Efficient Partitioning of Sequences with an Application to Sparse Matrix Computations

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 an efficient 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. We demonstrate how a solution to the partitioning problem can be used to speed up sparse matrix, dense vector multiplication on a SIMD computer.

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

[†]Department of Informatics, University of Bergen, N-5020 Bergen, Norway.
Email: Fredrik.Manne@ii.uib.no

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:

The computational complexity of forming the product $Ax = y$ where A is an $M \times N$ sparse matrix and x is a dense vector, is linear in the number of nonzeros in A. When forming this product the elements in each column of A are multiplied by the same element from x. We consider how this can be performed on a parallel computer containing p processors connected in a line (or ring) where each processor has local memory. One obvious way this can be done is to divide the columns of A among the processors and distribute each element of x to the appropriate processor. In this setting the multiplication algorithm would progress in two stages. In the first stage each processor multiplies the values in A with the corresponding element from x and accumulates its local contribution to y. In the second stage the processors communicates in order to sum the values of y. The computational complexity of such a method will be limited by the processor that gets assigned the highest number of nonzeros from A. This gives rise to the following partitioning problem:

Given a vector $v \in Z^N$ that contains the number of nonzeros in each column of A, partition v into p consecutive intervals such that the maximal sum of the elements in each interval is minimized.

In Sections 3 and 4 we develop a new efficient algorithm to solve this problem and a number of variants of it. If we instead of processors coupled in a line, have an $m \times n$ processor array, the partitioning approach can be used as a heuristic to achieve a good load balance: Form vectors v and w respectively containing the number of elements in each column and row. The vector v is then partitioned into n intervals and w into m intervals so as to minimize the maximal sum of the elements in each interval. In Section 5 we will show a practical example of such a method.

For an example of how solutions to the partitioning problem can be used to speed up computations in pipelined environments see [4].

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 number of variants of the partitioning problem are described and solved in Section 4. In Section 5 we demonstrate how the algorithm can be used to speed up sparse matrix, dense vector multiplication on a SIMD computer. 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 [10] 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:

$$f(\sigma_i, \sigma_{i+1}, ..., \sigma_j) \geq 0 \tag{1}$$

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

$$f(\sigma_{i+1}, \sigma_{i+2}, ..., \sigma_j) < f(\sigma_i, \sigma_{i+1}, ..., \sigma_j) < f(\sigma_i, \sigma_{i+1}, ..., \sigma_{j+1}) \tag{2}$$

for $0 \leq i \leq j < n-1$.
The problem is then:

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

For most problems of interest we expect the following criteria to be satisfied:

1. The function $f(\sigma_i)$ can be computed in time $O(1)$.

2. Given $f(\sigma_i, \sigma_{i+1}, ..., \sigma_j)$ we can calculate f in time $O(1)$, where either i or j has been increased or decreased by one.

A straightforward example of such a function is $f(\sigma_i, \sigma_{i+1}, ..., \sigma_j) = \Sigma_{l=i}^{j} |\sigma_l|$, given that $\sigma_l \neq 0$. To simplify the notation we write $f(i, j)$ instead of $f(\sigma_i, \sigma_{i+1}, ..., \sigma_j)$. We also denote the interval $\{\sigma_i, \sigma_{i+1}, ..., \sigma_j\}$ by $[i, j]$. The cost of a partition is the cost of the most expensive interval.

The first reference to the MinMax problem that we are aware of is by Bokhari [4] who presented an $O(n^3 p)$ algorithm using a bottleneck-path algorithm. Anily and Federgruen [2] and Hansen and Lih [9] independently presented the same dynamic programming algorithm with time complexity $O(n^2 p)$. Manne and Sørevik [10] then presented an $O(p(n-p)\log p)$ algorithm based on iteratively improving a given partition. They also described a bisection method for finding an approximate solution which runs in time $O(n \log(f(0, n-1)/\epsilon))$, where ϵ is the desired precision.

