algorithms

MDPI

Article

Efficient Algorithms for the Maximum Sum Problems

Sung Eun Bae [1], Tong-Wook Shinn [1,*] and Tadao Takaoka [1,2]

[1] Algorithm Research Institute, Christchurch 8053, New Zealand; sung.bae@ari.org.nz (S.E.B.); tadao.takaoka@ari.org.nz (T.T.)
[2] Computer Science and Software Engineering, University of Canterbury, Christchurch 8140, New Zealand
* Correspondence: tongwook.shinn@ari.org.nz

Academic Editors: Bruno Carpentieri and Spyros Kontogiannis
Received: 9 August 2016; Accepted: 26 December 2016; Published: 4 January 2017

Abstract: We present efficient sequential and parallel algorithms for the maximum sum (MS) problem, which is to maximize the sum of some shape in the data array. We deal with two MS problems; the maximum subarray (MSA) problem and the maximum convex sum (MCS) problem. In the MSA problem, we find a rectangular part within the given data array that maximizes the sum in it. The MCS problem is to find a convex shape rather than a rectangular shape that maximizes the sum. Thus, MCS is a generalization of MSA. For the MSA problem, $O(n)$ time parallel algorithms are already known on an (n, n) 2D array of processors. We improve the communication steps from $2n - 1$ to n, which is optimal. For the MCS problem, we achieve the asymptotic time bound of $O(n)$ on an (n, n) 2D array of processors. We provide rigorous proofs for the correctness of our parallel algorithm based on Hoare logic and also provide some experimental results of our algorithm that are gathered from the Blue Gene/P super computer. Furthermore, we briefly describe how to compute the actual shape of the maximum convex sum.

Keywords: maximum sub-array; maximum convex sum; parallel algorithm

1. Introduction

We face a challenge to process a big amount of data in the age of information explosion and big data [1]. As the end of Moore's law comes in sight, however, the extra computing power is unlikely obtained from a single processor computer [2]. Parallel computing is obviously a direction to faster computation. However, efficient utilization of the parallel architecture is not an effortless task. New algorithms often need to be designed for specific problems on specific hardware.

In this paper, we investigate the maximum subarray (MSA) problem and the maximum convex sum (MCS) problem, the latter being a generalization of the former. We design efficient parallel algorithms for both problems and an improved sequential algorithm for the MCS problem.

The MSA problem is to find a rectangular subarray in the given two-dimensional (2D) data array that maximizes the sum in it. This problem has wide applications from image processing to data mining. An example application from image processing is to find the most distinct spot, such as brightest or darkest, in the given image. If all pixel values are non-negative, solving the MSA problem will return the trivial solution of the whole array. Thus, we normalize the input image (e.g., by subtracting the mean value), such that the relatively brighter spots will have positive sums, while the relatively darker spots will have negative sums. Then, solving the MSA problem on the normalized image can give us the brightest spot in the image. In data mining, suppose we spread the sales amounts of some product on a two-dimensional array classified by customer ages and annual income. Then, after normalizing the data, the maximum subarray corresponds to the most promising customer range.

We now consider the MCS problem that maximizes the sum in a convex shape. The definition of the word "convex" is not exactly the same as that in geometry. We define the convex shape as the

joining of a W-shape and an N-shape, where W stands for widening and N for narrowing. We call this shape the WN-convex shape or WN-shape for short. In W, the top boundary goes up or stays horizontal when scanned from left to right, and the bottom boundary goes down or stays horizontal. Fukuda et al. defines a more general rectilinear shape [3], but for simplicity, we only consider the WN-shape. The paper is mainly devoted to computing the maximum sum, and one section is devoted to the computation of the explicit convex shape that provides the sum.

We give an example to illustrate how solving the MCS problem can provide a much more accurate data analysis compared to solving the MSA problem. We compared the size of the hole in the ozone layer over Antarctica between 2006 and 2011 by solving both the MSA problem and the MCS problem on the normalized input images (Figure 1a,d, respectively) of the Antarctic region. Figure 1b,e is from solving the MSA problem on the input images, while Figure 1c,f is from solving the MCS problem on the same set of input images. Solving the MCS problem clearly provides a much more accurate representation of the ozone hole. The numeric value returned by the MCS is also 22%~24% greater than that by the MSA on both occasions. Intuitively similar gains from the WN-shape compared to the rectangular shape can be foreseen for other types of data analysis.

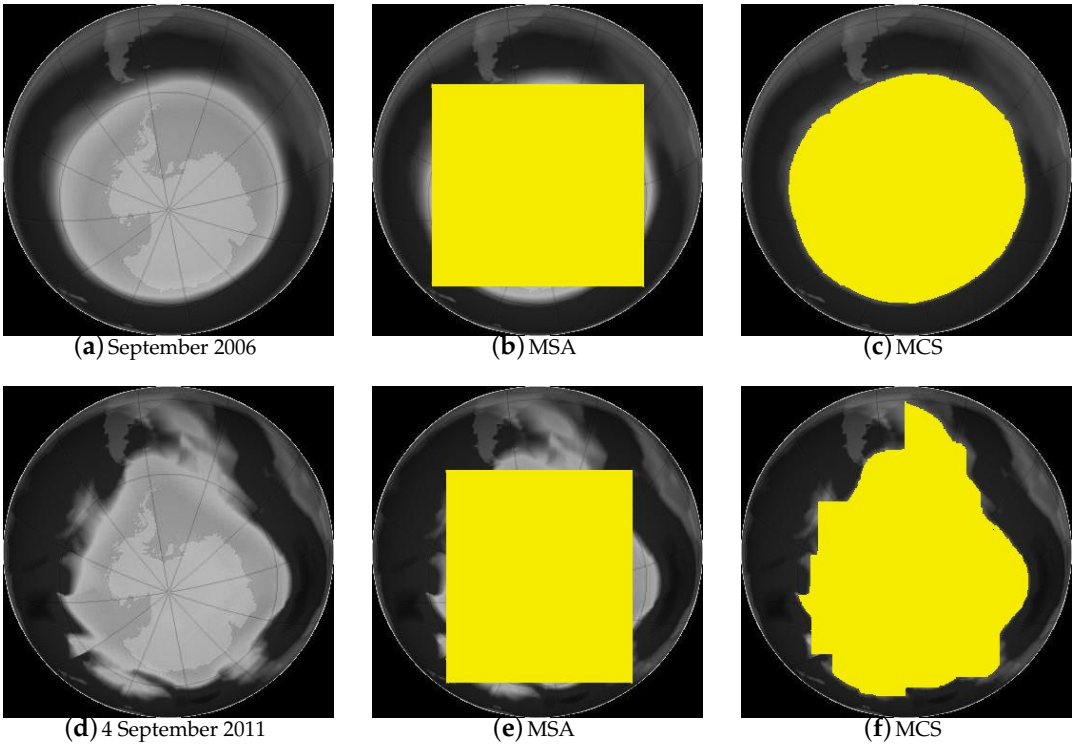

Figure 1. Ozone hole identified by the maximum subarray (MSA) and maximum convex sum (MCS) algorithms (Courtesy of NASA).

In this paper, we assume that the input data array is an (n, n) square two-dimensional (2D) array. It is a straightforward exercise to extend the contents of this paper to (n, m) rectangular 2D input data arrays.

The typical algorithm for the MSA problem by Bentley [4] takes $O(n^3)$ time on a sequential computer. This has been improved to a slightly sub-cubic time bound by Tamaki and Tokuyama [5] and also Takaoka [6]. For the MCS problem, an algorithm with $O(n^3)$ time is given by Fukuda et al. [3], and an algorithm with a sub-cubic time bound is not known.

Takaoka discussed a parallel implementation to solve the MSA problem on a PRAM [6]. Bae and Takaoka implemented a range of parallel algorithms for the MSA problem on an (n, n) 2D mesh array architecture based on the row-wise or column-wise prefix sum [7–9]. A parallel algorithm for the MSA

problem was also implemented on the BSP/CGMarchitecture, which has more local memory and communication capabilities with remote processors [10].

