Efficient Parallel Algorithms for the Maximum Subarray Problem [*]

Tadao Takaoka
Department of Computer Science
University of Canterbury
Christchurch, New Zealand

Abstract

Parallel algorithm design is generally hard. Parallel program verification is even harder. We take an example from the maximum subarray problem and and show those two problems of design and verification.

The best known communication steps for a mesh architecture for the maximum subarray problem is $2n - 1$. We give a formal proof for the parallel algorithm on the mesh architecture based on Hoare logic. The main part of the proof is to establish several space/time invariants with three indices (i, j, k). The indices (i, j) pair specifies the invariant at the (i, j) grid point of the mesh and k specifies the k-th step in the computation. Then ignoring additive constants, the communication steps are improved to $(3/2)n$ steps and finally n steps, which is optimal in terms of communication steps. Also the first algorithm is implemented on a Blue Gene parallel computer and performance measurements conducted are shown.

1 Introduction

The maximum subarray problem is to find a rectangular subarray in the given (n, n)-two dimensional array that maximizes the sum in it. If the array elements are non-negative, we have the trivial solution of the whole array. Thus we subtract the mean value, or another anchor value depending on applications. This problem has wide applications from graphics to data mining. In graphics, the maximum subarray corresponds to a brightest spot in the given graphic image. In data mining, suppose we spread the sale amounts of some product on a two dimensional array classified by ages and annual income. Then the maximum subarray corresponds to the most promising customer range.

The typical algorithm by Bentley [2] takes $O(n^3)$ time on a sequential computer. This has been improved to slightly sub-cubic by Tamaki and Tokuyama [8], and Takaoka [7]. When the data is large, such as (1024, 1024) in graphics applications, those time complexities are prohibitive. This is more so, when we need to process video images in dynamic changing situations. An obvious choice is parallel computing. In Takaoka [7], a parallel implementation on a PRAM is discussed. As parallel computers such as GPU are becoming readily available, we need to devise a parallel algorithm implementable on those parallel computers. In Bae and Takaoka [4], Bae [3] a parallel algorithm was implemented on an (n, n)-two dimensional architecture based on the row-wise prefix sum. In this paper, we implement an algorithm based on the column-wise prefix sum with different data flow on a two-dimensional mesh architecture. See Fig. 1. Our algorithm performs the computation in $2n-1$ steps, where steps mean communication steps. Each cell executes a few statements per communication step. Thus our algorithm is cost optimal with respect to the prefix sum-based sequential algorithm. A parallel algorithm for the same problem was implemented on the BPS/CGM architecture, which has more local memory and communication capabilities with remote processors [1].

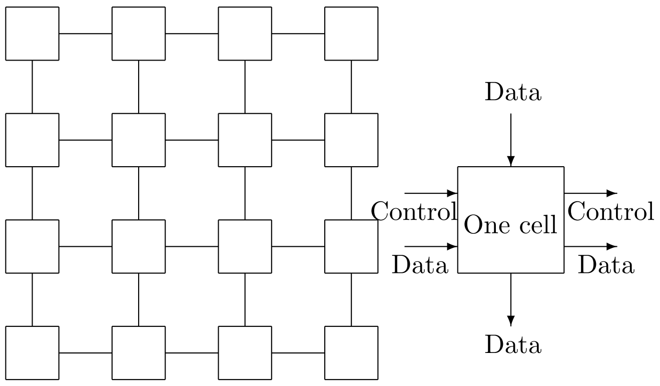

Figure 1: Two-dimensional architecture

We give a formal proof for the algorithm. It is based on the space-time invariants defined on the architecture. The proof leads us to a further speed-up of the computation. The data flow in the first implementation is from left to right and from top to down. The proof reveals that the processors to the right are idling at the the beginning. We first extend the data flow to operate in both directions horizontally to get the $(3/2)n$ steps result. Then we further extend data flow to operate in both directions vertically, i.e., data flow in four directions, so that the solution can be obtained at the center. By this method we achieve n steps, which is optimal. Algorithms are given by pseudo code.

