Universitat Politècnica de Catalunya

AMPP Final Project Report

Parallelization of Smith-Waterman Algorithm

Author:
Iuliia Proskurnia
Arinto Murdopo
Muhammad Anis uddin Nasir

Supervisor:
Josep R. Herrero
Dani Jimenez-Gonzalez

January 16, 2012

Contents

List of Figures

1 Introduction

The Smith–Waterman algorithm is a well-known algorithm for performing local sequence alignment for determining similar regions between two nucleotide or protein sequences. Proteins are made by aminoacid sequences and similar protein structure has similar aminoacid sequence.In this project we did the parallel implementation of the Smith-Waterman Algorithm using Message Passing Interface code.

To compare two aminoacid sequence, initially we have to align the sequences to compare them. To find the best alignment between two sequences the algorithm initially populates a matrix H of size $N \times N$ (N is size of sequence) using a scoring criteria. It requires a scoring matrix (cost of matching of two symbols) and a gap penalty for mismatch of two symbols. After populating the matrix H we can obtain the optimum local alignment by tracking back the matrix starting with the highest value in the matrix.

In our implementation of Smith-Waterman algorithm we populated the matrix H in parallel using multiple processes running of multicore machines. We used pipelined computation to achieve specific degree of parrallelism and compared different parallelizing techniques to find optimum parallelization technique for the problem. We started parallelizing our code using different blocking sizes B at the column level. Furthermore, we also introduced parallelization using different levels of interleave I at the row level.

For performance measurement we created the performance model of both the implementations for two interconnection networks which are *linear* and *2D-Mesh* interconnection network. We executed our code for evaluation on Altix machine using different values of parameter Δ (gap penalty), B (column interleaving factor) and I (row interleaving factor) to empirically find optimum B and I for the problem. We also calculated the optimum B and I by finding the global minima of the equations of the performance model.

2 Main Issues and Solutions

2.1 Available Parallelization Techniques

We can achieve pipelining with both blocking at column and row level. Blocking at column level can be interpreted in different ways.

1. Each processor P_i processes B complete columns of the matrix before doing any communication.

2. Each processor P_i processes B complete columns. However after processing B columns of a row of the matrix it does a communication to next processor.

3. Each processor P_i processes B complete columns. However after processing B columns of a set of rows of the complete B columns of the matrix it does a communication.

4. Each processor P_i processes N/P complete rows. After processing B columns of those N/P rows, it does a communication.

Among above mentioned techniques, we choosed the last one because it provides us with most optimum pipelined computation using the scheme.

2.2 Blocking Technique

2.2.1 Solution 1: Using Scatter and Gather

Based on chosen technique from our available parallelization techniques, we developed this following solution. Note that in our solution here we already incorporated I (Interleave factor), but we set the I to 1.

At the first step, process with rank 0 (which is the master process) reads all the necessary files which are two protein sequence files. The reading result is stored in **short* a** and **short* b**. Other than that, it also allocates enough memory to store the resulting matrix as shown in code snippet below

```
1 {
2     //note that sizeA is the total number of rows that we need
          process. We round up N if N is not divisible by
          total_processes as shown below. I is set to 1 here.
3
4     if (N % (total_processes * I) != 0) {
5         sizeA = N + (total_processes * I) - (N % (
              total_processes * I));//to handle case where N is not
                divisible by (total_processes * I)
6         } else {
7             sizeA = N;
8         }
9
```

```
10          read_files(in1, in2, a, b, N - 1);//in1 = input file 1,
                in2 = input file 2, a = resulting reading from in1, b
                 = resulting reading from in2
11          chunk_size = sizeA / (total_processes * I); //number of
                rows that each processes needs to work on
12          CHECK_NULL((h_all_ptr = (int *) calloc(N * (sizeA+1),
                sizeof(int)))); //resulting data
13          CHECK_NULL((h_all = (int **) calloc((sizeA+1), sizeof(
                int*)))); //contain list of pointer
14
15          for(i = 0; i <= sizeA; i++)
16              h_all[i] = h_all_ptr + i * N;//put the pointer in an
                    array
17
18          //initialize the first row of resulting matrix with 0
19          for(i = 0; i < N; i++)
20          {
21                  h_all[0][i] = 0;
22          }
23
24  }
```

Every process reads the PAM matrix, and master process performs broadcast of N and B value.

```
1       MPI_Bcast(&chunk_size, 1, MPI_INT, 0, MPI_COMM_WORLD); //
            Broadcast chunk size, which is the number of calculated
            rows by each slave
2       MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD); //Broadcast N
3       MPI_Bcast(&B, 1, MPI_INT, 0, MPI_COMM_WORLD); //Broadcast B
4       MPI_Bcast(&I, 1, MPI_INT, 0, MPI_COMM_WORLD); //Broadcast I
```

Then each process needs to allocate enough memory to receive **chunk_size**. Other than process with rank 0, they need to allocate memory to receive the whole part of protein 2 (which has size equals to **N**).

```
1       CHECK_NULL((chunk_a = (short *) calloc(sizeof(short),
            chunk_size));//slave process will obtain it from master
            process
2       if (rank != 0) {
3           CHECK_NULL((b = (short *) malloc(sizeof(short) * (N))));
                //slave process will obtain it from master process
4       }
5
6       MPI_Bcast(b, N, MPI_SHORT, 0, MPI_COMM_WORLD); //broadcast
            protein 2 to every process
```

Now, let's go to the parallel part, first we calculate how many blocks that we will process. We calculate, **total_blocks** variable and also **last_block** variable. **last_block** variable contains the size of the last block to process if N is not divisible by B (N %B != 0)

```
1       int total_blocks = N / B + (N % B == 0 ? 0 : 1);
2       int last_block = N % B == 0 ? B : N % B;
```

Then we scatter 1st protein sequence(in here we store it in a), with size of each scattered part equals to **chunk_size**. After each process receives each scattered part, the computation begins for process with rank 0. It will not wait to receive any data from other process and directly calculate the 1st block of data. Meanwhile other proces with rank **r**, will wait for data from process with rank **r-1**. The data sent between process here is the last row of calculated block (which is an array of **short** with size equals to B.

After a process receive the required data, each process performs computations for received data. In the end, each process with rank **r** will send **the last row of calculated block with size B** to neighboring process with rank **r+1**.

In the end, we perform gather to combine the result. Note that **current_interleave** variable is set to 0 and **I** is set to 1 here because we're not using interleaving factor. Code snippet below show how to implement this functionality

```
1  for (int current_interleave = 0; current_interleave < I;
       current_interleave++) {
2          MPI_Scatter(a + current_interleave * chunk_size *
               total_processes,
3              chunk_size, MPI_SHORT, chunk_a, chunk_size,
                   MPI_SHORT, 0, MPI_COMM_WORLD);//chunk_a is the
                   receiving buffer
4          int current_column = 1;
5          for (i = 0; i < chunk_size + 1; i++) h[i][0] = 0;
6
7          for (int current_block = 0; current_block < total_blocks
               ; current_block++) {
8              // Receive
9              int block_end = MIN2(current_column - (current_block
                   == 0 ? 1 : 0) + B, N);
10             if (rank == 0 && current_interleave == 0) { //if
                   rank 0 is processing the first block, it doesn't
                   need to receive any thing
11                 for (int k = current_column; k < block_end; k++)
                       {
12                     h[0][k] = 0;//init row 0
13                 }
14             } else {
15                 int receive_from = rank == 0 ? total_processes -
                       1 : rank - 1;//receive from neighboring
                       process
16                 int size_to_receive = current_block ==
                       total_blocks - 1 ? last_block : B;
17                 MPI_Recv(h[0] + current_block * B,
                       size_to_receive, MPI_INT, receive_from, 0,
                       MPI_COMM_WORLD, &status);
18             }
19             // Process
20             for (j = current_column; j < block_end; j++,
```

```
                current_column++) {
21              for (i = 1; i < chunk_size + 1; i++) {
22                              diag = h[i-1][j-1] + sim[chunk_a
                                    [i - 1]][b[j - 1]];
23                              down = h[i-1][j] + DELTA;
24                              right = h[i][j-1] + DELTA;
25                              max = MAX3(diag,down,right);
26                              if (max <= 0)  {
27                                      h[i][j] = 0;
28                              } else {
29                                      h[i][j] = max;
30                              }
31                      }
32              }
33
34              // Send
35              if (current_interleave + 1 != I || rank + 1 !=
                    total_processes) {
36                  int send_to = rank + 1 == total_processes ? 0 :
                        rank + 1;
37                  int size_to_send = current_block == total_blocks
                         - 1 ? last_block : B;
38                  MPI_Send(h[chunk_size] + current_block * B,
                        size_to_send, MPI_INT, send_to, 0,
                        MPI_COMM_WORLD);
39                  print_vector(h[chunk_size] + current_block * B,
                         size_to_send);
40              }
41
42              // Gathering result
43              MPI_Gather(hptr + N, N * chunk_size, MPI_INT,
44              h_all_ptr + N + current_interleave * chunk_size *
                    total_processes * N,
45              N * chunk_size, MPI_INT, 0, MPI_COMM_WORLD);
46      }
```