In this paper, we implement algorithms for the MSA and MCS problems based on the column-wise prefix sum on the 2D mesh architecture, as shown by Figure 2. This architecture is also known as a systolic array, where each processing unit has a constant number of registers and is permitted to only communicate with directly-connected neighbours. This seemingly inflexible architecture is well suited to be implemented on ASICs or FPGAs.

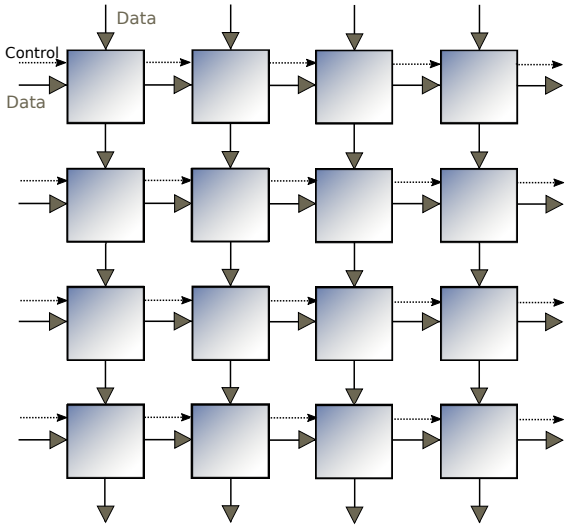

Figure 2. Two-dimensional architecture.

Our most efficient parallel algorithms complete the computation in $n-1$ communication steps for the MSA problem and $4n$ communication steps for the MCS problem, respectively. In each step, all cells execute a constant number of statements in parallel. Thus, our algorithms are cost optimal with respect to the cubic time sequential algorithms [3,4].

We give a formal proof for the parallel algorithm for computing the maximum W (a part of computing the MCS problem). The proof is based on the space-time invariants defined on the architecture.

In a 2D array architecture, some processors to the right are sitting idle in the early stage of computation waiting for inputs to arrive. This is because the data only flows from left to right and from up to down (Figure 2). If appropriate, we attempt to maximize the throughput by extending data flows to operate in four directions. This technique reduces the number of communication steps by a constant factor.

Algorithms are given by pseudocode.

2. Parallel Algorithms for the MSA Problem

In this section, we improve parallel algorithms for the MSA problem on a mesh array and achieve the optimal n communication steps. Furthermore, we show how the programming techniques in this section lead to efficient parallel algorithms for the MCS problem in the later sections.

2.1. Sequential Algorithm

The computation in Algorithm 1 [8] proceeds with the strip of the array from position k to position i. See Figure 3. The variable $column[i][j]$ is the sum of array elements in the j-th column from position k to position i in array a. The variable $sum[i][j]$, called a prefix-sum, is the sum of the strip from Position 1 to position j. Within this strip, variable j sweeps to compute $column[i][j]$ by adding $a[i][j]$ to $column[i-1][j]$. Then, the prefix sum of this strip from Position 1 to position j is computed

by adding $column[i][j]$ to $sum[i][j-1]$. The variable $min\_sum[i][j]$ is the minimum prefix sum of this strip from Position 1 to position j. If the current sum is smaller than $min\_sum[i][j]$, $min\_sum[i][j]$ is replaced by it. $sol[i][j]$ is the maximum sum in this strip so far found from Position 1 to position j. It is computed by taking the maximum of $sol[i][j-1]$ and $sum[i][j] - min\_sum[i][j]$, expressed by max_sum in the figure. After the computation for this strip is over, the global solution, S, is updated by $sol[i][n]$. This computation is done for all possible i and k, taking $O(n^3)$ time.

Algorithm 1 MSA: sequential.

```
1:  for 1 ≤ i ≤ n do
2:      min_sum[i][0] ← ∞; sum[i][0] ← 0; sol[i][0] ← −∞;
3:  end for
4:  S ← −∞;
5:  for k ← 1 to n do
6:      for j ← 1 to n do
7:          column[k − 1][j] ← 0;
8:      end for
9:      for i ← k to n do
10:         for j ← 1 to n do
11:             column[i][j] ← column[i − 1][j] + a[i][j];
12:             sum[i][j] ← sum[i][j − 1] + column[i][j];
13:             min_sum[i][j] ← MIN{sum[i][j], min_sum[i][j − 1] };
14:             max_sum[i][j] ← sum[i][j] − min_sum[i][j];
15:             sol[i][j] ← MAX{max_sum[i][j], sol[i][j − 1]};
16:         end for
17:         if sol[i][n] > S then
18:             S ← sol[i][n];
19:         end if
20:     end for
21: end for
```

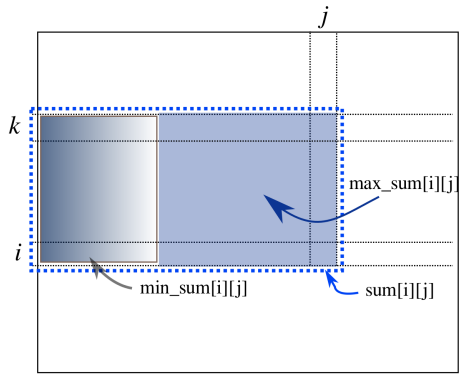

Figure 3. Strip-based sequential computation.

2.2. Parallel Algorithm 1

Algorithm 2 is a parallel adaptation of Algorithm 1. The following program is executed by a processing unit at the (i, j) grid point, which we refer to as $cell(i, j)$. Each $cell(i, j)$ is aware of its position (i, j). Data flow is from left to right and from top to bottom. The control signals are fired at the left border and propagate right. When the signal arrives at $cell(i, j)$, it accumulates the column sum "$column$", the sum "sum" and updates the minimum prefix sum "min", etc. Figure 4 illustrates the information available to $cell(i, j)$ at step k. At step $2n - 1$, $cell(n, n)$ will have computed the maximum sum. We assume that all corresponding instructions in all cells are executed at the

same time, that is they are synchronized. We will later make some comments on asynchronous computation.

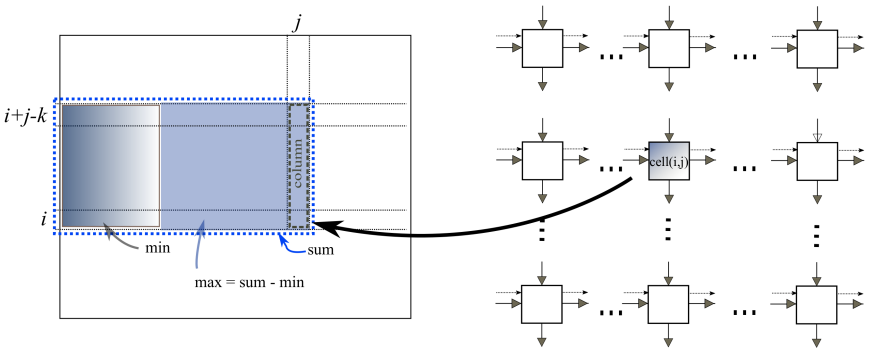

Figure 4. Illustration for Algorithm 2: The coverage of $cell(i, j)$ at step k.

Algorithm 2 MSA Parallel 1.