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 a 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 calculation. 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 [13] or by more problem dependent methods such as [12]. We did not find that the results presented in [13] directly applicable to the Anily and Federgruen algorithm. Thus our result is achieved by taking advantage of the specifics of 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 monotonically increasing as i decreases. As a consequence the optimal value of j in (5) also decreases monotonically as i decreases.

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 $g(i,k)$.

We start by showing 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 $1 \leq k \leq p$.*

Proof: Let $R = \{r_0, r_1, ..., r_k\}$ be a partitioning implied by $g(i,k)$ and let $s = \min\{j \mid r_j \in R \ \wedge \ r_j > i'\}$. Let further $R' = \{r'_0, r'_1, ..., r'_k\}$ define a partitioning of $[i', n-1]$ into k intervals where $r'_0 = i'$, and $r'_j = r_{s+j-1}$ for $1 \leq j \leq k-s$ and where $r'_j = n$ for $k-s < j \leq k$. Let γ be the cost of R'. Then from (2) we have $f(r'_0, r'_1) \leq f(r_{s-1}, r_s)$ and for $0 < j \leq k-s$, $f(r'_j, r'_{j+1}) = f(r_{s+j-1}, r_{s+j})$. For j such that $k-s < j < k$, we have $f(r'_j, r'_{j+1}) = 0$. This implies that $\gamma \leq g(i,k)$. Since $g(i',k) \leq \gamma$ we see that $g(i',k) \leq g(i,k)$. □

From Lemma 1 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 $g(i,k) = g(j+1,k-1)$. □

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 $g(n, k-1) = 0$. □

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 $f(i, s_{i,k}) \leq \gamma$. 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 $g(s_{i,k}, k-1) < \gamma$ 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

```
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 $f(i,j) \leq g(j+1,k-1)$                                          (8)
            $g(i,k) := g(j+1,k-1)$;
        else
            if $f(i,i) \geq g(i+1,k-1)$                                      (9)
                $g(i,k) := f(i,i)$;
                $j := i$;
            else
                while $f(i,j-1) \geq g(j,k-1)$ 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-do
end-do
```

Figure 1: The new algorithm for solving the MinMax problem

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 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 [10]. 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 actuall 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 Related problems

In [10] a number of problems related to the MinMax problem were described. In this section we show how each of these problems can be solved by slight modifications of our main algorithm. Some of these problems are solved asymptotically faster by the present algorithms for all values of p, whereas in other cases the value of p determines which algorithm is asymptotically fastest.

4.1 Bounded intervals

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 following 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 operated 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:

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

Both in practical implementations and in the theoretical discussion one should redefine infinity in the threshold operators to a fixed high value (with respect to the possible