Once the result is gathered, process with rank 0 deallocates the memory and perform optional verification result. The verification result is obtained by comparing the resulting parallel version of h matrix (by using *h_all*) with serial version of h matrix (by using *hverify*)

```
1   if (rank == 0) {
2       if(verifyResult == 1){
3           Max = 0;
4           xMax = 0;
5           yMax = 0;
6           CHECK_NULL((hverifyptr = (int *) malloc(sizeof(int)
                *(N+1)*(N+1))));
7           CHECK_NULL((hverify = (int **) malloc(sizeof(int*)*(
                N+1))));
8           /* Mount hverify[N][N] */
9           for(i=0;i<=N;i++)
10              hverify[i]=hverifyptr+i*(N+1);
11          for (i=0;i<=N;i++) hverify[i][0]=0;
```

```
12              for (j=0;j<=N;j++) hverify[0][j]=0;
13
14              for (i=1;i<=N;i++)
15                  for (j=1;j<=N;j++) {
16                      diag    = hverify[i-1][j-1] + sim[a[i-1]][b[
                            j-1]];
17                      down    = hverify[i-1][j] + DELTA;
18                      right   = hverify[i][j-1] + DELTA;
19                      max=MAX3(diag,down,right);
20                      if (max <= 0)  {
21                          hverify[i][j]=0;
22                          }
23                      else if (max == diag) {
24                          hverify[i][j]=diag;
25                          }
26                      else if (max == down) {
27                          hverify[i][j]=down;
28                          }
29                      else {
30                          hverify[i][j]=right;
31                          }
32                      if (max > Max){
33                          Max=max;
34                          xMax=i;
35                          yMax=j;
36                          }
37                  }
38
39                  int verFailFlag = 0;
40                  for(i=0;i<=N-1;i++){
41                      for(j=0;j<=N-1;j++){
42                          if(h_all[i][j] != hverify[i][j]){
43                              printf("Verification fail!\n");
44                              printf("h_all[i][j] = %d, hverify[i
                                    ][j] = %d\n", h_all[i][j],
                                    hverify[i][j]);
45                              verFailFlag = -1;
46                              break;
47                             }
48                          }
49
50                      if(verFailFlag != 0){
51                          break;
52                          }
53                  }
54
55                  if(verFailFlag ==0)
56                  {
57                      printf("Verification success!\n");
58                  }
59
60          }
61
62          free(hverifyptr);
```

```
63          free(hverify);
64          free(a);
65          free(h_all_ptr);
66          free(h_all);
67      }
68
69      free(b);
70      free(chunk_a);
71      free(h);
72      free(hptr);
73
74      MPI_Finalize();
```

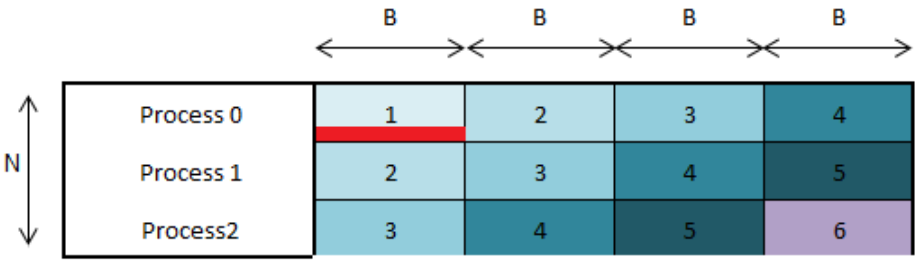

Figure 1: Blocking Communication

To summarize this technique, Figure 1 shows the dividing of block in a matrix. The number inside the block indicates the step. The red portion in block 1 indicate the amount of data (which is B integers) that is sent from process 0 to process 1 in the end of calculation of block 1, in step 1.

2.2.2 Solution 1: Linear-array Model

First, we use linear-array topology to model our solution. Here is the model for communication part of our chosen blocking technique

1. Broadcasting **chunk_size, N, B, and I**

   $t_{comm-bcast-4-int} = 4 \times (t_s + t_w) \times \log_2(p)$

2. Broadcasting of 2nd protein sequence (vector **b**)

   $t_{comm-bcast-protein-seq} = (t_s + t_w \times N) \times \log_2(p)$

3. Scattering **chunk_size** for each process to compute

   Note that the size of chunk_size is the following

   $chunk\_size = \frac{N}{p}$

   Therefore communication time for scattering is shown below

   $t_{comm-scatter-protein-seq} = t_s \times \log_2(p) + t_w \times \frac{N}{p} \times (p-1)$

4. Sending shared data

   To start the first block of computation, process with rank 0 does not need to wait for any data from other processes. That means we only have $(\frac{N}{B}+p-2)$ stages for sending shared data. The shared data is the last row of current finished block which consists of B items.Therefore putting all of them together, communication time to send shared data is

   $t_{comm-send-shared-data} = (\frac{N}{B} + p - 2) \times (t_s + (B \times t_w))$

5. Gathering calculated data

   Finally, we need to perform gather to combine all calculated data. Note that every process will need to combine $N \times chunk\_size$ data, which equals to $N \times \frac{N}{P}$ amount of data. Therefore the communication time for this step is given by

   $t_{comm-gather} = t_s \times \log_2(p) + t_w \times \frac{N}{p} \times N \times (p-1)$

6. Putting all the communication time together

   $t_{comm-all} = t_{comm-bcast-4-int}+t_{comm-bcast-protein-seq}+t_{comm-scatter-protein-seq}+ t_{comm-send-shared-data} + t_{comm-gather}$

   $t_{comm-all}(B) = (6\log_2(p)+p-1)t_s+((4+N)\log_2(p)+N+\frac{(N+N^2)(p-1)}{p})t_w+ \frac{Nt_s}{B} + (p-2) \times B \times t_w$

Now we calculate the calculation time for this blocking technique. Note that in our blocking technique we have $\frac{N}{B}+p-1$ stages of block-calculation. In each block-calculation, we need to compute $\frac{N}{p} \times B$ points. Therefore, if we represent time to compute one point as t_c, we obtain this following calculation time model

$t_{calc} = (\frac{N}{B} + p - 1) \times (\frac{N}{p} \times B) \times t_c$
$t_{calc} = (\frac{N^2}{p} + NB - \frac{NB}{p}) \times t_c$
$t_{calc} = (\frac{N^2}{p} + (\frac{N \times (p-1)}{p}) \times B) \times t_c$

Final model can be obtained by adding calculation time and communication time

$t_{total} = t_{comm} + t_{calc}$
$t_{total}(B) = (6\log_2(p)+p-1)t_s+((4+N)\log_2(p)+N+\frac{(N+N^2)(p-1)}{p})t_w+ \frac{Nt_s}{B} + (p-2) \times B \times t_w + (\frac{N^2}{p} + (\frac{N \times (p-1)}{p}) \times B) \times t_c$

2.2.3 Solution 1: Optimum B for Linear-array Model

To find optimum B for linear array model, we need to calculate derivative of final model of the linear topology with respect to B , and set the derivative to 0 as shown below

$$\frac{dt_{total}(B)}{dB} = 0$$

And, using obtained model from section 2.2.2 we obtain this following equation

$$\frac{-N}{B^2}t_s + (p-2)t_w + \frac{N(p-1)}{p} \times t_c = 0$$

$$(p-2)t_w + \frac{N(p-1)}{p} \times t_c = \frac{N}{B^2}t_s$$

$$B^2 = \frac{Nt_s}{(p-2)t_w + \frac{N(p-1)}{p} \times t_c}$$

$$B^2 = \frac{pNt_s}{p(p-2)t_w + N(p-1) \times t_c}$$

$$B = \sqrt{\frac{pNt_s}{p(p-2)t_w + N(p-1) \times t_c}}$$

Using assumption that P is very small in comparison with N, we simplify the equation above into this following

$$B \approx \sqrt{\frac{t_s}{t_c}}$$

2.2.4 Solution 1: 2-D Mesh Model

Using the same steps as in section 2.2.2, here is the 2-D Mesh Model of solution 1.