Initialization
1: **for all** $0 \leq i, j \leq n$ in parallel **do**
2: $column[i][j] \leftarrow 0$; $min[i][j] \leftarrow \infty$;
3: $control[i][[j] \leftarrow 0$; $sum[i][j] \leftarrow 0$;
4: **end for**
5: **for** $0 \leq i \leq n$ in parallel **do**
6: $control[i][0] \leftarrow 1$; $sol[i][0] \leftarrow -\infty$;
7: **end for**

Main
8: **for** $k \leftarrow 1$ to $2n-1$ **do**
9: **for all** $1 \leq i, j \leq n$ in parallel **do**
10: **if** $control[i][j-1] = 1$ **then**
11: $column[i][j] \leftarrow column[i-1][j] + a[i][j]$;
12: $sum[i][j] \leftarrow sum[i][j-1] + column[i][j]$;
13: $min[i][j] \leftarrow \mathbb{MIN}\{min[i][j-1], sum[i][j]\}$;
14: $max[i][j] \leftarrow sum[i][j] - min[i][j]$;
15: $sol[i][j] \leftarrow \mathbb{MAX}\{sol[i-1][j], sol[i][j-1], sol[i][j], max[i][j]\}$;
16: $control[i][j] \leftarrow 1$;
17: **end if**
18: **end for**
19: **end for**

2.3. Parallel Algorithm 2

This algorithm (Algorithm 3) does communication bi-directionally in a horizontal way. For simplicity, we assume n is even. The (n, n) mesh is divided into two halves, left and right. The left half operates in the same way as Algorithm 2. The right half operates in a mirror image, that is control signals go from right to left initiated at the right border. All other data also flow from right to left. At the centre, that is at $(i, n/2)$, $cell(i, n/2)$ performs "$centre[i] \leftarrow max[i][n/2] + max[i][n/2+1]$", which adds the two values that are the sums of strip regions in the left and right whose heights are equal and, thus, can be added to form a possible solution crossing over the centre. At the end of the k-th iteration, all properties in Algorithm 2 hold on the left half, and the properties in the mirror image hold on the right half. In addition, we have that $centre[i]$ is the value of the maximum subarray that lies above or that is touching the i-th row and crosses over the centre line.

Algorithm 3 MSA Parallel 2.

Initialization
1: **for all** $0 \leq i, j \leq n$ in parallel **do**
2: $column[i][j] \leftarrow 0; min[i][j] \leftarrow \infty;$
3: $control[i][[j] \leftarrow 0; sum[i][j] \leftarrow 0;$
4: **end for**
5: **for** $0 \leq i \leq n$ in parallel **do**
6: $control[i][0] \leftarrow 1; control[i][n+1] \leftarrow 1;$
7: **end for**
Main
8: **for** $k \leftarrow 1$ to $(3/2)n - 1$ **do**
9: **for all** $1 \leq i, j \leq n$ in parallel **do**
10: **if** $1 \leq j \leq n/2$ **then** ▷ left half
11: **if** $control[i][j-1] = 1$ **then**
12: $column[i][j] \leftarrow column[i-1][j] + a[i][j];$
13: $sum[i][j] \leftarrow sum[i][j-1] + column[i][j];$
14: $min[i][j] \leftarrow \mathbb{MIN}\{min[i][j-1], sum[i][j]\};$
15: $max[i][j] \leftarrow sum[i][j] - min[i][j];$
16: $sol[i][j] \leftarrow \mathbb{MAX}\{sol[i-1][j], sol[i][j-1], sol[i][j], max[i][j]\};$
17: $control[i][j] \leftarrow 1;$
18: **end if**
19: **end if**
20: **if** $n/2 + 1 \leq j \leq n$ **then** ▷ right half
21: **if** $control[i][j+1] = 1$ **then**
22: $column[i][j] \leftarrow column[i-1][j] + a[i][j];$
23: $sum[i][j] \leftarrow sum[i][j+1] + column[i][j];$
24: $min[i][j] \leftarrow \mathbb{MIN}\{min[i][j+1], sum[i][j]\};$
25: $max[i][j] \leftarrow sum[i][j] - min[i][j];$
26: $sol[i][j] \leftarrow \mathbb{MAX}\{sol[i-1][j], sol[i][j+1], sol[i][j], max[i][j]\};$
27: $control[i][j] \leftarrow 1;$
28: **end if**
29: **end if**
30: **if** $j = n/2$ **then** ▷ $cell(i, n/2)$ processes $centre[i]$
31: $centre[i] \leftarrow max[i][n/2] + max[i][n/2+1];$
32: **if** $centre[i] < centre[i-1]$ **then**
33: $centre[i] \leftarrow centre[i-1];$
34: **end if**
35: **end if**
36: **end for**
37: **end for**
Finalization step
38: Let $cell(n, n/2)$ do $solution \leftarrow \mathbb{MAX}\{sol[n][n/2], sol[n][n/2+1], centre[n]\};$

The strip $cell(i, j)$ processes is $a[i + j - k, ..., i][1, ..., j]$ in the left half, and that in the right half is $a[i + n - j + 1 - k, ..., i][j, ..., n]$. Thus, the cells $cell(i, n/2)$ and $cell(i, n/2 + 1)$ processes the strips of the same height in the left half and the right half. Communication steps are measured by the distance from $cell(1, 1)$ to $cell(n, n/2)$ or, equivalently, from $cell(1, n)$ to $cell(n, n/2 + 1)$, which is $(3/2)n - 1$. By adding the finalization step, we have $(3/2)n$ for the total communication steps.

2.4. Parallel Algorithm 3

In Algorithm 4, data flow in four directions. The array is divided into two halves; left and right, as in the previous section. Column sums c and prefix sums s accumulate downwards as before, whereas column sums d and prefix sums t accumulate upwards. See Figure 5.

Algorithm 4 MSA Parallel 3: initialization.

1: **for all** $0 \leq i, j \leq n+1$ in parallel **do**
2: $c[i][j] \leftarrow 0; d[i][j] \leftarrow 0;$
3: $min[i][j] \leftarrow \infty; control[i][[j] \leftarrow 0;$
4: $s[i][j] \leftarrow 0; t[i][j] \leftarrow 0;$
5: **end for**
6: **for** $0 \leq i \leq n+1$ in parallel **do**
7: $control[i][0] \leftarrow 1; control[i][n+1] \leftarrow 1;$
8: **end for**
9: **for** $k \leftarrow 1$ to $n-2$ **do**
10: **for all** $1 \leq i, j \leq n$ in parallel **do**
11: **if** $1 \leq j \leq n/2$ **then**
12: **if** $control[i][j-1] = 1$ **then**
13: $c[i][j] \leftarrow c[i-1][j] + a[i][j];$
14: $s[i][j] \leftarrow s[i][j-1] + c[i][j];$
15: $d[i][j] \leftarrow d[i+1][j] + a[i][j];$
16: $t[i][j] \leftarrow t[i][j-1] + d[i][j];$
17: $u[i][j] \leftarrow s[i][j] + t[i+1][j];$
18: $min[i][j] \leftarrow \mathbb{MIN}\{min[i][j-1], u[i][j]\};$
19: $max[i][j] \leftarrow u[i][j] - min[i][j];$
20: $sol[i][j] \leftarrow \mathbb{MAX}\{sol[i-1][j], sol[i+1][j], sol[i][j-1], sol[i][j]\};$
21: $sol[i][j] \leftarrow \mathbb{MAX}\{sol[i][j], max[i][j]\};$
22: $control[i][j] \leftarrow 1;$
23: **end if**
24: **end if**
25: **if** $n/2+1 \leq j \leq n$ **then**
26: **if** $control[i][j+1] = 1$ **then**
27: $c[i][j] \leftarrow c[i-1][j] + a[i][j];$
28: $s[i][j] \leftarrow s[i][j+1] + c[i][j];$
29: $d[i][j] \leftarrow d[i+1][j] + a[i][j];$
30: $t[i][j] \leftarrow t[i][j+1] + d[i][j];$
31: $u[i][j] \leftarrow s[i][j] + t[i+1][j];$
32: $min[i][j] \leftarrow \mathbb{MIN}\{min[i][j+1], u[i][j]\};$
33: $max[i][j] \leftarrow u[i][j] - min[i][j];$
34: $sol[i][j] \leftarrow \mathbb{MAX}\{sol[i-1][j], sol[i+1][j], sol[i][j+1], sol[i][j]\};$
35: $sol[i][j] \leftarrow \mathbb{MAX}\{sol[i][j], max[i][j]\};$
36: $control[i][j] \leftarrow 1;$
37: **end if**
38: **end if**
39: **if** $j = n/2$ **then** ▷ $cell(i, n/2)$ performs the following
40: $centre[i] \leftarrow max[i][n/2] + max[i][n/2+1];$
41: **if** $centre[i] < centre[i-1]$ **then**
42: $centre[i] \leftarrow centre[i-1];$
43: **end if**
44: **if** $centre[i] < centre[i+1]$ **then**
45: $centre[i] \leftarrow centre[i+1];$
46: **end if**
47: **end if**
48: **end for**
49: **end for**
50: **if** $i = n/2$ and $j = n/2$ **then** ▷ $cell(n/2, n/2)$ processes *solution*
51: $solution \leftarrow \mathbb{MAX}\{sol[n/2][n/2], sol[n/2][n/2+1], centre[n/2]\};$
52: **end if**

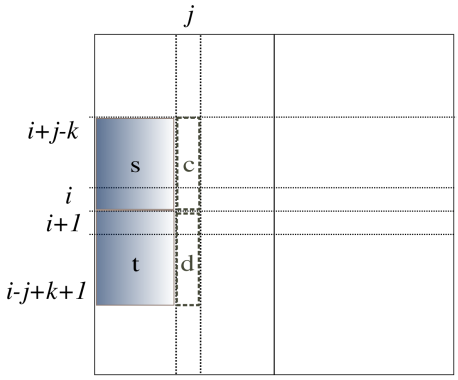

Figure 5. Illustration for Algorithm 4.