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

    for $i := n-1$ down to $0$ do
        if $f_{p-k}(i,j) \leq g(j+1,k-1)$
            $g(i,k) := g(j+1,k-1);$
        else
            while $f_{p-k}(i,j-1) \geq g(j,k-1)$ do
                $j := j-1;$
            $g(i,k) := \min\{f_{p-k}(i,j), g(j,k-1)\};$
            if $g(i,k) = g(j,k-1)$
                $j := j-1;$
        end-else
    end-do
end-do
```

Figure 2: The algorithm for solving the generalized MinMax problem

range of $f(i,j)$ values). A $g(i,k)$ value exceeding "∞" indicates that the corresponding bounded MinMax problem is unsolvable. If this is the case then the value of $g(l,k)$ will also exceed "∞" for $0 \leq l < i$.

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.

4.2 Circular list

We now consider the case where the data is given as a circular list. In this case we need only p delimiters to partition $[0, n-1]$ into p intervals and we do not require that $r_0 = 0$. The pth interval will now be $[r_{p-1}, r_0]$. We call this problem *the circular MinMax problem.*

Suppose that there exists an optimal partition where some delimiter is placed in position l. Then if we start our ordinary algorithm with σ_l as the first element, it will produce an optimal solution. To find such an l we divide $[0, n-1]$ into p intervals such that no interval contains more than $\lceil \frac{n}{p} \rceil$ elements. Let $[i, j]$ be the most expensive interval in this partition of cost $f(i, j) = \gamma$. It follows that a partition of minimum cost must be of cost $\leq \gamma$. A partition that contains no delimiter in the interval $[i, j]$ will be of cost $> \gamma$. Thus at least one delimiter must be placed in this interval. From this we see that if we start our algorithm from each place in the interval $[i, j]$ then the solution with the minimum value is an optimal partition. Since $[i, j]$ contains at most $\lceil \frac{n}{p} \rceil$ elements and each application of the algorithm takes $O(p(n-p))$ time, we see that the total time to find an optimal solution is $O(n(n-p))$. This should be compared with the Manne and Sørevik algorithm that solves this problem in time $O(p(n-p)\log p)$.

It is possible to speed up the algorithm in the case of a circular list by applying the following observation:

Lemma 6 *Let $[i, j]$ be as above and let $[i, l]$, $l < j$ be the places from which we have started the algorithm. Let $\gamma_{i,l}$ be the cost of the best partition found so far. If $f(i-1, l+1) \geq \gamma_{i,l}$ then $\gamma_{i,l}$ is optimal.*

Proof: Let $\gamma_{\min}$ be the cost of an optimal partitioning. Assume that $f(i-1, l+1) \geq \gamma_{i,l}$ and that $\gamma_{\min} < \gamma_{i,l}$. Since $\gamma_{i,l}$ is not optimal then in an optimal partitioning there can be no delimiters in the interval $[i, l]$. Thus in an optimal partitioning some interval must contain $[i-1, l+1]$. This shows that $\gamma_{\min} \geq f(i-1, l+1) \geq \gamma_{i,l}$ which contradicts the assumption that $\gamma_{\min} < \gamma_{i,l}$. Thus we can conclude that $\gamma_{i,l}$ must be an optimal solution if $f(i-1, l+1) \geq \gamma_{i,l}$. $\square$

As noted in Section 4.1 the bounded MinMax problem with U_k independent of k can be solved by the original MinMax algorithm. Thus we can also solve this bounded MinMax problem in the circular case.

4.3 Other problems

As noted in [10] one may wish to solve the MinMax problem for each value of p. When solving the problem for p it is also possible to generate $g(0, k)$ for $1 \leq k < p$. This would increase the time complexity to $O(np)$. Thus the time complexity of solving the problem for each p, $1 \leq p \leq n$, is $O(n^2)$. This is an improvement of a factor of $\log n$ over the algorithm presented in [10]. To solve the circular problem for each value of p, $1 \leq p < n$, one can start this algorithm with $p = n$ from each place in $[0, n-1]$. This gives a time complexity of $O(n^3)$. This is also an improvement by a factor of $\log n$ over the previously best algorithm [10].

Finally, we mention a similar problem to the MinMax problem called the MaxMin problem where the object is to find a partion such that the cost of the *least* expensive interval is *maximized*. This can be solved by a method analogous to that presented in Section 3 to solve the MinMax problem.

5 Parallel sparse matrix-vector multiplication