[*]This research was supported by the EU/NZ Joint Project, Optimization and its Applications in Learning and Industry (OptALI).

2 Sequential algorithm

We modify a sequential algorithm based on row-wise prefix sums introduced in [4] to the one based on column-wise prefix sums, to develop Algorithms 1, 2 and 3.

Algorithm - sequential

```
for $i := 1$ to $n$ do
{$min\_sum[i][0] := \infty; sum[i][0] := 0;$
  $sol[i][0] := -\infty;$ }
$S := -\infty;$
for$k := 1$ to $n$ do {
  for $j := 1$ to $n$ do $column[k-1][j] := 0;$
  for $i := k$ to $n$ do {
    for $j := 1$ to $n$ do{
      $column[i][j] := column[i-1][j] + a[i][j];$
      $sum[i][j] := sum[i][j-1] + column[i][j];$
      $min\_sum[i][j] :=$
        $min\{sum[i][j], min\_sum[i][j-1];$ };
      $max\_sum[i][j] := sum[i][j] - min\_sum[i][j];$
      $sol[i][j] := max\{max\_sum[i][j], sol[i][j-1]\};$
    } /* $j$ */
      if $solution[i][n] > S$ then $S := solution[i][n];$
  } /* $i$ */
} /* $k$ */
```

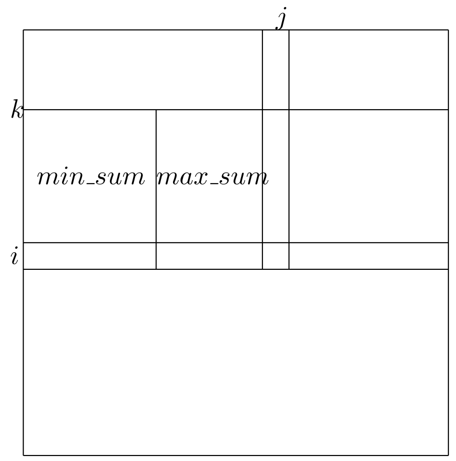

Figure 2: Strip-based sequential computation

The computation proceeds with the strip of the array from position k to position i. 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, expressed by “min_sum” in the figure. 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.

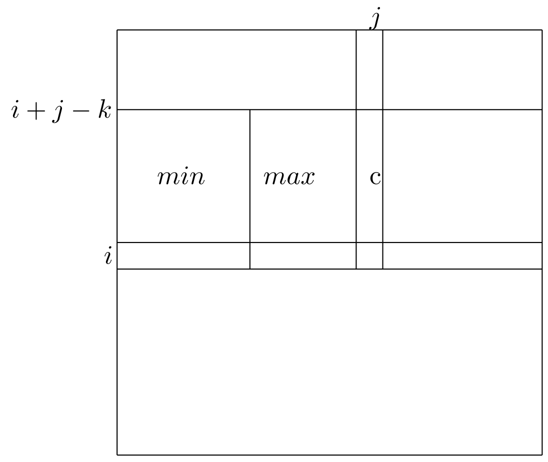

Figure 3: Illustration for Algorithm 1 (c for $column$)

3 Parallel algorithm Algorithm 1

The following is a parallel algorithm corresponding to the sequential algorithm in the previous section. The following program is executed by a 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 the cell (i, j), it accumulates the column sum “$column$” (c in the figure), the sum “sum”, and update min_sum, etc. We assume 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.

