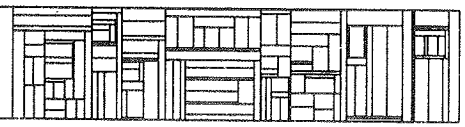
ISSN 0105-8517

EXPERIMENTS WITH A FAST STRING SEARCHING ALGORITHM

Peter Møller-Nielsen
Jørgen Staunstrup

DAIMI PB-167
October 1983

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55



EXPERIMENTS WITH A FAST STRING SEARCHING ALGORITHM

Peter Møller-Nielsen and Jørgen Staunstrup
Computer Science Department
Aarhus University
DK-8000 Aarhus C, Denmark

October 1983

Abstract

Consider the problem of finding the first occurrence of a particular pattern in a (long) string of characters. Boyer and Moore [1] found a fast algorithm for doing this. Here we consider how this algorithm behaves when executed on a multiprocessor. It is shown that a simple implementation performs very well. This claim is based on experiments performed on the Multi-Maren multiprocessor [4].

1. INTRODUCTION

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 [1] have found a very fast algorithm for doing such a string-search, their algorithm is sublinear, i.e. on the average it needs fewer than "i" character comparisons, to find an occurrence at index "i". Here we suggest an implementation of this algorithm which reduces the average running time further by searching a number of substrings simultaneously. This implementation is formulated as a so-called problem-heap algorithm [4].

2. PROBLEM-HEAP FORMULATION OF THE 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 [1]. Each of the processes looks as follows:

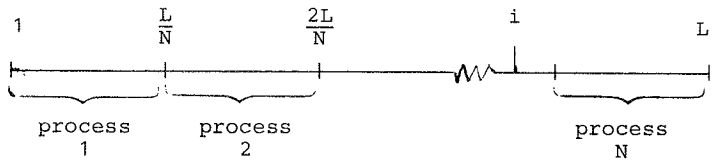
```
CYCLE
   receive (substring);
   search (substring);  "using the Boyer-Moore algorithm"
   IF match
     THEN report (position);
END
```

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

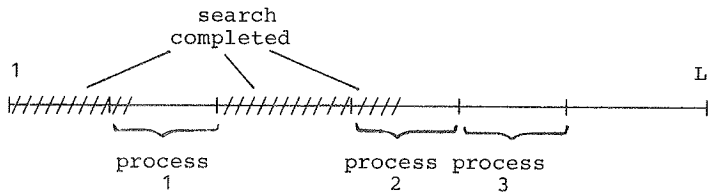
2.1 The problem-heap

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.

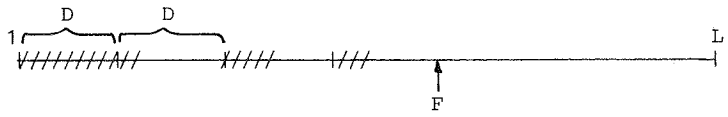
The most obvious way to distribute a string of length L to N processes is by dividing it into N substrings of equal length and then let each process search through one of these:



But since the leftmost occurrence of the pattern is not known in advance, this organization is very inefficient since the work done by all processes working to the right of "i" is lost. Instead relatively short substrings are given to processes. Hence the following illustrates a typical snapshot of a search:



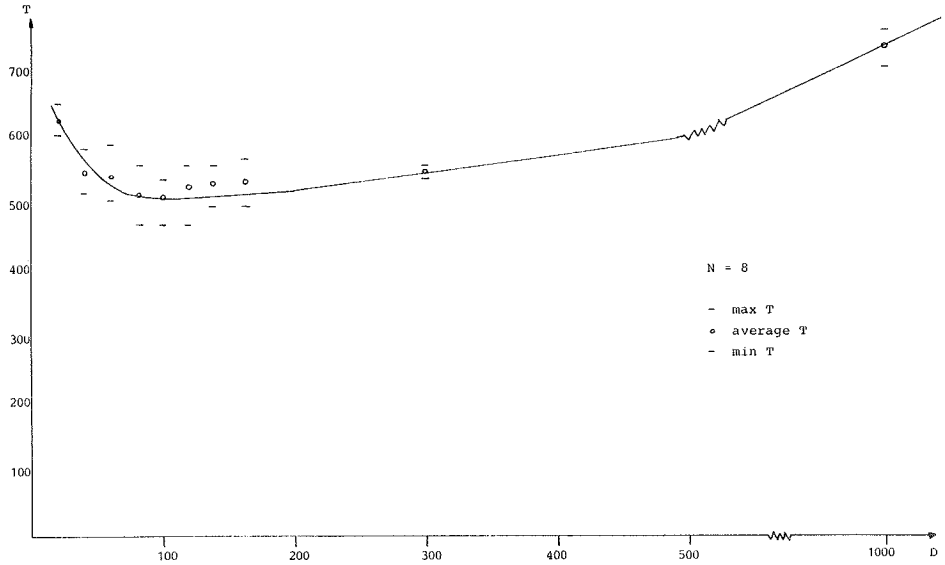
The details of the string administration, i.e. keeping track of which strings have been given to which processes can be worked out in many different ways, e.g. static or dynamic determination of the substring length. It turns out that the following simple administration performs very well: all substrings are of the same fixed length D, where D≥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:



The obvious question is of course which D should be chosen? A simple analysis (see appendix) shows that 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. The following diagram shows the relationship between the measured average running time T and the value of D (for a particular choice of the above parameters):



3. PERFORMANCE OF THE STRING-SEARCHING ALGORITHM

To demonstrate the performance of the algorithm it has been implemented on the Multi-Maren multiprocessor [4]. Two series of experiments were conducted with this implementation. One served to find the optimal D, a result is shown above. The other served to analyze the performance of the algorithm. Rather than give the performance as an absolute running time which is very dependent of hardware and implementation characteristica, the performance is given as a relative speed-up, i.e. the ratio of its running time using one process to the running time using N processes:

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

The advantage of using the speed-up as a performance measure is of course that it eliminates machine and implementation details.

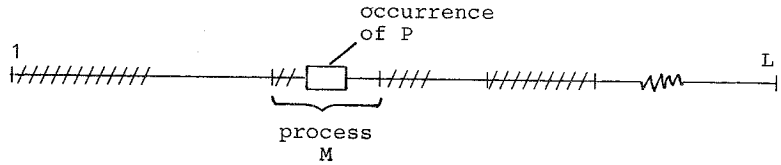
Below is shown the measured speed-up for the above described implementation of the string-search. For each N the optimal value of D has been used.

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

The table 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 completely 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, i.e. periods of time when the work of one or more processors does not contribute towards the solution, finding the leftmost occurrence of the pattern.

3.1 Braking loss

The first kind of loss which can be identified is braking loss, which is the superfluous work done by processes during termination of the algorithm. Consider a process, M, which finds an occurrence of the pattern; other processes may already be working on substrings to the right of M



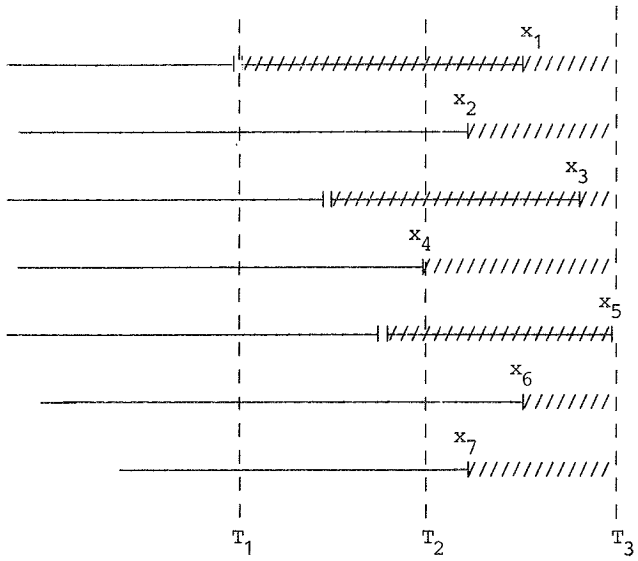
The work done to the right of the occurrence is of course superfluous. Consider now the processes working to the left of M, these should of course finish the search of the substrings they are currently working on, otherwise an occurrence before the one found by M could be missed. To get an estimate of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

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 in $[T_1,T_2[$ are immediately put to work on a new substring to the right of the occurrence, hence each of them loses a whole period, i.e. the time it takes to search a substring. On the average this should be N/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 shown below illustrates this situation:



The crossed sections (//////) represent the braking loss. The braking loss can be divided into two contributions. The first contribution is the (N-1)/2 processes losing a whole period, i.e. the periods marked by |////////| , this amounts to (N-1)/2 Dc, where c is a constant. Our measurements show that this contribution, although dominant, is not a satisfactory estimate of the braking loss. The residual (shown as /////// in the diagram) gives a significant contribution to the braking loss also. To estimate the residue consider the time instance T_2. At this instance, 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 residual contribution, R, to the loss is:

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

To estimate the average value of R 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

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

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 [5]