We will in this section show how the partitioning algorithms presented in Sections 3 and 4 can be used to speed up sparse matrix-vector multiplication on a parallel SIMD-computer. In particular we have implemented a method due to Ogielski and Aiello [11] to do sparse matrix-vector multiplication, on a MasPar MP-2 computer. We start by giving a short description of the MP-2 computer followed by a description of the matrix-vector multiplication algorithm. Finally we demonstrate how the partitioning algorithm can be used to speed up the matrix-vector multiplication algorithm.

5.1 The MasPar MP-2 system

The MasPar MP-2 system is a massively parallel SIMD computer. It is an upgrade of the older MP-1 system [3], incorporating more powerful processor elements while using the same communication subsystem. The MP-2 consists of two parts: a high-performance work station, which acts as a front-end for the system, and a data parallel unit (DPU). The DPU contains between 1024 (1K) and 16384 (16K) processor elements. They are arranged in a 2-dimensional, toroidal-wrapped grid called the processor array. The DPU also contains an array control unit (ACU), which provides an interface between the front-end and the processor elements.

All the processor elements receive the same instruction from the ACU at the same time and execute it on their local data. However, individual processor elements can disable themselves based on logical expressions and they can also use indirect references when referring to local data.

The MP-2 provides two types of communication between the processor elements called *Xnet* and *Router*. Xnet communication is the faster, but more restricted procedure. It follows the grid lines of the processor array. Processor elements can send data any distance to the north, south, west, and east, as well as to the northwest, northeast, southwest, and southeast. The grid lines wrap around, so each processor element always has a neighbor in each of these eight directions.

Router communication allows each processor to send data to any other processor in the processor array. This makes it more flexible than the Xnet, but slower.

Each processor element is a 32-bit load/store arithmetic processor with 40 32-bit registers and 64Kb of RAM. There is no floating point hardware and all floating point operations are thus implemented in software. If we define the average time of a floating point operation (flop) as $\alpha = \frac{1}{2}(Mult + Add)$, the peak speed of a single processor element is 0.1412 Mflops for 64-bit arithmetic. A 16K processor machine would thus have the peak performance of 2314 Mflops.

Comparing the speed of arithmetic to communication on the MP-2, we obtain the ratio

$$\frac{Xnet[1]}{\alpha} = 0.8.$$

Thus floating point arithmetic is actually *more* expensive (by 20%) than sending the value to the nearest neighbor in the processor array.

5.2 The algorithm

Let A be $M \times N$ sparse matrix and x a vector of length N. We have implemented an algorithm for calculating the matrix-vector product $Ax = y$ on a rectangular processor array of dimension $m \times n$. We assume that $m \ll M$ and $n \ll N$. The algorithm presented here is based on an algorithm due to Ogielski and Aiello [11]. It operates in three stages: In the first stage the entries of x are spread across the processor array. In the second stage each element $a_{i,j}$ is multiplied by x_j, and in the final stage the result of the multiplications are accumulated across the processor array to form y.

First we describe how A is mapped to the processor array. Let $R = \{r_0, r_1, ..., r_m\}$

define a partitioning of the rows of A into m intervals such that $\lfloor M/m \rfloor \leq r_i - r_{i-1} \leq \lceil M/m \rceil$, $0 < i \leq m$. Similarly let $C = \{c_0, c_1, ..., c_n\}$ define a partitioning of the columns of A into n intervals such that $\lfloor N/n \rfloor \leq c_i - c_{i-1} \leq \lceil N/n \rceil$, $0 < i \leq m$. This gives rise to a block partitioning of A into mn blocks each of size at most $\lceil M/m \rceil \times \lceil N/n \rceil$. We denote the i, jth block by $A_{i,j}$. The non-zeros of $A_{i,j}$ are mapped to processor (i, j) and stored in a one dimensional array *elem*. Only the local row and column indices of each non-zero $a_{i,j}$ are stored. They are computed relative to the number of rows and columns from A that are mapped to each processor row and column.

