# Practical Sequence Partitioning

---

Thomas Haukland, *thomash@ii.uib.no*

Submitted for the degree of Candidatus Scientiarum at the Department of Informatics, University of Bergen, Norway. Written under the sage and generous guidance of Fredrik Manne, fredrikm@ii.uib.no.

Thomas Haukland, July 2001

# Contents

**1 Introduction** **5**

1.1 Problem formulation . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.2 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

1.3 Notation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.4 Scope of Experiments . . . . . . . . . . . . . . . . . . . . . . . . 8

**2 Bounds on the optimal solution** **9**

**3 Probing** **10**

3.1 Linear Probe . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3.2 Binary Probe . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.3 Tree Probe . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

3.4 Notes on performance . . . . . . . . . . . . . . . . . . . . . . . . 15

**4 Balance-Points** **18**

**5 Algorithms** **20**

5.1 Bisection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

5.2 OM: Dynamic Programming . . . . . . . . . . . . . . . . . . . . . 27

5.3 Nicol's Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

5.4 MS: Leftist Partitioning . . . . . . . . . . . . . . . . . . . . . . . 35

5.5 PA: The Bidding algorithm . . . . . . . . . . . . . . . . . . . . . 38

5.6 HNC: Recursive Balance-points . . . . . . . . . . . . . . . . . . . 40

5.7 Frederickson's algorithm . . . . . . . . . . . . . . . . . . . . . . . 44

**6 Experimental results** **48**

6.1 Previously published experiments . . . . . . . . . . . . . . . . . . 48

6.2 Experiments on larger problems . . . . . . . . . . . . . . . . . . . 49

**7 Conclusion** **53**

# List of Algorithms


1 Linear/Greedy probe . . . . . . . . . . . . . . . . . . . . . . . . . 11

2 Binary probe . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

3 Place separator . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

4 Probing in a binary tree . . . . . . . . . . . . . . . . . . . . . . . 14

5 Bisection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

6 Nicol's algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . 34

# 1 Introduction

## 1.1 Problem formulation

A common problem associated with parallel computing is load-balancing. When a task is divided into sub-tasks and distributed among several processors, the total time needed to solve the task is the time used by the most heavily loaded processor. Therefor, one of the primary goals of load-balancing is to minimize the load of the most heavily loaded processor. Without any constraints on the sub-tasks or processors, this problem is known to be NP-hard[8].

When certain restrictions are placed on sub-tasks or processors, varieties of the problem arise which may be solved in polynomal and even linear time. In this work we consider the case were every processor has the same capacity and each sub-task is associated with a positive number expressing the amount of time needed to solve the sub-task at any processor. Also, if sub-task $i$ is assigned to processor $j$, sub-task $i+1$ must be assigned to processor $j$ or $j+1$. If there are $p$ processors, this means that the original sequence of sub-tasks will be divided into $p$ continuous partitions. Definition 1 formalizes the problem, and practical applications are discussed in Chapter 1.2.

**Definition 1 The MinMax-problem**: *A sequence of* **n** *numbers is to be partitioned into* **p** *continuous intervals, so that the cost of the most expensive interval is minimized.*

The cost of an interval is the sum of the numbers in the interval. Consider this sequence of 8 numbers partitioned into 4 intervals:

$$[\,2\;3\;|\;6\;1\;|\;2\;2\;4\;|\;5\,]$$

The most expensive interval here is the third, consisting of $2, 2$ and $4$, and having a cost of 8(which is called the *bottleneck-cost*).

This problem was first discussed as the *path-partitioning* problem, and solved by Becker, Perl, and Scach in 1982[2]. Their algorithm worked on trees[1], and used $O(p^3 n)$ time to solve the problem. It is also known as the *chains-on-chains* problem and Bokhari and Iqbal shows how a similar problem, taking into account communication costs between the $p$ intervals, can be transformed to the MinMax-problem with a simple $O(n)$ operation[5]. Bokhari also published an $O(n^3 p)$ algorithm[4] solving the problem in 1988, and extensive research has been done on the subject since.

Very few experimental results from actual implementations of the algorithms are published. This work is an attempt to implement a representative selection of the existing algorithms, based on performance and approach. First, in Chapter 2, some important bounds are discussed, which are later used to optimize the algorithms. In Chapter 3, a technique called probing and different variants of it is introduced. This technique is used by several of the algorithms. Chapter 4 defines a property of the MinMax-problem called a balance-point. Finding balance-points is at the core of some of the algorithms.

Chapter 5 treats each algorithm separately, discussing it's strengths and weaknesses, and possibilities of improvements. Afterwards, in Chapter 6, experiments are performed on all of the implemented algorithms, comparing efficiency on problems of various sizes. A summary of the results is presented in Chapter 7.

## 1.2 Motivation

As an example of the MinMax-problem, consider an image consisting of $n \times n$ pixels, where an operation has to be performed on every pixel of a specific color, and the result of this operation depends on the color of the adjacent pixels. When distributing this image among processors for parallel execution of the operation, one way of implicitly minimizing communication costs between processors is to assign countinous row-stripes of the image to the

[1] A sequence is the same as a path, which in turn is a special kind of tree.

processors. The cost of each task(a row) will be the number of pixels of the specific color in that row. See Kutluca et.al.[10] for a detailed discussion of image-space decomposition.

Other applications include signal-processing[4] and computations/simulations associated with a communications network[13]. Pinar and Aykanat[15] also propose to apply the problem to parallelization of sparse matrix vector multiplication.

## 1.3 Notation

The following notation will be used throughout this work.

| **Symbol** | **Value** | **Description** |
|---|---|---|
| $n$ | | number of elements in the sequence. |
| $p$ | | number of partitions. |
| $s_i$ | | separators or delimiters. $s_0 = 0, s_p = n, s_{i-1} \leq s_i$ and $s_i \in \{0, n\}$ when $0 < i < p$. |
| $W_i$ | | value of element number $i$. |
| $W_{i,j}$ | $\sum_{k=i}^{j-1} W_k, i < j$ | cost of elements $i$ through $j - 1$. |
| $W$ | $W_{s_0,s_p} = W_{0,n} = \sum_{i=0}^{n} W_i$ | total cost of all elements in the sequence. |
| $\pi$ | | a partition defined by $s_0, s_1, ..., s_p$. |
| $w(\pi)$ | $\max(W_{s_{i-1},s_i} 0 < i \leq p)$ | the cost of the most expensive interval in $\pi$. |
| $W_{opt}$ | $\min(w(\pi_i) \forall i)$ | The cost defining the optimal partition. |
| $T_i$ | $\sum_{j=1}^{i} W_j$ | $T$ is the prefix-sum array. |

## 1.4 Scope of Experiments

All algorithms are implemented [2] in the C programming language, on the Linux operating system. The experiments have been carried out on an Intel Celeron running at 768 MHz, with 128KB internal cache and 128MB RAM. Performance is measured by two variables; time and iterations. Time is in microseconds, and starts after the problem have been read from file. The number of iterations is a counter which is incremented each time a loop of the program is executed. This number is similar to the asymptotic running time, as it does not take into consideration the constant factors of the implementation. To get stable execution times, the algorithms have been run several times on the same problems, and only the best(shortest) execution times have been used.

If not noted otherwise, the numbers of the sequences are randomly generated. Only 64-bit integers have been used. The size of the problems are restricted by the amount of memory on a given computer, and no sequences larger than 1048576 numbers have been used.

[2]The source is available at http://www.ii.uib.no/~ thomash/hf/source/

# 2 Bounds on the optimal solution

The upper and lower bounds presented in this chapter will be used to improve the performance of several of the algorithms which solve the MinMax-problem.

If the cost of the most expensive interval, the bottleneck cost, is known, the $p$ delimiters can be placed by a greedy algorithm taking $O(n)$ time. Iterate over the sequence, summing elements, inserting a delimiter before the element which will make the sum exceed the bottleneck value. Hence, the actual placement of the delimiters is irrelevant, the bottleneck cost is a sufficient solution to the MinMax-problem. Since any algorithm needs to examine each element in the sequence, adding an $O(n)$ operation does not affect the asymptotic running time. The greedy algorithm is discussed further in the next chapter.

The lowest possible value of the optimal solution is $W/p$. This means every interval has the same cost, and no better partitioning exists. Notice that if the largest element, $W_{max}$, is larger than $W/p$, this is a lower bound on the optimal solution.

The best known upper boundary[3] is $W/p + W_{max}$. To see this, use a greedy approach to partition the sequence. Add elements to the current interval until adding another one means the cost of the interval exceeds or equals $W/p$, then add one more. If the $p-1$ first intervals cost $W/p$ or more, the last interval can not cost more than $W/p$, therefore the bottleneck cost can never be more than $W/p + W_{max}$.

[3] Choi and Narahari[6] proposed $2W/p$ as an upper boundary, but this only applies when $W_{max}$ is less than $W/p$, and then $W/p + W_{max} < 2W/p$.

# 3 Probing

Several algorithms for solving the MinMax-problem use a method for testing whether a value defines a valid partitioning the sequence, called a *probe*. The probe takes at least three parameters; a representation of the sequence, the number of intervals($p$) and a value. TRUE or FALSE is returned depending on whether the value defines a valid partitioning or not. Consider $p = 4$ and the sequence:

$$S = [\,2\,3\,6\,1\,2\,2\,4\,5\,].$$

A probe given $S$, $p = 4$, and $value = 5$ should return FALSE, since it is impossible to partition $S$ into 4 intervals where the maximum cost is not more than 5. Increase the value to e.g. 9 and probe should return TRUE. Note that a probe will return TRUE for all values greater than or equal to the optimal solution(8 for $p = 4$), and FALSE for all values less than the optimal solution. The rest of this chapter will discuss different probe-algorithms in detail.

## 3.1 Linear Probe

A greedy probe-algorithm, Algorithm 1, iterates over the sequence once, inserting a delimiter whenever the running sum exceeds the value in question. At the end of the sequence, if less than $p$ delimiters has been inserted, the value is a feasible solution. The running time of this algorithm is $O(n)$. Algorithm 1 makes it possible to probe only a part of the sequence. The sequence will be probed from the index given by the parameter 'start', until the index given by 'stop' is reached. How this can be useful is discussed in Chapter 4. Although this is useful and feasible for the following probe-algorithms, the details have been omitted for simplicity of reading. Obviously, if a number in the sequence is greater than the probing-value, FALSE should be returned. For simplicity, the code to handle this has been omitted.