Algorithm 1

```
Initialization
for all $i, j$ between 0 and $n$ do in parallel
  {$column[i][j] := 0;$ $min[i][j] := \infty;$
  $control[i][[j] := 0;$ $sum[i][j] := 0;$}
for all $i$ in parallel do {$control[i][0] = 1;$
  $sol[i][0] := -\infty;$ }
Main
  for $k := 1$ to $2n - 1$ do
    for all $i, j$ between 1 and $n$ do in parallel {
      if $control[i][j-1] = 1$ then {
        $column[i][j] := column[i-1][j] + a[i][j];$
        $sum[i][j] := sum[i][j-1] + column[i][j];$
        $min[i][j] :=$
          $minimum(min[i][j-1], sum[i][j]);$
        $max[i][j] := sum[i][j] - min[i][j];$
        $sol[i][j] := maximum(sol[i-1][j],$
          $sol[i][j-1], sol[i][j], max[i][j]);$
        $control[i][j] := 1;$
      }
    } ** i, j **
  } ** k **
```

We prove the correctness of this parallel program in a framework of Hoare logic [5] based on a restricted form of that in Owicki and Gries [6]. The latter is too general to cover our problem. We keep the minimum extension of Hoare logic to our mesh architecture. The meaning of $\{P\}S\{Q\}$ is that if P is true before program (segment) S and S stops, 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 do $S\{P \wedge \sim B\}$, 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 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. Note the difference between the wordings "sum of" and "sum in". Indices are attached to assertion names when necessary.

At time k (at the end of the k-th iteration) the following hold.
For $i = 1, ..., n$ and $j = 1, ..., k$
P_0 : $control[i][j] = 1$
P_1 : $c[i][j]$ is the column sum of $a[i+j-k, ..., i][j]$, that is, the sum of the j-th column of array a from position $i + j - k$ to position i
P_2 : $s[i][j]$ is the sum of $a[i + j - k, ..., i][1, ..., j]$
P_3 : $min[i][j]$ is the minimum of $s[i][l], l = 1, ..., j$
P_4 : $max[i][j]$ is the maximum of the sum of $a[i + j - k, ..., i][l, ..., j], 1 \leq l \leq j$.
P_5 : $sol[i][j]$ is the maximum sum in $a[i + j - k, ..., i][1, ..., j]$, that is, the sum of the maximum subarray of array portion $a[i + j - k, ..., i][1, ..., j]$.

The above are equivalent to
$1 \leq i \leq n \wedge 1 \leq j \leq k \Rightarrow P_0, ..., P_5$.
From this we obviously have $P_0(k) = true, ..., P_5(k) = true$ for $k = 0$.

For each P_0, ..., P_5 we omit indices i and j. Using the time index k, we prove $\{P_0(k-1)\}cell(i,j)\{P_0(k)\}$, ..., $\{P_5(k-1)\}cell(i,j)\{P_5(k)\}$.

We use the following proof rules. Let x_1, ..., x_n be local variables in $cell(1), ..., cell(n)$. 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 $cell(1), ..., cell(n)$ is shown by $[cell(1)||...||cell(n)]$.

Parallel assignment rule

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

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

Rule for 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, ..., P_5 in our program. S is a parallel program $[cell(1)||...||cell(n)]$. 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. How to implement this part depends on the parallel computing environment used. See Section 6. The proof for each $\{P(k-1)\}cell(i,j)\{P(k)\}$ for $P = P_0, ..., P_5$ is given in Appendix.

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

Proof. From the second rule for a sequential loop, we have $P_5(2n-1)$ at the end.
$P_5(2n-1)$ at $cell(n,n)$
$\Leftrightarrow sol[n][n]$ is the maximum sum in $a[n + n - 2n + 1, ..., n][1, ..., n]$
$\Leftrightarrow sol[n][n]$ is the maximum sum in $a[1, ..., n][1, ..., n]$.

4 Algorithm 2

This algorithm 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 1. 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 flows from right to left. At the center, that is, at $(i, n/2)$, $cell(i, n/2)$ performs "$center[i] := max[i][n/2] + max[i][n/2+1]$", 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 over the center. At the end of the k-th iteration, all assertions in Algorithm 1 hold on the left half and the assertions in mirror image hold on the right half. In addition, we have that $center[i]$ is the value of the maximum subarray that lies above or touching the i-th row and crosses over the center line.

