Efficient Partitioning of Sequences

Bjørn Olstad* and Fredrik Manne†

Abstract

We consider the problem of partitioning a sequence of n real numbers into p intervals such that the cost of the most expensive interval, measured with a cost function f is minimized. This problem is of importance for the scheduling of jobs both in parallel and pipelined environments. We develop a straightforward and practical dynamic programming algorithm that solves this problem in time $O(p(n-p))$, which is an improvement of a factor of $\log p$ compared to the previous best algorithm. A number of variants of the problem are also considered.

1 Introduction

The scheduling of jobs to processors so as to minimize some cost function is an important problem in many areas of computer science. In many cases, these problems are known to be NP-hard [6]. Thus if scheduling problems are to be solved optimally in polynomial time, they must contain enough restrictions to make them tractable. In this paper we will consider one such problem. To motivate why this particular problem is of interest consider the following example from Bokhari[3]:

In communication systems it is often the case that a continuous stream of data packages have to be received and processed in real time. The processing can among other things include demodulation, error correction and possibly decryption of each incoming data package before the contents of the package can be accessed [7]. Assume that n computational operations are to be performed in a pipelined fashion on a continuous sequence of incoming data packages. If we have n processors we can assign one operation to each processor and connect the processors in parallel. The time to process the data will now be dominated by the processor that has to perform the most time consuming operation. With this mapping of the operations to the processors, a processor will be idle once it is done with its operation and have to wait until the

*Department of Computer Systems and Telematics, The Norwegian Institute of Technology, N-7034 Trondheim-NTH, Norway. Email: Bjoern.Olstad@idt.unit.no

†Norsk Hydro a.s, N-5020 Bergen, Norway. Email: fmanne@bg.nho.hydro.com

processor that has the most time consuming operation is done, before it can get a new data package. This is inefficient if the time to perform each task varies greatly. Thus to be able to utilize the processors more efficiently we get the following problem: Given n consecutively ordered tasks, each taking $f(i)$ time, and p processors. Partition the tasks into p consecutive intervals such that the maximum time needed to execute the tasks in each interval is minimized.

In [3] it is also described how a solution to this problem can be used in parallel processing as compared to pipelined.

The outline of this paper is as follows: In Section 2 we describe the main partitioning problem and give an overview of recent work. In Section 3 we develop a new efficient algorithm for solving this problem. A generalization of the partitioning problem is described and solved in Section 4. In the final section we summarize and point to areas of future work.

2 A partitioning problem

We will in this section give a formal definition of the main partitioning problem and also recapitulate previous work. The problem as stated in [11] is as follows:

Let the two integers $p \leq n$ be given and let $\{\sigma_0, \sigma_1, ..., \sigma_{n-1}\}$ be a finite ordered set of bounded real numbers. Let $R = \{r_0, r_1, ..., r_p\}$ be a set of integers such that $r_0 = 0 \leq r_1 \leq ... \leq r_{p-1} \leq r_p = n$. Then R defines a partition of $\{\sigma_0, \sigma_1, ..., \sigma_{n-1}\}$ into p intervals: $\{\sigma_{r_0}, \sigma_1, ..., \sigma_{r_1-1}\}$, $\{\sigma_{r_1}, ..., \sigma_{r_2-1}\}$,...,$\{\sigma_{r_{p-1}}, ..., \sigma_{r_p-1}\}$. If $r_i = r_{i+1}$ then the interval $\{\sigma_{r_i}, ..., \sigma_{r_{i+1}-1}\}$ is empty.

Let f be a function, defined on intervals taken from the sequence $\{\sigma_0, \sigma_1, ..., \sigma_{n-1}\}$, that satisfies the following two conditions of non-negativity and monotonicity:

$$f(\sigma_i, \sigma_{i+1}, ..., \sigma_j) \;=\; f(\;[i,j]\;) \;=\; f(i,j) \geq 0 \tag{1}$$

for $0 \leq i, j \leq n-1$, with equality if and only if $j < i$.

$$f(i+1, j) < f(i,j) < f(i, j+1) \tag{2}$$

for $0 \leq i \leq j < n-1$. In addition we will only assume the following mild conditions on the computability of the set-function f to be satisfied: (1) The function $f(\sigma_i)$ can be computed in $O(1)$ time and (2) Given $f(i,j)$ we can calculate f in $O(1)$ time, when either i or j has been increased or decreased by one. The problem is then:

MinMax Find a partition R such that the associated cost $\max_{i=0}^{p-1}\{f(r_i, r_{i+1}-1)\}$ is minimum over all partitions of $\{\sigma_0, ..., \sigma_{n-1}\}$.