With this mapping of A the elements $x_{c_k}, x_{c_k+1}, ..., x_{c_{k+1}-1}$ are needed on each processor in column k in order to perform the multiplications. The element x_{c_k+l}, $0 \leq l < c_{k+1} - c_k$, is stored in position $\lfloor l/m \rfloor$ of the array *x_local* on the processor $(l \bmod m, k)$. Thus in effect the elements of x which are needed on each processor column are wrapped cyclically around the processors in that column. The segment of x which is mapped to each processor column is padded with zeros so that each processor stores $\lceil N/(mn) \rceil$ elements from x. The reason for using this storage scheme for x is that it requires little space and as we shall describe, the values of x can now be spread efficiently across each processor column. The vector y is stored in a similar row-oriented way: Element y_{r_k+l}, $0 \leq l < r_{k+1} - r_k$, is stored in position $\lfloor l/n \rfloor$ of the array *y_local* on the processor $(k, l \bmod n)$. Just as it did for x each processor stores $\lceil M/(mn) \rceil$ elements from y.

Before the multiplications are performed each processor in column k receives a copy of the elements $x_{c_k}, x_{c_k+1}, ..., x_{c_{k+1}-1}$. To do this each processor starts by copying the value of *x_local*$[0]$ into a temporary variable z. Thus the processor in row l of column k will have $z = x_{c_k+l}$. The variable z is then copied into the array *temp* before it is sent to the nearest neighbor to the north. The storing and sending of z into the array *temp* is repeated $m - 1$ times until $x_{c_k}, x_{c_k+1}, ..., x_{c_k+m-1}$ has been spread across processor column k. The entire process is then repeated for each consecutive value in *x_local* (i.e. $\lceil M/m \rceil$ times) at which point each processor has a complete copy of $x_{c_k}, x_{c_k+1}, ..., x_{c_{k+1}-1}$ in *temp*. The elements from x in each processor column arrive at different times on different processors. The effect of this is that the element x_{c_k+l} will on processor (s, k) be stored in position

$$(l - s) \bmod M + \lfloor l/M \rfloor * M \tag{15}$$

of *temp*.

Each processor now forms its local contribution to y. This is done by multiplying each element of $A_{i,j}$ with the appropriate element in *temp*. The result of each multiplication is accumulated in the array *answer*.

When the multiplications are done the values in *answer* are accumulated across each processor row to form y. This is done in a similar way to spreading x. To assure that the value of y_{r_k+l} is stored on processor $(k, l \bmod n)$ the processor (k, i) accumulates its contribution to y_{r_k+l} in position

$$(l - i - 1) \bmod N + \lfloor l/N \rfloor * N \tag{16}$$

of *answer*.

In the algorithm presented here we have not included the calculation of the pointers into the arrays *x_local* and *y_local* as given by (15) and (16). The reason for this is that before the algorithm can be executed the local indices on each processor have to be computed. So instead of storing the local indices we apply (15) and (16) at once. It is also the case in many applications where matrix-vector multiplication is used, that the same structural matrix is multiplied many times. If this is so then the indices have only to be calculated before the first time.

The complete algorithm is shown in Figure 3. In the algorithm the variables iy and ix contain the row and column index of each processor. The statment **sendE**$[1].z = z$ is the use of the Xnet primitive, where **E** gives the direction of sending (in this case east).

5.3 Load balancing

The expected execution time of the program in Figure 3 is given by the following formula:

$$\gamma_1 * m * \lceil N/(mn) \rceil + \gamma_2 * \max_{i,j} |A_{ij}| + \gamma_3 * n * \lceil M/(mn) \rceil \tag{17}$$

The first term accounts for the distribution of x, the second for the multiplications and the last for setting *answer* to zero and for performing the row-sums. On the MP-2 we found that $\gamma_1 = 8$, $\gamma_2 = 54$ and $\gamma_3 = 17$ measured in micro-seconds.

If A contains blocks of non-zeros then the number of nonzeros assigned to each processor might be uneven giving a large value of $\max_{i,j} |A_{ij}|$. In order to circumvent this problem, Ogielski and Aiello [11] suggested that one initially permutes the rows and columns of A randomly. Thus in effect we compute $PAQ^TQx = z$ where P and Q are permutation matrices and $z = Py$. The values of y can easily be derived from z.