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

Hence

$$R = \sum_{i=1}^{N} (N-i)\ \frac{1}{N+1}\ Dc$$

$$= \sum_{i=1}^{N-1} i\ \frac{1}{N+1}\ Dc$$

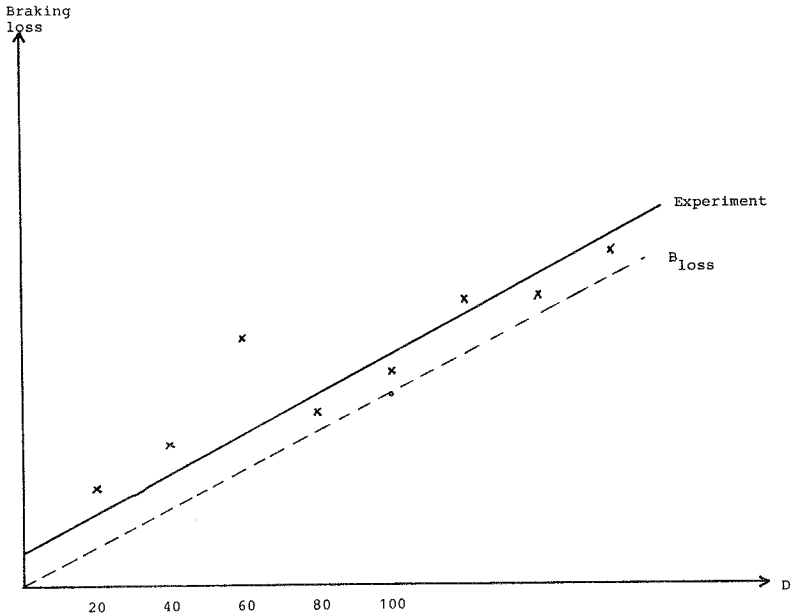
$$= \frac{N(N-1)}{2(N+1)}\ Dc$$

So the total braking loss is

$$B = \frac{(N-1)}{2} Dc + \frac{N}{2}\ (\frac{N-1}{N+1})\ Dc$$

$$= (\frac{N^2}{N+1} - \frac{1}{2})\ Dc$$

Some experiments were conducted to measure the actual braking loss, the following diagram shows the experimentally measured values against the predicted values for B_{loss}:



3.2 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 work 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:

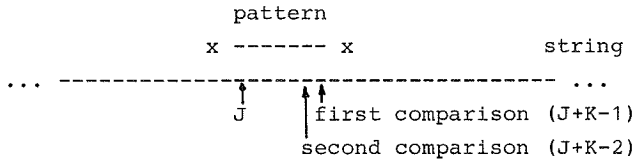
$$\frac{i}{D}$$

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 quite different. 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.



To check for an occurrence of the pattern at index J one compares the characters in the string with the pattern starting with the character in 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 [1], [2], and [3]. 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 [2].

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.

3.3 Saturation loss

In all the experiments reported above the string was placed in a common store accessible to all processes whereas all code and local variables were placed in private stores accessible to one process only. Only one process at a time may read or write the common store. Hence there might be a short delay, saturation loss, on each access to the common store while other processes access it. The following experiment was made to demonstrate that this delay is insignificant for the string searching algorithm. A version of the program was made where a copy of the string was placed in each of the local stores so that several processes referencing the string simultaneously would not interfere. The running time of this version differed by less than 1% from the above mentioned results. From this we may conclude that saturation loss can be ignored.

3.4 Running time estimation

It is claimed that braking and iteration loss explain the deviation from linear speed-up (see the table in section 3), i.e. that

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

To verify that no source 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):

N	$\frac{T(1)}{N}$	$\frac{B_{loss}}{N}$	$\frac{I_{loss}}{N}$	Predicted T(N)	Measured T(N)
4	899	49	25	973	981
8	450	30	25	505	516

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

4. CONCLUSION

A fast multiprocessor string searching algorithm has been presented. It is a simple modification of an algorithm by Boyer and Moore which makes it possible to search many substrings simultaneously. These searches can be done by the independent processors of a multiprocessor. The algorithm has been implemented on a multiprocessor and a very good speed-up has been achieved. The deviation from linear speed-up could be explained by identifying two kinds of lost processing power called braking and iteration loss. The magnitude of both of these was expressed analytically and these expressions were verified experimentally.

Acknowledgement

Thorkil Naur initiated this work by making the first attempt to construct a multiprocessor implementation of the Boyer-Moore algorithm as a term project in the fall of 1980.

References

[1] A Fast String Searching Algorithm, R.S. Boyer and J.S. Moore, C. ACM 20, 10, 1977.

[2] On improving the worst case running time of the Boyer-Moore string matching algorithm, Z. Galil, C. ACM 22, 9, 1979.

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

[4] Early Experience from a Multiprocessor Project, P. Møller-Nielsen and J. Staunstrup, Computer Science Department, Aarhus University, January 1982.

[5] An Introduction to Probability Theory and Its Applications, vol. II, W. Feller, Wiley 1970.

APPENDIX

In section 2.1 it was claimed that the optimal value of D depends on a number of quantities, e.g. the position of the occurrence which are usually not known before the search starts. Below the optimal D is calculated for a particular string and pattern. This also makes it possible to compare the calculated optimal D with the experimentally estimated optimal D.

The two sources of loss: braking loss and iteration loss determine the optimal D; to reduce the braking loss, D should be small whereas D should be large to reduce the iteration loss. The optimal D is therefore the one that minimizes the total loss:

$$\text{Loss}(D) = \frac{i \cdot O}{D} + \left(\frac{N}{N+1} - \frac{1}{2}\right) Dc$$

The minimal value of Loss is obtained for

$$D_{opt} = \sqrt{\frac{2(N+1)\,i \cdot O}{c(2N^2-N-1)}}, \quad N>1$$

The following table compares the calculated D_{opt} with the experimentally estimated D_{opt}

N	calculated D_{opt}	experimental D_{opt}
2	258	300
4	143	200
8	92	100

The disagreement between the experimentally estimated and the calculated optimal D is due to the very big variation in the experimental results. For $50 \leq D \leq 400$ this variation covers any differences caused by choosing different D. To make a reliable experimental estimation a very large number of experiments should be run.