Bokhari [3] presents the MinMax problem and gives an $O(n^3p)$ algorithm using a bottleneck-path algorithm. Anily and Federgruen [2] and Hansen and Lih [8] independently presented the same dynamic programming algorithm with time complexity $O(n^2p)$. Manne and Sørevik [11] then presented an $O(p(n-p)\log p)$ algorithm based on iteratively improving a given partition. They also described a bisection method based on a simple $O(n)$ feasibility test for finding an approximate solution which runs in time $O(n\log(f(0,n-1)/\epsilon))$, where ϵ is the desired precision.

Frederickson [4, 5] has studied partitioning of trees with parametric search techniques and gives a good account on related work. Frederickson restricts the weight function to ordinary addition and develops an intricate algorithm that solves this class of sequence partitioning problems in $O(n)$ time. The optimal parametric search algorithm is a carefully designed bisection method in the space of n^2 candidate values generated by all possible intervals in the sequence. Frederickson's algorithm assumes that $f(i,j)$ is computable in $O(1)$ time for all i and j after an $O(n)$ preprocessing algorithm. This assumption is valid if $f(i,j)$ is defined by an associative and invertible operator on the elements in the interval $[i,j]$. Our algorithm has the advantage of being straightforward and practical. In addition, our mild conditions on the computability of f includes non-associative operators that for example occur in signal regression problems [9]. The asymptotic running time for our algorithm and that of Frederickson both equal $O(np)$ for the generalized MinMax problem given in Section 4. Both algorithms also have $O(np)$ running time for the problem of solving the MinMax problem for p' segments with $p' = 1,..,p$. The last problem is relevant in signal analysis where p is not known a priori but rather a crucial parameter for optimization.

In the next section we show how the dynamic programming algorithm first presented by Anily and Federgruen can be improved to run in $O(p(n-p))$ time.

3 A new algorithm

In this section we describe a new efficient dynamic programming algorithm for solving the MinMax problem. We start by describing the algorithm by Anily and Federgruen.

Let $g(i,k)$ be the cost of the most expensive interval in an optimal partition of $[i, n-1]$ into k intervals where $1 \leq k \leq p$ and $0 \leq i \leq n$. The cost of the optimal solution is then given by $g(0,p)$. Once $g(0,p)$ is known, the positions of the delimiters can be obtained by a straightforward $O(n)$ calculation that grows intervals from left to right. A delimiter is inserted each time $f(i,j)$ has exceeded $g(0,p)$. The following boundary conditions apply to g:

$$g(i,1) = f(i, n-1) \tag{3}$$

$$g(i, n-i) = \max_{i \leq j < n} f(j,j) \tag{4}$$

Note that for $n - i < k \leq p$ we have $g(i,k) = g(i, n-i)$.

The following recursion, first presented by Anily and Federgruen [2], shows how $g(i,k)$ can be computed for $2 \leq k < n-i$:

$$g(i,k) = \min_{i \leq j \leq n-k} \max\{f(i,j), g(j+1, k-1)\} \tag{5}$$

This formula suggests that if one has access to each value of $g(j+1, k-1)$, $i \leq j \leq n-k$, then $g(i,k)$ can be computed by looking up $n-k-i+1$ values of g and by calculating $n-k-i+1$ values of f. This gives a total time complexity of $O(n^2 p)$ and a space complexity of $O(n)$ to compute $g(0,p)$.

There are two basic ways that the complexity of a dynamic programming algorithm can be reduced: either by general methods such as presented in [1, 13] or by more problem dependent methods such as [12]. The reader is referred to [1] and the references therein for a recent account on algorithmic improvements on dynamic programming algorithms using array searching. It should be noted that our weight function satisfies the Monge condition described in [1] if $f(i,j)$ is raised to a high enough power. Our result is achieved by an approach similar to [12] taking advantage of the monotonicity properties which are derived for the given problem.

In the rest of this section we show how $g(0,p)$ can be computed in $O(p(n-p))$ time. This result is due to the fact that $g(i,k)$ is decreasing in i. As a consequence the optimal value of j in (5) increases as i increases.

If $R = \{r_0 = i, r_1, ..., r_k = n\}$ defines a partitioning of $[i, n-1]$ of cost $g(i,k)$ we will say that R is *implied* by $g(i,k)$.