As an example of how the greedy probe works, consider giving it $S$ as described above, $p = 3$, $value = 9$, $start = 1$, and $stop = n$. The probe will

then insert delimiters like this:

$$S = [\,2\;3\,|\,6\;1\;2\,|\,2\;4\,|\,5\,].$$

Since the last delimiter is inserted without including the last element, FALSE is returned.

```
probe(sequence[], start, stop, value, intervals)
p_left ← intervals
sum ← 0
for i = start to stop do
   sum ← sum + sequence[i]
   if (sum > value) then
      p_left ← p_left − 1
      sum ← sequence[i]
   end if
end for
if (p_left > 0) then
   return TRUE
else
   return FALSE
end if
```

Algorithm 1: Linear/Greedy probe

## 3.2 Binary Probe

By precomputing the prefix-sum-array of the sequence, the asymptotic running time of each probe can be reduced for some values of $p$ and $n$. Using the sequence $S$ from above, the prefix-sum-array $T$ would look like this:

$$T = [\,2,\ 5,\ 11,\ 12,\ 14,\ 16,\ 20,\ 25\,].$$

The cost of the interval containing elements $S_3$ through $S_6$: $[\,6,\ 1,\ 2,\ 2\,]$ is then given by $T_6 - T_2 = 16 - 5 = 11$. In general, let $W_i$ be the cost of element $i$, and $W_{i,j}$ the cost of $W_i + W_{i+1} + ... + W_j, 0 < i \leq j \leq n$. If $T_0$ is set to 0, $W_{i,j}$ can be expressed as $T_j - T_{i-1}$.

With $T$ available, it is possible to perform a binary search for the position of each delimiter as described in Algorithm 3. The pseudo-code for the binary probe is given in Algorithm 2.

Note that if $i$ is returned from 'place_separator', the element with index $i$ in the sequence belongs in the interval to the right of this delimiter. If 'place_separator' is called with e.g. the above $T$ and probe-value 15, then 6 is returned, indicating this placement of a delimiter:

$$[\,2\;3\;6\;1\;2\mid 2\;4\;5\,]$$

If a delimiter is placed in position $n+1$, the probe-value was used to partition the sequence into $p$ or fewer intervals, and TRUE is returned. Else, the last interval did not contain element $n$, and the probe-value does not define a valid partitioning, hence FALSE is returned.

This changes the complexity of a probe from $O(n)$ to $O(p \log n)$ which is better when $p = o(n/\log n)$. Note that it takes $O(n)$ time to construct the prefix-sum-array $T$, but this cost is negligible if the probe is executed often enough.

```
probe(T[],value,intervals)
sum ← value
for i = 1 to intervals do
  ix ← place_separator(T[], sum)
  if (ix = n + 1) then
    return TRUE
  end if
  sum ← value + T[ix − 1]
end for
return FALSE
```

Algorithm 2: Binary probe

## 3.3 Tree Probe

Similar to the prefix-sum-array is a binary tree having the numbers of the sequence as it's leaves and the sum of all the leaves as the root. The probe

**place_separator**(T[],value)
*low* $\leftarrow 0$
*high* $\leftarrow n+1$
**while** $((high - low) > 1)$ **do**
  *middle* $\leftarrow (high + low)/2$
  **if** $(T[middle] > value)$ **then**
    *high* $\leftarrow$ *middle*
  **else**
    *low* $\leftarrow$ *middle*
  **end if**
**end while**
**return** *high*

Algorithm 3: Place separator

using this tree will place each of the $p$ delimiters by working it's way from the root to the correct leaf in a fashion very similar to 'place_separator'.

The pseudo-code for this algorithm is listed in Algorithm 4. If the weight of the left child of the current node, together with the 'carry', is less than the sum defining the current delimiter, the current interval will have room for all the leaves below the left child. The search continues on the right child, to find out if there is room for even more leaves. Else, if the weight of the left child exceeds the 'sum' and 'carry', this means that the current delimiter should be placed somewhere between the leaves of the left child. If the current node is a leaf, the placement of a delimiter has been found, and 'sum' is updated for the search of the next delimiter which will start at the root again.

This approach has the same complexity as the binary probe, $O(p \log n)$, but has slightly higher constant factors, and is consequently not used in any experiments in the following chapters.

Han, Narahari, and Choi[9] suggested building $p$ binary search-trees, each containing $n/p$ leaves, which would make the running time for each probe $O(p \log n/p)$. This is a minor asymptotic improvement, and has not been tested in this work.

```
probe(tree,value,intervals)
sum ← value
for i = 0 to intervals do
   carry ← 0
   node ← root
   while (node ≠ leaf) do
      if (left_child(node).weight + carry < sum) then
         carry ← carry + left_child(node).weight
         node ← right_child(node)
      else
         node ← left_child(node)
      end if
   end while
   sum ← carry + value
end for
if (carry = root.weight) then
   return TRUE
else
   return FALSE
end if
```

Algorithm 4: Probing in a binary tree

### 3.4 Notes on performance

The binary probe is asymptotically faster than the linear when $p = o(\frac{n}{\log n})$. However, since the data the linear probe needs is most probably already in the cache, this intersection of performance occurs before $p$ reaches $\frac{n}{\log n}$ in reality. I.e. the constant factors of the linear probe are smaller than those of the binary probe. This is illustrated by Figure 1. Theoretically, the two probes should perform equally when $p = n/\log n = 2048/11 = 186$, as is the case in the first graph depicting the iteration count[4]. Beacuse of the smaller constant factors, the performance of the linear probe exceeds the binary probe already for $p$ greater than 120, when performance is measured in time. The graphs are obtained by running multiple probes for each value of $p$. A smart probe will determine which of the binary or linear method it should use, depending on the value of $p$ and $n$.

Although binary and tree probe have the same complexity($O(p \log n)$), the binary probe has lower constant factors. Figure 2 is a comparison of the three probes discussed in this chapter. As can be seen from the graph, the tree-probe loses more and more to the other two, as $p$ grows larger.

[4]See Chapter 1.4 for a description of how this is measured.

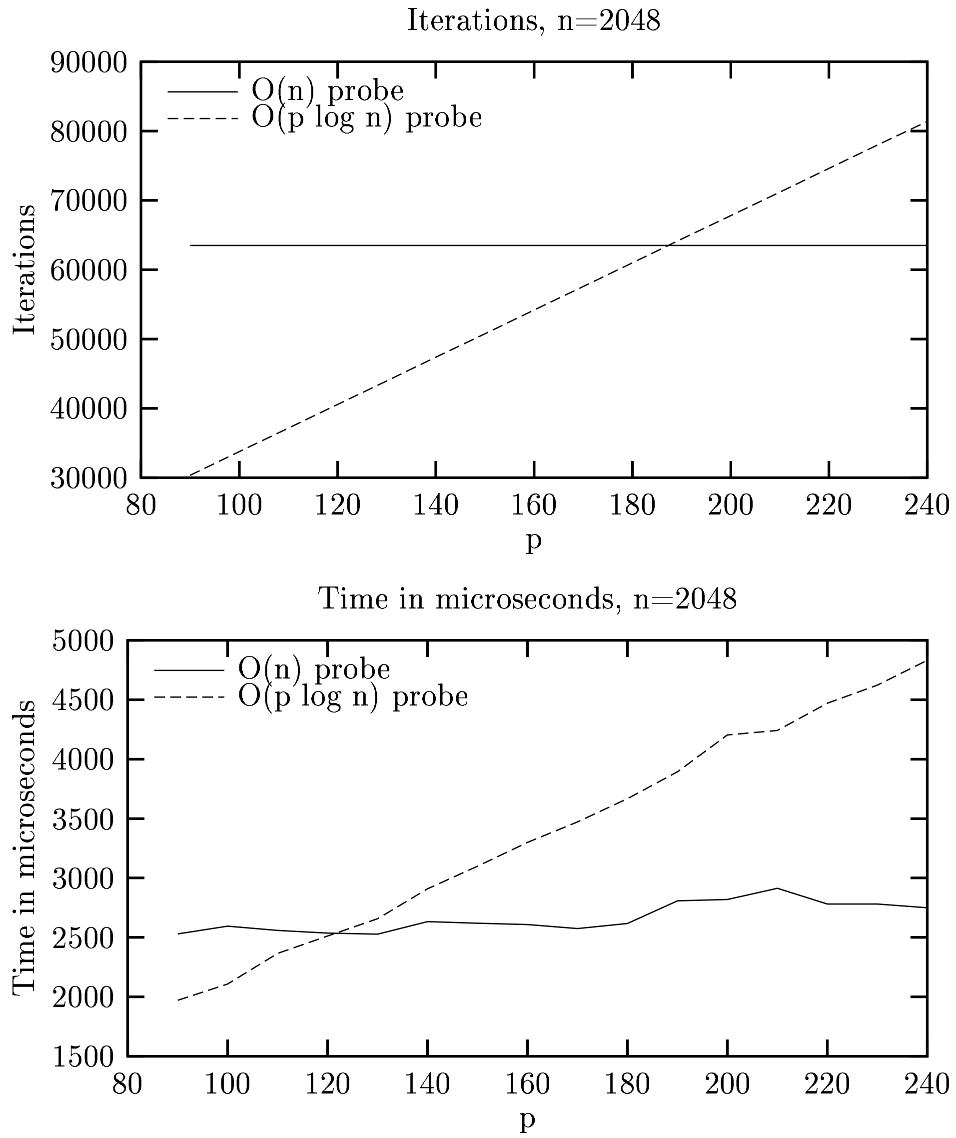


Figure 1: A comparison of the binary and the linear probe.

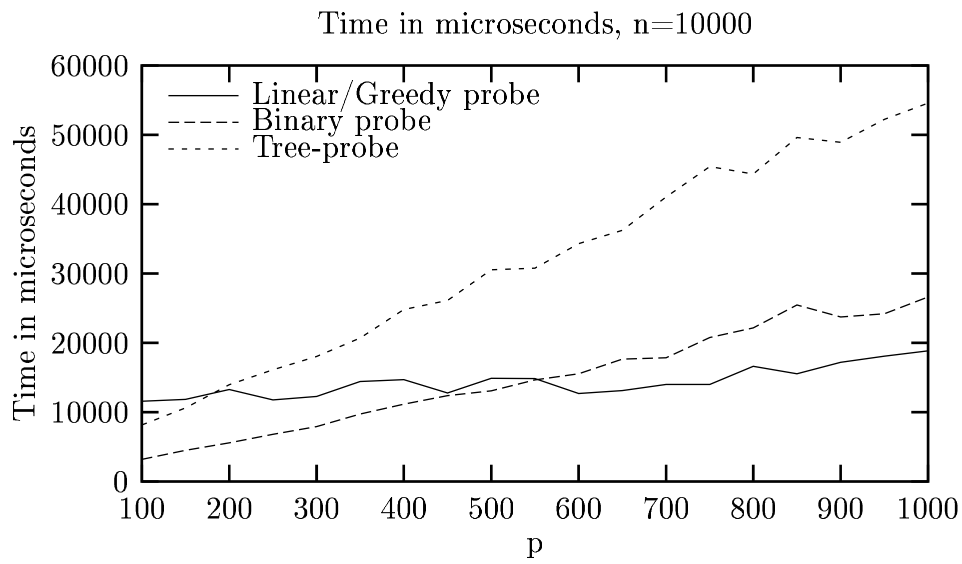


Figure 2: A comparison of the different probing-algorithms

# 4 Balance-Points

Finding so-called balance-points is central to several of the algorithms presented in this text. To help explain the principle, it is useful to define the function $g(i, j, l)$ as the cost of the most expensive interval in an optimal partitioning of $[W_i, W_{i+1}, ..., W_j]$ into $l$ intervals. The solution to the MinMax-problem is then given as $g(1, n, p)$.