1. Broadcasting **chunk_size, N, B, and I**

   $t_{comm-bcast-4-int} = 4 \times 2 \times (t_s + t_w) \times \log_2(\sqrt{p})$

2. Broadcasting of 2nd protein sequence (vector **b**)

   $t_{comm-bcast-protein-seq} = 2 \times (t_s + t_w \times N) \times \log_2(\sqrt{p})$

3. Scattering **chunk_size** for each process to compute

   Note that the size of chunk_size is the following

   $chunk\_size = \frac{N}{p}$

Communication time for scattering in 2-D Mesh model can be modeled using hypercube. It is similar as the communication time for scattering in Linear Array model.[1]

$t_{comm-scatter-protein-seq} = t_s \times \log_2(p) + t_w \times \frac{N}{p} \times (p-1)$

4. Sending shared data
   Since sending shared data is using primitive send and receive, the communication time for this part in 2 D mesh model also does not change.

   $t_{comm-send-shared-data} = (\frac{N}{B} + p - 2) \times (t_s + (B \times t_w))$

5. Gathering calculated data

   Communication time for gathering is using same formula as scattering, but different size of data that is gathered.

   $t_{comm-gather} = t_s \times \log_2(p) + t_w \times \frac{N}{p} \times N \times (p-1)$

6. Putting all the communication time together

   $t_{comm-all} = t_{comm-bcast-4-int} + t_{comm-bcast-protein-seq} + t_{comm-scatter-protein-seq} + t_{comm-send-shared-data} + t_{comm-gather}$

   $t_{comm-all}(B) = ((10\log_2(\sqrt{p}) + \log_2(p) + p - 1) \times t_s + ((8+2N)\log_2(\sqrt{p}) + \frac{N\times(p-1)}{p} + N + \frac{N^2\times(p-1)}{p}) \times t_w + \frac{N}{B} \times t_s + (p-2)B \times t_w$

Calculation time does not change between 2-D mesh model and Linear Array model, therefore the calculation time is

$t_{calc} = (\frac{N^2}{p} + (\frac{N\times(p-1)}{p}) \times B) \times t_c$

Putting all together

$t_{total} = t_{comm} + t_{calc}$

$t_{total}(B) = \frac{N^2}{p} \times t_c + (10\log_2(\sqrt{p}) + \log_2(p) + p - 1) \times t_s + ((8+2N)\log_2(\sqrt{p}) + \frac{N\times(p-1)}{p} + N + \frac{N^2\times(p-1)}{p}) \times t_w + \frac{N}{B} \times t_s + (p-2)B \times t_w + ((\frac{N\times(p-1)}{p}) \times B) \times t_c$

2.2.5 Solution 1: Optimum B for 2-D Mesh Model

We need to calculate derivative of final model of the 2-D Mesh model with respect to B , and set the derivative to 0 as shown below

$$\frac{dt_{total}(B)}{dB} = 0$$

And, using obtained model from section 2.2.4 we obtain this following equation

$$\frac{-N}{B^2}t_s + (p-2)t_w + \frac{N(p-1)}{p}t_c = 0$$

$$(p-2)t_w + \frac{N(p-1)}{p} \times t_c = \frac{N}{B^2}t_s$$

$$B^2 = \frac{Nt_s}{(p-2)t_w + \frac{N(p-1)}{p}t_c}$$

$$B^2 = \frac{pNt_s}{p(p-2)t_w + N(p-1)t_c}$$

$$B = \sqrt{\frac{pNt_s}{p(p-2)t_w + N(p-1)t_c}}$$

$$B \approx \sqrt{\frac{t_s}{t_c}}$$

As we observed here, the optimum B does not change when we use 2-D Mesh to model the communication. Using our solution 1, the usage of 2-D mesh model only affect the broadcast time. And refering to total time equation with respect to B($t_{total}(B)$), broadcast time is only a constant and it disappears when we calculate $\frac{dt_{total}(B)}{dB}$.

2.2.6 Solution 2: Using Send and Receive

In the second solution, we used Send and Receive methods provided in MPI library for communicating among the processes. In this implementation every process reads the input file. Every process also reads the similarity matrix.

After reading the files each process calculates the number of rows that it has to process and declares the required memory. Process with rank 0 declares the matrix H of size N * N. In our implementation data distribution is fair among all the process. In case of number of rows in the list are not divisible among all the processes we give one more row to each process starting from the master process. Figure 2 shows the distribution of data in case where data is not equally divisible among the processes.

Each process calculates the block size that it needs to communicate with its neighbour. Filling starts by master process and other process waits to receive the block to start processing. Master communicates its first block, with its neighbour, after processing its required number of rows for the first block. Below mentioned is the code snippet for filling the matrix at all the process.

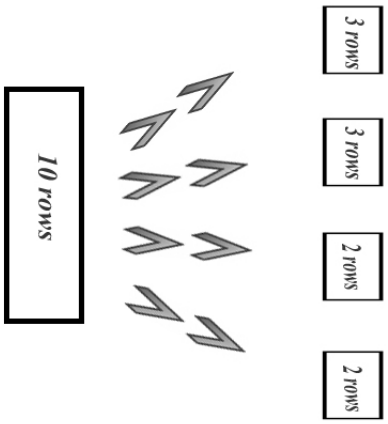

Figure 2: Data Partitioning among processes

```
1  if ( id == 0)
2      {
3          for(i=0;i<ColumnBlock;i++)
4          {
5              for(j=1;j<=s;j++)
6              {
7                  for(k=i*B+1;k<=(i+1)*B && k<=b[0] && k <= N;k++)
8                  {
9                      int RowPosition;
10                     if(id < r)
11                         RowPosition = id*((N/p)+1)+j;
12                     else
13                         RowPosition = (r*((N/p)+1))+((id-r)*(N/p
                               ))+j;
14
15                     diag = h[j-1][k-1] + sim[a[RowPosition]][b[k
                           ]];
16                         down = h[j-1][k] + DELTA;
17                         right = h[j][k-1] + DELTA;
18                         max = MAX3(diag,down,right);
19                     if (max <= 0)  {
20                         h[j][k] = 0;
21                     } else {
22                         h[j][k] = max;
23                     }
24                     chunk[k-(i*B+1)] = h[j][k];
25                 }
26             }
27             MPI_Send(chunk,B,MPI_SHORT,id+1,0,MPI_COMM_WORLD);
28         }
29     }else
30     {
```

```
31          for(i=0;i<ColumnBlock;i++)
32          {
33              MPI_Recv(chunk,B,MPI_SHORT,id-1,0,MPI_COMM_WORLD,&
                    status);
34              for(z=0;z<B;z++)
35              {
36                  if((i*B+z+1) <= N)
37                      h[0][i*B+z+1] = chunk[z];
38              }
39              for(j=1;j<=s;j++)
40              {
41                  int RowPosition;
42                  if(id < r)
43                      RowPosition = id*((N/p)+1)+j;
44                  else
45                      RowPosition = (r*((N/p)+1))+((id-r)*(N/p))+j
                            ;
46
47                  for(k=i*B+1;k<=(i+1)*B && k<=b[0] && k <= N;k++)
48                  {
49                      diag = h[j-1][k-1] + sim[a[RowPosition]][b[k
                            ]];
50                      down = h[j-1][k] + DELTA;
51                      right = h[j][k-1] + DELTA;
52                      max = MAX3(diag,down,right);
53                  if (max <= 0)
54                      h[j][k] = 0;
55                  else
56                      h[j][k] = max;
57
58                  chunk[k-(i*B+1)] = h[j][k];
59
60                  }
61              }
62              if(id != p-1)
63                  MPI_Send(chunk,B,MPI_SHORT,id+1,0,MPI_COMM_WORLD
                        );
64          }
65      }
```

At the end every process sends its portion of the matrix H to the master process using the Send method available in the MPI library. Below mentioned is the code snippet of gathering process.

```
1  if(id ==0)
2      {
3          int row, col;
4          for(i=1;i<p;i++)
5          {
6              MPI_Recv(&row,1,MPI_INT,i,0,MPI_COMM_WORLD,&status);
7              CHECK_NULL((recv_hptr = (int *) malloc(sizeof(int)*(
                   row)*(N))));
8
9              MPI_Recv(recv_hptr,row*N,MPI_INT,i,0,MPI_COMM_WORLD
```

```
                    ,&status);
10
11              for(j=0;j<row;j++)
12              {
13                  int RowPosition;
14                  if(i < r)
15                      RowPosition = (i*((N/p)+1))+j+1;
16                  else
17                      RowPosition = (r*((N/p)+1))+((i-r)*(N/p))+j
                            +1;
18
19                  for(k=0;k<N;k++)
20                      h[RowPosition][k+1]=recv_hptr[j*N+k];
21              }
22              free(recv_hptr);
23          }
24      }
25      else
26      {
27          MPI_Send(&s,1,MPI_INT,0,0,MPLCOMM_WORLD);
28          CHECK_NULL((recv_hptr = (int *) malloc(sizeof(int)*(s)*(
                N))));
29
30          for(j=0;j<s;j++)
31          {
32              for(k=0;k<N;k++)
33              {
34                  recv_hptr[j*N+k] = h[j+1][k+1];
35              }
36          }
37          MPI_Send(recv_hptr,s*N,MPI_INT,0,0,MPLCOMM_WORLD);
38          free(recv_hptr);
39      }
```

Once the result is gathered, process with rank 0 deallocates the memory. and perform optional verification result.