It can easily be shown that $g(i,k)$ increases monotonically as i decreases:

Lemma 1 *Let i, i' be integers such that $0 \leq i \leq i' \leq n$. Then $g(i,k) \geq g(i',k)$ for $1 \leq k \leq p$.*

From Lemma 1 it follows that for fixed k the function g increases monotonically in the size of the interval to be partitioned. The following lemma shows how in certain cases we can compute $g(i,k)$ using the first interval $[i+1, j]$ in a partitioning implied by $g(i+1,k)$ and the value of $g(j+1, k-1)$.

Lemma 2 *Let $[i+1, j]$ be the first interval in a partitioning implied by $g(i+1,k)$, $k > 1$. If $f(i,j) \leq g(j+1,k-1)$ then $g(i,k) = g(j+1,k-1)$.*

Proof: If $f(i,j) \leq g(j+1,k-1)$ then $f(i+1,j) < g(j+1,k-1)$. Since $g(i+1,k) = \max\{f(i+1,j), g(j+1,k-1)\}$ we have $g(i+1,k) = g(j+1,k-1)$. Together with $g(i,k) \leq \max\{f(i,j), g(j+1,k-1)\} = g(j+1,k-1)$ this shows that $g(i,k) \leq g(i+1,k)$. From Lemma 1 we know that $g(i,k) \geq g(i+1,k)$ and it follows that $g(i,k) = g(i+1,k)$ and thus $g(i,k) = g(j+1,k-1)$. $\square$

Note that Lemma 2 could also have been stated: If $f(i,j) \leq g(i+1,k)$ then $g(i,k) = g(i+1,k)$. The present formulation was chosen in order to emphasize the relationship to (5).

We now discuss how $g(i,k)$ can be computed efficiently if Lemma 2 does not apply. For this purpose we need the following definition:

Definition 3 *Let i and k be integers such that $0 \leq i < n$ and $2 \leq k \leq p$. Further let $s_{i,k}$ be an integer, $i \leq s_{i,k} < n$, such that $f(i, s_{i,k}-1) < g(s_{i,k}, k-1)$ and $f(i, s_{i,k}) \geq g(s_{i,k}+1, k-1)$. Then $s_{i,k}$ is a balance point.*

It follows from Definition 3 that $\max\{f(i, s_{i,k}), g(s_{i,k}+1, k-1)\} = f(i, s_{i,k})$ and that $\max\{f(i, s_{i,k}-1), g(s_{i,k}, k-1)\} = g(s_{i,k}, k-1)$. We now show that $s_{i,k}$ is well defined.

Lemma 4 *The balance point $s_{i,k}$ exists and is unique.*

Proof: The result follows directly from the following two facts: (1) $f(i,j)$ is a *strictly monotonically increasing* function of j on $i-1 \leq j \leq n$ for which $f(i,i-1)=0$, and (2) $g(j,k-1)$ is a *monotonically decreasing* function of j on $i-1 \leq j \leq n$ for which $g(n,k-1)=0$. $\square$

We now state our main theorem. It tells us how the balance point $s_{i,k}$ can be used to compute $g(i,k)$.

Theorem 5 *For $k \geq 2$, $g(i,k) = \min\{f(i,s_{i,k}), g(s_{i,k},k-1)\}$.*

Proof: We consider two cases: Suppose first that $f(i,s_{i,k}) \leq g(s_{i,k},k-1)$. Since $f(i,s_{i,k}) \geq g(s_{i,k}+1,k-1)$, a partitioning of $[i,n-1]$ into k intervals that costs less than $f(i,s_{i,k})$ must have a first interval $[i,j]$ where $j < s_{i,k}$. The cost of such a partitioning is $\gamma = \max\{f(i,j), g(j+1,k-1)\}$. From Lemma 1 it follows that $g(s_{i,k},k-1) \leq g(j+1,k-1)$ and since $g(j+1,k-1) \leq \gamma$ we have $g(s_{i,k},k-1) \leq \gamma$. By the assumption $f(i,s_{i,k}) \leq g(s_{i,k},k-1)$ we have that $f(i,s_{i,k}) \leq \gamma$ in contradiction to the definition of j. Thus it follows that $g(i,k) = f(i,s_{i,k})$.