Algorithm 2

```
Initialization
for all $i, j$ between 0 and $n$ do in parallel
    {$column[i][j] := 0; min[i][j] := -\infty;$
      $control[i][j] := 0; sum[i][j] := 0;$}
for all $i$ do in parallel
    {$control[i][0] := 1; control[i, n+1] := 1;$ }
Main
    for $k := 1$ to $(3/2)n - 1$ do
        for all $i = 1, ..., n, j = 1, ..., n$ do in parallel
            if $1 \leq j \leq n/2$ then /*** left half ***/
                if $control[i][j-1] = 1$ then {
                    $column[i][j] := column[i-1][j] + a[i][j];$
                    $sum[i][j] := sum[i][j-1] + column[i][j];$
                    $min[i][j] :=$
                        $minimum(min[i][j-1], sum[i][j]);$
                    $max[i][j] := sum[i][j] - min[i][j];$
                    $sol[i][j] := maximum(sol[i-1][j],$
                        $sol[i][j-1]); sol[i][j], max[i][j]);$
                    $control[i][j] := 1;$
                }
            if $n/2 + 1 \leq j \leq n$ then /*** right half ***/
                if $control[i][j+1] = 1$ then {
                    $column[i][j] := column[i-1][j] + a[i][j];$
                    $sum[i][j] := sum[i][j+1] + column[i][j];$
                    $min[i][j] :=$
                        $minimum(min[i][j+1], sum[i][j]);$
                    $max[i][j] := sum[i][j] - min[i][j];$
                    $sol[i][j] :=$
                        $maximum(sol[i-1][j], sol[i][j+1]),$
                        $sol[i][j], max[i][j]);$
                    $control[i][j] := 1;$
                }
            if $j = n/2$ then {
            /*** $cell(i, n/2)$ processes $center[i]$ ***/
                $center[i] := max[i][n/2] + max[i][n/2+1];$
                if $center[i] < center[i-1]$ then
                    $center[i] := center[i-1];$
            }
    } / ** i, j **/
  } /** k **/
***Finalization step ***
  Let $cell(n, n/2)$ do
    $solution = maximum(sol[n][n/2],$
        $sol[n][n/2 + 1], center[n]);$
```

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 cell $cell(i, n/2)$ and $cell(i, n/2 + 1)$ process 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.

5 Algorithm 3

In this algorithm data flows 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 4.

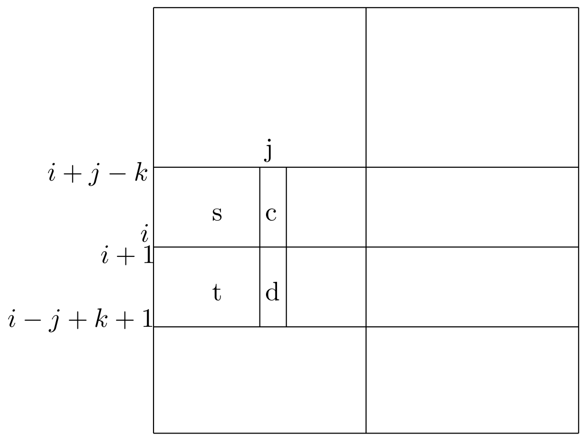

Figure 4: Illustration for Algorithm 3

The proof of Algorithm 1 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 width 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.

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 invariant 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.

