

| |
|------------|
| HOMEWORK V |
|------------|

DUE : November 30, 2011

READ :

, Related portions of Chapters 2, 3, 4 and Appendix H of the Hennessy book
 , Related portions of Chapter 1, 2, 3, 4 and 7 of the Jordan book

ASSIGNMENT : There are **six** problems.

Solve all homework and exam problems as shown in class and past exam solutions.

1) Define the following terms related to parallelism and computational methods :

- a) Degree of parallelism
- b) Computational granularity

2) Consider the following piece of **high-level** language loop :

```

for (i=1 ; i < 100 ; i = i + 1) {
    a[i] = b[i] + c[i] ;    /* S1 */
    b[i] = a[i] + d[i] ;    /* S2 */
    a[i+1] = a[i] + e[i] ; /* S3 */
}
  
```

i) List the dependencies.

ii) Rewrite the high-level language loop based on your dependence list so that there is loop-level parallelism. That is, all iterations are independent of each other such that loop body statements for all iterations can be performed in parallel. Therefore, it would not matter whether iteration 28 is done first or iteration 43, the result will be correct. Note that you will **NOT** compile the loop to an assembly code.

3) Develop a **sequential** Binary Search algorithm to search element “**k**” on one-dimensional array **A** whose “**n**” elements are already ordered. If the search is successful, you will return the “index” of the array, i.e. $A[\text{index}] = k$. Otherwise, you will return **-1**. Specify the time complexity of your algorithm. What is the time complexity of your algorithm ?

4) Consider the **sequential** Binary Search algorithm worked on in Question 3 above. Convert the sequential binary search algorithm to a **parallel** binary search PRAM algorithm to search element “**k**” on one-dimensional vector “**A**” with “**n**” ordered elements on “**p**” processors. If a processor finds “**k**,” it returns “index” where $A[\text{index}] = k$. Note that “index” is a global variable. If “**k**” is

not found by any processor, then, “index” contains -1.

- Indicate the time complexity of the algorithm. Can it be cost efficient ? Explain.
- Make observations relevant to the execution of your PRAM algorithm, including the data decomposition, load balancing, the communication graph, etc.

5) Develop a cost efficient dot-product PRAM algorithm on two vectors, “A” and “B” with ”p” processors. Store the result in “k” which is a global variable. In order to keep the result of each processor’s dot product result, use vector “D” with “p” elements. To store the result in “k,” processor 0 copies D[0] to k.

- Indicate the time complexity of your algorithm.
- Make observations relevant to the execution of your PRAM algorithm, including the data decomposition, load balancing, the communication graph, etc.

6) Develop a dot-product SIMD algorithm on two vectors, “A” and “B” with “p” processing elements. Each one stores the result in “k” which is a local variable. Processing Element 0 computes the final value of “k.” Then, it writes to global “k” location which you will **not** show.

The SIMD has a 2-d square mesh interconnection network.

- Indicate the time complexity of your algorithm.
- Make observations relevant to the execution of your SIMD algorithm, including the data decomposition, load balancing, the communication graph, etc.

RELEVANT QUESTIONS AND ANSWERS

Q1) Consider the following piece of old code :

```
loop : L.D          F2, 0(R1)          ; Statement S1. R1 points at vector X & is initially (16)10
      ADD.D         F0, F0, F2        ; S2. F0 is variable k & is initially 0
      DADDI         R1, R1, #(-8)10   ; S3
      BNEZ          R1, loop          ; S4
      S.D           F0, 0(R2)         ; S4
```

R1 and F0 are already initialized. Assume that R1 is initially (16)₁₀.

This code is written for the unpipelined processor, since, for example, the ADD.D is right after the L.D., even though the ADD.D depends on the L.D. Therefore, this code is written for *machine model number 0*. The corresponding algorithm is as follows :

```

for (i = 1 ; i <= 2 ; i++)
    k = k + X[i] ;

```

Determine the time complexity of the algorithm.

A1)

```

for (i = 1 ; i <= 2 ; i++)
    k = k + X[i] ;

```

) **O(n)**

The algorithm is a **sequential** algorithm.

We see that for n elements, the loop body is executed n times. The time complexity is O(n), a **linear** time complexity.

Q2) Consider the **old** code written for the unpipelined CPU (*machine model number 0*) below :

| | | | |
|--------------|-------|--------------------------------------|---|
| | L.D | F8, #0 | ; Load value zero (0) to F8 |
| | DADDI | R1, R0, #(64) ₁₀ | ; Memory accesses are commented below : |
| loop: | L.D | F0, 0(R _A) | ; R _A points at vector A |
| | L.D | F2, 0(R _B) | ; R _B points at vector B |
| | L.D | F4, 0(R _C) | ; R _C points at vector C |
| | MUL.D | F6, F0, F2 | |
| | ADD.D | F8, F8, F6 | |
| | MUL.D | F10, F4, F8 | |
| | S.D | F10, 0(R _C) | ; Stores to vector C |
| | DADDI | R1, R1, #(-1) ₁₀ | |
| | DADDI | R _A , R _A , #8 | ; R _A is advanced |
| | DADDI | R _B , R _B , #8 | ; R _B is advanced |
| | DADDI | R _C , R _C , #8 | ; R _C is advanced |
| | BNEZ | R1, loop | |
| | S.D | F8, 0(R _k) | ; Stores to scalar <i>k</i> out of the loop. |

The corresponding algorithm is as follows :

```

k = 0 ;
for (i = 0 ; i < 64 ; i = i + 1)
{
    k = k + (A[i] * B[i])    /* S1 : Implements the dot product */
    C[i] = k * C[i] ;      /* S2 : Implements the other operation */
}

```

a) Determine the time complexity of the algorithm.

b) List the dependencies

A2) a)

```
k = 0 ;
for (i = 0 ; i < 64 ; i = i + 1)
{
    k = k + (A[i] * B[i])      /* S1 */
    C[i] = k * C[i] ;        /* S2 */
}
```

O(n)

The algorithm is a **sequential** algorithm. We see that for n elements, the loop body is executed n times. The time complexity is O(n), a **linear** time complexity.

- b) From S1 to S2 there is true dependence on k.
- From S2 to S2 there is false dependence on C[i].
- From S2 to S1, there is a loop-carried false dependence on k

Q3) Consider the following **old** MIPS code for the unpipelined MIPS processor (*machine model number 0*) :

```
loop :   L.D      F0, 0(R1)      ; Load from vector A
        ADD.D   F0, F0, F1      ; F1 is already initialized with the constant value "b"
        MUL.D   F0, F0, F2      ; F2 is already initialized with the constant value "c"
        DIV.D   F0, F0, F3      ; F3 is already initialized with the constant value "d"
        S.D     F0, 0(R1)      ; Store to vector A
        DADDI   R1, R1, #8      ; Advance the vector A pointer
        DADDI   R2, R2, #(-1)10 ; Decrement the loop counter
        BNEZ    R2, loop       ; Branch back if not the end
```

This code is without delayed loads, without delayed branches, and without any consideration for the latencies of functional units, etc.

The corresponding algorithm is as follows :

```
For (i = 0 ; i < n ; i++)
    A[i] = ((A[i] + b) * c) / d ) O(n)
```

- a) Determine the time complexity of the algorithm.
- b) Is there a loop-level parallelism in the high-level code ? Explain.

A3) a)

```
For (i = 0 ; i < n ; i++)
    A[i] = ((A[i] + b) * c) / d ) O(n)
```

The algorithm is a **sequential** algorithm. We see that for n elements, the loop body is executed n times. The time complexity is O(n), a **linear** time complexity.

- b) The code has loop-level parallelism since A[i] does not depend on any other A[i], but itself. Thus **all** n iterations can run in parallel. The code is embarrassingly parallel !

Q4) Consider the following **old** MIPS code for the unpipelined MIPS processor (*machine model number 0*) :

```

loop :   L.D      F0, 0(R2)          ; Load from vector B
          L.D      F1, 0(R3)          ; Load from vector C
          L.D      F2, 0(R4)          ; Load from vector D
          MUL.D    F3, F1, F2
          ADD.D    F4, F0, F2
          DIV.D    F5, F0, F2
          ADD.D    F6, F3, F4
          SUB.D    F7, F6, F5
          S.D      F7, 0(R1)          ; Store to vector A
          DADDI   R1, R1, #8           ; Advance the A pointer
          DADDI   R2, R2, #8           ; Advance the B pointer
          DADDI   R3, R3, #8           ; Advance the C pointer
          DADDI   R4, R4, #8           ; Advance the D pointer
          DADDI   R5, R5, #(-1)10    ; Decrement loop counter which has (64)10 initially
          BNEZ    R5, loop             ; Branch back if not the end
    
```

The corresponding algorithm is as follows :

```

For (i = 0 ; i < 64 ; i++)
    A[i] = (C[i] * D[i]) + (B[i] + D[i]) - (B[i] / D[i]) ;
    
```

- a) Determine the time complexity of the algorithm.
- b) Is there a loop-level parallelism in the high-level code ? Explain.