Definition 2 explains a balance-point using the $g()$ function, and the probe discussed in Chapter 3.1.

**Definition 2** *$i^*$ is the k'th balance-point if:*

$$probe(i^*, n, g(1, i^* - 1, k), p - k) = FALSE$$

*and*

$$probe(i^* + 1, n, g(1, i^*, k), p - k) = TRUE$$

I.e. $g(1, i^* - 1, k)$ does not define a valid partitioning of $[W_{i^*}, ..., W_n]$ into $p - k$ intervals, but $g(1, i^*, k)$ defines a valid partitioning of $[W_{i^*+1}, ..., W_n]$ into $p - k$ intervals. Since $g(i, j, l)$ is a monotonically increasing function of $j$, the balance point exists and is unique[5].

[5]See Manne and Sørevik[12] for a detailed discussion.

### Properties of balance-points

Han, Narahari, and Choi[9] first discussed the following property of the generalized balance-point:

**Theorem 1** *If $i^*$ is the $k$'th balance-point,*

$$g(1, n, p) = \min\{g(1, i^*, k), g(i^*, n, p - k)\}$$

The idea is that the balance-point element tips the scale of where the most expensive interval is. If element $i^*$ is partitioned with the first $k$ intervals, this is where the most expensive interval is found, otherwise it will be among the $p - k$ last intervals. The optimal solution is minimized on cost, so both options are checked and the smallest is chosen.

# 5 Algorithms

In this chapter, several different algorithms for solving the MinMax-problem are presented. Although there exists other algorithms, they have been left out on purpose, either because their asymptotic running time was very poor, or because the approach was too similar to one of the presented algorithms.

After the introduction of each algorithm follows an attempt to increase its efficiency, accompanied by a graph illustrating the results.

Table 1 lists the asymptotic running times of the algorithms which are discussed in this chapter.

| **Algorithm** | **Asymptotic running time** |
|---|---|
| Frederickson[6] | $O(n)$ |
| HNC | $O(n + p^{1+\epsilon})$ |
| OM | $O((n-p)p)$ |
| Nicol | $O(p^2 \log^2 n)$ |
| PA | $O(np^2)$ |
| MS | $O((n-p)p \log p)$ |
| Bisection | $O(p \log n \log W_{max})$ |

Table 1: Asymptotic running time of the implemented algorithms

[6]The algorithm implemented in this work have running time $O(n \log \log n)$.

## 5.1 Bisection

This algorithm is due to Iqbal[11]. Using the upper and lower bounds on the optimal solution discussed in Chapter 2, it is possible to perform a binary search for $W_{opt}$. Recall from Chapter 3 that a probe returns TRUE for all values higher than or equal to the optimal, and FALSE for all lower values. After a completed probe on the value of the upper boundary and the lower boundary divided by 2, this value defines a new upper boundary if the probe returned TRUE, or a new lower boundary if the probe returned FALSE. To illustrate one iteration of the algorithm, consider the sequence $S$ from Chapter 3:

$$S = [\,2\,3\,6\,1\,2\,2\,4\,5\,].$$

Let $p = 4$. The upper boundary from Chapter 2 is $(S_1 + ... + S_8)/4 + 6 = 13$ rounded upwards, and the lower boundary is $(S_1 + ... + S_8)/4 = 6$ rounded downwards. The first value to be probed is $(13+6)/2 = 10$, which will return TRUE. 10 now defines the new upper boundary, and the process is repeated by testing $(10 + 6)/2 = 8$.

It has been assumed that this technique could not achieve exact accuracy when dealing with real numbers. The search has simply been terminated when the difference between the upper and lower bounds is below a predefined $\epsilon$. As will be shown, it is possible to get perfect accuracy without increasing the asymptotic running time.

### The Algorithm

To avoid aborting the search before the optimal solution has been found, we modify the probe-algorithm, so that it provides a way of mapping arbitrary probe-values, to the closest value arising from the sequence. The optimal solution will always be in the interval between the lower and upper bound. Using this modified probe, the upper and lower bounds will eventually take on the same value, and the optimal solution is found. The pseude-code for the algorithm is listed in Algorithm 5, and is explained in the next paragraphs.

We establish the upper boundary by setting the variable *high* to $\frac{W}{p} + W_{max}$ and the lower boundary by setting the variable *low* to $\frac{W}{p}$. The probing value *middle* is set to $\frac{high+low}{2}$ and an attempt is made to partition the sequence into $p$ intervals, none of which cost more than *middle*, using a modified probe.

The probe function performs a binary search[7] to locate the position of each delimiter, as defined by *middle*. When one is found, two variables are updated; if the cost of the current interval exceeds *real_ cost*, update *real_ cost*, and if if the cost of the current interval including the "next" element is smaller than *next_ cost*, update *next_ cost*.

When a probe is completed, *real_ cost* will hold the cost of the most expensive interval if *middle* defined a valid partitioning. Else, *middle* will have to be increased to at least *next_ cost* to have a chance of defining a valid partitioning.

If the probe finds that *middle* defines a possible partitioning, it returns TRUE and *real_ cost*, and *high* is updated to the value of *real_ cost*. Else, if the *middle* was too small to define a partitioning, FALSE and *next_ cost* is returned, and *low* is updated to *real_ cost*.

When $low = high$, the optimal partitioning has been found.

### Asymptotic running time

Since the search interval is of size $O(W_{max})$ at the start, and is halved at each iteration of the outer loop, the *binary_ probe* procedure is called $\log W_{max}$ times [8], in which *place_ separator* ($O(logn)$) is called $p$ times, yelding a total complexity of:

$$O(p \log n \log W_{max})$$

This is difficult to compare to other algorithms, because of the factor

[7] In the actual implementation, the smart probe referred to in Chapter 3.4 is used to ensure that the fastest probe algorithm is used.

[8] The $\log W_{max}$ factor is slightly smaller, since *high* is always set to *real_boundary* which is smaller than or equal to *middle*.

## Bisection

```
low ← W_tot/p
high ← low + W_max
while low ≤ high do
   middle ← (low + high)/2
   (result, tmp) ← binary_probe(middle)
   if result = TRUE then
      high ← tmp
   else
      low ← tmp
   end if
end while
return high
```

## Binary_probe

```
sum ← value
increase ← MAXINT
real_cost ← 0
previous_ix ← 0
for i = 0 to p do
   ix ← place_separator(sum)
   if (T[ix] − T[previous_ix]) > real_cost then
      real_cost = T[ix] − T[previous_ix]
   end if
   if (T[ix + 1] − T[previous_ix]) < increase then
      increase ← T[ix + 1] − T[previous_ix]
   end if
   if ix = n then
      return (TRUE, real_cost)
   end if
   sum ← value + T[ix]
   previous_ix ← ix
end for
return (FALSE, increase)
```

Algorithm 5: Bisection

$\log W_{max}$ which is not present in other running times. In practical applications though, this factor will never be larger than the number of bits used to represent the numbers. Two important exceptions to this assumption exists. First, if floats are used, it may be as big as $\log 10^{(2^x)} = O(2^x)$, where $x$ is the number of bits used to represent the exponent of the float. Second, if a number-implementation with variable number of bits, like the GMP library[9], is used, there is no limit to this factor.

To find a running time independent of data size, imagine all the possible values arising from one sequence, $\sum_{k=i}^{j} W_k, 0 < i \leq j \leq n$, sorted in an array $C$. The size of this array is then $\binom{n}{2} = O(n^2)$. At the start of the bisection algorithm, *high* and *low* will point to somewhere inbetween the elements of $C$. When *middle* is probed, and either *high* or *low* is updated, the interval containing possible solutions will shorten, and at least one element of $C$ will be excluded.

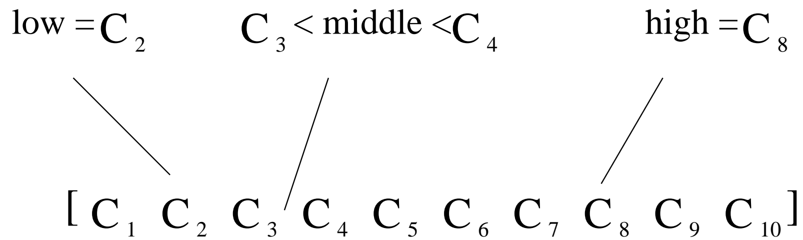


Figure 3: If probe(middle) returns TRUE, *high* is set to $C_3$, else *low* is set to $C_4$.

This analysis leads to an asymptotic running time of:

$$O(n^2 p \log n)$$

**Improvements**

The two improvements presented here can be applied to all algorithms based on probing.

[9]GNU Multiple Precision: http://www.gnu.org/software/gmp.html

First, we keep track of where each previous probe placed delimiters. If probe returned TRUE, the $p$ delimiters placed will serve as upper bounds for the optimal position of delimiters for subsequent calls to probe. Instead of setting 0 as the lower bound for each delimiter, we use the position of the previous delimiter as this bound. The effect of this modifications is minor, but can be compared to building $p$ binary search trees, as discussed in chapter 3, in that the running time of a probe will converge on $O(p \log n/p)$. Note that this only applies to the binary probe.

Another improvement to the probe, is to check that the probing-value still defines a possible solution. After having placed delimiter $k$ at position $i$, if $(W_i + ... + W_n)/(p-k)$ is greater than the probing-value, then we can abort the probe and return FALSE. This follows as a corollary from the upper bound explored in Chapter 2: If the mean cost of the remaining intervals exceeds the probing-value, this probing-value can not define a valid partitioning of the sequence into $p$ intervals. This improvement apply to both the linear and the binary probe.

Figure 4 shows the performance of the bisection-algorithm before and after the improvements have been implemented. Since the most significant improvement only applies to the binary probe, values of $p$ larger than $n/\log n = 4096$ does not lead to much difference in performance, since the linear probe is then used.

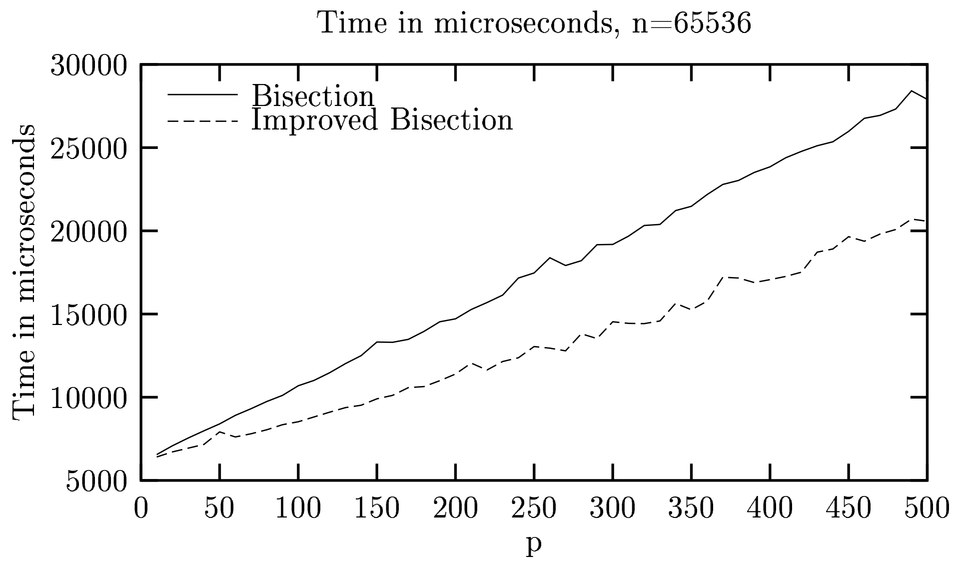


Figure 4: Comparison of bisection with and without minor improvements for small values of $p$.