Assume now that $f(i,s_{i,k}) > g(s_{i,k},k-1)$. Since $f(i,s_{i,k}-1) < g(s_{i,k},k-1)$ a partitioning of $[i,n-1]$ into k intervals that costs less than $g(s_{i,k},k-1)$ must have the first interval $[i,j]$ such that $j \geq s_{i,k}$. The cost of such a partitioning is $\gamma = \max\{f(i,j), g(j+1,k-1)\}$. Since $f(i,s_{i,k}) \leq f(i,j)$ and $f(i,j) \leq \gamma$ it follows that $f(i,s_{i,k}) \leq \gamma$. Together with the assumption that $f(i,s_{i,k}) > g(s_{i,k},k-1)$ this shows that $g(s_{i,k},k-1) < \gamma$ in contradiction to the definition of j and therefore $g(i,k) = g(s_{i,k},k-1)$. $\square$

We can now show how $g(i,k)$ and the first interval implied by $g(i,k)$ can be computed efficiently from the following information: The first interval $[i+1,j]$ implied by $g(i+1,k)$ and $g(l,k-1)$ for $i < l \leq j+1$.

If $f(i,j) \leq g(j+1,k-1)$ then as noted in Lemma 2 we have $g(i,k) = g(j+1,k-1)$ and the size of the first interval implied by $g(i,k)$ is $[i,j]$.

If $f(i,j) > g(j+1,k-1)$ then we locate $s_{i,k}$ and apply Theorem 5. From the definition of $s_{i,k}$ and Lemma 4 we see that $f(i,j) > g(j+1,k-1)$ implies that $i \leq$

$s_{i,k} \leq j$. We first test if $f(i,i) \geq g(i+1, k-1)$. If this is true then $s_{i,k} = i$ (since $f(i, i-1) = 0$) and from Theorem 5 if follows that $g(i,k) = f(i,i)$ and the first interval contains only σ_i. If $f(i,i) < g(i+1, k-1)$ then we know that $i < s_{i,k}$.

Assume now that $f(i,j) > g(j+1, k-1)$ and $f(i,i) < g(i+1, k-1)$. To locate $s_{i,k}$ we compute $f(i, j-1)$ and compare with $g(j, k-1)$. If $f(i, j-1) < g(j, k-1)$, then $j = s_{i,k}$. If $g(j, k-1) \leq f(i, j-1)$ we reduce j by one and repeat the process. This way we will eventually get j such that $f(i, j-1) < g(j, k-1)$ and $f(i,j) \geq g(j+1, k-1)$. From Definition 3 it follows that $j = s_{i,k}$. We can now compute $g(i,k)$ by applying Theorem 5. The size of the first interval is $[i,j]$ if $f(i,j) < g(j, k-1)$ and $[i, j-1]$ if $f(i,j) \geq g(j, k-1)$.

From the above it is clear that to compute $g(i,k)$ we only make use of $g(l, k-1)$ where $i < l \leq j+1$. This implies that for a fixed value of k we only need to compute $g(i,k)$ for $p - k \leq i \leq n - k$ to be able to compute $g(0,p)$.

Before giving the complete algorithm we note that (4) can be transformed into the following recursive formula:

$$g(n-i, i) = \max\{f(n-i, n-i), g(n-i+1, i-1)\}.$$

We use (3) to compute $g(i, 1)$ for $p - 1 \leq i < n$. The complete algorithm is shown in Figure 1.

Now we show that the time complexity of this algorithm is $O(p(n-p))$. First we argue that under the assumptions on f made in Section 2, we can calculate each needed value of f in $O(1)$ time. It is clear that this is true when evaluating $g(i,1)$ in (6). In (7) and (9) we evaluate f on only one element which can be done in $O(1)$ time. If we ignore (9) then each calculation (except one) of $f(i,j)$ in (8), (10) and (11) is directly preceded by one of the following calculations: $f(i+1, j), f(i, j+1), f(i, j-1)$. The only exception occurs when calculating $f(i,j)$ in (8) for $i < n-k-1$ and (12) was true for $i+1$. Then $f(i+1, j+1)$ was calculated in (11) prior to (8). Since the argument of f is shifted only by two elements from (11) to (8) we can still calculate $f(i,j)$ in (8) using only $O(1)$ time. (Note also that in this case $f(i+1, j)$ was calculated in (10)). From this we see that we can calculate each needed value of f in $O(1)$ time.