The structure of Algorithm 4 reveals that at the end of the k-th iteration, $s[i][j]$ is the sum of $a[i+j-k,...,i][1,...,j]$ and $t[i+1][j]$ is the sum of $a[i+1,...,i-j+k+1][1,...,j]$. The height of each subarray is $k-j+1$. Since the widths of those two areas are the same, we can have the prefix sum $u[i][j] = s[i][j] + t[i+1][j]$ that covers $a[i+j-k,...,i-j+k+1][1,...,j]$, the height of which is $2(k-j+1)$. That is, spending k steps, we can achieve twice as much height as that in Algorithm 3.

The solution array *sol* is calculated as before, but the result is sent into three directions; up, down and right in the left half and up, down and left in the right half. We have the property that $sol[i][j]$ is the maximum sum in subarray $a[i+j-k,...,i-j+k+1][1,...,j]$ in the left half. Substituting $i = n/2, j = n/2$, and $k = n-1$ yields the subarray $a[1,...,n][1,...,n/2]$. Similarly, $sol[n/2][n/2+1]$ is the maximum sum in the subarray $a[1,...,n][n/2+1,...,n]$. For simplicity, we deal with the maximum subarray whose height is an even number. For a general case, see the note at the end of this section.

The computation proceeds with $n-2$ steps by k and the last step of comparing the results from $cell(n/2, n/2)$ and $cell(n/2+1, n/2)$, resulting in $n-1$ steps in total.

Note that we described the algorithm for the solution whose height is an even number. This fact comes from the assignment statement "$u[i][j] \leftarrow s[i][j] + t[i+1][j]$" where the heights of subarrays whose sums are s and t are equal. To accommodate a height of an odd number, we can use the value of t one step before, whose height is one shorter. To accommodate such odd heights, we need to almost double the size of the program by increasing the number of variables.

3. Review of Sequential Algorithm for the MCS Problem

We start from describing a sequential algorithm for W based on column sums, given by Fukuda et al. [3]. We call the rightmost column of a W-shape the anchor column of W. The array portion of general array b, $b[i,...,k][j,...,l]$, is the rectangular portion whose top left corner is (i,j), and the bottom right corner is (k,l). The column $b[i,...,k][j,...,j]$ is abbreviated as $b[i,...,k][j]$. In Algorithm 5, $W[i][k][j]$ is a W-shape based on the anchor column of $a[k,...,i][j]$. The sum of this anchor column is given by $column[i][k][j]$.

The computation proceeds with the strip of the array from row k to row i (Figure 6) based on dynamic programming. Within this strip from row k to row i, variable j sweeps to compute $W[i][k][j]$ by adding $column[i][k][j]$ to $W[i][k][j-1]$ (Case 1 of Figure 7) or extending a W-shape downward or upward by one (Cases 2 and 3 of Figure 7, respectively). Note that the width of the strip is given by $t+1$. That is, we go through from thinner to thicker strips, so that the results for thinner ones are available when needed for Cases 2 and 3. Obviously, Algorithm 5 takes $O(n^3)$ time.

Algorithm 5 Sequential algorithm for W.

Initialization
1: **for all** $0 \leq i, j, k \leq n$ **do**
2: $W[i][k][j] \leftarrow 0$ ▷ $W[i][k][j]$ stores maximum W for anchor column $column[k, \ldots, i][j]$
3: **end for**

Main
4: **for** $j \leftarrow 1$ to n **do**
5: $sum[0][j] \leftarrow 0$
6: **for** $i \leftarrow 1$ to n **do**
7: $sum[i][j] \leftarrow sum[i-1][j] + a[i][j]$
8: **end for**
9: **end for**
10: **for** $t \leftarrow 1$ to $n-1$ **do**
11: **for** $k \leftarrow 1$ to $n-t$ **do**
12: $i \leftarrow k+t$
13: **for** $j \leftarrow 1$ to n **do**
14: $column[i][k][j] \leftarrow sum[i][j] - sum[k-1][j]$ ▷ sum of $a[k, \ldots, i][j]$
15: $cand_1 \leftarrow W[i][k][j-1] + column[i][k][j]$
16: $cand_2 \leftarrow W[i-1][k][j] + a[i][j]$
17: $cand_3 \leftarrow W[i][k+1][j] + a[k][j]$
18: $W[i][k][j] \leftarrow \mathbb{MAX}\{cand_1, cand_2, cand_3\}$
19: **end for**
20: **end for**
21: **end for**

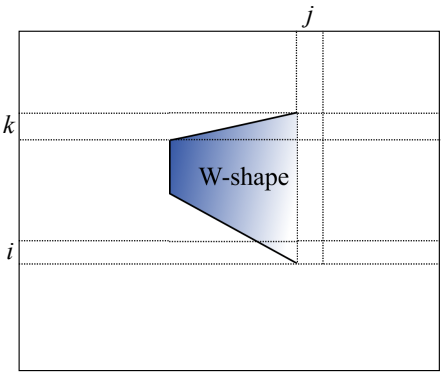

Figure 6. Sequential computation with position indices i, j, k.

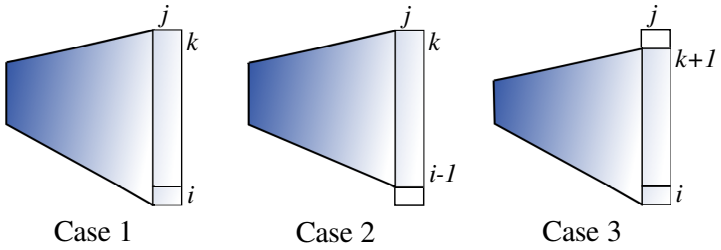

Figure 7. Three cases for candidates.

After all W are computed, we compute the N-shape in array N for all anchor columns by a mirror image of the algorithm for W. Then, the finalization of the WN-shape is given by Algorithm 6.

Algorithm 6 Combining W and N.

```
1: S ← −∞
2: for j ← 1 to n do
3:     for k ← 1 to n do
4:         for i ← k to n do
5:             S ← 𝕄𝔸𝕏{S, W[i][k][j] + N[i][k][j] − column[i][k][j]}
6:         end for
7:     end for
8: end for
```

Note that the W-shape and the N-shape share the same anchor column, meaning we need to subtract one $column[i][k][j]$. This computation is done for all possible i, j and k, taking $O(n^3)$ time, resulting in $O(n^3)$ time for the maximum convex sum.

4. Improved Sequential Algorithm

We can observe that Algorithm 5 not only takes $O(n^3)$ time, but also requires $O(n^3)$ memory. The reason for this memory requirement is that the algorithm stores maximum W and maximum N for all possible anchor columns.

We can improve the memory requirement of Algorithm 5 significantly to $O(n^2)$ with a simple modification. Firstly, we observe that there is no reason to keep all maximum W and maximum N for all $O(n^3)$ possible anchor columns. Instead, we can iterate over the possible anchor column sizes and compute the $O(n^2)$ maximum W and N for the given anchor column size in each iteration, thereby computing the maximum WN for the given anchor column size in each iteration. Thus, in each iteration, we only need $O(n^2)$ memory, and there is no need to store maximum W and N values from previous iterations. Note that W and N on shorter columns are available for the t-th iteration.