A4) a)

```

For (i = 0 ; i < 64 ; i++)
    A[i] = (C[i] * D[i]) + (B[i] + D[i]) - (B[i] / D[i]) ;
    
```

- b) The code has loop-level parallelism that **all** 64 loop iterations can be run in parallel. For n elements, the loop body is executed n times. The time complexity is O(n), a **linear** time complexity.

Q5) Consider the following algorithm :

```

For (i = 0 ; i < 64 ; i = i + 1)
    { A[i] = (B[i] * c) + D[i] ; /* S1 */
      D[i] = A[i] / e } /* S2 */
    
```

- a) List the dependencies. Explain clearly whether there is loop-level parallelism or not.

A5) a)

```

For (i = 0 ; i < 64 ; i = i + 1)
    { A[i] = (B[i] * c) + D[i] ; /* S1 */
      D[i] = A[i] / e } /* S2 */ } O(n)
    
```

The dependencies are as follows :

=> From S1 to S2 true dependency on A[i]

=> From S1 to S2 antidependence on D[i]

The code has loop-level parallelism since iterations on A[i] and D[i] are independent of each other. Thus **all** n iterations can run in parallel. It does not matter which iteration is performed when, the result will be correct.

Q6) Consider the following **sequential** algorithm :

```

For (j = 1 ; j < n ; j++)
{
    K[j] = K[j] + K[j - 1] ;      /* S1 */
    A[j] = K[j] * B[j] + C[j] ;   /* S2 */
    D[j] = E[j] * B[j] - C[j] ;   /* S3 */
}

```

Vectors K, A, B, C, D and E have “n” elements each, but only last “n-1” elements are changed.

a) List the dependencies. Explain clearly whether there is loop-level parallelism or not.

b) What is the time complexity of the high-level code ? Explain.

c) Develop the corresponding **PRAM** algorithm with “p” processors, named PRAMABD.

Explain the time complexity of the PRAM algorithm.

Make observations relevant to the execution of the PRAM algorithm, including the data decomposition, load balancing, the communication graph, synchronization, etc.

A6) a) The **sequential** algorithm is given :

| | | |
|--|--|--|
| <pre> For (j = 1 ; j < n ; j++) { K[j] = K[j] + K[j - 1] ; /* S1 */ A[j] = K[j] * B[j] + C[j] ; /* S2 */ D[j] = E[j] * B[j] - C[j] ; /* S3 */ } </pre> | $\left. \vphantom{\begin{matrix} \text{For } (j = 1 ; j < n ; j++) \\ \{ \\ K[j] = K[j] + K[j - 1] ; \\ A[j] = K[j] * B[j] + C[j] ; \\ D[j] = E[j] * B[j] - C[j] ; \\ \} \end{matrix}} \right) O(n)$ | <p>a) The dependencies are as follows :</p> <p>=> From S1 to S2 true dependency on K[i]</p> <p>=> From S1 to S1 antidependence on K[i]</p> <p>=> From S1 to S1 loop-carried true dependence on K[i]</p> <p>=> From S2 to S1 loop-carried false dependence on K[i]</p> |
|--|--|--|

There is **no** loop-level parallelism ! Iterations on A[i] and D[i] are independent of each other. But there is a loop-carried true dependence on K[i]. Thus, one can generate two loops, one with S1 and the other one with S2 and S3. In the second loop, **all** n iterations can run in parallel. That is, in the second loop, it does not matter which iteration is performed when, the result will be correct.

b) For “n” elements, the loop body is executed “n-1” times. The time complexity is O(n), **linear** time complexity.

c) The PRAM algorithm is as follows :

PRAMABD (CREW PRAM)

Global : $n ; p ; s ; A[0, \dots, (n-1)] ; B[0, \dots, (n-1)] ; C[0, \dots, (n-1)] ; D[0, \dots, (n-1)] ; E[0, \dots, (n-1)] ; K[0, \dots, (n-1)]$

Local : $i ; j ; my_id$

Begin

$s = n/p$

Spawn ($P_1, P_2, \dots, P_{(p-1)}$)

For all P_i where $0 \leq i \leq (p-1)$ do

For ($j = 1 ; j < s ; j++$)

$K[i*s + j] = K[i*s + j] + K[i*s + j - 1]$

Endfor

For ($j = 0 ; j < s ; j++$)

If $myid = j$ and $myid \neq 0$ then

for ($k = 0 ; k < s ; k++$)

$K[i*s + k] = K[i*s + k] + K[i*s + k - 1]$

Else

If ($myid = 0$ and $j = 1$) or ($myid > 0$ and $j \geq 0$) then

$D[i*s + j] = E[i*s + j] * B[i*s + j] - C[i*s + j]$

Endif

Endfor

For ($j = 0 ; j < s ; j++$)

If ($myid = 0$ and $j = 1$) or ($myid > 0$ and $j \geq 0$) then

$A[i*s + j] = K[i*s + j] * B[i*s + j] + C[i*s + j]$

Endfor

Endfor

End

$O(n/p)$

$O(n/p)$

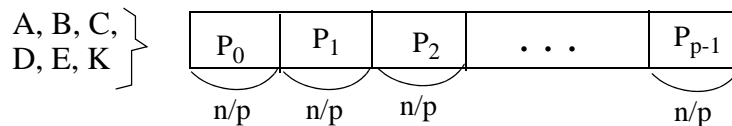
$O(n)$

$O(n/p)$

The parallel time complexity is $O(n)$ since there is no loop-level parallelism. The middle loop takes $O(n)$ time : $O(p) * O(n/p) = O(n)$! $(p-1)$ processors update their K vector elements taking $O(n)$ time. This time complexity is expected as the algorithm has a completely sequential part.

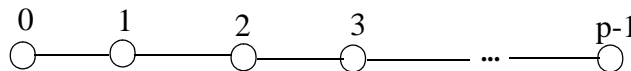
The cost : $O(n) * O(p) = O(np)$. The cost is not equal to the sequential time complexity. Since the cost is not equal to the sequential time complexity, the algorithm is **not** cost efficient.

The initial data decompositions are as shown below. The decompositions change lightly during the execution.



In the middle loop, processors 1 to $p-1$ use the last element of vector K in the domain of the processor whose id number is one less

There is communication among the processors to pass new values of K in a linear array fashion :



Load balancing is good except in the $O(p)$ loop where gradually most of the processors become idle.

Q7) Develop a **cost efficient PRAM** algorithm that corresponds to the following **sequential** algorithm :

```

for (i = 0 ; i < n ; i++) do
  k = A[i]
  if (k = 0) then
    for (j = 0 ; j < 63 ; j++) do
      C[i][j] = B[i][j] + q
    endfor
  endif
endfor

```

The algorithm is called “CONDSCALE” standing for “Conditional Scaling.” Matrices “B” and “C” have “n” rows and 64 columns. Also, note that “n” can be quite large.

- Indicate the time complexity of your PRAM algorithm.
- Make observations relevant to the execution of your PRAM algorithm, including the data decomposition, load balancing, the communication graph, etc.

A7) The sequential time complexity is $O(n)$:

```

for (i = 0 ; i < n ; i++) do
  k = A[i]
  if (k = 0) then
    for (j = 0 ; j < 63 ; j++) do
      C[i][j] = B[i][j] + q
    endfor
  endif
endfor

```

The outer “for” loop is executed “n” times while the inner “for” loop is always executed 64 times. Since 64 is small compared with a large n, the time complexity of the inner loop is considered constant time, $O(1)$. Thus, 64 is ignored for large “n.”