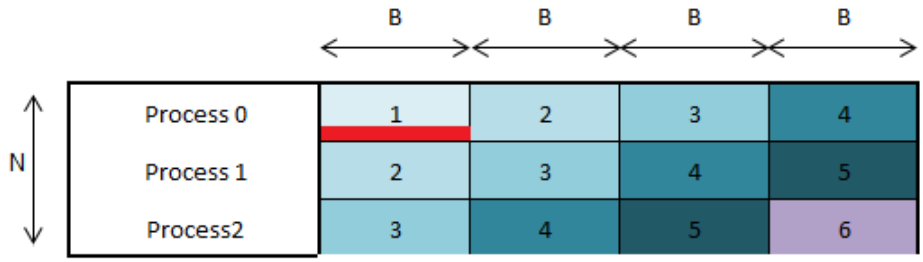

Figure 3: Blocking Communication

As reflected in Figure 3, the dividing of block in Solution 2 is same with solution 1. But, instead of using scatter and gather to distribute data,

solution 2 uses primitive sends and receives.

2.2.7 Solution 2: Linear-array Model

Initially we calculated the performance model for the Linear interconnection Network. The timing diagram could be found in the Appendix C.

1. In solution 2 every process calculates the *N/p*B* number of values before communicating a chunk with the other process. It takes *(N/B)+p-1* steps in total for computation.Below mentioned is the equation for computation.

   $t_{comp1} = (\frac{N}{B} + p - 1) \times (\frac{N}{p} \times B) \times t_c$

2. After computation step each process communicates a Block with its neighbour process. There are *(N/B)+p-2* steps of communication among all the processes.

   $t_{comm1} = (\frac{N}{B} + p - 2) \times (t_s + B \times t_w)$

3. After completing their part of matrix H every process sends it to the master process.

   $t_{comm2} = (t_s + \frac{N}{p} \times N \times t_w)$

4. In the end master process puts all the partial result in the matrix H to finalize the matrix H.

   $t_{comp2} = (t_s + \frac{N}{p} \times N \times t_w)$

The total time can be calculated by combining all the communication times.

$t_{total} = t_{comp1} + t_{comm1} + t_{comp2} + t_{comm2}$

$t_{total} = (\frac{N}{B} + p - 1) \times (\frac{N}{p} \times B) \times t_c + (\frac{N}{B} + p - 2) \times (t_s + B \times t_w) + (t_s + \frac{N}{p} \times N \times t_w) + (t_s + \frac{N}{p} \times N \times t_w)$

2.2.8 Solution 2: Optimum B for Linear-array Model

To find optimum B for linear array model, we need to calculate derivative of final model of the linear topology with respect to B , and set the derivative to 0 as shown below

$$\frac{dt_{total}(B)}{dB} = 0$$

And, using obtained model from section 2.2.7 we obtain this following equation

$$\frac{-N}{B^2} t_s + (p-2) t_w + \frac{N(p-1)}{p} t_c = 0$$

$$(p-2)t_w + \frac{N(p-1)}{p}t_c = \frac{N}{B^2}t_s$$

$$B^2 = \frac{Nt_s}{(p-2)t_w + \frac{N(p-1)}{p}t_c}$$

$$B^2 = \frac{pNt_s}{p(p-2)t_w + N(p-1)t_c}$$

$$B = \sqrt{\frac{pNt_s}{p(p-2)t_w + N(p-1)t_c}}$$

2.2.9 Solution 2: 2-D Mesh Model

We calculated the performance model for the 2D-Mesh interconnection Network. And we found that there is no difference between the Linear Array Model and 2-D Mesh model because the difference between them is mainly in the time to perform broadcasting and this solution does not involve any broadcasting of element from root to other processes in the system.

2.3 Blocking-and-Interleave Technique

2.3.1 Solution 1: Using Scatter and Gather

Taking into account not only Blocking size B but also Interleave size I, we developed solution below. First step is to allocate memory for all necessary variables in each processes. Master process also allocates memory for the final matrix where all the partial results will be stored. All slave processes will also allocate memory for partial result matrices which eventually will be send to the master process.

```
1  main(int argc, char *argv[]) {
2
3      {...}
4
5      int B, I;
6
7      MPI_Init(&argc, &argv);
8      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9      MPI_Comm_size(MPI_COMM_WORLD, &total_processes);
10
11     if (rank == 0) {
12         chunk_size = sizeA / (total_processes * I);
13
14         CHECK_NULL((h_all_ptr = (int *) calloc(N * (sizeA+1),
               sizeof(int)))); //resulting data
15         CHECK_NULL((h_all = (int **) calloc((sizeA+1), sizeof(
               int*)))); //contain list of pointer
```

```
16          for(i = 0; i < sizeA; i++)
17              h_all[i] = h_all_ptr + i * N;
18
19          //initialize the first row of resulting matrix with 0
20          for(i = 0; i < N; i++)
21          {
22                  h_all[0][i] = 0;
23          }
24
25      }
26
27      MPI_Bcast(&chunk_size, 1, MPI_INT, 0, MPLCOMM_WORLD);
28      MPI_Bcast(&N, 1, MPI_INT, 0, MPLCOMM_WORLD);
29      MPI_Bcast(&B, 1, MPI_INT, 0, MPLCOMM_WORLD);
30      MPI_Bcast(&I, 1, MPI_INT, 0, MPLCOMM_WORLD);
31
32      CHECK_NULL((hptr = (int *) malloc(sizeof(int) * (N) * (
            chunk_size + 1))));
33      CHECK_NULL((h = (int **) malloc(sizeof(int*) * (chunk_size +
             1))));
34      for(i = 0; i < chunk_size + 1; i++)
35         h[i] = hptr + i * N;
36
37      CHECK_NULL((chunk_a = (short *) calloc(sizeof(short),
            chunk_size)));
38      if (rank != 0) {
39          CHECK_NULL((b = (short *) malloc(sizeof(short) * (N))));
40      }
41      MPI_Bcast(b, N, MPI_SHORT, 0, MPLCOMM_WORLD);
```

The master process scattering *vector A* to each process partially. Each interleave step there will be send part of the *vector A*. Sequence of code for the interleave 0 will be the same as in previous section but only with one exception that the last process will send its results to the first process. Each process receives *size B* data from previous one before processing next B columns. Each process sends data after processing B columns to the next processes but the last process sends the data to the first(master) one if it's not the last stage.

