first character of the occurrence, which in this case is proportional to the number of iterations.

The table given below shows the running time of the string searching algorithm (for finding the leftmost occurrence). The measured running time is compared with the predicted running time obtained by using the formula:

$$T(N) = \frac{T(1) + Sep_{loss} + Br_{loss}}{N}$$

Where Br_{loss} is the braking loss and Sep_{loss} the separation loss. The numbers stem from an experiment where a string of 10000 characters were searched for an occurence of a pattern 10 characters long.

N	Sep_{loss}	Br_{loss}	Predicted $T(N)$	Measured $T(N)$	$S(N)$
4	100	216	978	981	3.7
8	200	248	506	516	7.0

[2]

4.4 Saturation loss

In our experiments the problem-heap was a global data structure which all the processes could reference, but only one at a time (mutual exclusion). When two processes try to reference the problem-heap at the same time, one of them must wait which introduces *saturation loss*. Let G_1 and G_2 be the times it takes for the two processes to complete their references to the problem-heap, the diagram in Figure 8 illustrates the loss they can experience. This kind of loss may happen at each reference to the problem-heap, i.e. once in each iteration of all the processes, see the skeleton in section 3.1. Furthermore, the saturation loss usually grows with the number of processes. To analyze the saturation loss we have used traditional techniques from queueing theory. Although the emphasis there has been on

[2] A different value of D was used to obtain $T(4)$ and $T(8)$.

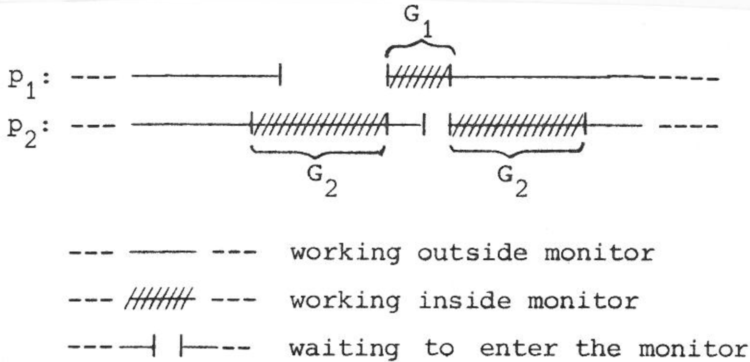


Figure 8: Time diagram illustrating saturation loss

PAC 078 8

hardware saturation, which is not significant in any of our experiments, the same techniques and results can be applied to software saturation (saturation loss).

Note, that there is a distinction between starvation loss and saturation loss. When processes attempt to get problems in or out of the heap, there may be some loss because they need mutually exclusive access to the heap, and therefore need to wait for each other. In contrast, the starvation loss 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.

By subdividing the heap one can reduce the saturation loss, but usually at the expense of increasing one or more of the other sources of loss, e.g. starvation loss [Møller-Nielsen and Staunstrup 1983].

5 Granularity

An important issue to consider when designing a multiprocessor algorithm is its **granularity**, i.e. what is the proper size of the subtasks which are performed in parallel. If too fine a granularity is used there might be a significant separation or saturation loss. On the other hand a very coarse division can give starvation or braking loss. We cannot offer any general rule for making the compromise between the two. But an analysis of the four sources of loss will show which is the dominant kind. If it is separation

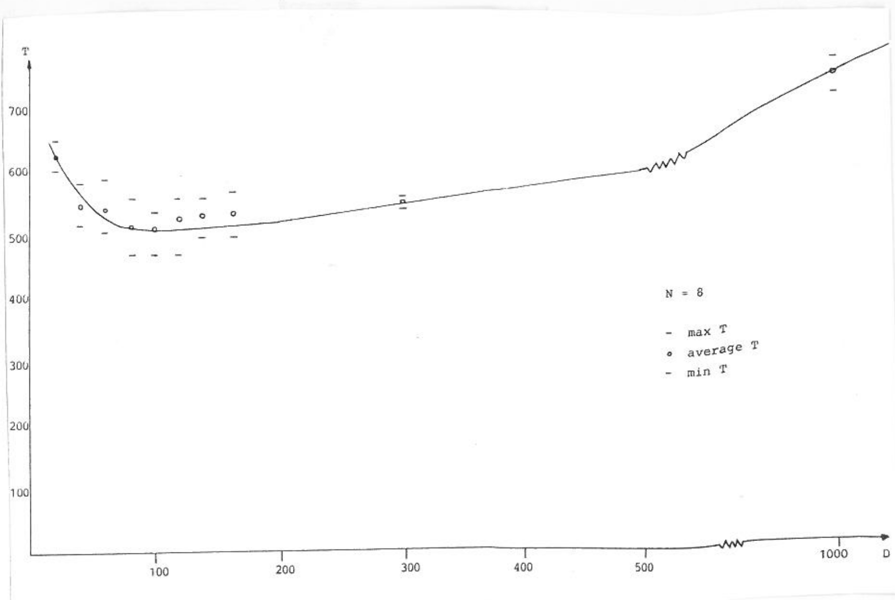


Figure 9: Loss as a function of the subproblem size D

3
PAC 078 9
110/

loss a larger granularity should be attempted and conversely if braking or starvation loss dominates.