The PRAM algorithm :

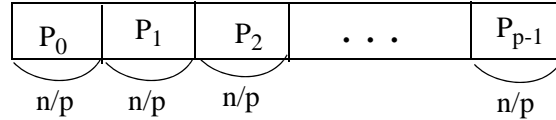
```

CONDSCALE (CREW PRAM)
Global : n ; p ; A[0,...(n-1)] ; B[0,...(n-1)][0,...63] ; C[0,...(n-1)][0,...63] ; q
Local : j, k, l ; i
Begin
  ---
  Spawn (P1, P2, ..., P(p-1))
  For all Pi where 0 ≤ i ≤ (p-1) do
    for (j=0 ; j ≤ (n/p)-1 ; j++) do
      k = A[(i*n/p) + j]
      if (k = 0) then do
        for (l = 0 ; l ≤ 63 ; l++) do
          C[(i*n/p) + j][l] = B[(i*n/p) + j][l] + q
        endfor
      endif
    endfor
  endfor
End

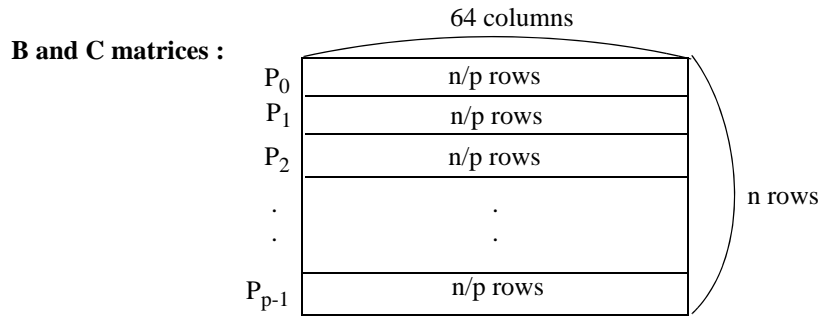
```

The parallel time complexity is $O(n/p)$ since each processor executes the outer loop pointed by the above two parallel lines n/p times. The cost is the multiplication of the parallel time complexity and the number of processors : $O(n/p) * O(p) = O(n)$ Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decomposition : the A vector decomposition is as shown below. The decomposition does **not** change during the execution:



The B and C matrices are decomposed to the processors rowwise and the decomposition does **not** change during the execution :



The communication graph : The processors do **not** communicate with each other at all during the execution. Thus, the algorithm is **embarrassingly parallel** ! There is **no communication graph** !

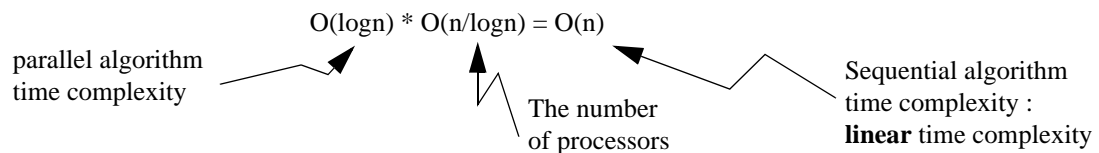
The load balance : All the processors have the same amount of work and do not wait for results from other processors (since it is embarrassingly parallel). The processors are busy all the time during the execution. Thus, the **load balance** is very good.

Q8) Develop a cost efficient DAXPY algorithm for two n -element vectors, “**X**” and “**Y**,” for a PRAM. Indicate the time complexity of your algorithm. Make observations relevant to the execution of the algorithm, including the data decomposition, load balancing, the communication graph, etc.

A8) The algorithm is shown below.

The time complexity : $O(\text{local_domain_size}) = O(n/p) = O(n/(n/\log n)) = O(\log n)$, a **logarithmic** time complexity.

The algorithm is cost efficient since



DAXPY (CREW PRAM)

Global : a, s, X[0, 1, ..., (n - 1)], Y[0, 1, ..., (n - 1)], p, local_domain_size

Local : i ; j

Begin

s = logn

p = n/(logn)

local_domain_size = n/p

SPAWN (P₁, P₂, ..., P_(p-1))

for all P_i where 0 <= i <= (p - 1) do

for j = 0 to (local_domain_size - 1) do

Y[(i * s) + j] = (a * X[(i * s) + j]) + Y[(i * s) + j]

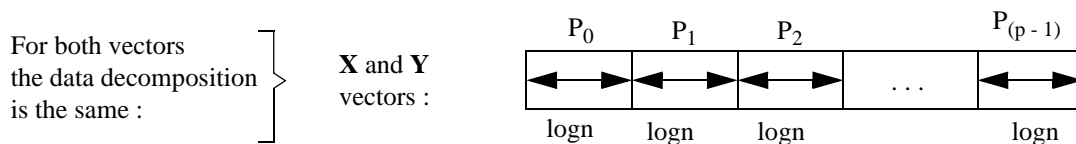
O(local_domain_size)

endfor

endfor

end

The data decomposition throughout the calculation is static and straightforward as follows :



There is no communication among the processors : an **embarrassingly parallel** algorithm ! There is **no communication graph** !

Load balancing is good since all processors are busy until the computation completes.

Q9) Consider the following sequential algorithm :

For (j = 0 ; j < n ; j++)

C[j] = 0 ;

For (r = 0 ; r < n ; r++)

C[j] = C[j] + (A[j, r] * B[j, r]) ;

Endfor

Endfor

$$\begin{array}{ccc}
 \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} & \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} & \longrightarrow (17 \ 53) \\
 \mathbf{A} & \mathbf{B} & \mathbf{C} \\
 \hline
 (1 * 5) + (2 * 6) & \longrightarrow & 5 + 12 = 17 \\
 (3 * 7) + (4 * 8) & \longrightarrow & 21 + 32 = 53
 \end{array}$$

It works on two **n x n** matrices named **A** and **B** and generates an **n**-element vector named **C**. An example with n = 2 and arbitrary values is also shown on page 2.

Develop the corresponding cost-efficient **PRAM** algorithm with “**p**” processors, named PRAMABC. **Explain** the time complexity of the PRAM algorithm.

Make observations relevant to the execution of the algorithm, including the data decomposition, load balancing, the communication graph, synchronization, etc.

A9) The PRAM algorithm is shown below.

PRAMABC (CREW PRAM)

Global : $A[0, \dots, (n-1)][0, \dots, (n-1)]$; $B[0, \dots, (n-1)][0, \dots, (n-1)]$; $C[0, \dots, (n-1)]$; n ; p ; s

Local : i ; j ; r

Begin

$s = n/p$

Spawn ($P_1, P_2, \dots, P_{(p-1)}$)

For all P_i where $0 \leq i \leq (p-1)$ do

for ($j = 0$; $j < s$; $j++$)

$C[i*s + j] = 0$

for ($r = 0$; $r \leq n - 1$; $r++$) do

$C[i*s+j] = C[i*s + j] + (A[i*s+j, r] + B[i*s+j, r])$

endfor

endfor

End

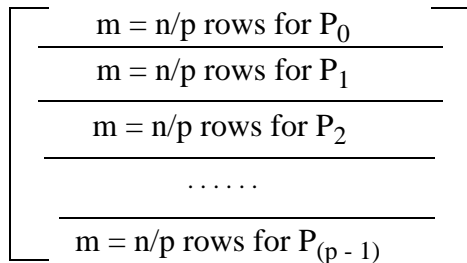
$O(n^2/p)$

The parallel time complexity is $O(n^2/p)$ as each processor executes the loop pointed by the two parallel lines n^2/p times.

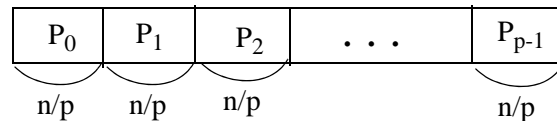
The cost : $O(n^2/p) * O(p) = O(n^2)$. The time complexity of the sequential algorithm is $O(n^2)$ since the inner loop in the algorithm is executed “n” times for each iteration of the outer loop which is executed n times. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decompositions are as shown below. The decompositions do **not** change during the execution :

The initial data decomposition of matrices A and B is rowwise :



C vector :



There is no communication among the processors : an **embarrassingly parallel** algorithm ! There is **no communication graph** !

Load balancing is good since all processors are busy until the computation completes.

Q10) Consider the following sequential algorithm :

For ($i = 0$; $i < n$; $i++$)
{ $A[i] = (C[i] * D[i]) + (B[i] + D[i]) - (B[i] / D[i])$;
$E[i] = F[i] + G[i]$; }

Develop the corresponding cost-efficient **PRAM** algorithm, named **PRAMABCDEFGF**.

Explain the time complexity of the PRAM algorithm.

Make observations relevant to the execution of the algorithm, including the data decomposition, load balancing, the communication graph, etc.