Finally after calculating all partial matrices each process sends its result to the master process (It happens interleave times).

```
1
2  for (int current_interleave = 0; current_interleave < I;
       current_interleave++) {
3      MPI_Scatter(a + current_interleave * chunk_size *
           total_processes,
4      chunk_size, MPI_SHORT, chunk_a, chunk_size, MPI_SHORT, 0,
           MPLCOMM_WORLD);
5      int current_column = 1;
6      for (i = 0; i < chunk_size + 1; i++) h[i][0] = 0;
7      for (int current_block = 0; current_block < total_blocks;
           current_block++) {
```

```
8           // Receive
9           int block_end = MIN2(current_column - (current_block ==
                0 ? 1 : 0) + B, N);
10          if (rank == 0 && current_interleave == 0) {
11              for (int k = current_column; k < block_end; k++) {
12                  h[0][k] = 0;
13              }
14          } else {
15              int receive_from = rank == 0 ? total_processes - 1 :
                    rank - 1;
16              int size_to_receive = current_block == total_blocks
                    - 1 ? last_block : B;
17              MPI_Recv(h[0] + current_block * B, size_to_receive,
18                  MPI_INT, receive_from, 0, MPI_COMM_WORLD, &
                        status);
19          }
20          // Process
21          for (j = current_column; j < block_end; j++,
                current_column++) {
22              for (i = 1; i < chunk_size + 1; i++) {
23                  diag = h[i-1][j-1] + sim[chunk_a[i - 1]][b[j -
                        1]];
24                  down = h[i-1][j] + DELTA;
25                  right = h[i][j-1] + DELTA;
26                  max = MAX3(diag,down,right);
27                  if (max <= 0)  {
28                      h[i][j] = 0;
29                  } else {
30                      h[i][j] = max;
31                  }
32              }
33          }
34          // Send
35          if (current_interleave + 1 != I || rank + 1 !=
                total_processes) {
36              int send_to = rank + 1 == total_processes ? 0 : rank
                    + 1;
37              int size_to_send = current_block == total_blocks - 1
                    ? last_block : B;
38              MPI_Send(h[chunk_size] + current_block * B,
                    size_to_send,
39                  MPI_INT, send_to, 0, MPI_COMM_WORLD);
40          }
41      }
42      MPI_Gather(hptr + N, N * chunk_size, MPI_INT,
43          h_all_ptr + N + current_interleave * chunk_size *
                total_processes * N,
44          N * chunk_size, MPI_INT, 0, MPI_COMM_WORLD);
45  }
46      MPI_Finalize();
47  }
48  {...}
```

To summarize the interleave realization illustrated on Figure 4.

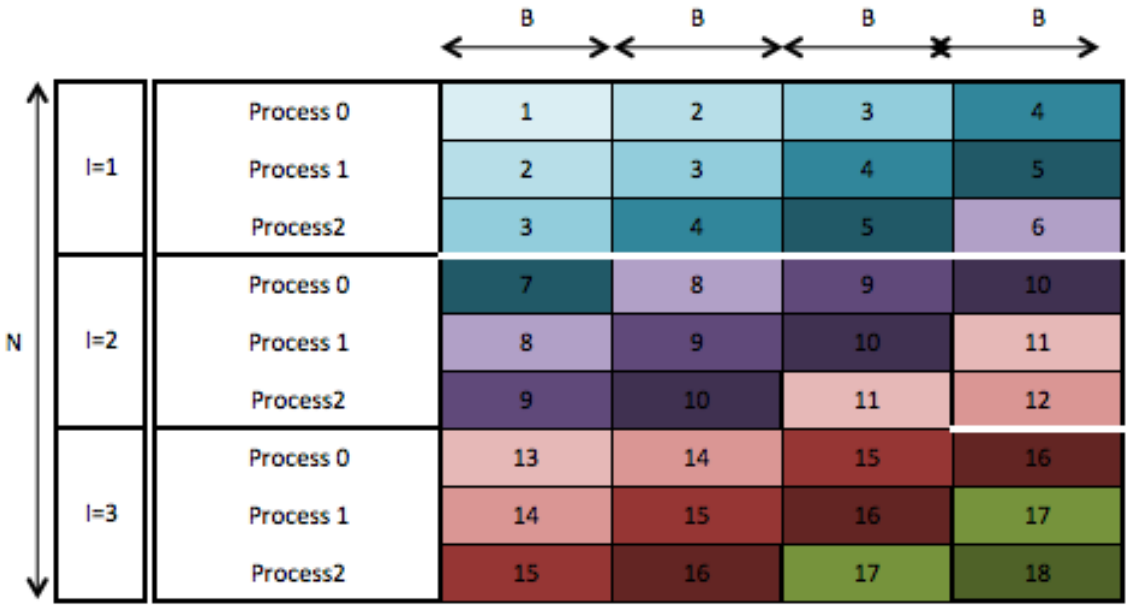

Figure 4: Blocking and interleave communication

2.3.2 Solution 1: Linear-Array Model

Here is linear array model for communication part for blocking technique with interleave

1. Broadcasting **chunk_size**, **N**, **B**, and **I**

   $t_{comm-bcast-4-int} = 4 \times (t_s + t_w) \times \log_2(p)$

2. Broadcasting of 2nd protein sequence (vector **b**)

   $t_{comm-bcast-protein-seq} = (t_s + t_w \times N) \times \log_2(p)$

3. Scattering **chunk_size** for each process to compute

   Note that the size of chunk_size is the following

   $chunk\_size = \frac{N}{p \times I}$ where I is the interleave factor.

   And scattering is performed I times. Therefore, the communication cost of scattering is

   $t_{comm-scatter-protein-seq} = I \times (t_s \times \log_2(p) + t_w \times \frac{N}{p \times I} \times (p-1))$

4. Sending shared data

   To start the first block of computation, process with rank 0 does not need to wait for any data from other processes. And, we need to take

note that in each interleave except the last interleave, last process ($(p-i)^{th}$ process) needs to send N data to process 0. Therefore, for $I-1$ occurences, we need $(\frac{N}{B}+p-1)$ pipeline stages for sending data, and for the last Interleave step (the I^{th} steps), we will have $(\frac{N}{B}+p-2)$ stages for sending data. The shared data is the last row of current finished block which consists of B items.Therefore putting all of them together, communication time to send shared data is

$t_{comm-send-shared-data} = (I-1) \times (\frac{N}{B}+p-1) \times (t_s + (B \times t_w)) + (\frac{N}{B} + p-2) \times (t_s + (B \times t_w))$

5. Gathering calculated data

   We need to perform gather to combine all calculated data in every interleave step. Note that every process will need to combine $N \times chunk\_size$ data, which equals to $N \times \frac{N}{P \times I}$ amount of data. This gather procedure is repeated I times. Therefore the communication time for this step is given by

   $t_{comm-gather} = I \times (t_s \times \log_2(p) + t_w \times \frac{N}{p \times I} \times N \times (p-1))$

6. Putting all the communication time together

   $t_{comm-all} = t_{comm-bcast-4-int} + t_{comm-bcast-protein-seq} + t_{comm-scatter-protein-seq} + t_{comm-send-shared-data} + t_{comm-gather}$

   Simplifying the equation with respect to B (by separating constant of the equation with the component of the equation containing B, so that we can easily calculate the derivative of the equation to obtain maximum B), we obtain this following equation

   $t_{comm-all}(B) = ((5+2I)log_2(p) + (p-1)(I-1) + (p-2)) \times t_s + ((4+N)log_2(p) + \frac{N}{p}(p-1) + \frac{N^2}{p}(p-1) + I - 1 + N) \times t_w + \frac{IN}{B} \times t_s + ((I-1)(p-1) + p - 2)B \times t_w$

   Simplyfing the equation with respect to I, we obtain this following equation

   $t_{comm-all}(I) = ((5+2I)log_2(p) - 1) \times t_s + ((4+N)log_2(p) + \frac{N}{p}(p-1) + \frac{N^2}{p}(p-1) + B) \times t_w + (\frac{N}{B} + p - 1)(t_s + Bt_w)I$

Now we calculate the calculation time for this blocking technique. Note that in our blocking technique we have $I \times (\frac{N}{B} + p - 1)$ stages of block-calculation. In each block-calculation, we need to compute $\frac{N}{p \times I} \times B$ points. Therefore, if we represent time to compute one point as t_c, we obtain this following calculation time model

$t_{calc} = I \times (\frac{N}{B} + p - 1) \times (\frac{N}{p \times I} \times B) \times t_c$

I will be canceled and we obtain this following

$t_{calc} = (\frac{N^2}{p} + NB - \frac{NB}{p}) \times t_c$
$t_{calc} = (\frac{N^2}{p} + (\frac{N \times (p-1)}{p}) \times B) \times t_c$

Final model can be obtained by adding calculation time and communication time, and here is the final equation with respect to B

$t_{total} = t_{comm} + t_{calc}$

$t_{total}(B) = ((5+2I)log_2(p)+(p-1)(I-1)+(p-2)) \times t_s + ((4+N)log_2(p)+ \frac{N}{p}(p-1) + \frac{N^2}{p}(p-1) + I - 1 + N) \times t_w + \frac{IN}{B} \times t_s + ((I-1)(p-1) + p - 2)B \times t_w + (\frac{N^2}{p} + (\frac{N \times (p-1)}{p}) \times B) \times t_c$