As an example, consider again the string searching algorithm. Here the granularity is determined by the size, D, of the substrings in the problem-heap. The braking loss decreases with D, whereas the separation loss grows. In Figure 9 it is shown how very small and very large values of D increases the total loss. Fortunately there is a very wide range of D values where the loss is close to its optimal value. The diagram in Figure 9 stem from an experiment where a string of 10000 characters where searched for an occurence of a 10 character pattern. The size of D (x-axis) is given in characters.

6 Summary

During the four years we have experimented with the Multi-Maren multiprocessor a number of algorithms have been tried. Initially, we picked algorithms more or less at random, but after a period the problem-heap algorithms emerged. After this a majority of the algorithms we have implemented has been problem-heap algorithms. So far, problem-heap algorithms have been written for the following tasks: quadrature, root-searching, fixpoint calculation, string searching, sorting, optimization (branch and bound), in-

terpretation (Prolog and SASL). There is no reason to believe that this list cannot be extended, it is given here to indicate the variety of the tasks we have experimented with.

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 we have found 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 be welcomed.

We do *not* claim that the class of problem-heap algorithms is universal. There are other clearly distinct classes of multiprocessor algorithms, e.g. the vector and pipeline algorithms. But the problem-heap algorithms have some nice properties as they can be performed by:

- *an arbitrary number of processes.* The algorithm does not have to be rewritten when the number of processes is increased or decreased. This means that extra processing power can be added without changing the algorithm (the program text).

- *asynchronous processes.* The algorithm can be executed by processes running with different and varying speeds.

- *symmetric processes.* All processors execute the same algorithm (code). This is much easier to handle than writing a different algorithm for each processor.

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, otherwise the slowest process creates a bottleneck.

Acknowledgments

Many of our colleagues and students at the Computer Science Department at Aarhus University contributed to the Multi-Maren laboratory. Brian Koblenz and Eric Jul made many corrections to the Danglish of earlier version of this paper. We are grateful to Paul Frederickson, Los Alamos National Laboratory for his encouragement in preparing this paper.

References

[Architecture 1984:] **Computer Architecture Technical Committee Newsletter** IEEE Technical Committee on Computer Architecture, February 1984.

[DeWitt, Finkel and Solomon 1984:] **The CRYSTAL Multicomputer: Design and Implementation Experience**, Technical Report 553, Dept. of Computer Science, University of Wisconsin-Madison, September 1984.

[Finkel and Manber 1985:] DIB- A Distributed Implementation of Backtracking, **Proceedings from Fifth International Conference on Distributed Computing Systems**, Denver, May 1985.

[Feller 1970] **An Introduction to Probability Theory and its Applications II**, W. Feller, Wiley, New York 1970.

[Hoare 1962:] Quicksort, **Computer Journal** 5, 1, 1962.

[Jones and Gehringer 1980:] **The** $CM^{\star}$ **Multiprocessor Project: a Research Review** , A. K. Jones and E. F. Gehringer (editors), Computer Science Department, Carnegie-Mellon University, July 1980.

[Lai and Sahni 1984:] Anomalies in parallel Branch and Bound Algorithms, T. H. Lai and S. Sahni, **Comm. ACM 27**, 6, 1984.

[Lusk and Overbeek 1983] **Implementation of Monitors with Macros: A Programming Aid for the Hep and other Parallel Processors**, E. L. Lusk and R. A. Overbeek, ANL-83-97, Argonne National Laboratory, Argonne, Illinois, 1983.

[Møller-Nielsen and Staunstrup 1983:] Saturation in a Multiprocessor, P. Møller-Nielsen and J. Staunstrup, **Proceedings from IFIP 83**, North Holland 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.

[Oleinick 1982], **Parallel Algorithms on a Multiprocessor**, P. N.

Oleinick, UMI Research Press, Ann Arbor, MI, 1982.

[Wilkes 1977:] Beyond today's Computers, M. Wilkes, **Proceedings from IFIP 77**, North Holland 1977.