A10) The PRAM algorithm is as follows :

```

PRAMABCDEFGF (CREW PRAM)
Global : A[0,...(n-1)] ; B[0,...(n-1)] ; C[0,...(n-1)] ; D[0,...(n-1)] ; E[0,...(n-1)] ; F[0,...(n-1)] ;
        G[0,...(n-1)] ; n ; p ; s
Local : i ; j
Begin
---
    s = n/p

    Spawn (P1, P2, ..., P(p-1))

    For all Pi where 0 ≤ i ≤ (p-1) do
        for (j=0 ; j ≤ s - 1 ; j++) do
            A[i*s+j] = (C[i*s+j] * D[i*s+j]) + (B[i*s+j] + D[i*s+j]) - (B[i*s+j] / D[i*s+j])
            E[i*s+j] = F[i*s+j] + G[i*s+j] ;
        endfor
    endfor
End
    
```

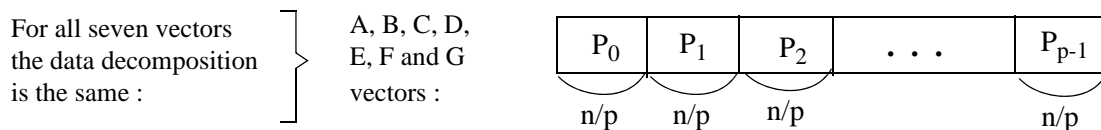
) O(n/p)

The parallel time complexity is O(n/p) since each processor executes the loop pointed by the two parallel lines n/p times.

The cost : O(n/p) * O(p) = O(n)

The time complexity of the sequential algorithm is O(n) since the loop in the algorithm is executed “n” times. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decomposition : all the A, B, C, D, E, F and G vector decompositions are as shown below. The decompositions do **not** change during the execution:



There is no communication among the processors : an **embarrassingly parallel** algorithm ! There is **no communication graph** !

Load balancing is good since all processors are busy until the computation completes.

Q11) Consider the following **sequential** algorithm :

```

For (j = 0 ; j < n ; j++)
  A[j] = (B[j] + D[j]) ;
  E[j] = 1 / F[j] ;
Endfor
c = 0 ;
For (i = 0 ; i < n ; i++)
  c = c + A[i] ;
Endfor

```

Develop the corresponding cost-efficient **PRAM** algorithm with “p” processors, named PRAMABDF.

Explain the time complexity of the PRAM algorithm.

Make observations relevant to the execution of the algorithm, including the data decomposition, load balancing, the communication graph, synchronization, etc.

A11) The PRAM algorithm is as follows :

PRAMABDF (CREW PRAM)

Global : c ; n ; p ; s ; A[0,...(n-1)] ; B[0,...(n-1)] ; D[0,...(n-1)] ; E[0,...(n-1)] ; F[0,...(n-1)] ; TEMP[0,...(p-1)]

Local : i ; j ; my_id

Begin

s = n/p

Spawn (P₁, P₂, ..., P_(p-1))

For all P_i where 0 ≤ i ≤ (p-1) do

For (j=0 ; j<s ; j++)

A[i*s + j] = B[i*s + j] + D[i*s + j]

) O(n/p)

Endfor

TEMP[i] = 0

For (j = 0 ; j < ((n/p) - 1) ; j++)

TEMP[i] = TEMP[i] + A[i*s + j]

) O(n/p)

Endfor

For (j=0 ; j<(logp) ; j++)

If imod2^j = 0 and 2i + 2^j < p then

TEMP[i] = TEMP[i] + TEMP[2i + 2^j]

) O(logp)

Endif

Endfor

If my_id = 0 then c = TEMP[0]

For (j=0 ; j<s ; j++)

E[i*s + j] = 1/F[i*s + j]

) O(n/p)

Endfor

Endfor

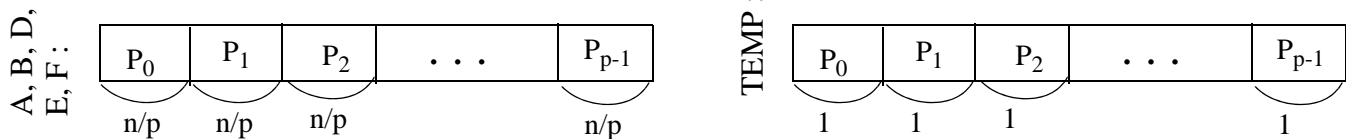
End

The parallel time complexity is O(n/p) since each processor executes three large loops n/p times.

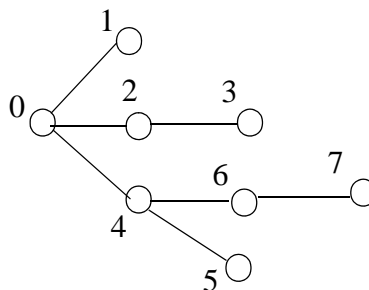
The time complexity of the sequential algorithm is O(n) since the two loops in the algorithm are executed “n” times.

The cost : O(n/p) * O(p) = O(n). The cost is equal to the sequential time complexity. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decompositions are as shown below. The decompositions do **not** change during the execution :



There is no communication among the processors in all the loops except the one with $O(\log p)$ time complexity. There the communication is tree like. For an 8-processor system it looks like as follows :



Load balancing is good except in the $O(p)$ loop where gradually most of the processors become idle.

Q12) Consider the following sequential algorithm :

```

For (i = 0 ; i < n ; i++)
  { A[i] = B[i] * C[i] ;
    D[i] = D[i] + E[i] ;
    F[i] = F[i] - G[i] ; }

```

Develop the corresponding cost-efficient **PRAM** algorithm, named **PRAMABCDEFGH**.

Explain the time complexity of the PRAM algorithm.

Make observations relevant to the execution of the algorithm, including the data decomposition, load balancing, the communication graph, etc.

A12) The PRAM algorithm is as follows :

PRAMABCDEFGH (CREW PRAM)

Global : $A[0, \dots, (n-1)] ; B[0, \dots, (n-1)] ; C[0, \dots, (n-1)] ; D[0, \dots, (n-1)] ; E[0, \dots, (n-1)] ; F[0, \dots, (n-1)] ; G[0, \dots, (n-1)] ; n ; p ; s$

Local : $i ; j$

Begin

$s = n/p$

Spawn ($P_1, P_2, \dots, P_{(p-1)}$)

For all P_i where $0 \leq i \leq (p-1)$ do

for ($j=0 ; j \leq s-1 ; j++$) do

$A[i*s+j] = B[i*s+j] * C[i*s+j]$

$D[i*s+j] = D[i*s+j] * E[i*s+j]$

$F[i*s+j] = F[i*s+j] + G[i*s+j]$;

endfor

endfor

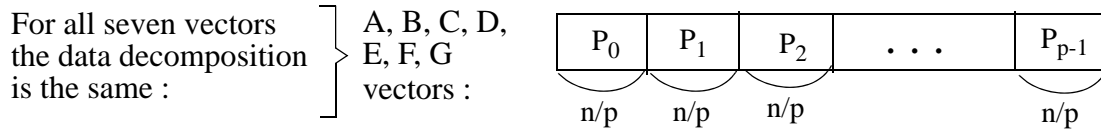
End

The parallel time complexity is $O(n/p)$ since each processor executes the loop pointed by the two parallel lines n/p times.

The cost : $O(n/p) * O(p) = O(n)$

The time complexity of the sequential algorithm is $O(n)$ since the loop in the algorithm is executed “n” times. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decomposition : all the A, B, C, D, E, F and G vector decompositions are as shown below. The decompositions do **not** change during the execution:



There is no communication among the processors : an **embarrassingly parallel** algorithm ! There is **no communication graph** !

Load balancing is good since all processors are busy until the computation completes.

Q13) Develop an algorithm, called “NORMALIZE” for a 2-d square Mesh SIMD computer with “p” processing elements. The corresponding sequential NORMALIZE algorithm is as follows :

```
for (i = 0 ; i < n ; i = i + 1)
  { if (i modulo 2 = 0 then A[i] = (A[i] + A[i - 1])/cons1
    else A[i] = (A[i])/cons1 }
```

The variable “cons1” is a constant and “n” is an even number. When “i” is zero, “i - 1” means “n - 1.” A node can send a value to another node by directly specifying the id number of the destination node :

```
send (25, var_d)
```

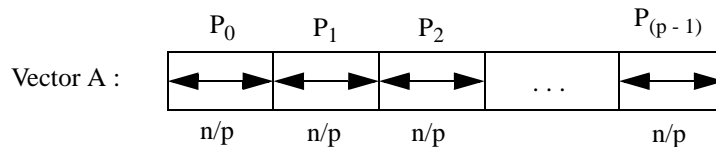
A node whose id number is contained in “myid” can send a value to another node the following way also :

```
send ((myid + 2), tmp_var)
```

- Indicate the time complexity of your SIMD algorithm.
- Make observations relevant to the execution of your SIMD algorithm, including the data decomposition, load balancing, the communication graph, etc.