## 5.2 OM: Dynamic Programming

Anily and Federgruen[1] first presented a recursion which can be used to solve the MinMax-problem with dynamic programming. Recall the function $g(i, j, k)$ from Chapter 4, and notice that $g(i, j, 1)$ is $W_i + ... + W_j$ for $0 < i \leq j \leq n$. For $k > 1$ we get :

$$g(1, j, k) = \min_{1 \leq l \leq j} \{\max\{g(1, l, k-1), g(l, j, 1)\}\}$$

This means testing all the possible positions for the $(k-1)$'th balance-point, and choosing the one leading to the smallest cost. Using this recursion, it is possible to construct the $p \times n$ array consisting of the solutions of $g(0, j, k), 0 < j \leq n, 0 < k \leq p$, with a time complexity of $O(pn^2)$. To find each $g()$ value, we need to look up and compare $n$ previous values.

To illustrate, the array arising from the sequence $S = [2\,3\,6\,1\,2\,2\,4\,5]$ when $p = 4$ is presented in Table 2. The dynamic programming algorithm, AF, will fill out this array one row at a time, from left to right, until $g(0, n, p)$ is found at the bottom right of the array.

| **p/n** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 5 | 11 | 12 | 14 | 16 | 20 | 25 |
| 2 | 2 | 3 | 6 | 7 | 9 | 11 | 11 | 13 |
| 3 | 2 | 3 | 6 | 6 | 6 | 6 | 8 | 11 |
| 4 | 2 | 3 | 6 | 6 | 6 | 6 | 6 | 8 |

Table 2: The optimal solution, $g(1, 8, 4) = 8$, is found at the bottom right.

### A refined dynamic programming algorithm

Choi and Narahari[6], and independently Olstad and Manne[14] discovered that it is not necessary to check all possible elements to find a balance-point. The key observation, which they use to develop improved algorithms, is that

the $k$'th balance-point of the sequence $[S_1, ..., S_j, S_{j+1}]$ is located to the right of the $k$'th balance-point of the sequence $[S_1, ..., S_j]$.

The OM-algorithm fills out the array of $g()$-values in the same fashion as the AF-algorithm: one row at a time from top to bottom, from left to right. For $k = 1$, $g(0, j, k) = T[j]$. For $k > 1$, $g(0, j, k)$ is calculated by keeping track of where the $k-1$'th balance-point was located for $g(0, j-1, k)$. Let $i^*$ be the $k-1$'th balance-point for the problem $g(0, j-1, k)$. To find $g(0, j, k)$, increase $i^*$ until it meets the criteria for a balance-point, as specified in Definition 2. When the balance-point has been found, the value of $g(0, j, k)$ is found by using Theorem 1. According to this, $g(0, j, k) = \min\{g(0, i^*, k-1), g(i^*, j, 1)\}$, which is calculated in constant time since $g(0, i^*, k-1)$ is already known, and $g(i^*, j, 1) = T[j] - T[i^* - 1]$. Another observation used to reduce the complexity of the algorithm, is that every interval has room for at least one element. I.e. it is not necessary to compute the value of $g(0, j, k)$ for $j < k$ and $j > p - k$. Table 3 illustrate which values of $g()$ the OM-algorithm calculates.

| **p/n** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 5 | 11 | 12 | 14 | | | |
| 2 | | 3 | 6 | 7 | 9 | 11 | | |
| 3 | | | 6 | 6 | 6 | 6 | 8 | |
| 4 | | | | 6 | 6 | 6 | 6 | 8 |

Table 3: The OM-algorithm only calculate $p(n-p+1)$ values of $g(0, j, k)$.

### Asymptotic running time

For each $k, 0 < k \leq p$, $g(1, j, k)$ is calculated for $k \leq j \leq n - p + k$. For each $g()$, the balance-point is found. Since the total cost of finding all balance points for a fixed $k$ is $O(n-p)$, this can be amortized into the cost of iterating through the sequence for each $k$. The total time complexity is then:

$$O(p(n-p))$$

### Improvements

By carefully analyzing the bounds explored in Chapter 2, we can limit the number of $g()$-values we need to calculate. Notice that for any $k$, we only need to know $g()$ for the $k$'th balance-point, since this value will be used when calculating $g()$-values for the $k+1$'th row. The upper and lower bounds on the optimal solution, implicitly limits the position of delimiters, hence also the position of balance-points. Instead of starting at index $k$ at each row, we start at the index indicated by the sum $(T[n]/p)(k-1)$. I.e. for each $k$, skip as many elements as possible, without the cost of these exceeding $T[n]/p$. Do the same operation backwards from the end of the row, to find out when we can stop. In general, we can skip $p-k$ intervals of cost $W[n]/p$ at the end of each row[10].

This improvement is an extension of the observation mentioned above, that each interval has room for one element. The increase in performance is dependent on the spacing of data in the sequence, and the asymptotic running time remains unchanged.

Figure 5 illustrates the effect of the this improvement. As $p$ grows closer to $n$, the running time of both algorithms decrease and finally becomes linear when $p = n$.

[10]The implementation figures out the bounds for the balance-points in an O(n) preprocessing-step and stores them in an array.

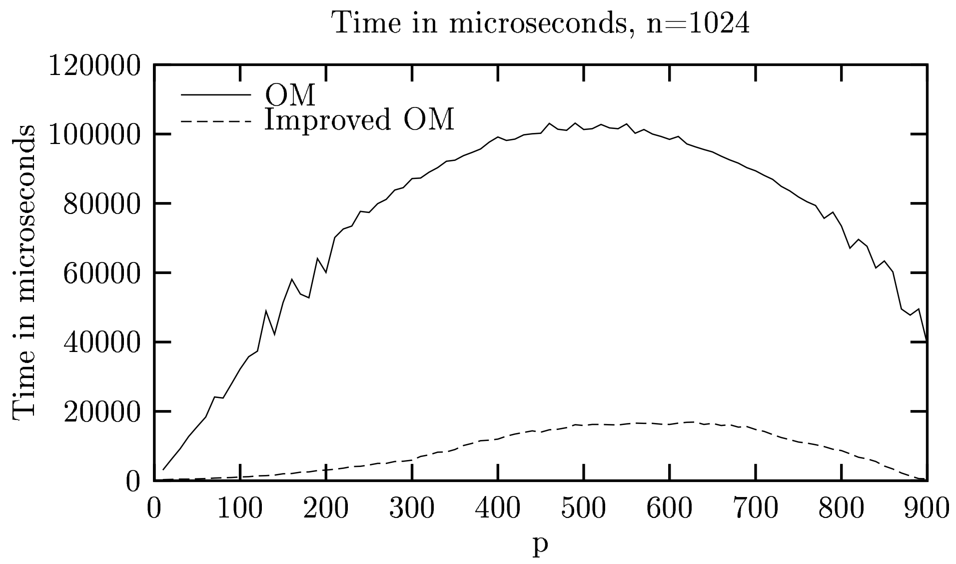


Figure 5: The OM-algorithm with and without improvement.

## 5.3 Nicol's Algorithm

Nicol's algorithm[13] starts by performing a binary search for the first balance-point, $i^*$, which is described in Chapter 4. I.e. the smallest $i$ for which $probe(S, i+1, n, g(1,i,1), p-1) = TRUE$. Recall from Theorem 1, that when this element($i^*$) is found, the solution to the MinMax-problem, $g(1,n,p)$, is $\min\{g(1,i^*,1), g(i^*,n,p-1)\}$. Nicol stores the cost of the first interval, $g(0,i^*,1)$, as a possible solution. The solution to $g(i^*,n,p-1)$ is found by repeating the process on this sub-problem. After $p$ costs have been stored, the smallest one is the optimal solution to the MinMax-problem. Figure 6 illustrates the the first step of the algorithm.

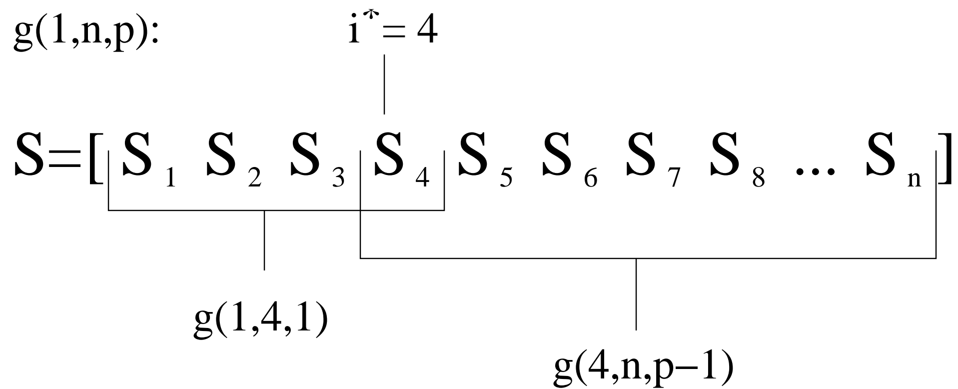


Figure 6: After $i^*$ has been found at index 4, the cost of $g(1,4,1)$ is stored, and $g(4,n,p-1)$ is solved by repeating the procedure on this sub-problem.

#### Asymptotic running time

To find each balance-point, a binary search is performed, using a probe in each step. The probe uses $O(p \log n)$ time, the binary search uses $O(\log n)$ time, and since this has to be repeated $p$ times, the total asymptotic running time is:

$$O(p^2 \log^2 n)$$

**Improvements**

When testing whether element $i$ is the $k$'th balance-point, the probe only needs to consider the sub-sequence $[S_i...S_n]$, as noted in Chapter 4. Figure 7 illustrates what happens when the algorithm tries to locate the position of the second balance-point. The sequence $S_8...S_n$ is probed with the value $g(4,7,1)$ as a step in the binary search for $i^*$, the second balance-point.

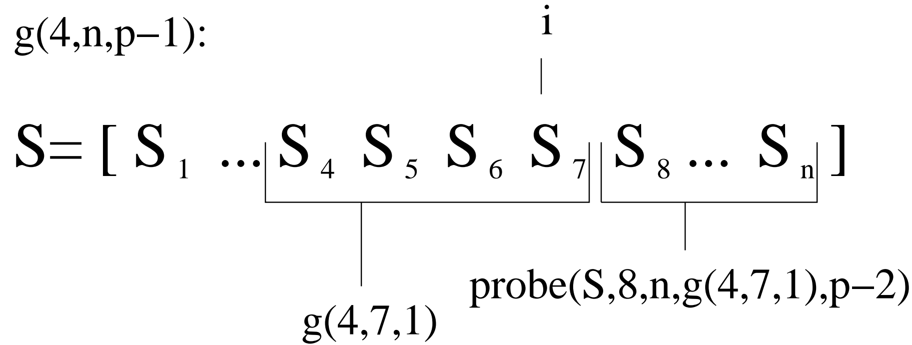


Figure 7: The sequence $S_8...S_n$ is probed with the value $g(4,7,1)$.