Algorithm 7 is the pseudocode for achieving the $O(n^2)$ memory bound. The pseudocode has been simplified, since much of the details have already been provided in Section 3.

Algorithm 7 Sequential algorithm for *MCS*.

```
1: for t ← 1 to n do
2:     Compute maximum W for all anchor columns of size t
3:     Compute maximum N for all anchor columns of size t
4:     Combine W and N for all anchor columns of size t
5:     Store the current maximum WN
6: end for
```

We note that the reduction in the memory bound is very significant in practical terms. The difference between $O(n^2)$ memory and $O(n^3)$ memory, if we take image processing as an example, is the difference between being able to process a mega-pixel image entirely in memory and having to resort to paging the results in an incomparably slow execution time.

5. Parallel Algorithm for MCS

We now give a parallel algorithm that corresponds to the sequential algorithm described in Section 3. Algorithm 8 is executed by the cell at the (i, j) grid point. Each $cell(i, j)$ is aware of its position (i, j). Data flow is from left to right and from top to down. The control signals are fired at the left border and propagate right. When the signal arrives at $cell(i, j)$, it starts to accumulate the column sum $column$ and update top, W and sol. The value of $W[i][j]$ at time k is to hold the maximum W-value based on the anchor column in the j-th column from position $i + j - k$ to position i. This is illustrated in Figure 8. The value of $sol[i][j]$ is to hold the best W-value obtained at $cell(i, j)$ so far. The role of top

is to bring down the top value of anchor column to $cell(i,j)$ in time. The role of "old_W" is to provide the value of W one step before, with $old\_W[i][0]$ being undefined.

Algorithm 8 Parallel algorithm for W.

Initialization

for all $0 \leq i,j \leq n$ in parallel **do**
  $column[i][j] \leftarrow 0$
  $top[i][j] \leftarrow a[i][j]$
  $control[i][j] \leftarrow 0$
  $W[i][j] \leftarrow 0$
  $old\_W[i][j] \leftarrow 0$
end for
for $0 \leq i \leq n$ in parallel **do**
  $control[i][0] \leftarrow 0$
  $sol[i][0] \leftarrow -\infty$
  $a[i][0] \leftarrow 0$
  $sol[0][i] \leftarrow -\infty$
  $a[0][i] \leftarrow 0$
end for

Main

for $k \leftarrow 1$ to $2n-1$ **do**
  for all $1 \leq i,j \leq n$ in parallel **do**
    if $control[i][j-1] = 1$ **then**
      $column[i][j] \leftarrow column[i-1][j] + a[i][j]$
      $cand_1 \leftarrow W[i][j-1] + column[i][j]$
      $cand_2 \leftarrow W[i-1][j] + a[i][j]$
      $cand_3 \leftarrow old\_W[i][j] + top[i][j]$
      $old\_W[i][j] \leftarrow W[i][j]$
      $W[i][j] \leftarrow \mathbb{MAX}\{cand_1, cand_2, cand_3\}$
      $top[i][j] \leftarrow top[i-1][j]$
      $sol[i][j] \leftarrow \mathbb{MAX}\{sol[i-1][j], sol[i][j-1], sol[i][j], W[i][j]\}$
      $control[i][j] \leftarrow 1$
    end if
  end for
end for

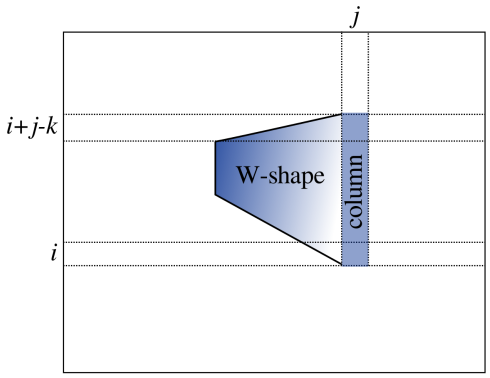

Figure 8. Illustration for Algorithm 8 with position indices (i,j) and time index k.

Note that the memory requirement for each cell in Algorithm 8 is constant. When we put W-shapes and N-shapes together, however, we need $O(n)$ space in each cell, as we will explain later on. We assume that all corresponding instructions in all cells are executed in parallel in a synchronized manner. We later make some comments regarding how synchronization is achieved in the actual implementation of the parallel algorithm.

We prove the correctness of Algorithm 8 in the framework of Hoare logic [11] based on a restricted form of that in Owicki and Gries [12]. The latter is too general to cover our problem. We keep the minimum extension of Hoare logic to our mesh architecture.

The meaning of Hoare's triple $\{P\}S\{Q\}$ is that if P is true before program (segment) S, and if S halts, then Q is true after S stops. The typical loop invariant appears as that for a while-loop; "while B do S". Here, S is a program, and B is a Boolean condition. If we can prove $\{P \wedge B\}S\{P\}$, we can conclude $\{P\}$ while B does $S\{P \wedge \sim B\}$, where $\sim B$ is the negation of B. P is called the loop invariant, because P holds whenever the computer comes back to this point to evaluate the condition B. This is a time-wise invariant as the computer comes back to this point time-wise. We establish invariants in each cell. They are regarded as time-space invariants because the same conditions hold for all cells as computation proceeds. Those invariants have space indices i and j and time index k. Thus, our logical framework is a specialization of Owicki and Gries to indexed assertions.

The main assertions are given in the following. $scope[a, ..., b][c, ..., d]$ is the rectangular array portion from the top-left corner (a, c) to the bottom-right corner (b, d) where a candidate for the solution can be found.

At the end of the k-th iteration, the following holds in $cell(i, j)$:

For $1 \leq i \leq n$ and $0 \leq j \leq k$:

$P_0 : control[i][j] = 1$
$P_1 : column[i][j]$ is the column sum of $a[i + j - k, ..., i][j]$,
which is the sum of the j-th column of array a from position $i + j - k$ to position i
$P_2 : top[i][j] = a[i + j - k][j]$
$P_3 : W[i][j]$ is the maximum W anchored at column $a[i + j - k, ..., i][j]$
$P_4 : old\_W[i][j]$ is the old value of $W[i][j]$
$P_5 : sol[i][j]$ is the maximum W in $scope[i + j - k, ..., i][1, ..., j]$

The above are combined to P, where:

$$P \Leftrightarrow (1 \leq i \leq n \wedge 0 \leq j \leq k \Rightarrow P_0 \wedge P_1 \wedge P_2 \wedge P_3 \wedge P_4 \wedge P_5) \wedge (1 \leq i \leq n \wedge k < j \leq n \Rightarrow Q)$$

Here, Q states that variables in each cell keep the initial values. In the following descriptions, we omit the second portion Q of the above logical formula.

We can prove that $P_0(k)$ to $P_5(k)$ are all true for $k = 0$ by checking the initialization. For each P_0 to P_5, we omit indices i and j. Using the time index k, we prove $\{P(k-1)\}cell(i, j)\{P_0(k)\}$ to $\{P(k-1)\}cell(i, j)\{P_5(k)\}$. We use the following rules of Hoare logic. Let $x_1 \leftarrow y_1$ to $x_n \leftarrow y_n$ be assignment statements in $cell(1)$ to $cell(n)$ in general. There can be several in each cell. We use one for simplicity. The meaning of y_i/x_i is that the occurrence of variable x_i in Q is replaced by y_i. Parallel execution of $cell(1)$ to $cell(n)$ is shown by $[cell(1)||cell(2)...||cell(n)]$.

Parallel assignment rule:

$$\frac{P \Rightarrow Q[y_1/x_1, y_2/x_2, ..., y_n/x_n], \quad \{Q[y_1/x_1, y_2/x_2, ..., y_n/x_n]\} cell(i) \{Q\} \ \textbf{for} \ 1 \leq i \leq n}{\{P\}[cell(1)||cell(2)...||cell(n)]\{Q\}}$$

Other programming constructs such as composition (semi-colon), if-then-else statement, etc., in sequential Hoare logic can be extended to the parallel versions. Those definitions are omitted, but the following rule for the if-then and for-loop for the sequential control structure, which controls a parallel program S from outside, is needed for our verification purpose.