Here is the final equation with respect to I

$t_{comm-all}(I) = ((5 + 2I)log_2(p) - 1) \times t_s + ((4 + N)log_2(p) + \frac{N}{p}(p-1) + \frac{N^2}{p}(p-1) + B) \times t_w + (\frac{N}{B} + p - 1)(t_s + Bt_w)I + (\frac{N^2}{p} + (\frac{N \times (p-1)}{p}) \times B) \times t_c$

2.3.3 Solution 1: Optimum B and I for Linear-array Model

Optimum B can be derived by calculating $\frac{dt_{total}(B)}{dB}$ and set the inequality to 0.

$$\frac{dt_{total}(B)}{dB} = 0$$

And, using obtained model from previous section we obtain this following equation

$$\frac{-IN}{B^2} t_s + ((I-1)(p-1) + (p-2))t_w + \frac{N(p-1)}{p} t_c = 0$$

$$((I-1)(p-1) + (p-2))t_w + \frac{N(p-1)}{p} t_c = \frac{IN}{B^2} t_s$$

$$B^2 = \frac{INt_s}{((I-1)(p-1) + (p-2))t_w + \frac{N(p-1)}{p} t_c}$$

$$B^2 = \frac{pINt_s}{((I-1)(p-1) + (p-2))pt_w + N(p-1)t_c}$$

$$B = \sqrt{\frac{pINt_s}{((I-1)(p-1)+(p-2))pt_w + N(p-1)t_c}}$$

$$B \approx \sqrt{\frac{INt_s}{(Nt_c + I)}}$$

However, we can not find optimum I for Blocking-and-Interleave technique because the derivation of $\frac{dt_{total}(I)}{dI}$ results in a constant as shown below

$$\frac{dt_{total}(I)}{dI} = 0$$

$$(\frac{N}{B} + p - 1)(t_s + Bt_w) = 0$$

Looking at equation of $dt_{total}(I)$, interleave factor only introduce more communication time when sending and receiving shared data. Therefore no optimum interleave level can be derived using this model.

2.3.4 Solution 1: 2-D Mesh Model

Using similar technique as what we have done in Linear-array model, here is the communication and computation model of 2-D Mesh Model

1. Broadcasting **chunk_size**, **N**, **B**, and **I**

   $t_{comm-bcast-4-int} = 4 \times 2 \times (t_s + t_w) \times \log_2(\sqrt{p})$

2. Broadcasting of 2nd protein sequence (vector **b**)

   $t_{comm-bcast-protein-seq} = 2 \times (t_s + t_w \times N) \times \log_2(\sqrt{p})$

3. Scattering **chunk_size** for each process to compute
   As what we discuss in section 2.2.4, scattering communication model between 2-D Mesh model and Linear Array model are equals.

   $t_{comm-scatter-protein-seq} = I \times (t_s \times \log_2(p) + t_w \times \frac{N}{p \times I} \times (p-1))$

4. Sending shared data Communication time for sending shared data also equal between 2-D Mesh model and Linear Array model.

   $t_{comm-send-shared-data} = (I-1) \times (\frac{N}{B} + p - 1) \times (t_s + (B \times t_w)) + (\frac{N}{B} + p - 2) \times (t_s + (B \times t_w))$

5. Gathering calculated data Gathering formula is equal to scattering except for the amount of data being gathered.

   $t_{comm-gather} = I \times (t_s \times \log_2(p) + t_w \times \frac{N}{p \times I} \times N \times (p-1))$

6. Putting all the communication time together

   $t_{comm-all} = t_{comm-bcast-4-int} + t_{comm-bcast-protein-seq} + t_{comm-scatter-protein-seq} + t_{comm-send-shared-data} + t_{comm-gather}$

   Simplifying the equation with respect to B (by separating constant of the equation with the component of the equation containing B, so that we can easily calculate the derivative of the equation to obtain maximum B), we obtain this following equation

   $t_{comm-all}(B) = (10\log_2(\sqrt{p}) + 2I\log_2(p) + (p-1)(I-1) + (p-2)) \times t_s + ((8+2N)log_2(\sqrt{p}) + \frac{N}{p}(p-1) + \frac{N^2}{p}(p-1) + I - 1 + N) \times t_w + \frac{IN}{B} \times t_s + ((I-1)(p-1) + p - 2)B \times t_w$

   Simplyfing the equation with respect to I, we obtain this following equation
   $t_{comm-all}(I) = (10\log_2(\sqrt{p}) + 2I\log_2(p) - 1) \times t_s + ((8+2N)log_2(\sqrt{p}) + \frac{N}{p}(p-1) + \frac{N^2}{p}(p-1) + B) \times t_w + (\frac{N}{B} + p - 1)(t_s + Bt_w)I$

2.3.5 Solution 1: Optimum B and I for 2-D Mesh Model

Optimum B can be derived by calculating $\frac{dt_{total}(B)}{dB}$ and set the inequality to 0.

$$\frac{dt_{total}(B)}{dB} = 0$$

And, using obtained model from previous section we obtain this following equation

$$\frac{-IN}{B^2}t_s + ((I-1)(p-1) + (p-2))t_w + \frac{N(p-1)}{p} \times t_c = 0$$

$$((I-1)(p-1) + (p-2))t_w + \frac{N(p-1)}{p} \times t_c = \frac{IN}{B^2}t_s$$

$$B^2 = \frac{INt_s}{((I-1)(p-1) + (p-2))t_w + \frac{N(p-1)}{p} \times t_c}$$

$$B^2 = \frac{pINt_s}{((I-1)(p-1) + (p-2))pt_w + N(p-1) \times t_c}$$

$$B = \sqrt{\frac{pINt_s}{((I-1)(p-1) + (p-2))pt_w + N(p-1) \times t_c}}$$

We observe that the resulting optimum B for 2-D Mesh model is equal to Linear Array model. As what we have discussed in section 2.2.5, 2-D Mesh model only differs in the broadcast time which act as constant in $t_{total}(B)$ equation and the constant disappear when we calculate the derivaion of the equation.

Similar to calculation of optimum I in Linear Array Model, we can not find optimum I for Blocking-and-Interleave technique because the derivation of $\frac{dt_{total}(I)}{dI}$ results in a constant as shown below

$$\frac{dt_{total}(I)}{dI} = 0$$

$$(\frac{N}{B} + p - 1)(t_s + Bt_w) = 0$$

2.3.6 Solution 1: Improvement

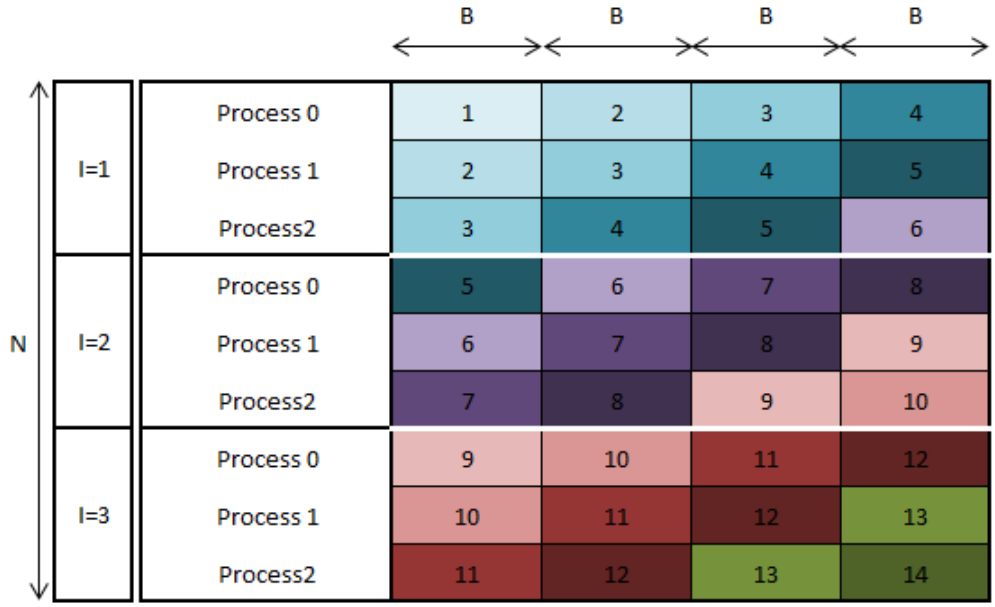

Figure 5: Blocking and Interleave Communication

The main idea of this improvement is moving the gathering final data process into the end of whole calculation in each process. That means, refering to Figure 5, gathering will be performed after step 14.