```
/* Distribute $x$ along the processor-columns */

for $i := 0$ to $\lceil N/(mn) \rceil - 1$ do
    $z := x\_local[i]$;
    temp$[i * n] := z$;
    for $j := 1$ to $m - 1$ do
        sendN[1].$z := z$;
        temp$[j + i * n] := z$;
    end-do
end-do

/* Set answer to zero */

for $i := 0$ to $\lceil M/(mn) \rceil * m - 1$ do
    answer$[i] := 0.0$;

/* Perform the multiplications */

for $i := 0$ to $|A_{iy,ix}| - 1$ do
    $a :=$ row$[i]$;
    $b :=$ column$[i]$;
    answer$[a] :=$ answer$[a]$ + temp$[b]$ * elem$[i]$;
end-do

/* Sum $y$ along the processor-rows */

for $i := 0$ to $\lceil M/(mn) \rceil - 1$ do
    $z :=$ answer$[i * m]$;
    for $j := 1$ to $n - 1$ do
        sendE[1].$z := z$;
        $z := z +$ answer$[j+i * m]$;
    end-do
    $y\_local[i] := z$;
end-do
```

Figure 3: Parallel matrix-vector multiplication

To avoid confusion we assume that $P = I$ and $Q = I$.

The quotient $\phi = \gamma_2/(\gamma_1 + \gamma_3)$ gives an indication of the relative importance of reducing $\max |A_{ij}|$ in terms of the total execution time of the algorithm. We note that in our implementation $\phi = 2.16$ where as Ogielski and Aiello [11] on an MP-1 have $\phi = 2.87$. How well a reduction of $\max |A_{ij}|$ translates into total speedup of the algorithm, is thus dependent on the speed of arithmetic compared to communication.

We now discuss how the partitioning algorithm described in Sections 3 and 4 can be used to speed up this code. Consider a partitioning of the columns of A such that no processor column gets more than l columns from A. Then if $l \leq m\lceil N/(mn)\rceil$ the time to spread x across the processor columns will not increase. To see what this means consider the case where $N = 50,000$, $m = 128$ and $n = 128$. Then with the Ogielski and Aiello partitioning scheme each processor column will get $\lfloor N/n \rfloor = 390$ or $\lceil N/n \rceil = 391$ columns from A and $\lceil N/(mn)\rceil = 4$ iterations are needed to spread x across the processor columns. However, as long no processor column gets assigned more than $m\lceil N/(mn)\rceil = 512$ columns we still only need 4 iterations to spread x. Similarly any partitioning of the rows of A where no interval is larger than $n\lceil M/(mn)\rceil$ will spend the same amount of time on summing the values of y along the processor rows.

Thus if mn does not divide M and N there is a certain degree of freedom as to how A is partitioned into blocks without increasing the time spent on distributing x and summing y. We take advantage of this fact to try to reduce $\max_{i,j} |A_{ij}|$, i.e. the middle term of (17).

We form a vector containing the number of elements in each column. Since a random permutation has been performed of the rows of A we can expect the non-zeros in each column of A to be evenly distributed. Thus by partitioning the vector into n intervals such that the sum of the elements of the most expensive interval is low we can expect $\max_{i,j} |A_{ij}|$ to be reduced. We partition the vector with the constraint that no interval contains more than $m\lceil N/(mn)\rceil$ elements. This is the bounded MinMax problem with U_k independent of k. Thus we can use the algorithm in Figure 1 with f modified according to (13). We perform a similar partition of the rows of A with the restriction that no interval contains more than $n\lceil M/(mn)\rceil$ elements. We will demonstrate the efficiency of this method on a number of test problems. Note also that since the processors on the MP-2 are toroidally wrapped, it is possible to perform a circular partitioning of the rows and columns of A.

Most matrices from finite element and finite difference applications tend to be very sparse with only a small constant number of nonzeros in each column. For very sparse problems there is not much to gain from trying different partitionings since the time

Matrix	Dim	Density	LB	Rand	RMF	BA	BMF
16-3D	4096	3.5%	36	100	135.03	59	182.89
21-3D	9261	2.3%	118	273	215.40	176	302.66
bcsstk31	35588	0.5%	369	683	259.85	450	355.03
bcsstk33	8738	3.5%	163	337	249.63	219	354.79
33 no-fill	8738	0.4%	19	37	113.89	37	113.89

Table 1: Results for the 16K processor MasPar MP-2

will be dominated by spreading x along the processor columns and summing y along the processor rows. Also if each row and column is sparse the random permutation of A tend to give a good load balance.

To demonstrate the efficiency of the partitioning method we use symmetric matrices on which symbolic permutation has been performed. This increases the density of the matrices. We note that these matrices might not be typical of the kind of matrices that one would want to perform matrix-vector multiplication with. Still as we show, they indicate that the method is potentially helpful whenever the matrix is not very sparse and contains dense rows and columns.

To generate test matrices we have used three-dimensional $k \times k \times k$ twenty-seven point grids on which we have performed symbolic factorization. They were preordered by Sparspak's automatic nested dissection [7]. We also include results on matrices from the Harwell-Boeing test collection [5]. These have been preordered by the minimum degree algorithms from Sparspak.