The improvements to the probe discussed in Chapter 5 are also implemented.

By keeping track of the lowest value probed successfully and the highest value probed unsuccessfully, it is possible to avoid many unnecessary probes. If probe returned TRUE for a value, it will return TRUE for all values higher than this. Conversely, if probe returned FALSE for a value, it will return FALSE for all values lower than this. This can be applied to succeeding probes on sub-sequences, since Nicol ensures either that previous delimiters are placed optimally, or that the optimal solution is already found. By continuously narrowing the upper and lower bounds for probing-values, Nicol's uses a parametric search technique. The behavior of the algorithm after this change is similar to the bisection-algorithm of Chapter 5.

Since this improvement is dependent on the actual values in the sequence, rather than $n$ or $p$ it is difficult to prove any impact on the asymptotic running time, but as the graphs will show, it has a significant practical effect.

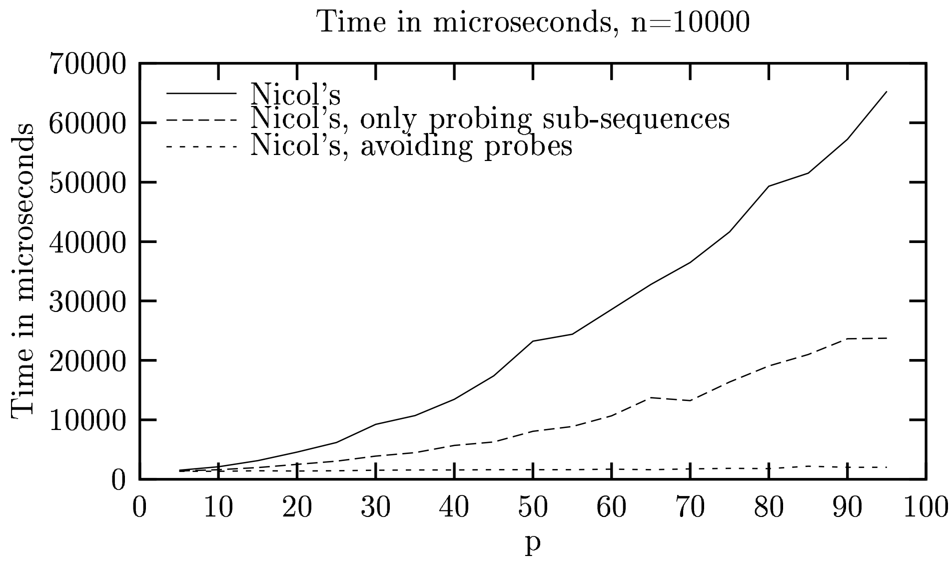


Figure 8: The performance of Nicol's algorithm.

Figure 8 illustrates the effect of the improvements discussed above. With all improvements implemented, the performance of the algorithm does not seem very dependent on the value of $p$.

The pseudocode in Algorithm 6 details the modifications needed. (See Chapter 2 for pseudocode of binary probe.)

```
Make prefixsum-array T
lowest_cost ← MAXINT
known_low ← T[n]/p
known_high ← known_low + W_max
prev_ix ← 0
for i = 0 to p do
  low ← prev_ix − 1
  high ← n
  repeat
    middle ← (low + high)/2
    sum ← T[middle] − T[prev_ix]
    if (sum ≥ known_high) then
      high ← middle
    else if (sum ≤ known_low) then
      low ← middle
    else if (probe(T, middle + 1, n, sum, p − i)) then
      high ← middle
      known_high ← high
    else
      low ← middle
      known_low ← low
    end if
  until ((high − low) > 1)
  lowest_cost ← min(lowest_cost, T[high] − T[prev_ix])
end for
return lowest_cost
```

Algorithm 6: Nicol's algorithm

## 5.4 MS: Leftist Partitioning

Becker, Perl, and Scach[2] first developed a shifting algorithm which solves the MinMax-problem in $O(p^3n)$ time. A shifting algorithm places delimiters(or cuts), and moves them to acheive better and better partitions. Although Becker et. al. only discussed partitioning trees, this also applies to sequences, or paths, since these are a special kind of trees. Becker and Perl[3] gave an overview of shifting algorithms.

Manne and Sørevik[12] improved the running time from $O(p^3n)$ to $O((n-p)p\log p)$ by introducing a binary heap.

### The algorithm

The idea is to find the most expensive interval, and moving the leftmost delimiter one position to the right until this is not the most expensive interval, and repeating the process until it does not produce a cheaper partitioning. To find the most expensive interval in constant time, they maintain a binary heap of the costs of the intervals. They show that this will produce an optimal partitioning if the partition the algorithm started with is what they call a leftist partition, meaning all the delimiters are to the left of their optimal placement. They propose to simply place the delimiters so that each of the $p-1$ first intervals contains one element to ensure a leftist partition.

### Asymptotic running time

In the worst case, all the $p$ delimiters have to be moved $(n-p)$ places, and for each move, the heap containing the $p$ costs has to be updated, $O(\log p)$, for a total complexity of:

$$O((n-p)p\log p)$$

### Improvements

By using the lower bound $\frac{W}{p}$, it is easy to construct a leftist partition by greedily filling up intervals. This partition will be closer to the optimal solution than what the authors suggest starting with, and thus saves iterations.

Assume the weight of the first interval is $\theta_1 \leq W/p$, using the initialization scheme discussed above. The second interval can then have weight $\theta_2 \leq (W - \theta_1)/(p-1)$ without placing the second delimiter, $s_2$, to the right of it's optimal position. The proof is by contradiction. If $s_2$ is placed to the right of its optimal position, $(\theta_1 + \theta_2) > \frac{2W}{p}$. If $\theta_1$ and $\theta_2$ take on their largest possible values, this means:

$$\frac{W}{p} + \frac{W - \frac{W}{p}}{p-1} > \frac{2W}{p}$$

$$\Rightarrow \frac{2(p-1)W}{p(p-1)} > \frac{2W}{p}$$

$$\Rightarrow \frac{2W}{p} > \frac{2W}{p} \quad \square$$

The proof can be extended to all $\theta$'s by induction. This improved initialization further increases the performance of the MS-algorithm. However, since the effect of this is dependent on the values of the data, it does not have any impact on the asymptotic running time.

Figure 9 demonstrates the difference in performance, caused by the improvements. When $p$ is very high, the initial placement of delimiters is almost optimal, hence very few adjustments have to be made.

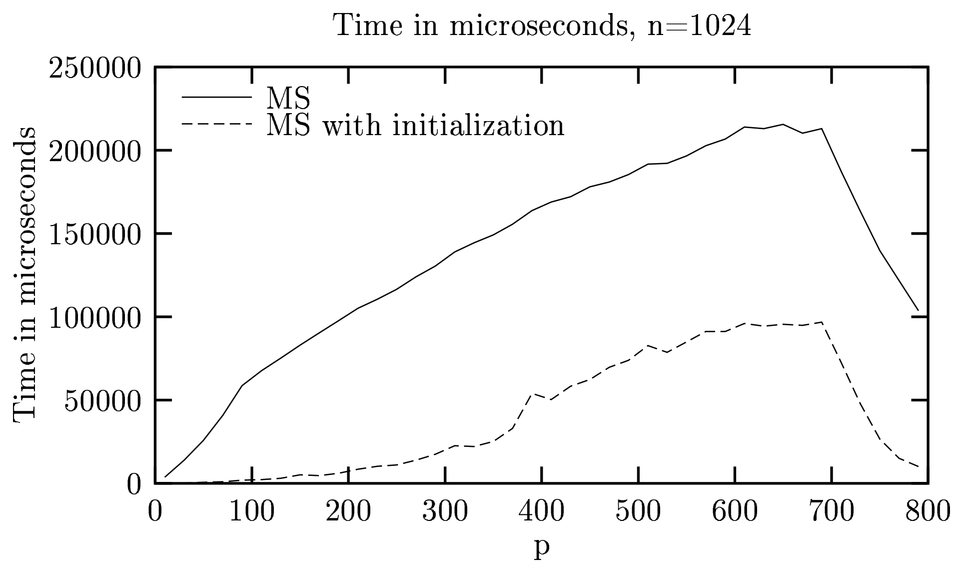


Figure 9: When $p$ is very high, the initial placement of delimiters is almost optimal, hence every few adjustments have to be made.

## 5.5 PA: The Bidding algorithm

This is another shifting algorithm(se chapter 5.4), by Pinar and Aykanat[15], which starts with $W/p$ as the possible solution, and places delimiters greedily until it finds out that this is impossible. The solution is then updated to the next lowest possible, previous delimiters are updated, and this is repeated until all delimiters are placed, and the current solution is returned as the optimal solution.

Figure 10 illustrates how the algorithm works on an example sequence. If $p = 4$, the lower bound on the optimal solution is $W/p = 15$. After having placed two delimiters according to this bound, it is clear that the rest of the sequence can not be partitioned with this bound. One of the delimiters must be moved, and the one leading to the smallest increase is chosen. The first delimiter is moved 1 position to the right, leading to a new bound of 17. The next delimiter is moved to include the next two elements, and the third and last delimiter is placed before the element of weight 7, and 17 is returned as the optimal solution.

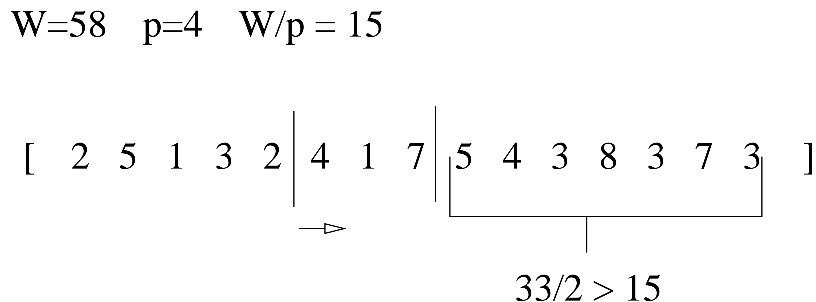


Figure 10: After the second delimiter has been placed, the current bound is increased.

Although very similar to Manne and Sørevik's algorithm[12], this approach eliminates the need of a heap to store costs in, thus saving an $O(\log p)$ factor of the running time. However, after updating the current cost, they need to check all subsequent intervals, possibly without moving any delimiters. This adds an $O(p)$ factor in comparison to Manne and Sørevik. In

theory, the difference is that while Manne and Sørevik shrinks the most expensive interval, Pinar and Aykanat expands the interval which leads to the smallest increase of the current cost.