To implement this improvement, we performed these following steps:

1. Allocate enough memory for each process, to hold $I \times N \times chunk\_size$. Note that $chunk\_size$ in this case is $\frac{N}{P \times I}$.

```
1  CHECK_NULL((hptr = (int *) malloc(sizeof(int) * (N) * I *
       (chunk_size
2 + 1))));//Instantiate temporary resulting matrix for each
       process
3    CHECK_NULL((h = (int **) malloc(sizeof(int*) * I * (
         chunk_size +
4 1))));//list of pointer
5
6    int ***hfin;
7    CHECK_NULL(hfin = (int ***) malloc(sizeof(int***) * I));
8
9    for(i = 0; i < (chunk_size + 1) * I; i++) {
10      h[i] = hptr + i * N;//put the prointer int the array
11   }
12
13   for(i = 0; i < I; i++) {
14      hfin[i] = h + i * (chunk_size + 1);
15   }
```

2. Change the way each process manipulates the data. Each process stores the data using *hfin*. *hfin* is a variable with type ****int*, therefore we need to store the data as shown in the following code snippet

```
1     for (int current_interleave = 0; current_interleave < I
          ; current_interleave++) {
2
3     MPI_Scatter(a + current_interleave * chunk_size *
          total_processes,
4           chunk_size, MPI_SHORT, chunk_a, chunk_size,
              MPI_SHORT, 0, MPI_COMM_WORLD);//chunk_a is
              the receiving buffer
5
6         int current_column = 1;
7         for (i = 0; i < chunk_size + 1; i++) hfin[
              current_interleave][i][0] = 0;
8
9         for (int current_block = 0; current_block <
              total_blocks; current_block++) {
10            // Receive
11            int block_end = MIN2(current_column - (
                  current_block == 0 ? 1 : 0) + B, N);
12            if (rank == 0 && current_interleave == 0) { //
                  if rank 0 is processing the first block, it
                  doesn't need to receive any thing
13                for (int k = current_column; k < block_end;
                      k++) {
14                    hfin[current_interleave][0][k] = 0;//
                          init row 0
15                }
16            } else {
17                int receive_from = rank == 0 ?
                      total_processes - 1 : rank - 1;//receive
```

```
                    from neighboring process
18                  int size_to_receive = current_block ==
                        total_blocks - 1 ? last_block : B;
19
20                  MPI_Recv(hfin[current_interleave][0] +
                        current_block * B, size_to_receive,
                        MPI_INT, receive_from, 0, MPI_COMM_WORLD
                        , &status);
21              }
22              for (j = current_column; j < block_end; j++,
                    current_column++) {
23                  for (i = 1; i < chunk_size + 1; i++) {
24                  diag = hfin[current_interleave][i-1][j-1] +
                         sim[chunk_a[i - 1]][b[j - 1]];
25                  down = hfin[current_interleave][i-1][j] +
                        DELTA;
26                  right = hfin[current_interleave][i][j-1] +
                        DELTA;
27                  max = MAX3(diag,down,right);
28                  if (max <= 0)  {
29                      hfin[current_interleave][i][j] = 0;
30                  } else {
31                      hfin[current_interleave][i][j] = max;
32                  }
33              }
34              }
35
36              // Send
37              if (current_interleave + 1 != I || rank + 1 !=
                    total_processes) {
38                  int send_to = rank + 1 == total_processes ?
                         0 : rank + 1;
39                  int size_to_send = current_block ==
                        total_blocks - 1 ? last_block : B;
40                  MPI_Send(hfin[current_interleave][
                        chunk_size] + current_block * B,
                        size_to_send, MPI_INT, send_to, 0,
                        MPI_COMM_WORLD);
41              }
42          }
43      }
```

Note that *hfin[i]* means it contains the data for the ith interleaving stage in each process.

3. Move gathering process into the end of all calculation as shown in the following code snippet

```
1       for (i = 0; i < I; i++) {
2           MPI_Gather(hptr + N + i * chunk_size * N, N *
                chunk_size, MPI_INT,
3               h_all_ptr + N + i * chunk_size *
                    total_processes * N,
```

```
4                 N * chunk_size, MPI_INT, 0, MPI_COMM_WORLD);
5         }
```

2.3.7 Solution 1: Optimum B and I for the Improved Solution

Here is the part of the model that are affected by the improved solution.

1. Sending shared data

   For the first *I - 1* interleaving stages the communication time is followed:

   $(I-1) \times (t_s + t_w \times B) \times \frac{N}{B}$

   Then the last interleaving stage consist of following amount of communication time:

   $(t_s + t_w \times B) \times (\frac{N}{B} + P - 2)$

   Therefore putting all of them together, communication time to send shared data is

   $(t_s + t_w \times B) \times (\frac{N}{B} + P - 2) + (I-1) \times (t_s + t_w \times B) \times \frac{N}{B}$

2. Computational time

   As well with sending and receive changes, time for computation are also improved.

   $(\frac{N}{B} \times B \times \frac{N}{P \times I} \times (I-1) + B \times \frac{N}{P \times I} \times (\frac{N}{B} + P - 1)) \times t_c$

Optimal B and I for Improved Solution

To calculate the optimal value we ignore all the communication time which is not going to influent the value of optimal B and I. For optimal B, we only have the following formula the calculation.

$t\_total\_improved(B) = (t_s + t_w \times B) \times (\frac{N}{B} + P - 2) + (I-1) \times (t_s + t_w \times B) \times \frac{N}{B} + (\frac{N}{B} \times B \times \frac{N}{P \times I} \times (I-1) + B \times \frac{N}{P \times I} \times (\frac{N}{B} + P - 1)) \times t_c$

$$\frac{dt\_total\_improved(B)}{dB} = 0$$

$$-\frac{(I-1) \times t_s \times N}{B^2} - \frac{N \times t_s}{B^2} + (P-2) \times t_w + (P-1) \times \frac{N}{P \times I} \times t_c = 0$$

$$B = \sqrt{\frac{I^2 \times t_s \times N \times P}{(P-2) \times t_w \times P \times I + (P-1) \times N \times t_c}}$$

$$B \approx \sqrt{\frac{I \times N \times t_s}{(P-2) \times t_w}}$$

However, for optimal I value, we need to consider also scatter time as well. Therefore we obtain this following formula for $t\_total\_improved(I)$

$t\_total\_improved(I) = I \times t_s \times \log_2(p) + (t_s + t_w \times B) \times (\frac{N}{B} + P - 2) + (I - 1) \times (t_s + t_w \times B) \times \frac{N}{B} + \frac{N}{B} \times B \times \times \frac{N}{P \times I} \times (I-1) + B \times \frac{N}{P \times I} \times (\frac{N}{B} + P - 1) \times t_c)$

$$\frac{dt\_total\_improved(I)}{dI} = 0$$

$$t_s \times \log_2(p) + (t_s + t_w \times B) \times \frac{N}{B} + \frac{N^2 \times B}{B \times P \times I^2} \times t_c - \frac{B \times N}{P \times I^2} \times (\frac{N}{B} + P - 1) \times t_c = 0$$

$$I = \sqrt{\frac{B^2 \times N \times (\frac{N}{B} + P - 1) \times t_c - N^2 \times B \times t_c}{B \times P \times t_s \times log_2(p) + (t_s + t_w \times B) \times N \times P}}$$

$$I \approx \sqrt{\frac{B \times N \times t_c}{t_s \times log_2(p) + N \times t_w + \frac{N}{B} \times t_s}}$$

2.3.8 Solution 2: Using Send and Receive

This implementation also takes in account the row interleave factor along with the column interleave. Every process calculates the number of rows it has to process at every interleave and initializes the memory. Master process declares the matrix H and use it for its partial processing as well.