A13) The code is shown below. Here, we assume that variable “s” is even and so “s - 1” is odd.

The initial data decomposition is as follows :



During the computation, nodes whose id numbers differ by one communicate in one direction to send a single data element, forming a **ring** :



NORMALIZE (2-D MESH SIMD)

```

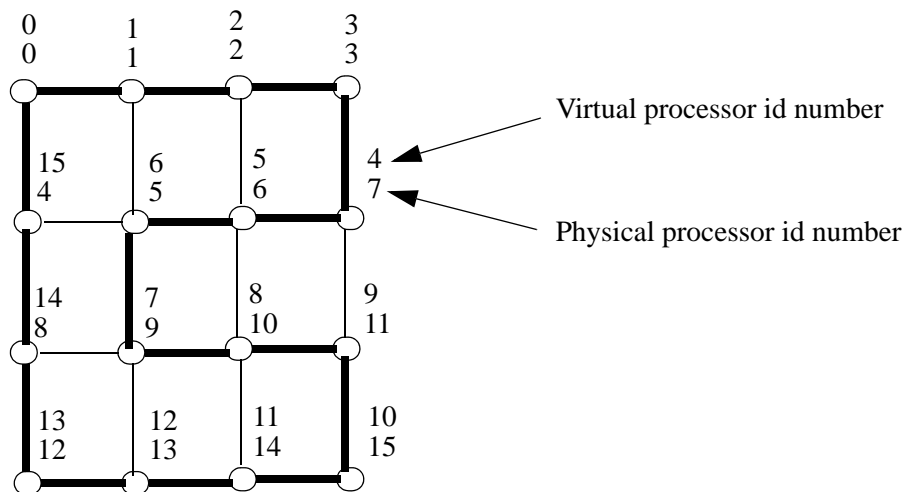
Local : myid, s, i, j ; local_A[0, 1, ..., ((n/p) - 1)], cons1, tmp
Begin
  ---
  s = n/p
  for all Pi where 0 <= i <= (p - 1) do
    if myid = (p-1) then send(0, local_A[s-1])
    else
      {
        local_A[s-1] = (local_A[s-1]/cons1)
        send((myid+1), local_A[s-1])
      }
    endif
    for (j = 1 ; j < (s - 1) ; j = j + 1) do
      if(jmod2=0) then local_A[j]=(local_A[j]+local_A[j - 1])/cons1
      else local_A[j]= (local_A[j])/cons1
    endif
  endfor
  receive (tmp)
  local_A[0]=(tmp + local_A[0])/cons1
endfor
end

```

Therefore the communication graph is a **ring** ! Any static direct interconnection network, other than the linear, tree, fat tree, perfect shuffle and star would be acceptable for the algorithm.

Since we have to use a square 2-D mesh, we have to consider how to embed the ring in the physical network. In parallel processing, there are the concepts of virtual and physical processor id numbers. A programmer would specify virtual processor id numbers in parallel programs. The mapping of virtual processor id numbers to physical processor id numbers must be done in the beginning of the computation. Systems typically have library programs to embed a regular graph to the physical interconnection network and also to map from virtual id numbers to physical id numbers. In the SIMD algorithm above, we use a system call before the “s = n/p” statement to embed the ring in the 2-d square mesh network with “p” nodes.

For the above ring graph, the embedding in the MESH for a 16-processor case is shown below :



The ring is embedded **dilation-1** in the communication network, so that near neighbor communication on the communication graph is near neighbor communication on the interconnection network.

There is a large number of elements per processor so that computations take a long time and can overlap with communication.

Load balancing is good since communication overlaps computation : every processor first sends the data element needed by the next processor on the ring then starts computing on its own elements. By the time a processor completes its computations on $((n/p) - 1)$ elements, the element sent by its ring neighbor is received. Processors complete their execution by working on the received element.

The time complexity : $O(n/p)$. This holds if computation overlaps with communication (as presumed above) or takes longer than the communication time, as the bulk of the work is done on (n/p) local vector elements.

Q14) Consider the sequential algorithm below :

```

For (j = 0 ; j < n ; j++)
  A[j] = (B[j] + D[j]) ;
  E[j] = 1 / F[j] ;
Endfor
c = 0 ;
For (i = 0 ; i < n ; i++)
  c = c + A[i] ;
Endfor
    
```

Develop the corresponding **SIMD** algorithm with “**p**” processing elements named SIMDABCDEF. The SIMD has a 2-d square mesh interconnection network.

Assume that the arrival of a message can be checked by testing a flag such as “**msg_recvd**” in the algorithm. When a message arrives the flag is set to 1 and cleared to 0 automatically after the message is read by the algorithm via a “receive()” statement.

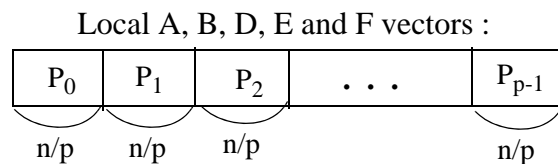
Indicate the time complexity of your algorithm. Make observations relevant to the execution of your SIMD algorithm, including the data decomposition, load balancing, the communication graph, etc. Is your algorithm cost efficient ? Explain.

A14) The SIMD algorithm is shown below.

The parallel time complexity is $O(n/p) + O(p)$ where the first component is due to computations and the second one is due to communication. The above algorithm overlaps computations and communication so that its processing elements perform computations while they also wait for messages to arrive. If the communication time is overlapped with computations, then the time complexity is $O(n)$. Then, the cost : $O(n/p) * O(p) = O(n)$

The data decompositions are as follows :

The decompositions do **not** change during the execution :



SIMDABCDEF (2-D MESH SIMD)

Local : myid, s ; i ; j ; local_A[0, 1, ..., ((n/p) - 1)] ; local_B[0, 1, ..., ((n/p) - 1)] ; local_D[0, 1, ..., ((n/p) - 1)] ; local_E[0, 1, ..., ((n/p) - 1)] ; local_F[0, 1, ..., ((n/p) - 1)] ; west, north, tmp ; i ; j ; s ; k ; l ; local_sum ; my_column ; my_row ; recvnum ; mrecv

Begin

for all P_i where $0 \leq i \leq (p - 1)$ do

 s = n/p

 l = \sqrt{p}

 recvnum = 0

 For (j=0 ; j<s ; j++)

 local_A[j] = local_B[j] + local_D[j]

) O(n/p)

 Endfor

 local_sum = 0

 For (j=0 ; j<s ; j++)

 local_sum = local_sum + local_A[j]

) O(n/p)

 Endfor

 For (j=0 ; j<s ; j++)

 If my_column = l - 1 and j = 0 then

 send(local_sum, west)

 Endif

 Else

 If (msg_rcvd and recvnum = 0)

 receive(tmp)

 recvnum = recvnum + 1

 local_sum = local_sum + tmp

 If my_column \neq 0

 send(local_sum, west)

 Endif

 Else

 If my_column = 0

 If my_row = l-1 then

 If (recvnum = 1)

 send(local_sum, north)

 Endif

 Else

 If (msg_rcvd and mrecv = 1)

 receive(tmp)

 mrecv = mrecv + 1

 local_sum = local_sum + tmp

 If my_row \neq 0 then

 send(local_sum, north) }

 Endif

 Endif

 Endif }

 Endif

 Endif

 E[j] = 1/F[j]

 Endfor

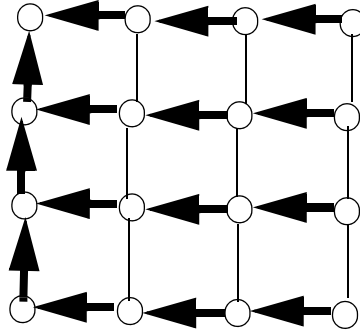
 If i = 0 then c = local_sum

Endfor

End

max(O(n/p), O(p))

There is no communication among the processors in the beginning then they communicate in a tree like fashion :



There is communication on the bold links.