### Asymptotic running time

Since all the delimiters are moved only in one direction, no more than $pn$ moves are performed. In the worst case, an increase of the current solution might lead to $O(p)$ iterations where no delimiters are moved. This leads to a total asymptotic running time of:

$$O(p^2n)$$

Since the algorithm already uses the lower bound from Chapter 2, it is not possible to use this to improve it.

## 5.6 HNC: Recursive Balance-points

Han, Narahari, and Choi[9] developed this algorithm, which is very similar to Nicol's[13]. Binary search is used to find $i^\star$, the $k$'th balance point[11]. For each $i$ tested, a recursive call solves the min-max problem on the sequence to the left of $i$, yielding a solution $W_a$, and $W_a$ is used to probe[12] the sequence to the right of $i$. If $i$ is the $k$'th balance point, determined by applying Definition 2, the sequence to the right is solved recursively($W_b$), and compared to $W_a$ to find the optimal solution. Figure 11 illustrates what HNC does after finding $i^*$, the $k$'th balance-point.

$$g(1, n, p) = \min\{W_a, W_b\}$$

$$S=[S_1 \;\; S_2 \;\; \ldots \;\; S_{i^*} \;\; \ldots \;\; S_n]$$

$$W_a = g(1, i^*, k) \qquad W_b = g(i^*, n, p-k)$$

Figure 11: After $i^*$ has been found, HNC recursively computes $W_b = g(i^*, n, p-k)$, and returns $\min\{W_a, W_b\}$ as the optimal solution.

### Asymptotic running time

The authors suggest an elaborate scheme to pick $k$'s, so that the asymptotic running time is minimized. Using this scheme, they show that the running time is

$$O(n + p\log^2 n\sqrt{\frac{\log p}{\log\log p}} \times c^{\sqrt{\log p \log\log n}})$$

[11] If $k = 1$ the algorithm is simply a recursive version of Nicol's.

[12] $p$ binary search-trees are used for probing. The running time of a probe is $O(p \log \frac{n}{p})$.

for some constant $c$. Asymptotically this is the same as

$$O(n + p\log^2 p\sqrt{\frac{\log p}{\log\log p}} \times c^{\sqrt{\log p \log\log p}})$$

since it is true when $n < p^2$, and since $O(n)$ will dominate when $n \geq p^2$. This expression can be reduced to

$$O(n + p^{1+\epsilon})$$

for any small $\epsilon > 0$.

**Notes**

Although the choice of the initial $k$ has a significant impact on the performance of the algorithm, the authors do not provide an exact method for doing this. They give these equations to describe $k$:

$$\frac{p}{k} = O((2\log n)^j)$$

and

$$j = O(\sqrt{\frac{\log p}{\log\log n}})$$

. The implementations experimented with here pick the initial $k$ as

$$\frac{p}{(logn)^{\sqrt{\log p/\log\log n}}}$$

.

In fact, the optimal choice of the first $k$ is dependent upon the actual values of the sequence, as well as $p$ and $n$. To find the optimal value of the first $k$ for a given problem, it is necessary to experiment. Figure 12 illustrates the change in performance when choosing different initial $k$'s. The formula above chooses 14 as the first $k$.

To choose $k$'s after the first one, exact formulas are given by Han et. al. . These formulaes grows very slowly with $n$ and $p$, and the result is that the depth of the recursion tree is kept at a minimum. For the problems experimented on in this work, the HNC algorithm never recursed deeper

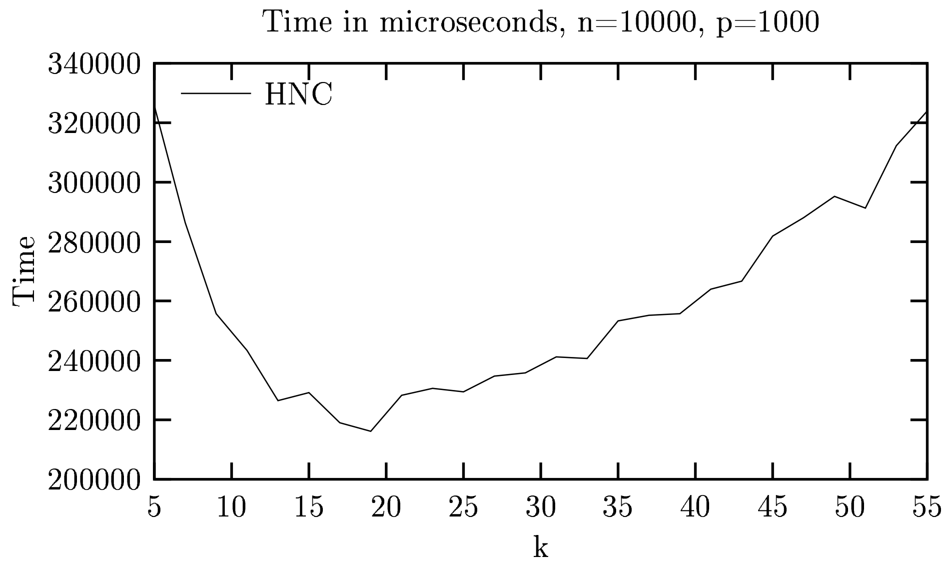


Figure 12: The effect of which balance-point is chosen first.

than 2 levels. Even so, recursion cost a lot using functional programming languages, and the performance of the algorithm would probably benefit from unrolling the recursion.

**Improvements**

It is possible to reduce the number of probes, by checking if the value to be probed with is in $< \frac{W}{p}, \frac{W}{p} + W_{max} >$. If not, it is possible to predict the result of a probe without executing it.

From Figure 13 it should be clear that the above mentioned improvement makes a substantial difference to the performance of HNC. Notice that the graphs for HNC and recursive Nicol's are identical until $p > 30$. This is because $k$ increases with $p$, but isnt larger than 1 until $p$ reaches 30.

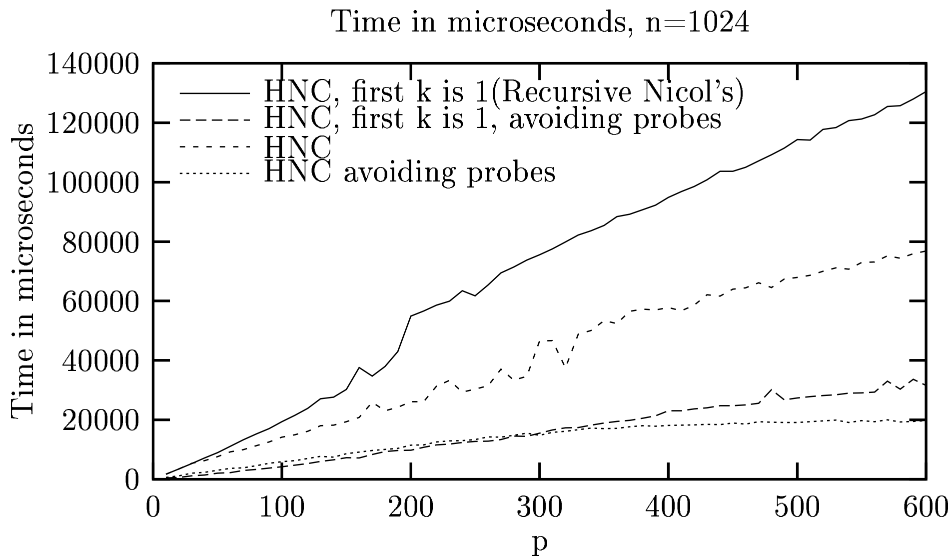


Figure 13: The performance of variants of the HNC algorithm.

**Implementation**

The implementations used to produce the graphs do not use $p$ binary trees for probing, as suggested by the authors. A binary probe of the sub-sequence is used instead. Asymptotically, this does not make a difference[13].

[13] The authors use an $O(p \log n)$ running time for the probe in the analysis.

## 5.7 Frederickson's algorithm

Frederickson's[7] is by far the most complex of the existing algorithms for the MinMax-problem. The approach is to search an $n \times n$ matrix of all possible solutions, gradually refining both the search interval and the probing procedure.

To illustrate, consider the sequence $S = [\,2\;3\;6\;1\;2\;2\;4\;5\,]$ from Chapter 3. $A$, the matrix of possible solutions looks like this:

| **x\y** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 5 | 11 | 12 | 14 | 16 | 20 | 25 |
| 2 | 0 | 3 | 9 | 10 | 12 | 14 | 18 | 23 |
| 3 | 0 | 0 | 6 | 7 | 9 | 11 | 15 | 20 |
| 4 | 0 | 0 | 0 | 1 | 3 | 5 | 9 | 14 |
| 5 | 0 | 0 | 0 | 0 | 2 | 4 | 8 | 13 |
| 6 | 0 | 0 | 0 | 0 | 0 | 2 | 6 | 11 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 9 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 |

Table 4: The elements are monotonically increasing from left to right and from bottom to top.

The element $A_{x,y}$ of the matrix represents the cost of the interval containing elements $x$ through $y$ in the sequence, i.e. $W_x + ... + W_y$. The elements of the matrix can be calculated in constant time once the prefix-sum-array, $T$ as described in Chapter 3, is computed. $A_{x,y} = T[y] - T[x-1]$, where $T[0] = 0$, thus it is not necessary to hold the matrix in a data-structure. Since the cost of all sub-sequences are in $A$, the optimal solution to the MinMax-problem must also be there.

The key to Frederickson's algorithm is the monotonic properties of $A$. For any sub-matrix of $A$, every element is equal to or less than the upper right element, and every element is equal to or greater than lower left element. This means that during a parametric search, an entire sub-matrix may be

discarded just by examining two elements of it. If the value of the upper right element of the sub-matrix is lower than the lower boundary on the optimal solution, or if the lower left element of the sub-matrix is greater than the upper boundary on the optimal solution, none of the values in the sub-matrix can be the optimal solution.

The algorithm works by splitting the matrix into sub-matrices, narrowing the search interval by finding the median of the matrices and probing it, and then discarding all matrices whose values fall outside the search interval. These steps are repeated until the size of the remaining matrices is $1 \times 1$. Only one distinct value will remain, and this is the optimal solution.

An elaborate dynamically improving probing scheme is developed to achieve linear running time. The key observation is that if a sub-matrix defined by two coordinates $(x_1, y_1)$ and $(x_2, y_2)$ does not contain any elements whose values are in the current search range, all probes with values in the search range on the sequence from $x_1$ to $x_2$ will produce the same partitioning. Frederickson calls this a dead interval, and develops an algorithm which gradually discovers more and more of these intervals. A dead interval can be probed in constant time by first applying a linear pre-processing procedure.

### Asymptotic running time

Frederickson develops several algorithms, each one building on and improving on the previous. His first uses a linear probe, and has an asymptotic running time of $O(n \log n)$. The next algorithm uses his special probe, looking for dead intervals once, and have a running time of $O(n \log \log n)$. Only these two algorithms are implemented in this work. By discovering dead intervals throughout the execution of the algorithm, the running time is improved to $O(n \log^* n)$.[14] Finally, linear running time is acheived through further improvements to the probing strategy.