Rule for if-then statement:

$$\frac{\{P \wedge B\} S \{Q\}, P \wedge \neg B \Rightarrow Q}{\{P\} \ \textbf{if} \ B \ \textbf{then} \ S\{Q\}}$$

In our proof, $P(k-1)$ corresponds to P, $control[i][j-1] = 1$ to B and $P(k)$ to Q.

Rule for the for-loop:

$$\frac{\{P(0)\}, \{P(k-1)\} S \{P(k)\}}{\{P(0)\} \ \textbf{for} \ k = 1 \ \textbf{to} \ n \ \textbf{do} \ S\{P(n)\}}$$

This P represents P_0 to P_5 in our program. S is the parallel program $[cell(1)||cell(2)...||cell(n)]$. Each $cell(i)$ has a few local variables and assignment statements. For an arbitrary array x, we regard $x[i][j]$ as a local variable for $cell(i,j)$. A variable from the neighbour, $x[i-1][j]$, for example, is imported from the upper neighbour. Updated variables are fetched in the next cycle. The proof for each $\{P(k-1)\} cell(i,j) \{P(k)\}$ for P is given in Appendix.

Theorem 1. *Algorithm 8 is correct. The result is obtained at* $cell(n,n)$ *in* $2n-1$ *steps.*

Proof. From the Hoare logic rule for the for-loop, we have $P_5(2n-1)$ at the end.

$$\begin{aligned} P_5(2n-1) \ \text{ at } \ cell(n,n) &\Leftrightarrow sol[n][n] \ \text{ is the maximum sum in } \ scope[n+n-2n+1, ..., n][1, ..., n] \\ &\Leftrightarrow sol[n][n] \ \text{ is the maximum sum in } \ scope[1, ..., n][1, ..., n] \end{aligned}$$

□

We used array $W[1..n][1..n]$ to compute the maximum W-shape. The value of $W[i][j]$ at $cell(i,j)$ is ephemeral in the sense that its value changes as computation proceeds. That is, $W[i][j]$ at time k holds the maximum W-shape anchored at column $a[i+j-k, ..., i][j]$, and at the next step, it changes to that value of W anchored at column $a[i+j-k-1, ..., i][j]$.

In order to combine W and N, we must memorize the maximum W and the maximum N for each anchor column. Thus, we need the three-dimensional array *store_W*, as well as the array *store_N*. Since in the final stage of WN computation, the same anchor column is added from W and N, we need to subtract the sum of the anchor column. For that purpose, the sum of anchor column $a[k, ..., i][j]$ is stored in $store\_column[i][k][j]$. Note that the computation of N goes from right to left.

In Algorithm 9, we provide the complete algorithm that combines W and N, where *sol* is dropped, and the initialization is omitted. The computation of W and N takes $2n-1$ steps, and WN takes n steps.

Selecting the maximum of $WN[1..n][1..n]$ can be done in parallel. Algorithm 10 and Figure 9a illustrate how to find the maximum in a 2D mesh in $2n$ steps, where the maximum can be retrieved at the bottom right corner. If we orchestrate the bidirectional data movement in each of four quarters of the mesh (Figure 9b), so that the maximum of each quarter can meet at the centre for the final selection, it can be done in $n+1$ steps. Therefore, Algorithm 9 takes total of $4n$ communication steps.

Algorithm 9 Combining W and N.

1: **for** $k \leftarrow 1$ to $2n-1$ **do**
2: **for all** $1 \leq i, j \leq n$ in parallel **do**
3: **if** $control[i][j-1] = 1$ **then** ▷ Compute W from left to right
4: $column[i][j] \leftarrow column[i-1][j] + a[i][j]$
5: $cand_1 \leftarrow W[i][j-1] + column[i][j]$
6: $cand_2 \leftarrow W[i-1][j] + a[i][j]$
7: $cand_3 \leftarrow old\_W[i][j] + top[i][j]$
8: $old\_W[i][j] \leftarrow W[i][j]$
9: $W[i][j] \leftarrow \mathbb{MAX}\{cand_1, cand_2, cand_3\}$
10: $store\_column[i][i+j-k][j] \leftarrow column[i][j]$
11: $store\_W[i][i+j-k][j] \leftarrow W[i][j]$
12: $top[i][j] \leftarrow top[i-1][j]$
13: $control[i][j] \leftarrow 1$
14: **end if**
15: **if** $control[i][j+1] = 1$ **then** ▷ Compute N from right to left
16: Mirrored operation of lines 4–13
17: **end if**
18: **end for**
19: **end for**
20: **for all** $1 \leq i, j \leq n$ in parallel **do**
21: $WN[i][j] \leftarrow -\infty$
22: **for** $k \leftarrow 1$ to i **do**
23: $anchor \leftarrow store\_column[i][k][j]$
24: $WN[i][j] \leftarrow \mathbb{MAX}\{WN[i][j], store\_W[i][k][j] + store\_N[i][k][j] - anchor\}$
25: **end for**
26: **end for**
27: Find the maximum of $WN[1..n][1..n]$ by Algorithm 10

Algorithm 10 Find the maximum of $a[1..n][1..n]$ in $2n$ steps.

1: **for all** $1 \leq i, j \leq n$ in parallel **do**
2: $M[i][j] \leftarrow a[i][j]$
3: **end for**
4: **for** $k \leftarrow 1$ to n **do**
5: **for all** $1 \leq i, j \leq n$ in parallel **do**
6: $M[i][j] \leftarrow \mathbb{MAX}\{M[i-1][j], M[i][j]\}$
7: **end for** ▷ Find the maximum in the same column
8: **end for**
9: **for** $k \leftarrow 1$ to n **do**
10: **for all** j in parallel **do**
11: $M[n][j] \leftarrow \mathbb{MAX}\{M[n][j-1], M[n][j]\}$
12: **end for** ▷ Find the maximum in the n-th row
13: **end for** ▷ $M[n][n]$ is the maximum

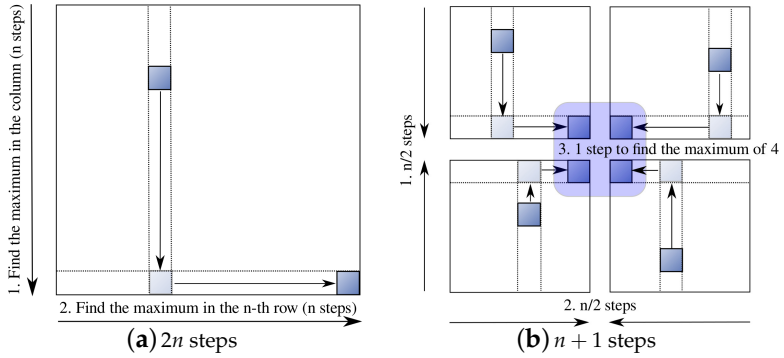

Figure 9. Maximum finding in 2D mesh.

6. Computation of the Boundary

So far, we computed the maximum sum of the WN-shape. For practical purposes, we sometimes need the shape itself. This problem was solved by Thaher [13] for the sequential algorithm (Algorithm 11). We show how the idea works in our parallel situation. For simplicity, we only show the boundary computation of the W-shape. We prepare the array $store\_d[i][k][j]$ to memorize the direction from which the algorithm took one of Case 1, Case 2 or Case 3. This part of enhancement in Algorithm 9 is shown below.

$$\text{Case 1} : store\_d[i][i+j-k][j] \leftarrow 1$$
$$\text{Case 2} : store\_d[i][i+j-k][j] \leftarrow 2$$
$$\text{Case 3} : store\_d[i][i+j-k][j] \leftarrow 3$$

Let array $boundary[i][j] = 1$, if (i, j) is on the boundary, and zero otherwise. Let (i, k, j) denote the j-th column from position k to position i, to represent the anchor column of some W-shape. Suppose the anchor column of the maximum W-shape after Algorithm 9 is (i_0, k_0, j_0). The following sequential algorithm goes backward from (i_0, k_0, j_0) guided by the data structure $store\_d[i][k][j]$. The correctness of the algorithm is obvious. The time $O(n)$ can be shown as follows.