The direction of communication and so the structure of the tree is shown by the arrows

Load balancing is good in the beginning since all processing elements are busy until the computation completes. During the communication sometimes the processing elements are idle.

Q15) Consider the following **sequential** algorithm :

```

k = 0
For (j = 0 ; j < n ; j++)
  { A[j] = B[j] + C[j] ;           /* S1 */
    C[j] = A[j] / D[j] - (Log10(E[j] * F[j])); /* S2 */
    k = k + A[j] + C[j] ;         /* S3 */
  }

```

Vectors A, B, C, D, E and F have “n” elements each.

- a) List the dependencies among S1, S2 and S3. Explain clearly whether there is loop-level parallelism or not.
- b) What is the time complexity of the sequential algorithm ? Explain.

A15) a) The **sequential** algorithm is given :

```

k = 0
For (j = 0 ; j < n ; j++)
  { A[j] = B[j] + C[j] ;           /* S1 */
    C[j] = A[j] / D[j] - (Log10(E[j] * F[j])); /* S2 */
    k = k + A[j] + C[j] ;         /* S3 */
  }

```

The dependencies are as follows :

- => From S1 to S2 true dependency on A[i]
- => From S1 to S2 antidependence on C[i]
- => From S1 to S3 true dependency on A[i]
- => From S2 to S3 true dependency on C[i]
- => From S3 to S3 loop-carried true dependence on k
- => From S3 to S3 loop-carried output dependence on k

There is a loop-carried true dependence and loop carried outputs dependence on k. Therefore, the code does not have loop-level parallelism.

b) For “n” elements, the loop body is executed “n” times. Thesequential time complexity is O(n), **linear** time complexity.

Q16) Consider the sequential algorithm in Question Q15 above. Develop the corresponding **cost-efficient PRAM** algorithm with “p” processors, named Q16. **Explain** the time complexity of the PRAM algorithm.

Make observations relevant to the execution of the PRAM algorithm, including the data decomposition, load balancing, the communication graph, synchronization, etc.

A16) The PRAM algorithm is as follows :

Q16 (CREW PRAM)

Global : n ; p ; s ; $A[0, \dots, (n-1)]$; $B[0, \dots, (n-1)]$; $C[0, \dots, (n-1)]$; $D[0, \dots, (n-1)]$; $E[0, \dots, (n-1)]$; $F[0, \dots, (n-1)]$; $G[0, \dots, (p-1)]$; k

Local : i ; j ; my_id

Begin

$s = n/p$

Spawn ($P_1, P_2, \dots, P_{(p-1)}$)

For all P_i where $0 \leq i \leq (p-1)$ do

$G[i] = 0$

For ($j = 1$; $j < s$; $j++$) -----
 $A[i*s + j] = B[i*s + j] + C[i*s + j]$
 $C[i*s + j] = A[i*s + j] / D[i*s + j] - (\text{Log}_{10}(E[i*s + j] * F[i*s + j]))$
 $G[i] = G[i] + A[i*s + j] + C[i*s + j]$ -----) $O(n/p)$

Endfor

For ($j=0$; $j < ((p/2) - 1)$; $j++$) -----
 If $i \bmod 2^j = 0$ and $2i + 2^j < p$ then -----
 $G[2i] = G[2i] + G[2i + 2^j]$ -----) $O(\log p)$
 Endif -----

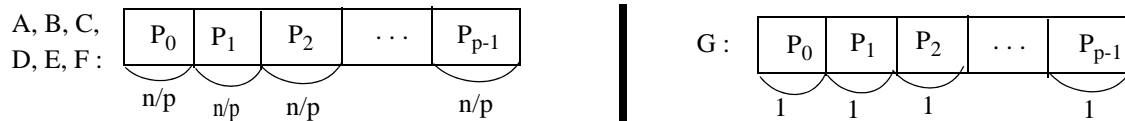
If $myid = 0$ then $k = D[0]$

Endfor

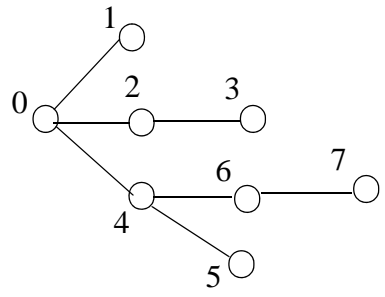
End

The parallel time complexity is $O(n/p)$ since each processor executes one large loop and contributes to another loop based on its id number no more than $\log p$ times : $O(n/p) + O(\log p) = O(n/p)$. The cost : $O(n/p) * O(p) = O(n)$. The cost is equal to the sequential time complexity. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decompositions are as shown below. The decompositions do **not** change during the execution :



There is no communication among the processors in the first the loop. But, in the second loop with $O(\log p)$ time complexity, there is communication in a tree-like fashion. For an 8-processor system, it looks like as follows :



Load balancing is good except in the $O(\log p)$ loop where gradually most of the processors become idle.

Q17) Consider again the sequential algorithm in Question 15 above. Develop the corresponding **SIMD** algorithm with “**p**” processing elements named Q17. The SIMD has a 2-d square mesh interconnection network.

Assume that the arrival of a message can be checked by testing a flag such as “**msg_recvd**” in the algorithm. When a message arrives the flag is set to 1 and cleared to 0 automatically after the message is read by the algorithm via a “**receive()**” statement as done in class.

Indicate the time complexity of your algorithm. Make observations relevant to the execution of your SIMD algorithm, including the data decomposition, load balancing, the communication graph, etc. Is your algorithm cost efficient? Explain.

A17) The SIMD algorithm is as follows :

Q17 (2-D MESH SIMD)

Local : myid, s ; i ; j ; A[0, 1, ..., ((n/p) - 1)] ; B[0, 1, ..., ((n/p) - 1)] ; C[0, 1, ..., ((n/p) - 1)] ; D[0, 1, ..., ((n/p) - 1)] ; E[0, 1, ..., ((n/p) - 1)] ; F[0, 1, ..., ((n/p) - 1)] ; Local_k ; tmp ; i ; j ; s ; my_row ; my_column

Begin

```

for all Pi where 0 <= i <= (p - 1) do
  s = n/p
  For (j = 1 ; j < s ; j++)
    A[i*s + j] = B[i*s + j] + C[i*s + j]
    C[i*s + j] = A[i*s + j] / D[i*s + j] - (Log10(E[i*s + j] * F[i*s + j]))
    Local_k = Local_k + A[i*s + j] + C[i*s + j]
  Endfor
  If my_column = 1-1 then
    If msg_sent = 0 then send(local_k, west)
  Else
    receive(tmp)
    local_k = local_k + tmp
    If my_column ≠ 0
      send(local_k, west)
    Endif
    If my_column = 0
      If my_row = 1-1 then
        send(local_k, north)
      Else
        receive(tmp)
        local_k = local_k + tmp
        If my_row ≠ 0 then
          send(local_k, north)
        }
      Endif
    Endif
  Endif
  If i = 0 then k = local_k
Endfor

```

O(n/p)

O(p^{1/2})

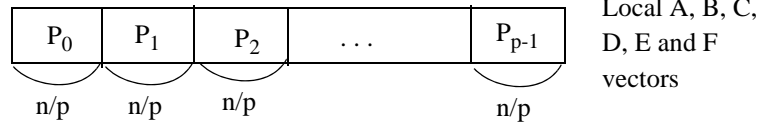
End

The parallel time complexity is $O(n/p)$, even though the communication time in the $O(p^{1/2})$ can be considerable. The algorithm cannot overlap computations and communication because of lack of loop-level parallelism. But, since addition is associative, we extract parallelism in a tree like fashion, with reduced parallelism as we move up the tree.

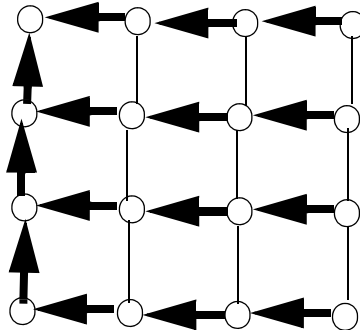
The cost : $O(n/p) * O(p) = O(n)$. The cost is equal to the sequential time complexity. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decomposition is as follows :

The decompositions do **not** change during the execution :



There is no communication among the processors in the first loop. Then they communicate in a tree like fashion :



There is communication on the bold links.

The direction of communication and so the structure of the tree is shown by the arrows

Load balance is good in the first loop. But, during communication the load balance gets worse.