Now we turn our attention to the overall time complexity of the algorithm. The initialization in (6) can be done in $O(n-p)$ time. In the innermost for-loop the only

```
for i := n − 1 down to p − 1 do                                          (6)
    g(i, 1) := f(i, n − 1);

for k := 2 to p do
    g(n − k, k) := max{f(n − k, n − k), g(n − k + 1, k − 1)};            (7)
    j := n − k;

    for i := n − k − 1 down to p − k do
        if f(i, j) ≤ g(j + 1, k − 1) then                                (8)
          g(i, k) := g(j + 1, k − 1);
        else
          if f(i, i) ≥ g(i + 1, k − 1) then                              (9)
            g(i, k) := f(i, i);
            j := i;
          else
            while f(i, j − 1) ≥ g(j, k − 1) do                           (10)
                j := j − 1;
            g(i, k) := min{f(i, j), g(j, k − 1)};                        (11)
            if g(i, k) = g(j, k − 1)                                     (12)
              j := j − 1;
          end-else
        end-else
    end-do
end-do
```

Figure 1: The new algorithm for solving the MinMax problem

statement that cannot be executed in $O(1)$ time is the while-loop (10). We argue that for a fixed value of k, (10) is not true more than $n-p-1$ times. The value of j is initially set to $n-k$. Whenever (10) is reached the value of j is reduced in steps of one until $j = s_{i,k}$. Since $s_{i,k} > i$ it follows that j will never be reduced below $i+1$ in (10). For fixed k the lowest value i can have is $p-k$. Thus for each value of k, (10) can at most be true $n-p-1$ times.

If we ignore (10) then the time complexity of the innermost for-loop is $O(n-p)$. Thus by amortizing the time spent on (10) over the time spent on the innermost for-loop we see that (10) can be regarded as taking constant time. The for-loop involving k is executed $p-1$ times giving a total time complexity of $O(p(n-p))$ for the algorithm. We observe that the algorithm degenerates to an $O(n)$ algorithm for $p=1$ and $p=n$.

As stated earlier this is an improvement by a factor of $\log p$ compared to the Manne and Sørevik algorithm [11]. It should be noted that the algorithm presented in this paper has time complexity $\Omega(p(n-p))$ on *every* input. The highest known lower bound for the Manne and Sørevik algorithm is also $\Omega(p(n-p))$ but for an actual set of values it might take less time.

In order to compute $g(i,k)$ for fixed values of i and k, we need only $g(j,k-1)$, $i < j \leq n-k+1$, and the length of the first interval implied by $g(i+1,k)$. Thus our algorithm like that of Anily and Federgruen can be implemented using only $O(n)$ space.

4 The generalized MinMax problem

When scheduling jobs to processors each processor might have limited storage for its job queue. In this section we will show how the MinMax problem can be solved with a size function on each interval and when we demand that the size of each interval be smaller than some preset limit.

Let $s(i)$ be the *size* of element σ_i where s is a function defined on consecutive intervals of $\{\sigma_0, \sigma_1, ..., \sigma_{n-1}\}$ such that s satisfies (1) and (2). We also assume that the time complexity of computing s is similar to that of f. We now have the following problem:

Bounded MinMax Given p positive real numbers $U_0, U_1, ..., U_{p-1}$, find an optimal partition for the MinMax problem with the constraint that $s(r_j, r_{j+1}-1) \leq U_j$ for $0 \leq j < p$.

We will first demonstrate how the bounded MinMax problem can be viewed as a special case of the generalized MinMax problem:

Generalized MinMax Find a partition R such that $\max_{i=0}^{p-1}\{f_i(\sigma_{r_i}, ..., \sigma_{r_{i+1}-1})\}$ is minimum over all partitions of $\{\sigma_0, ..., \sigma_{n-1}\}$.

Note that the generalized MinMax problem is defined with different cost functions for each of the intervals in the partition. This could be convenient if we were to distribute data on a sequence of processors where the processors operate at different speeds. We will require that each of the f_i functions satisfy the properties required by f in the original MinMax problem. We can now think of the bounded MinMax problem as a generalized MinMax problem where the cost functions are defined as follows:

$$f_k(i,j) = f(i,j) + T_k(\ s(i,j)\ ) \quad \text{for } 0 \leq k < p \tag{13}$$

T_k is a threshold operator that encodes the constraints in the bounded MinMax problem:

$$T_k(x) = \begin{cases} \infty & \text{if } x > U_k \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

We observe that the bounded MinMax problem can be solved with the original MinMax algorithm if U_k is independent of k. The cost function must in this case be updated according to (13).