[14]The function $\log^* k$ is the iterated logarithm of $k$, defined by $\log^* 1 = \log^* 2 = 1$ and $\log^* k = 1 + \log^* \lceil \log k \rceil$ for $k > 2$.

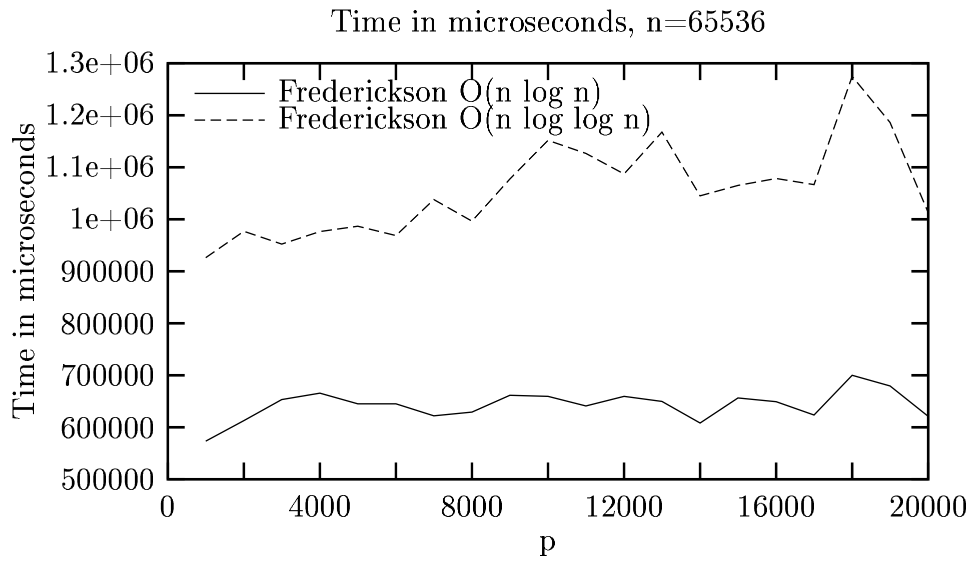


Figure 14: The performance of variants of the Frederickson algorithm.

**Performance**

As seen in Figure 14, the $O(n \log n)$ algorithm actually runs faster than the $O(n \log \log n)$ algorithm. The reason for this is that Fredrickson's improvements introduces large constant factors. Much more time is spent on improving the probe, than what is gained.

Figure 15 illustrates that when $n$ is high enough, the effect of the improvement is starting to show.

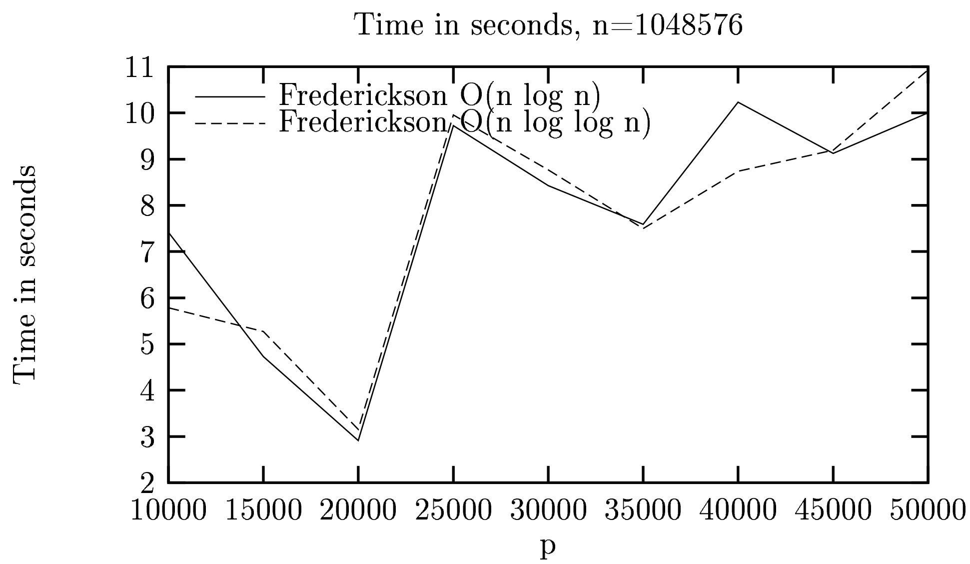


Figure 15: Performance of Frederickson for large problems.

# 6 Experimental results

In this chapter, we present tests of the algorithms from Chapter 6. They are tested on various problems, followed by a discussion of the results.

The algorithms are implemented in the C programming language, and tested on an Intel Celeron processor running at 768 MHz. See Chapter 1.4 for a detailed description of the conditions under which these experiments where conducted.

## 6.1 Previously published experiments

The authors of the PA-algorithm published experiments[15] on load balancing of various matrices from the linear programming domain[15]. The sequences to partition were obtained by multiplying the constraint matrix from a linear problem with its transpose, and collapsing the rows of the resulting matrix. Table 5 lists some properties[16] of the test-problems.

Table 6 lists the execution time in microseconds, of the algorithms from Chapter 5, when applied to the problems from Table 5. All implementations include the optimizations outlined in Chapter 5.

In general, Pinar and Aykanat found that PA was the fastest algorithm, closely followed by bisection, and that Nicol and OM were magnitudes slower. Of the algorithms in Table 6, Nicol, Bisection and PA seem to outperform the rest. Compared to the result of Pinar and Aykanat's experiments[17], Nicol's algorithm performs surprisingly well. It would seem like Pinar and Aykanat used a version of Nicol which was not optimized.

As expected, PA and MS are very fast for small $p$'s, but the running time for these algorithms increase drastically for large $p$. Aykanat, together with Kutluca and Kurc later published an article on image-space decomposition [10] in which Nicol's algorithm is recommended, with no mention of the PA-

[15] The problems are available at ftp://dollar.biz.uiowa.edu/pub/yyye/testprob/lp/

[16] For ken-11 and ken-18, the number of non-zeros is not the same as Pinar and Aykanat worked with. The reason for this is unknown, but the effect should be minimal.

[17] The "1D Decomposition" part of Table 2 in their paper.

| Matrix: | Rows/Cols: | Total nonzero: | Min pr. row | Max pr. row: |
|---|---|---|---|---|
| co9 | 10789 | 249205 | 1 | 707 |
| cq9 | 9278 | 221590 | 1 | 702 |
| cre-b | 9648 | 398806 | 1 | 904 |
| cre-d | 8926 | 372266 | 1 | 845 |
| ken-11 | 14694 | 82454 | 2 | 243 |
| ken-18 | 105127 | 609271 | 2 | 649 |
| mod2 | 34774 | 604910 | 1 | 941 |
| nl | 7039 | 105089 | 1 | 361 |
| pilot87 | 2030 | 238624 | 1 | 738 |
| world | 34506 | 582064 | 1 | 972 |

Table 5: Properties of sparse test-matrices.

algorithm. Frederickson and HNC, the asymptotically fastest algorithms, both have large constant factors which causes poor performance on relatively small problems.

## 6.2 Experiments on larger problems

The experiments of Pinar and Aykanat only considered very limited values of $p$, since their main focus was on applying the algorithms on load-balancing and not many SMP-machines with more than 256 processors existed. Cluster-technology, such as Beowulf[18], has since emerged to make higher values of $p$ more interesting[19].

When increasing the value of $p$, at some point the optimal solution will be the value of most expensive element in the sequence. For every $p$ larger than this, the optimal solution remains the same, since it must be larger than the value of the most expensive element, but decreases with $p$. For this

[18] http://www.beowulf.org

[19] Since communication between processors on these clusters are generally very expensive, minimizing communication cost is of special interest.

reason, the experiments are terminated before $p$ reaches $n$. The problems in Figure 16 and 17 are randomly generated, with numbers in the ranges $[4248, 2147438944]$ and $[0, 9999]$ respectively.

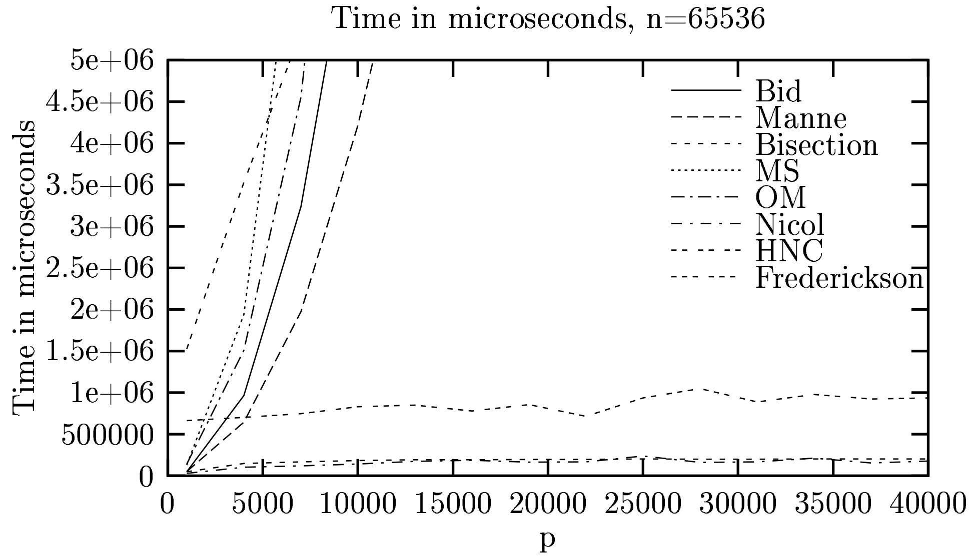


Figure 16: Performance-comparison of all implemented algorithms.

As seen in Figure 16, the algorithms based on moving delimiters all perform poorly when $p$ increases. The exception is Nicol, which from it's asymptotic running time should be much worse off. The modifications of the original algorithm are described on Page 32. When $p$ is larger than $n/\log n$ (4096 in this case), bisection switches from the $O(p \log n)$ probe to the linear probe(see Chapter 3), and after this, the running time is independent of $p$. Frederickson is slightly worse off than Nicol and bisection only because of high constant factors, but will probably outperform bisection if the numbers of the sequence are very large($\log W_{max} > n$).

Although not clear from Figure 16, the running time of HNC grows slower than that of PA and MS, and outperforms them on this problem when $p >$ 15000.

Figure 17 illustrates the performance of the three algorithms which are best suited for very large $n$ and $p$, applied to a sequence of 1048576 elements. To avoid overflow-problems, the randomly generated numbers of the sequence are between 0 and 10000, which benefits all three algorithms, especially bisection which beats Nicol with a narrow margin.