Algorithm 3

```
Initialization
for all $i, j$ between 0 and $n + 1$ do in parallel {
  $c[i][j] := 0; d[i][j] := 0;$
  $min[i][j] := -\infty; control[i][[j] := 0;$
  $s[i][j] := 0; t[i][j] := 0;$}
for all $i$ in parallel do
  {$control[i][0] := 1; control[i, n + 1] := 1;$ }
Main
for $k := 1$ to $n - 2$ do
  for all $i = 1, ..., n, j = 1, ..., n$ do in parallel
    if $1 \leq j \leq n/2$ {
      if $control[i][j - 1] = 1$ then {
        $c[i][j] := c[i - 1][j] + a[i][j];$
        $s[i][j] := s[i][j - 1] + c[i][j];$
        $d[i][j] := d[i + 1][j] + a[i][j];$
        $t[i][j] := t[i][j - 1] + d[i][j];$
        $u[i][j] = s[i][j] + t[i + 1][j];$
        $min[i][j] := minimum(min[i][j - 1], u[i][j]);$
        $max[i][j] := u[i][j] - min[i][j];$
        $sol[i][j] := maximum(sol[i - 1][j],$
          $sol[i + 1][j]), sol[i][j - 1], sol[i][j]);$
        $sol[i][j] := maximum(sol[i][j], max[i][j]);$
        $control[i][j] := 1;$
    }
    if $n/2 + 1 \leq j \leq n$ {
      if $control[i][j + 1] = 1$ then {
        $c[i][j] := c[i - 1][j] + a[i][j];$
        $s[i][j] := s[i][j + 1] + c[i][j];$
        $d[i][j] := d[i + 1][j] + a[i][j];$
        $t[i][j] := t[i][j + 1] + d[i][j];$
        $u[i][j] := s[i][j] + t[i + 1][j];$
        $min[i][j] := minimum(min[i][j + 1], u[i][j]);$
        $max[i][j] := u[i][j] - min[i][j];$
        $sol[i][j] := maximum(sol[i - 1][j],$
          $sol[i + 1][j], sol[i][j + 1]), sol[i][j]);$
        $sol[i][j] := maximum(sol[i][j], max[i][j]);$
        $control[i][j] := 1;$
      }
    }
    if $j = n/2$ {
    /*** $cell(i, n/2)$ performs the following. ***/
      $center[i] := max[i][n/2] + max[i][n/2 + 1];$
      if $center[i] < center[i - 1]$ then
          $center[i] = center[i - 1];$
      if $center[i] < center[i + 1]$ then
          $center[i] := center[i + 1];$
    }
  } /** i, j **/
} ** k **
if $i = n/2$ and $j = n/2$ then
/** $cell(n/2, n/2)$ processes $solution$ **/
  $solution := maximum(sol[n/2][n/2],$
        $sol[n/2][n/2 + 1], center[n/2];$
```

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. We described the algorithm for the solution whose height is an even number. This fact comes from the assignment statement "$u[i][j] := s[i][j] + t[i + 1][j]$" where the height 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.

6 Implementation issues

We implemented Algorithm 1 on the Blue Gene/P computer under the MPI/Parallel C program environment. There are many practical issues to be considered. We summarize just three issues here as representatives.

We can let each $cell(i,j)$ know its position (i,j) by a system call "MPI_Cart_coords".

The next issue is synchronization. We assumed the corresponding statements in all cells are executed in a synchronized way. If we remove this assumption, that is, if the execution goes in asynchronous manner, the algorithm loses its correctness. Most mesh computers run asynchronously, but have synchronization primitives. Suppose we have the simplest synchronization primitive "synchronize". This means that when all cells come to this primitive, they can go ahead. In MPI, this primitive is called "MPI_Barrier". To make a correct program, we 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. Also $column[i-1][j]$ in the rght-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 $x := x_1$.

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

```
$P$ : for all $i$ do in parallel $x[i] := x[i-1]$;
```

Here $x[i]$ is the current variable and $x[i-1]$ is a neighbour variable space-wise and time-wise. An asynchronous computer can make all values 0. For the intended outcome, we perform P_1, synchronize and *update*.

```
$P_1$ : for all $i$ do in parallel $x_1[i] := x[i-1]$;
synchronize;
update : for all $i$ do in parallel $x[i] := x_1[i]$;
```

For our mesh algorithm, Algorithm 1, omitting the initialization part, we make the program of the form.

```
for $k := 1$ to $2n-1$ do
begin $P_1$; synchronize; update end.
```

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. This means that given a large (n,n)-array, each processor is given its territory. Suppose, for example, we are given $(1024, 1024)$ array and only 16 processors are available. The input array is divided into sixteen $(256, 256)$ subarrays, to which the sixteen processors are assigned. Let us call the subarray for each processor its territory. Each processor simulates one step of Algorithm 1 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. 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 1, which is $O(n)$.