The algorithm for solving the generalized MinMax problem is given in Figure 2. We have not included the computations needed to calculate $g(i,1)$ since it is obvious how (6) must be changed to reflect the new conditions. Different cost functions can give solutions with empty intervals. Therefore the i-loop must be repeated n times. Since we might need $g(i,k-1)$ in order to compute $g(i,k)$ the if-test in (9) is not included in the algorithm in Figure 2.

The asymptotic time complexity of the algorithm is $O(np)$. This is still an improvement of a factor of $\log p$ compared to the algorithm presented by Manne and Sørevik for the bounded MinMax problem. The parametric search procedure by Fredrickson [4, 5] will also give a $O(np)$ solution when it is applied to the generalized MinMax problem, but as noted in Section 2 our simple and straightforward algorithm must be expected to have a much smaller constant term. The number of candidate values have increased to n^2p and the parametric search must explore p unrelated $n \times n$ sorted matrices.

5 Conclusion

We have in this paper shown how the complexity of the dynamic programming method used by Anily and Federgruen to solve the MinMax problem can be reduced from $O(n^2p)$ to $O(p(n-p))$. We obtain the improvement by taking advantage of the monotone

```
for k := 2 to p do
    g(n, k) := 0;
    j := n − 1;

    for i := n − 1 down to 0 do
        if f_{p−k}(i, j) ≤ g(j + 1, k − 1) then
            g(i, k) := g(j + 1, k − 1);
        else
            while f_{p−k}(i, j − 1) ≥ g(j, k − 1) do
                j := j − 1;
            g(i, k) := min{f_{p−k}(i, j), g(j, k − 1)};
            if g(i, k) = g(j, k − 1) then
                j := j − 1;
        end-else
    end-do
end-do
```

Figure 2: The algorithm for solving the generalized MinMax problem

properties of the cost functions. Where applicable, this technique seems to be a useful way of reducing the complexity of dynamic programming algorithms.

In [11] a number of problems related to the MinMax problem were described. Each of these problems can also be solved by slight modifications of our main algorithm. In particular, we solve the MinMax problem for p' segments with $p' = 1, .., p$ in $O(np)$ time and the circular version of the MinMax problem is solved in $O(n(n-p))$ time. We note that the bounded MinMax problem can be solved by the algorithm for the generalized MinMax problem even if we had used an independent size function s_i as long as each s_i satisfies (1) and (2).

The MinMax problem can be generalized to higher dimensions than one. In [10] it is described how a solution of the two dimensional partioning problem can be used to speed up sparse matrix-vector multiplication on a systolic array of processors.

6 Acknowledgement

The authors thank Bengt Aspvall and Tor Sørevik for constructive comments. We would also like to thank the referees for their many helpful comments and for bringing some interesting references to our attention.

References

[1] A. Aggarwal and J.K. Park, *Improved algorithms for economic lot size problems*, Operations Research, 41 (1993), pp. 549–571.

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

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

[4] G.N. Frederickson, *Optimal algorithms for partitioning trees and locating p-centers in trees*, Technical Report CSD-TR-1029, Purdue University, 1990.

[5] G.N. Frederickson, *Optimal algorithms for partitioning trees and locating p-centers in trees*, in Proceedings of 2nd ACM-SIAM Symposium on Discrete Algorithms, San Francisco, California, 1991, pp. 168–177.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability*, Freeman, 1979.

[7] F. Halsall, *Data Communications, Computer Networks and OSI*, Addison Wesley, 1988.

[8] P. Hansen and K.-W. Lih, *Improved algorithms for partitioning problems in parallel, pipelined, and distributed computing*, IEEE Trans. Comput., 41 (1992), pp. 769–771.

[9] W. Press, B. Flannery, S. Teukolsky and W. Vetterling, *Numerical Recipes in Pascal*, Cambridge University Press, 1989.

[10] F. Manne, *Load Balancing in Parallel Sparse Matrix Computations*, PhD thesis, University of Bergen, Norway, 1993.

[11] F. Manne and T. Sørevik, *Optimal partitioning of sequences*, Tech. Report CS-92-62, University of Bergen, Norway, 1992.

[12] B. Olstad and H. E. Tysdahl, *Improving the computational complexity of active contour algorithms*, in Proceedings of the 8th Scandinavian Conferance on Image Analysis, Tromsø, Norway, 1993.

[13] F. F. Yao, *Speed-up in dynamic programming*, SIAM J. Alg. Discrete Meth., 3 (1982), pp. 532–540.