Algorithm 11 Computing the boundary.

```
1: i ← i_0, j ← j_0, k ← k_0
2: while j > 0 and W[i][k][j] > 0 do
3:     boundary[i][j] ← 1
4:     boundary[k][j] ← 1
5:     D ← store_d[i][k][j]
6:     if D=1 then
7:         j ← j − 1
8:     else if D=2 then
9:         i ← i − 1
10:    else if D=3 then
11:        k ← k + 1
12:    end if
13: end while
```

The time is proportional to the number of changes on indices i, j and k. The index j can be reduced at most n times. For i and k, we observe $i - k$ decreases whenever i or k changes, resulting in $O(n)$ time for i and k. The $O(n)$ time for the boundary can be absorbed in $O(n)$ time of Algorithm 9. It will be easy to organize parallel computation of tracing back over array $store\_d[i][k][j]$ by the mesh architecture.

For example, we can convert the array index j to a processor index j and use a one-dimensional mesh architecture as shown below (Algorithm 12).

This version has $O(n)$ time, which provides no gain time-wise, but the $O(n^3)$ space requirement on a single processor can be eased.

Algorithm 12 Computing the boundary; j-th cell.

```
1: receive (i, k)
2: while j > 0 and W[i][k] > 0 do
3:     boundary[i][j] ← 1
4:     boundary[k][j] ← 1
5:     D ← store_d[i][k]
6:     if D=1 then
7:         send (i, k)-to j − 1-th processor
8:     else if D=2 then
9:         i ← i − 1
10:    else if D=3 then
11:        k ← k + 1
12:    end if
13: end while
    Main
14: for 1 ≤ j ≤ n in parallel do send (i_0, k_0) to j_0-th processor
15: end for
```

7. Implementation

We implemented Algorithm 9 on the Blue Gene/P computer under the MPI/Parallel C program environment. There were many practical issues to be considered. We summarize just three issues here as representatives.

Firstly, we cannot assume that each cell knows its own position within the mesh array. Depending on the architecture, additional computation is required for each cell to gather this information. Within the MPI environment, we can let each $cell(i, j)$ know its position (i, j) by the system call "MPI_Cart_coords()".

The second issue is synchronization. We assumed the corresponding statements in all cells are executed in a synchronized manner. If we remove this assumption, that is if the execution proceeds in an asynchronous manner, the algorithm loses its correctness.

In MPI, "MPI_Send()", "MPI_Recv()" and "MPI_Sendrecv()" functions are used to perform synchronous blocking communications [14]. As we call these functions in each step, no further mechanisms are necessary to ensure synchronization between cells as the function calls ensure that any given cell cannot progress one or more steps further than the rest. In other words, one cell may reach the end of the given step and tries to move onto the next step before others, but then, the cell must wait for the other cells to reach the same point due to the blocking nature of the MPI communication functions.

The third implementation issue is related to the number of available processors. As the number of processors is limited, for large n, we need to have what is called a coarse grain parallel computer. Suppose, for example, we are given a $(1024, 1024)$ input array and only 16 processors are available. The input array is divided into sixteen $(256, 256)$ sub-arrays, to which the sixteen processors are assigned. Let us call the sub-array for each processor its territory. Each processor simulates one step of Algorithm 8 sequentially. These simulation processes by sixteen processors are done in parallel. At the end of each simulation, the values in the registers on the right and bottom border are sent to the left and top borders of the right neighbour and the lower neighbour, respectively. The simulation of one step takes $O((n/p)^2)$ time, and $2n - 1$ steps are carried out, meaning the

computing time is $O(n^3/p^2)$ at the cost of $O(p^2)$ processors. When $p = 1$, we hit the sequential complexity of $O(n^3)$. If $p = n$, we have the time complexity of Algorithm 8, which is $O(n)$.

While resolving the third implementation issue, we must again face the second issue of synchronization. For each processor to simulate the parallel computation in its own territory in a coarse-grained implementation, we must take extra care to ensure synchronization between simulated cells within each territory. Specifically, we must double the number of variables, that is we prepare variable x_1 for every variable x. Let us associate the space/time index, (i, j, k) with each variable. Let us call $x(i, j, k)$ the current variable and the variable with indices different by one a neighbour variable. For example, $sol[i][j]$ in the right-hand side of the assignment statement is a time-wise neighbour, and that at the left-hand side is a current variable. Furthermore, $column[i-1][j]$ in the right-hand side is a neighbour variable space-wise and time-wise, and so on. If x is a current variable, change it to x_1. If it is a variable of a neighbour, keep it as it is. Let us call the modified program P_1. Now, we define "update" to be the set of assignment statements of the form $x \leftarrow x_1$.

Example 1. *Let P be a one-dimensional mesh program given by Algorithm 13, which shifts array x by one place. Let us suppose $x[0] = 0$ and $x[i]$ are already given.*

In Algorithm 13, $x[i]$ is the current variable, and $x[i-1]$ is a neighbour variable space-wise and time-wise. An asynchronous computer can make all values zero. For the intended outcome, we perform P_1, given by Algorithm 14, which includes "synchronize" and "update".

Algorithm 13 Program P.

```
1: for all i in parallel do
2:     x[i] ← x[i − 1]
3: end for
```

Algorithm 14 Program P_1.

```
1: for all i in parallel do
2:     x1[i] ← x[i − 1]
3: end for
4: synchronize
5: for all i in parallel do
6:     x[i] ← x1[i] /* update */
7: end for
```

For our mesh algorithm, Algorithm 8, omitting the initialization part, we make the program of the form that is given by Algorithm 15.

Algorithm 15 Synchronization for coarse grain.

```
1: for k ← 1 to 2n − 1 do
2:     P1
3:     synchronize
4:     update
5: end for
```

Algorithm 9 was executed on Blue Gene/P with up to 1024 cores. For the software side, the programming environment of MPI and the parallel C compiler, mpixlc, were used with Optimization Level 5. The results are shown in Table 1. Arrays of size (n, n) containing random integers were tested. The times for generating uniformly-distributed random numbers, as well as loading the data onto each processor were not included in the time measurement. On Blue Gene/P, the mesh architecture can be configured into a 2D mesh or 3D mesh. The time taken to configure the mesh into a 2D mesh array

for our algorithm was included in the time measurement. As we can see from the table, for small n, the configuration time dominates, and increasing the number of processors does not result in a decrease in the execution time. As the size of the input array increases, however, we can see noticeable improvements in the execution times with a larger number of processors.

Note that we were unable to execute the program under certain configurations as shown by 'x' in Table 1. This was due to memory requirements. With large n and small p, each processor must simulate a very large territory. With each cell in the territory requiring $O(n)$ memory to store the maximal values of W and N, the memory requirements can become prohibitive. This highlights the fact that parallelization is required not only to reduce the execution time, but also to handle larger input data.

Table 1. Time measurements on BG/P(seconds).

	$p^2 = 4$	16	64	256	1024
$n = 64$	0.1069	0.0329	0.0258	0.0251	0.0272
128	0.8435	0.2298	0.0989	0.1253	0.1252
256	7.2762	1.7506	0.4783	0.3783	0.6088
512	64.5455	15.8192	3.6038	1.3141	2.9883
1024	332.8847	137.6626	33.9193	7.7712	7.9303
2048	x	x	188.9598	87.5153	43.5064

8. Lower Bound

Algorithm 8 for W is not very efficient, as cells to the right are idling at the early stage. Suppose we are given an input array with the value a in the top-left cell and b in the bottom-right cell, while all other value are $-\infty$. Obviously the solution is a if $a > b$, and b otherwise. The values a and b need to meet somewhere. It is easy to see that the earliest possibility is at time $n - 1$. Our algorithm for W takes $2n - 1$ steps, meaning there is still a gap of n steps with the lower bound. This is a sharp contrast with the mesh algorithm for MSA (Algorithm 4) that completes in $n - 1$ steps. The role of Algorithm 8 is to establish an $O(n)$ time for the MCS problem on the mesh architecture.