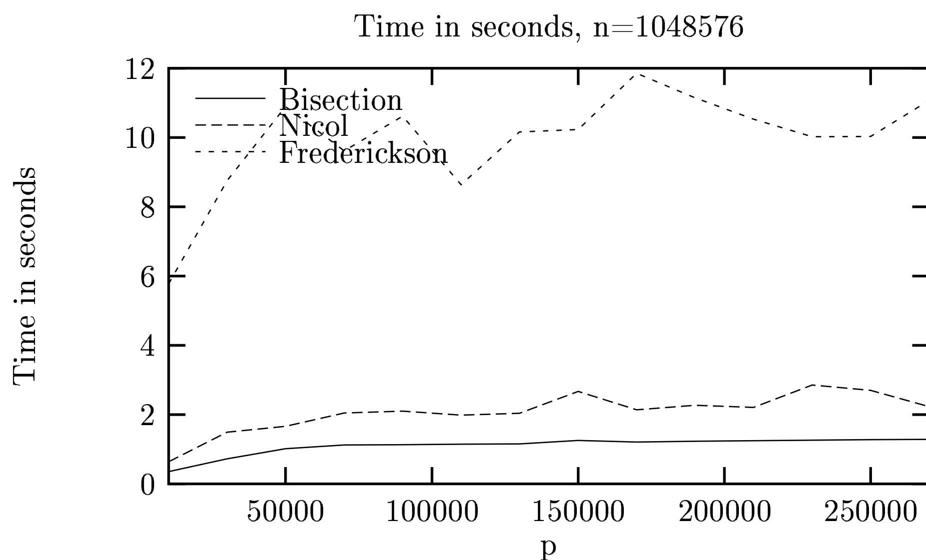


Figure 17: Performance-comparison of the algorithms best suited for large $n$ and $p$.

| Problem | p | $W_{opt}$ | MS | Nicol | Bis. | PA | OM | HNC | Fred. |
|---|---|---|---|---|---|---|---|---|---|
| **co9** | 16 | 15628 | 7257 | 1614 | 1567 | 1438 | 1037 | 29909 | 7159 |
| | 32 | 7871 | 7399 | 1650 | 1614 | 1682 | 1758 | 55816 | 7445 |
| | 64 | 3969 | 8313 | 2182 | 1789 | 2326 | 4072 | 82630 | 12121 |
| | 128 | 2039 | 14241 | 2312 | 2130 | 3879 | 17205 | 132942 | 14518 |
| | 256 | 1106 | 40716 | 3520 | 2707 | 12953 | 74971 | 197363 | 10497 |
| **cq9** | 16 | 13930 | 5715 | 1391 | 1322 | 1362 | 853 | 28107 | 11448 |
| | 32 | 6980 | 5996 | 1437 | 1239 | 1405 | 1209 | 46961 | 10814 |
| | 64 | 3512 | 6729 | 1748 | 1582 | 1509 | 3245 | 67917 | 14358 |
| | 128 | 1792 | 9662 | 2226 | 1986 | 3087 | 11429 | 113369 | 13357 |
| | 256 | 993 | 44646 | 3541 | 2448 | 11466 | 79649 | 162215 | 14137 |
| **cre-b** | 16 | 24988 | 6258 | 1350 | 1393 | 1343 | 694 | 30510 | 18336 |
| | 32 | 12564 | 6442 | 1671 | 1487 | 1356 | 994 | 62457 | 10153 |
| | 64 | 6306 | 6873 | 1745 | 1676 | 1530 | 2386 | 76977 | 17349 |
| | 128 | 3194 | 9228 | 2260 | 1974 | 2079 | 5867 | 132784 | 19306 |
| | 256 | 1714 | 24076 | 3610 | 2504 | 3193 | 14736 | 195534 | 26152 |
| **cre-d** | 16 | 23371 | 5688 | 1337 | 1251 | 1211 | 593 | 32731 | 22702 |
| | 32 | 11707 | 5645 | 1474 | 1319 | 1222 | 873 | 56598 | 9737 |
| | 64 | 5917 | 6167 | 1400 | 1499 | 1289 | 1897 | 79788 | 9498 |
| | 128 | 2992 | 8361 | 2349 | 1892 | 1873 | 5954 | 118246 | 24097 |
| | 256 | 1612 | 23349 | 2919 | 2372 | 2496 | 22023 | 186628 | 21728 |
| **ken-11** | 16 | 5164 | 9494 | 2134 | 2142 | 2064 | 1300 | 38506 | 24159 |
| | 32 | 2607 | 10448 | 2379 | 2287 | 2462 | 2962 | 85760 | 11130 |
| | 64 | 1305 | 11142 | 2594 | 2467 | 2641 | 5340 | 114972 | 30863 |
| | 128 | 688 | 21369 | 3296 | 2914 | 7611 | 30855 | 176352 | 52220 |
| | 256 | 345 | 35763 | 3963 | 3630 | 11541 | 75132 | 271674 | 9256 |
| **ken-18** | 16 | 38145 | 77857 | 15828 | 17021 | 15997 | 9911 | 559253 | 43643 |
| | 32 | 19083 | 79122 | 17167 | 16012 | 17015 | 12641 | 1188495 | 155864 |
| | 64 | 9605 | 86243 | 17881 | 17729 | 19295 | 27136 | 1520262 | 190719 |
| | 128 | 4805 | 93609 | 18812 | 17535 | 23146 | 54566 | 2358062 | 280242 |
| | 256 | 2489 | 195032 | 21880 | 19333 | 73966 | 303528 | 3594671 | 125709 |
| **mod2** | 16 | 37831 | 24362 | 5258 | 5170 | 5053 | 3058 | 165927 | 34385 |
| | 32 | 18925 | 24887 | 5373 | 5469 | 5194 | 3228 | 334317 | 31837 |
| | 64 | 9473 | 26580 | 5889 | 5929 | 5425 | 4017 | 442380 | 33778 |
| | 128 | 4746 | 27659 | 7169 | 6656 | 5984 | 6153 | 669967 | 29718 |
| | 256 | 2393 | 36780 | 10678 | 8225 | 10264 | 33394 | 1011793 | 42636 |
| **nl** | 16 | 6591 | 4049 | 870 | 988 | 898 | 619 | 18602 | 5719 |
| | 32 | 3312 | 4288 | 1122 | 1086 | 985 | 1251 | 49886 | 4546 |
| | 64 | 1681 | 5333 | 1216 | 1095 | 1414 | 2779 | 53578 | 5044 |
| | 128 | 862 | 7919 | 1950 | 1538 | 2346 | 11603 | 98579 | 6193 |
| | 256 | 469 | 44029 | 3110 | 1886 | 7301 | 67706 | 136854 | 9898 |
| **pilot87** | 16 | 15085 | 907 | 228 | 219 | 190 | 146 | 2770 | 3571 |
| | 32 | 7595 | 1101 | 348 | 302 | 267 | 349 | 6885 | 5877 |
| | 64 | 3840 | 1346 | 408 | 421 | 406 | 1024 | 9612 | 3823 |
| | 128 | 1977 | 2435 | 825 | 691 | 872 | 3612 | 16871 | 4568 |
| | 256 | 1065 | 7106 | 1307 | 760 | 3209 | 14852 | 32969 | 8776 |
| **world** | 16 | 36403 | 24329 | 5197 | 5239 | 5038 | 3042 | 168786 | 27003 |
| | 32 | 18220 | 25441 | 5698 | 5373 | 5184 | 3217 | 365711 | 43133 |
| | 64 | 9124 | 26584 | 6100 | 6046 | 5821 | 4717 | 460257 | 35887 |
| | 128 | 4584 | 30120 | 7040 | 6774 | 7127 | 12697 | 690922 | 37719 |
| | 256 | 2299 | 36537 | 8626 | 8064 | 9200 | 31917 | 1046781 | 43357 |

Table 6: Execution time in microseconds

# 7 Conclusion

In Chapter 5, many of the discussed algorithms were improved by using the bounds on the optimal solution, described in Chapter 2. Although this can not be shown to have an effect on the asymptotic running time, experiments proved that optimizations based on these bounds can make a significant difference.

Which algorithm to choose depends very much on the properties of the problem. $n$, $p$ and the size and distribution of the numbers in the sequence makes the performance of the different algorithms vary. For $p \leq 64$, Nicol, PA and bisection run fastest. When $p$ grows beyond this, Nicol and bisection outperforms PA. While Frederickson and HNC have the best asymptotic running times, very high constant factors makes these algorithms unattractive in practice. For practical experiments, bisection seems to perform best, and is easy to implement(as opposed to Frederickson), although it will have problems on extremely high numbers.

The bisection-algorithm was changed to produce perfect accuracy when working with real numbers, without increasing the asymptotic running time. Over several experiments, this algorithm acheived the best performance. Future research may find a better asymptotic running time than $O(n^2 p \log n)$ which was developed in this work, by exploiting the properties of the search space.

Anily and Federgruen[1] lists several problems similar to the MinMax-problem. It would be interesting trying to apply the optimizations developed in this work to algorithms for solving related problems.

# References

[1] S. Anily and A. Federgruen, *Structured partitioning problems*, Operations Research, 13 (1991), pp. 130–149.

[2] R. Becker, Y. Perl, and S. Schach, *An efficient algorithm for min-max tree partition*, Quaestiones Inform., (1982), pp. 27–30.

[3] R. I. Becker and Y. Perl, *The shifting algorithm technique for the partitioning of trees*, Disc. App. Math., (1995), pp. 15–34.

[4] S. H. Bokhari, *Partitioning problems in parallel, pipelined, and distributed computing*, IEEE Trans. Comput., 37 (1988), pp. 48–57.

[5] S. H. Bokhari and M. A. Iqbal, *Efficient algorithms for a class of partitioning problems*, Tech. Rep. ICASE Report No. 90-49, NASA Langley Research Center, 1990.

[6] H.-A. Choi and B. Narahari, *Algorithms for mapping and partitioning chain structured parallel computations*, in Proc. Intl. Conf. on Parallel Processing, vol. I, 1991, pp. 625–628.

[7] G. N. Frederickson, *Optimal parametric search algorithms in trees i: Tree partitioning*, 1992.

[8] M. R. Garey and D. S. Johnson, *Complexity results for multiprocessor scheduling under resource constraints*, SIAM J. Comput., 4 (1975), pp. 397–411.

[9] Y. Han, B. Narahari, and H.-A. Choi, *Mapping a chain task to chained processors*, Information Processing Letters, 44 (1992), pp. 141–148.

[10] T. M. K. Huseyin Kutluca and C. Aykanat, *Image-space decomposition algorithms for sort-first parallel volume rendering of unstructured grids*, in The Journal of Supercomputing, 2000, pp. 15(1):51–93.

[11] M. A. Iqbal, *Approximate algorithms for partitioning and assignment problems*, Int. J. Parallel Programming, (1991), pp. 341–361.

[12] F. Manne and T. Sørevik, *Optimal partitioning of sequences*, J. Alg., 19 (1995), pp. 235–249.

[13] D. M. Nicol, *Rectilinear partitioning of irregular data parallel computations*, J. Par. Dist. Comp., (1994), pp. 119–134.

[14] B. Olstad and F. Manne, *Efficient partitioning of sequences*, IEEE Trans. Comput., 44 (1995), pp. 1322–1326.

[15] A. Pinar and C. Aykanat, *Sparse matrix decomposition with optimal load balance*, in Proceedings of the 4th International Conference on High Performance Computing, 1997, pp. 224–229.