The results are shown in Table 1. The matrix "33 no-fill" contain the lower triangular structure of "bcsstk33" before adding the fill elements. The reported results are the median improvement given by the partitioning algorithm on 11 initial random permutations of each matrix. "Dim" gives the dimension of each matrix and "Density" gives the percentage of the elements that are nonzero. "LB" gives $\lceil |A|/(nm) \rceil$ which is a theoretical lower bound for $\max_{i,j} |A_{i,j}|$. "Rand" gives $\max_{i,j} |A_{i,j}|$ after A has been randomly permuted and partitioned into even size blocks. "RMF" is the number of MFlops achieved on the MP-2 with this layout of A. "BA" gives $\max_{i,j} |A_{i,j}|$ after the partitioning algorithm has been used and "BMF" gives the MFlop rate with this layout.

We note that on the filled matrices the MFlop rate increases between 35% and 43% by performing a balanced partitioning of A. On the matrix without fill no increase occurred. As can be seen by comparing the matrices bcsstk31 and bcsstk33 without fill it is not only the density of the matrices that determines if the balanced partition gives a positive effect. Other factors that might influence this is the dimension of the matrix compared to the size of the processor array and the presence of rows and columns that are relatively more dense than the rest of the matrix.

We have also tried a cyclical partitioning of the rows and columns of A. However, this did not result in any significant speed-up compared to the ordinary partitioning algorithms.

The improvement that can be achieved with this method must be compared to the time taken by the partitioning algorithm. For iterative methods such as the conjugate gradient method [8] one has to multiply the same matrix with a dense vector many times. Thus the time to perform the partitioning can be distributed over the number of matrix-vector operations that need to be performed.

Another application where sparse matrix-vector multiplication is needed, is when solving sparse triangular systems by the means of *partitioned inverses* [1]. This method involves repeated multiplications of a series of lower triangular sparse matrices and a dense vector.

6 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 properties of the cost functions. Where applicable, this technique seems to be a useful way of reducing the complexity of dynamic programming algorithms. We have shown how our algorithm can be modified to solve a number of variants of the MinMax problem. Based on these algorithms we demonstrated how sparse matrix-vector multiplication on a SIMD computer can be made more efficient.

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).

In the matrix-vector code presented in Section 5 we performed a random permutation of A prior to the partitioning. This in itself gives relatively good load balance. For other problems where it is not possible to perform such a permutation we believe that one can expect greater speed-up from the partitioning algorithm.

We note that other methods than the one presented in this paper could have been used to speed up the sparse matrix-vector multiplication algorithm. Instead of using a randomized permutation of the rows and columns of A, one could use some more sophisticated way to determine permutations to even out the number of nonzeros allocated to each processor. Also, our algorithm only gives an approximation to the 2 dimensional partitioning problem. It would be of interest to know if it was possible to find an optimal solution to this problem. The partitioning problem can also be generalized to higher dimensions than 2.

7 Acknowledgment

The authors thank Andrew Ogielski for help with implementing the matrix-vector multiplication algorithm in Section 5. They also thank Bengt Aspvall and Tor Sørevik for constructive comments.

References

[1] F. L. Alvarado and R. Schreiber, *Optimal parallel solution of sparse triangular systems*, SIAM J. Sci. Statist. Comput., 14 (1993).

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

[3] T. Blank, *The MasPar MP-1 architecture*, in Proceedings of IEEE Compcon Spring 1990, IEEE, February 1990.

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

[5] I. Duff, R. Grimes, and J. Lewis, *Sparse matrix test problems*, ACM Trans. Math. Software, 15 (1989), pp. 1–14.

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

[7] A. George and J. W. H. Liu, *Computer Solutions of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[8] G. H. Golub and C. F. V. Loan, *Matrix Computations*, North Oxford Academic, 2 ed., 1989.

[9] 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.

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

[11] A. T. Ogielski and W. Aiello, *Sparse matrix computations on parallel processor arrays*, SIAM J. Sci. Comput., 14 (1993), pp. 519–530.

[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.