Based on the above methods for implementation, we implemented Algorithm 1 on the Blue Gene computer at the University of Canterbury. The version was BlueGene/P with 4096 processors, called cores. For the software side, the programming environment of MPI and the parallel C compiler, mpixlc, were used with optimization level 5. The timing results are shown below. The unit of time is second. (n,n) random arrays are tested. The time for generating uniformly distributed random numbers is not included in the time measurement. The mesh architecture can be figured into a 2-D mesh or 3-D mesh. We figured it into 2-D and included the configuration time in the measurement. The data items were loaded appropriately into processors and loading time was excluded from time measurement. As we can see from the table below, for small n, the configuration time dominates and a large number of processors has no effect. As the size of array increases, however, the speed increases with a large number of processors.

n	$p^2 = 1$	$p^2 = 16$	$p^2 = 25$	$p^2 = 100$
50	0.03275	0.00627	0.01274	0.03378
100	0.05143	0.02742	0.02869	0.04643
200	0.14794	0.12909	0.10898	0.11304
500	1.21697	1.32867	1.09698	2.29324
1000	7.72829	8.62130	6.28106	2.96971
1500	22.3838	27.8474	19.1569	7.24681
2000	49.7604	63.9644	42.7302	15.1321
2500	95.3663	120.392	82.2593	26.7386

7 Lower bound

Algorithms 2 and 3 are not very efficient for practical purposes. Rather their roles are to show the optimal bound of communication steps. Suppose we have a value a in the top-left cell and b in the bottom-right cell. All others are -1. 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 the earliest possibility is at time n-1. Thus Algorithm 3 is optimal in communication steps. The role of Algorithm 2 is a bridging step to Algorithm 3.

8 Concluding remarks

We gave a formal proof to a parallel algorithm for the maximum subarray problem. The formal proof for the other two parallel algorithms can be given in a similar way, but the details are omitted. The formal proof is not only for the reliability of the algorithm, but also it clarifies what is going inside the algorithm. In fact, the other two parallel algorithms, Algorithm 2 and Algorithm 3, have been developed by the insight into the data flow, given by the formal proof of Algorithm 1. We simplified the proof by synchronizing everything. The asynchronous version with (synchronize, update) in Section 6 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.

The algorithms are of fine-grain in the sense that each array element, or pixel in graphics, is processed by a processor. When the given array is large, such as (1024, 1024), this is not practical. That is, we need to develop a parallel algorithm of coarse grain. This means one processor will take care of some portion of the given array. When each processor finishes one step of k, the time index, it can communicate with the neighbour for data transmission. In fact this version has been implemented on the BlueGene parallel computer with 4096 processors, and we observed a remarkable speed-up with 100 processors. The experiment conducted was rather of small scale. A larger experiment with a larger number of processors will be carried out in the future.

If we analyze dynamic images, such as video images, data loading becomes a big issue. Data must come through the top or left border processors. For the sake of speed, data must be processed in a pipeline manner, that is, data must be fed into the mesh architecture while the previous image is still being processed. We already have this pipe-lined version of Algorithm 1, which must be tested on the BlueGene for time measurement.

In our architecture, communication with the right neighbour and left neighbour cannot be done at the same time, that is, they are done one by one. To speed up Algorithm 2 and Algorithm 3, we need a more advanced architecture with bi-directional communication capabilities.

Appendix

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

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

Proof for $\{P_2(k-1)\}cell(i,j)\{P_2(k)\}$. At time $k-1$, $s[i][j-1]$ is the sum of $a[i+j-1-(k-1), ..., i][1, ..., j-1] = a[i+j-k, ..., i][1, ..., j-1]$. At time k, "$s[i][j] := s[i][j-1] + c[i][j]$" is performed for $i = 1, ..., n$ and $j = 1, ..., k$ in parallel. Thus $s[i][j]$ is the sum of $a[i+j-k, ..., i][1, ..., j]$, and $\{P_1(k-1)\}cell(i,j)\{P_1(k)\}$ holds.