Q18) Consider the following **sequential** algorithm :

```

For (j = 1 ; j < n ; j++)
{
    K[j] = K[j] + K[j - 1] ;      /* S1 */
    A[j] = K[j] * B[j] + C[j] ;  /* S2 */
    D[j] = E[j] * B[j] - C[j] ;  /* S3 */
}
    
```

Vectors K, A, B, C, D and E have “n” elements each, but only last “n-1” elements are changed.

Develop the corresponding **SIMD** algorithm with “p” processing elements named Q18. The SIMD has a 2-d square mesh interconnection network.

Assume that the arrival of a message can be checked by testing a flag such as “**msg_recvd**” in the algorithm. When a message arrives the flag is set to 1 and cleared to 0 automatically after the message is read by the algorithm via a “receive()” statement as done in class.

Indicate the time complexity of your algorithm. Make observations relevant to the execution of your SIMD algorithm, including the data decomposition, load balancing, the communication graph, etc. Is your algorithm cost efficient ? Explain.

A18) The SIMD algorithm is as follows :

Q18 (2-D MESH SIMD)

Local : myid, s ; i ; j ; A[0, 1, ..., ((n/p) - 1)] ; B[0, 1, ..., ((n/p) - 1)] ; C[0, 1, ..., ((n/p) - 1)] ; D[0, 1, ..., ((n/p) - 1)] ; E[0, 1, ..., ((n/p) - 1)] ; K[0, 1, ..., ((n/p) - 1)] ; temp ; resultsent ; east ; i ; j ; s ; j

Begin

for all P_i where $0 \leq i \leq (p - 1)$ do

 s = n/p

 For (j=1 ; j<s ; j++)) $O(n/p)$

 K[j] = K[j] + K[j - 1]

 Endfor

 If myid = 0 then

 send (K[s - 1], mylineararrayneighbor)

 resultsent = 1

 Endif

 For (j = 0 ; j < s ; j++).....

 If resultsent = 0 and msg_rcvd then receive (temp)

 K[s - 1] = K[s - 1] + temp

 For (j = 0 ; j < s ; j++)

 If j = 0 then K[j] = K[j] + temp

 else K[j] = K[j] + K[j - 1]

 Endif

 Endfor) $O(n/p)$

 If myid < (p-1) then

 send(K[s - 1], mylineararrayneighbor)

 resultsent = 1

 Endif

 Endif

 If (myid = 0 and j = 1) or (myid > 0 and j \geq 0) then D[j] = E[j] * B[j] - C[j]

 Endfor) $O(n)$

 While resultsent = 0 and myid < (p - 1) then.....

 If msg_rcvd then receive (tmp)

 K[s - 1] = K[s - 1] + tmp

 For (j = 0 ; j < s ; j++)

 If j = 0 then K[j] = K[j] + temp

 else K[j] = K[j] + K[j - 1]

 Endif

 Endfor

 If myid < (p-1) then

 send(K[s - 1], mylineararrayneighbor)

 resultsent = 1

 Endif

 Endif

 Endwhile) $O(?)$

 For (j = 0 ; j < s ; j++).....

 If (myid = 0 and j = 1) or (myid > 0 and j \geq 0) then A[j] = K[j] * B[j] + C[j]

 Endfor) $O(n/p)$

Endfor

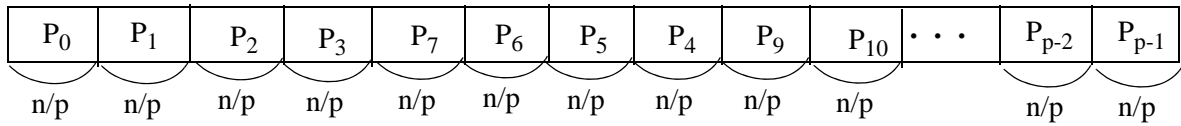
End

The parallel time complexity is $O(n)$ since there is no loop-level parallelism. The algorithm overlaps computations and commuincation so that processing elements perform computations while they also wait for messages to arrive. But, depending on the interconnection network, the overlapping may not be sufficient, then PEs would wait additional time whose complexity depends on the interconnection network and is shown as $O(?)$ below.

The cost is $(n)*O(p) = O(np)$. This is not equal to the sequential time complexity and so the algorithm is **not** cost efficient.

The data decomposition is such that groups of “s” elements are assigned to the PEs based on their position in the mesh network :

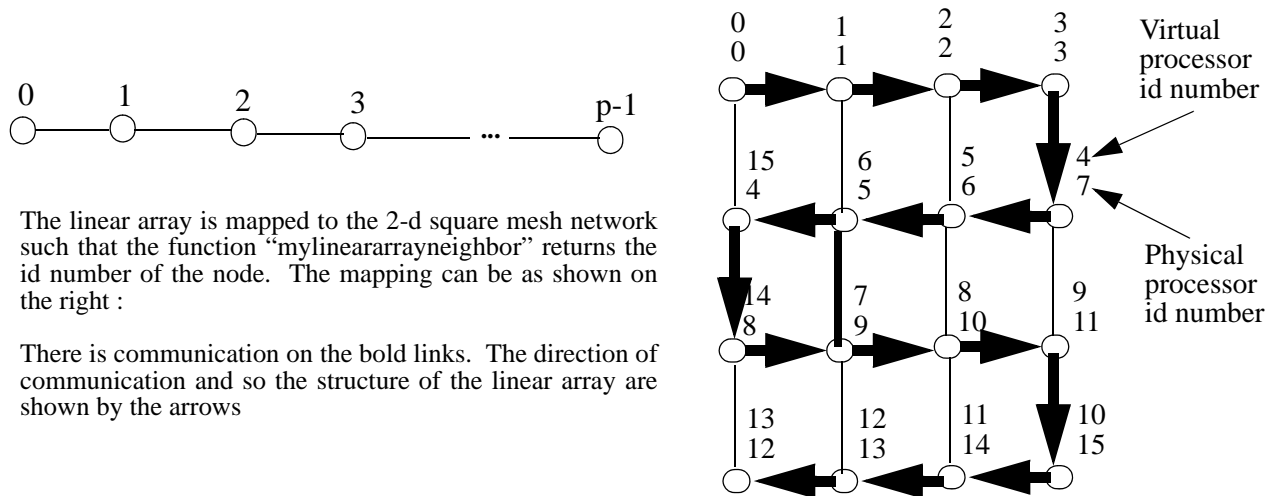
Local A, B, C, D, E and K vectors for a 4x4 mesh network :



The decompositions above do not change during the execution, except for vector K. For vector K, in the middle loop, the last element of each domain vector K for PEs from 0 to (p-2) is used by PEs from 1 to (p-1).

In addition, the assignment of the domains to the PEs is not as done before. The assignment follows the position of each PE on the 2-d square mesh network. Above, the assignment is given for a 4x4 mesh network.

There is communication among the processors to pass new values of K. The processors communicate in a linear array fashion. Since we have to use a square 2-D mesh, we have to consider how to embed the linear array in the physical network. We will use virtual processor id numbers. The programmer specifies virtual processor id numbers. The mapping of virtual processor id numbers to physical processor id numbers is done in the beginning of the computation. Library programs embed the linear array to the 2-D square mesh network and also map from virtual id numbers to physical id numbers. In the SIMD algorithm above, we use a system call before the “s = n/p” statement to embed the linear array in the 2-d square mesh network with “p” nodes.



The linear array is mapped to the 2-d square mesh network such that the function “mylineararrayneighbor” returns the id number of the node. The mapping can be as shown on the right :

There is communication on the bold links. The direction of communication and so the structure of the linear array are shown by the arrows

Load balance is good since processing elements are busy until the computation completes, with the exception that in the while loop they are idle if the interconnection network takes a long time.

Q19) Consider the following **sequential** algorithm :

```

For (j = 0 ; j < n ; j++)
{ A[j] = A[j] + e                               /* S1 */
  If (A[0] ≠ 0 & j ≠ 0) then A[j] = A[j] + A[0] /* S2 */
  B[j] = A[j] * B[j] * f                         /* S3 */
  C[j] = D[j] * g                                 /* S4 */
}

```

Vectors A, B, C and D have “n” elements each.

- List the **data** dependencies among S1, S2, S3 and S4. Explain clearly whether there is loop-level parallelism or not.
- What is the time complexity of the sequential algorithm ? Explain.
- Develop the corresponding cost-efficient **PRAM** algorithm with “p” processors, named Q3. **Explain** the time complexity of the PRAM algorithm.