9. Concluding Remarks

We gave an $O(n)$ time parallel algorithm for solving the MSA and MCS problem with n^2 processors and a formal proof for the algorithm to compute the W-shape, a part of the MCS problem. The formal proof for the N-shape can be given in a similar way. The formal proof not only ensures correctness, but also clarifies what is actually going on inside the algorithm.

The formal proof was simplified by assuming synchronization. The asynchronous version with (synchronize, update) in Section 7 would require about twice as much complexity for verification, since we double the number of variables. Once the correctness of the synchronized version is established, that of the asynchronous version will be acceptable without further verification.

It is open whether there is a mesh algorithm for the MCS problem with the $O(1)$ memory requirement in each cell. Mesh algorithms are inherently easy to implement on an FPGA and, thus, can be considered for practical applications.

Appendix A. Proof for Invariants

In the following, $sum(k, ..., i; j)$ is the sum of array portion $a[k, ..., i][j]$. We assume $control[i][j-1] = 1$.

Proof. For $\{P(k-1)\}cell(i,j)\{P_0(k)\}$: At the beginning of the k-th iteration, $control[i][j] = 1$ for $j = 0, ..., k-1$, equivalently $control[i][j-1] = 1$ for $j = 1, ..., k$. $cell(i,j)$ performs "$control[i][j] \leftarrow 1$" for $j = 1, ..., k$. Thus, we have $\{P(k-1)\}cell(i,j)\{P_0(k)\}$ for $j = 1, ..., k$. □

Proof. For $\{P(k-1)\}cell(i,j)\{P_1(k)\}$: At time $k-1$, $column[i-1][j] = sum(i-1+j-(k-1),...,i-1;j) = sum(i+j-k,...,i-1;j)$. "$column[i][j] \leftarrow column[i-1][j]+a[i][j]$" is performed for $i = 1,...,n$ and $j = 1,...,k$ in parallel. Thus, $\{P(k-1)\}cell(i,j)\{P_1(k)\}$ holds. □

Proof. For $\{P(k-1)\}cell(i,j)\{P_2(k)\}$: At time $k-1$, $top[i-1][j] = a[i+j-1-(k-1)][j] = a[i+j-k][j]$. At time k, "$top[i][j] \leftarrow top[i-1][j]$" is performed for $i = 1,...,n$ and $j = 1,...,k$ in parallel. Thus, $top[i][j] = a[i+j-k][j]$, and $\{P(k-1)\}cell(i,j)\{P_2(k)\}$ holds. □

Proof. For $\{P(k-1)\}cell(i,j)\{P_3(k)\}$: At time $k-1$, $W[i][j-1]$ is the maximum of W-shapes anchored at column $a[i+(j-1)-(k-1),...,i][j-1] = a[i+j-k,...,i][j-1]$. By adding the sum of column $a[i+j-k,...,i][j]$, $cand_1$ is made. $W[i-1][j]$ is the maximum of the W-shape anchored at column $a[i-1+j-(k-1),...,i-1][j] = a[i+j-k,...,i-1][j]$. By adding $a[i][j]$, $cand_2$ is made. $old\_W[i][j]$ is $W[i][j]$ at time $k-1$, which is the maximum of W-shapes anchored at column $a[i+j-(k-1),...,i][j] = a[i+j-k+1][j]$. By adding $top[i][j] = a[i+j-k][j]$, $cand_3$ is made. Since the maximum of the W-shape based on column $a[i+j-k,...,i][j]$ is chosen from only those three possibilities, $W[i][j]$ is correctly computed. □

Proof. For $\{P(k-1)\}cell(i,j)\{P_4(k)\}$: After $cell(i,j)$, the value of $old\_W[i][j]$ changes from $W[i][j]$ at $k-2$ to $k-1$. Thus, $\{P(k-1)\}cell(i,j)\{P_4(k)\}$ holds. □

Proof. For $\{P(k-1)\}cell(i,j)\{P_5(k)\}$: The scopes for the four candidates are given below.

- $sol[i-1][j]$ at $k-1$: $scope[i-1+j-(k-1),...,i-1][1,...,j] = scope[i+j-k,...,i-1][1,...,j]$
- $sol[i][j-1]$ at $k-1$: $scope[i+(j-1)-(k-1)..i][1,...,j-1] = scope[i+j-k,...,i][1,...,j-1]$
- $sol[i][j]$ at $k-1$: $scope[i+j-(k-1),...,i][1,...,j] = scope[i+j-k+1,...,i][1,...,j]$
- $W[i][j]$ at k is anchored at column $a[i+j-k,...,i][j]$ with $scope[i+j-k,...,i][1,...,j]$

□

There are only four candidates for $sol[i][j]$ from those four scopes. Thus, $\{P(k-1)\}cell(i,j)\{P_5(k)\}$ holds.

Acknowledgments: This research was supported by the EU/NZ Joint Project, Optimization and its Applications in Learning and Industry (OptALI). The experiments were conducted utilising high performance computing resources from the New Zealand eScience Infrastructure (NeSI), especially the High Performance Computing unit at the University of Canterbury (UCHPC). [NeSI-UCHPC]URL https://www.nesi.org.nzorhttp://www.hpc.canterbury.ac.nz

Author Contributions: All authors contributed to the design of algorithms; T.S. provided the codes; T.S. and S.E.B. designed and performed the experiments; T.T. provided the formal proofs; S.E.B provided the figures; T.T. wrote more of the initial draft, and all authors contributed to the finalization of the paper

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bell, G. Tony hey and alex szalay: Beyond the data deluge. *Science* **2009**, *323*, 1297–1298.
2. Pavlus, J. The search for a new machine. *Sci. Am.* **2015**, *312*, 48–53.
3. Fukuda, T.; Morimoto, Y.; Morishita, S.; Tokuyama, T. Data mining with optimized two-dimensional association rules. *ACM Trans. Database Syst.* **2001**, *26*, 179–213.
4. Bentley, J.L. Perspective on performance. *Commun. ACM* **1984**, *27*, 1087–1092.
5. Tamaki, H.; Tokuyama, T. Algorithms for the maximum subarray problem based on matrix multiplication. In Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1998), San Francisco, CA, USA, 25–27 January 1998; pp. 446–452.
6. Takaoka, T. Efficient algorithms for the maximum subarray problem by distance matrix multiplication. *Electr. Notes Theor. Comput. Sci.* **2002**, *61*, 191–200.
7. Bae, S.E. Sequential and Parallel Algorithms for Generalized Maximum Subarray Problem. Ph. D Thesis, University of Canterbury, Christchurch, New Zealand, 2007.

8. Bae, S.E.; Takaoka, T. Algorithms for the Problem of *K* Maximum Sums and a VLSI Algorithm for the *K* Maximum Subarrays Problem. In Proceedings of the 7th International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN 2004), Hong Kong, China, 10–12 May 2004; pp. 247–253.
9. Takaoka, T. Efficient parallel algorithms for the maximum subarray problem. In Proceedings of the 12th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2014), Auckland, New Zealand, 20–23 January 2014; Volume 152, pp. 45–50.
10. Alves, C.E.R.; Caceres, E.N.; Song, S.W. BSP/CGM Algorithms for Maximum Subsequence and Maximum Subarray. In Proceedings of the 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 19–22 September 2004; pp. 139–146.
11. Hoare, C.A.R. An axiomatic basis for computer programming. *Commun. ACM* **1969**, *12*, 576–580.
12. Owicki, S.; Gries, D. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM* **1976**, *19*, 279–285.
13. Thaher, M. Efficient Algorithms for the Maximum Convex Sum Problem. PhD Thesis, University of Canterbury, Christchurch, New Zealand, 2014.
14. IBM: Parallel Environment Runtime Edition for Linux: MPI Programming Guide (SC23-7285-01) (7/2015). Available online: http://publib.boulder.ibm.com/epubs/pdf/c2372851.pdf (accessed on 29 December 2016).