Proof for $\{P_3(k-1)\}cell(i,j)\{P_3(k)\}$.
At time $k-1$, $min[i][j-1]$ is the minimum of $s[i][l], l = 1, ..., j-1$. At time k, "$min[i][j] := minimum(min[i][j-1], s[i][j])$" is performed for $i = 1, ..., n$ and $j = 1, ..., k$ in parallel. Thus $min[i][j]$ is the minimum of $s[i][l], l = 1, ..., j$. Therefore $\{P_3(k-1)\}cell(i,j)\{P_3(k)\}$ holds.

Proof for $\{P_4(k-1)\}cell(i,j)\{P_4(k)\}$. At time k, $min[i][j]$ is the minimum of $s[i][l], l = 1, ..., j$. At time k, "$max[i][j] := s[i][j] - min[i][j]$" is performed for $i = 1, ..., n$ and $j = 1, ..., k$ in parallel. Thus $max[i][j]$ is the maximum of the sum of $a[i+j-k, ..., i][l, ..., j]$ for $1 \leq l \leq j$. Therefore $\{P_4(k-1)\}cell(i,j)\{P_4(k)\}$ holds.

Proof for $\{P_5(k-1)\}cell(i,j)\{P_5(k)\}$.
At time $k-1$, $sol[i][j-1]$ is the maximum sum in
$a[i+j-1-(k-1), ..., i][1, ..., j-1]$
$= a[i+j-k, ..., i][1, ..., j-1]$,
and $sol[i-1][j]$ is the maximum sum in
$a[i-1+j-(k-1), ..., i-1][1, ..., j]$
$= a[i+j-k, ..., i-1][1, ..., j]$.
At time k,
$sol[i][j] := maximum(sol[i-1][j], sol[i][j-1],$
$sol[i][j], max[i][j])$
is performed for $i = 1, ..., n$ and $j = 1, ..., k$ in parallel. The first two cases do not cover $a[i][j]$. The last two cases cover $a[i][j]$. $sol[i][j]$ is the solution for $cell(i,j)$ for the time up to $k-1$, which does not cover row $i+j-k$, and $max[i][j]$ is the maximum sum of the strip that ends at column j. Thus $sol[i][j]$ is the maximum sum in $a[i+j-k, ..., i][1, ..., j]$, and $\{P_5(k-1)\}cell(i,j)\{P_5(k)\}$ holds.

Acknowledgment The author is thankful to Robin Candy who implemented Algorithm 1 on the Blue Gene and conducted time measurements. He also appreciates useful discussions on the maximum subarray problem with Sung Eun Bae. Finally he is grateful to the supercomputer centre of University of Canterbury, Blue Fern, who offered a free use of Blue Gene for the research on the mesh algorithms.

References

[1] C. E. R. Alves, E. N. Caceres and S. W. Song: BPS/CGM Algorithms for Maximum Subsequence and Maximum Subarray, EuroPVM/MPI 2004, LNCS 3241: 139-146 (2004)

[2] Jon Louis Bentley: Perspective on Performance. Commun. ACM 27(11): 1087-1092 (1984)

[3] Sung Eun Bae: Sequential and Parallel Algorithms for Generalized Maximum Subarray Problem. Ph. D thesis. University of Canterbury (2007)

[4] Sung Eun Bae and Tadao Takaoka: Algorithms for the Problem of K Maximum Sums and a VLSI Algorithm for the K Maximum Subarrays Problem. I-SPAN 2004: 247-253 (2004)

[5] C.A.R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576-580 (1969).

[6] S. Owicki and D. Gries: Verifying properties of parallel programs: An axiomatic approach. Communications of the ACM, 19(5):279-285 (1976)

[7] Tadao Takaoka: Efficient Algorithms for the Maximum Subarray Problem by Distance Matrix Multiplication. Electr. Notes Theor. Comput. Sci. 61: 191-200 (2002)

[8] Hisao Tamaki, Takeshi Tokuyama: Algorithms for the Maximum Subarray Problem Based on Matrix Multiplication. SODA 1998: 446-452 (1998)