Make observations relevant to the execution of the PRAM algorithm, including the data decomposition, load balancing, the communication graph, etc.

A19) The **sequential** algorithm is given :

```

For (j = 0 ; j < n ; j++)
{ A[j] = A[j] + e
  If (A[0] ≠ 0 & j ≠ 0) then A[j] = A[j] + A[0]
  B[j] = A[j] * B[j] * f
  C[j] = D[j] * g
}

```

| | |
|----------|---|
| /* S1 */ | a) The dependencies are as follows : => From S1 to S2 true dependency on A[i] => From S1 to S2 output dependency on A[i] => From S1 to S2 antidependence on A[i] => From S1 to S3 true dependence on A[i] => From S2 to S3 true dependence on A[i] => From S1 to S2 loop-carried true dependence on A[i] |
| /* S2 */ | |
| /* S3 */ | |
| /* S4 */ | |
| /* S4 */ | |

There is no loop-level parallelism due to the loop-carried dependency.

- For “n” elements, the loop body is executed “n” times. The sequential time complexity is O(n), **linear** time complexity.
- The PRAM algorithm is below.

The parallel time complexity is O(n/p) since each processor executes the loop. Processor 0 also executes the first computation which is negligible. Therefore, the time complexity is : O(n/p).

The cost : O(n/p) * O(p) = O(n).

The cost is equal to the sequential time complexity. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

Q3 (CREW PRAM)

Global : $n ; p ; s ; A[0, \dots, (n-1)] ; B[0, \dots, (n-1)] ; C[0, \dots, (n-1)] ; D[0, \dots, (n-1)] ; e ; f ; g$

Local : $i ; j ; my_id$

Begin

$s = n/p$

$A[0] = A[0] + e$

Spawn ($P_1, P_2, \dots, P_{(p-1)}$)

For all P_i where $0 \leq i \leq (p-1)$ do

For ($j = 0 ; j < s ; j++$)

If $my_id \neq 0$ and $j \neq 0$ then $A[i*s + j] = A[i*s + j] + e + A[0]$

$B[j] = A[j] * B[j] * f$

$C[i*s + j] = D[i*s + j] * g$

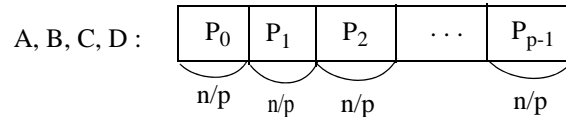
Endfor

Endfor

End

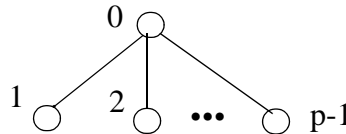
$O(n/p)$

The data decompositions are as shown below. The decompositions do **not** change during the execution :



Load balancing is good. In the loop, processor 0 is less busy than the other processors for only one iteration which is negligible.

There is communication in the loop whose graph is a tree :



Q20) Consider again the sequential algorithm in Question Q19. Develop the corresponding **SIMD** algorithm with “ p ” processing elements named QSIMD. The SIMD has a 2-d square mesh interconnection network.

Assume that the arrival of a message can be checked by testing a flag such as “**msg_rcvcd**” in the algorithm. When a message arrives the flag is set to 1 and cleared to 0 automatically after the message is read by the algorithm via a “receive()” statement as done in class.

Indicate the time complexity of your algorithm. Make observations relevant to the execution of your SIMD algorithm, including the data decomposition, load balancing, the communication graph, etc. Is your algorithm cost efficient ? Explain.

A20) The SIMD algorithm is as follows :

QSIMD (2-D MESH SIMD)

Local : my_id, s ; i ; j ; A[0, 1, ..., ((n/p) - 1] ; B[0, 1, ..., ((n/p) - 1)] ; C[0, 1, ..., ((n/p) - 1)] ; D[0, 1, ..., ((n/p) - 1)] ; e ; f ; g ; tmp ; resultreceived

Begin

for all P_i where $0 \leq i \leq (p - 1)$ do

 s = n/p

 For (j = 0 ; j < s ; j++)

 A[j] = A[j] + e

 If my_id = 0 and (j = 0) then

 send (A[j], mylineararrayneighbor)

 Endif

 C[j] = D[j] * g

 If my_id \neq 0 & msg_rcvd then

 receive (tmp)

 resultreceived = 1

 if my_id \neq (lastprocessorinlineararray) then

 send (tmp, mylineararrayneighbor)

 Endfor

 If my_id \neq 0 & resultreceived = 0 then

 while (msg_rcvd = 0)

 receive (tmp)

 resultreceived = 1

 if my_id \neq (lastprocessorinlineararray) then

 send (tmp, mylineararrayneighbor)

 Endif

 Endif

 For (j=0 ; j < s ; j++)

 If my_id \neq 0 and (j \neq 0) then If temp \neq 0) then A[j] = A[j] + temp

 B[j] = A[j] * B[j] * f

 Endfor

Endfor

End

O(n/p)

O(p)

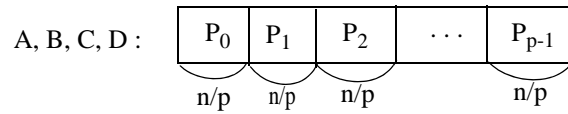
O(n/p)

Each processor, except processor 0, executes all three loops. In the first loop, processor 0 sends its A[0] value to its linear array neighbor. The other processors perform calculations on "C" and at the same time check if they received A[0]. If yes, they send it to their linear array neighbor. This way, communication is overlapped with calculations. Processor 0 also calculates "C" elements in the first loop. If by the end of the first loop a processor, other than processor 0, has not received A[0], it waits until it receives it and sends it to its linear array neighbor if its id number is not lastprocessorinlineararray. The waiting time is highly interconnection network dependent. In the third loop, all processors are busy. Then, the parallel time complexity is $O(n/p) O(n/p) + O(p) + O(n/p) = O(n/p)$.

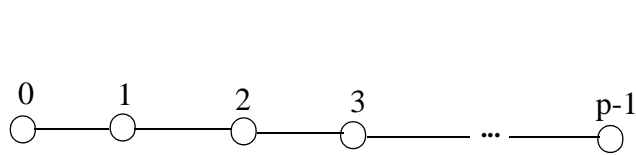
The cost : $O(n/p) * O(p) = O(n)$.

The cost is equal to the sequential time complexity. Since the cost is equal to the sequential time complexity, the algorithm is cost efficient.

The data decompositions are as shown below. The decompositions do **not** change during the execution :

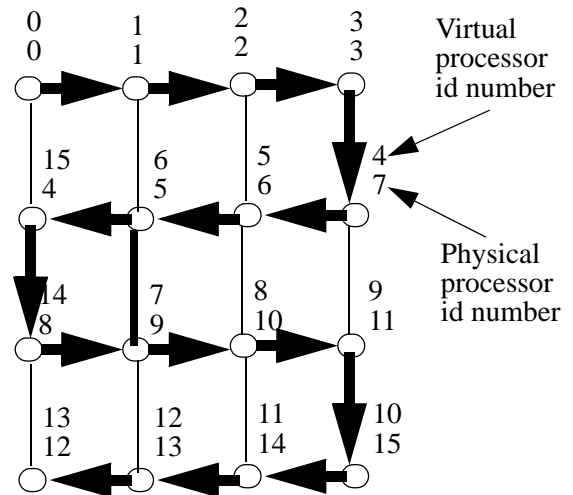


There is communication among the processors to pass $A[0]$. The processors communicate in a linear array fashion. Since we have to use a square 2-D mesh, we have to consider how to embed the linear array in the physical network. We will use virtual processor id numbers. The programmer specifies virtual processor id numbers. The mapping of virtual processor id numbers to physical processor id numbers is done in the beginning of the computation. Library programs embed the linear array to the 2-D square mesh network and also map from virtual id numbers to physical id numbers. In the SIMD algorithm above, we use a system call before the “ $s = n/p$ ” statement to embed the linear array in the 2-d square mesh network with “ p ” nodes.



The linear array is mapped to the 2-d square mesh network such that the function “mylineararrayneighbor” returns the id number of the node. The mapping can be as shown on the right :

There is communication on the bold links. The direction of communication and so the structure of the linear array are shown by the arrows



Load balance is good in the first and third loops since processing elements are busy until the loops complete. In the second (while) loop some processors are idle if they have not received $A[0]$. The waiting time depends on the inter-connection network.