Each process process N/(p*I) number of rows in every interleave and communicates the block with its neighbour process. Last process communicates its block with the master process and do not perform any communication in the last interleave.

```
1  if ( id == 0)
2          {
3              for(i=0;i<ColumnBlock;i++)
4              {
5                  CHECK_NULL((chunk = (int *) malloc(sizeof(int)*(
                       B))));
6
7                  for(j=1;j<=s;j++)
8                  {
9                      for(k=i*B+1;k<=(i+1)*B && k<=b[0] && k <= N;
                           k++)
10                     {
11                         int RowPosition;
12
13                         if( (interleave*p+id) < r)
14                             RowPosition = (interleave*(N/(p*I)
                                   +1)*p) + id*((N/(p*I)+1)) + j;
15                         else
```

```
16                          RowPosition = (r*(N/(p*I)+1)) + (
                                interleave*p+id-r)*(N/(p*I)) + j;
17
18                      diag = h[RowPosition-1][k-1] + sim[a[
                            RowPosition]][b[k]];
19                      down = h[RowPosition-1][k] + DELTA;
20                      right = h[RowPosition][k-1] + DELTA;
21                      max = MAX3(diag,down,right);
22
23                      if (max <= 0)  {
24                          h[RowPosition][k] = 0;
25                      } else {
26                          h[RowPosition][k] = max;
27                      }
28                      chunk[k-(i*B+1)] = h[RowPosition][k];
29                  }
30              }//communicate toe partial block to next process
31              MPI_Send(chunk,B,MPI_INT,id+1,0,MPI_COMM_WORLD);
32              free(chunk);
33          }
34      //end filling matrix H[][] at master
35      }else if (id != p-1)
36      {//filling matrix at other processes
37
38          for(i=0;i<ColumnBlock;i++)
39          {
40              CHECK_NULL((chunk = (int *) malloc(sizeof(int)*(
                    B))));
41
42              MPI_Recv(chunk,B,MPI_INT,id-1,0,MPI_COMM_WORLD,&
                    status);
43              for(z=0;z<B;z++)
44              {
45                  if((i*B+z) <= N)
46                      h[0][i*B+z+1] = chunk[z];
47              }
48              for(j=1;j<=s;j++)
49              {
50                  int RowPosition;
51
52                  if( (interleave*p+id) < r)
53                      RowPosition = (interleave*(N/(p*I)+1)*p)
                            + id*((N/(p*I)+1)) + j;
54                  else
55                      RowPosition = (r*(N/(p*I)+1)) + (
                            interleave*p+id-r)*(N/(p*I)) + j;
56
57
58                  for(k=i*B+1;k<=(i+1)*B && k<=b[0] && k <= N;
                        k++)
59                  {
60                      diag = h[j-1][k-1] + sim[a[RowPosition
                            ]][b[k]];
61                      down = h[j-1][k] + DELTA;
```

```
62                          right = h[j][k-1] + DELTA;
63                          max = MAX3(diag,down,right);
64                          if (max <= 0)
65                              h[j][k] = 0;
66                          else
67                              h[j][k] = max;
68
69                          chunk[k-(i*B+1)] = h[j][k];
70                      }
71                  }
72                  MPI_Send(chunk,B,MPI_INT,id+1,0,MPI_COMM_WORLD);
73                  free(chunk);
74              }//end filling matrix at other processes
75          }else//start filling matrix at last process
76          {
77              for(i=0;i<ColumnBlock;i++)
78              {
79                  CHECK_NULL((chunk = (int *) malloc(sizeof(int)*(
                        B))));
80
81                  MPI_Recv(chunk,B,MPI_INT,id-1,0,MPI_COMM_WORLD,&
                        status);
82                  for(z=0;z<B;z++)
83                  {
84                      if((i*B+z) <= N)
85                          h[0][i*B+z+1] = chunk[z];
86                  }
87
88                  free(chunk);
89                  for(j=1;j<=s;j++)
90                  {
91                      int RowPosition;
92                      if( (interleave*p+id) < r)
93                          RowPosition = (interleave*(N/(p*I)+1)*p)
                                + id*((N/(p*I)+1)) + j;
94                      else
95                          RowPosition = (r*(N/(p*I)+1)) + (
                                interleave*p+id-r)*(N/(p*I)) + j;
96
97
98                      for(k=i*B+1;k<=(i+1)*B && k<=b[0] && k <= N;
                            k++)
99                      {
100                         diag = h[j-1][k-1] + sim[a[RowPosition
                                ]][b[k]];
101                         down = h[j-1][k] + DELTA;
102                         right = h[j][k-1] + DELTA;
103                         max = MAX3(diag,down,right);
104                         if (max <= 0)
105                             h[j][k] = 0;
106                         else
107                             h[j][k] = max;
108
109
```

```
110                         }
111                     }
112                 }
113             }
```

After filling the partial matrix H, every process sends the partial result to the master process at every interleave. Below mentioned is the code snippet of master gathering the partial result after every interleave.

```
1  if(id ==0)
2          {
3              int row, col;
4              for(i=1;i<p;i++)
5              {
6                  MPI_Recv(&row,1,MPI_INT,i,0,MPI_COMM_WORLD,&
                       status);
7                  CHECK_NULL((recv_hptr = (int *) malloc(sizeof(
                       int)*(row)*(N))));
8
9                  MPI_Recv(recv_hptr,row*N,MPI_INT,i,0,
                       MPI_COMM_WORLD,&status);
10
11                 for(j=0;j<row;j++)
12                 {
13                     int RowPosition;
14
15                     if( (interleave*p+i) < r)
16                         RowPosition = (interleave*(N/(p*I)+1)*p)
                               + i*((N/(p*I)+1)) + j+1;
17                     else
18                         RowPosition = (r*(N/(p*I)+1)) + (
                               interleave*p+i-r)*(N/(p*I)) + j+1;
19
20                     for(k=0;k<N;k++)
21                         h[RowPosition][k+1]=recv_hptr[j*N+k];
22
23                 }
24                 free(recv_hptr);
25             }
26         }
27         else
28         {
29             MPI_Send(&s,1,MPI_INT,0,0,MPI_COMM_WORLD);
30             CHECK_NULL((recv_hptr = (int *) malloc(sizeof(int)*(
                   s)*(N))));
31
32             for(j=0;j<s;j++)
33             {
34                 for(k=0;k<N;k++)
35                     recv_hptr[j*N+k] = h[j+1][k+1];
36             }
37             MPI_Send(recv_hptr,s*N,MPI_INT,0,0,MPI_COMM_WORLD);
38
39             free(recv_hptr);
```

```
40          }
```

To summarize the interleave realization illustrated in Appendix D.

2.3.9 Solution 2: Linear-array Model

1. Every process calculates the *(N/(p*I))*B* number of values in every interleave before communicating a chunk with the other process. It takes *((N/B)+p-1)*I* steps in total for computation.Below mentioned is the equation for computation.

   $t_{comp1} = I \times (\frac{N}{B} + p - 1) \times (\frac{N}{p \times I} \times B) \times t_c$

2. After computation step each process communicates a Block with its neighbour process. There are *(N/B)+p-2* steps of communication among all the processes.

   $t_{comm1} = (I-1) \times (\frac{N}{B} + p - 1) \times (t_s + B \times t_w) + (\frac{N}{B} + p - 2) \times (t_s + B \times t_w)$

3. After completing their part of matrix H every process sends it to the master process.

   $t_{comm2} = (t_s + \frac{N}{(p \times I} \times N \times t_w) \times I$

4. In the end master process puts all the partial result in the matrix H to finalize the matrix H.

   $t_{comp2} = I \times (t_s + \frac{N}{(p \times I)} \times N \times t_w)$

The total execution time can be calculated by combining all the times.

$t_{total} = t_{comp1} + t_{comm1} + t_{comp2} + t_{comm2}$

$t_{total} = I \times (\frac{N}{B} + p - 1) \times (\frac{N}{p \times I} \times B) \times t_c + (I-1) \times (\frac{N}{B} + p - 1) \times (t_s + B \times t_w) + (\frac{N}{B} + p - 2) \times (t_s + B \times t_w) + (t_s + \frac{N}{(p \times I} \times N \times t_w) \times I + I \times (t_s + \frac{N}{(p \times I)} \times N \times t_w)$

2.3.10 Solution 2: Optimum B and I for Linear-array Model

Optimum B can be derived by calculating $\frac{dt_{total}(B)}{dB}$ and set the inequality to 0.

$$\frac{dt_{total}(B)}{dB} = 0$$

And, using obtained model from previous section we obtain this following equation

$$\frac{-IN}{B^2} t_s + ((I-1)(p-1) + (p-2)) t_w + \frac{N(p-1)}{p} = 0$$

$$((I-1)(p-1)+(p-2))t_w + \frac{N(p-1)}{p} = \frac{IN}{B^2}t_s$$

$$B^2 = \frac{INt_s}{((I-1)(p-1)+(p-2))t_w + \frac{N(p-1)}{p}}$$

$$B^2 = \frac{pINt_s}{((I-1)(p-1)+(p-2))pt_w + N(p-1)}$$

$$B = \sqrt{\frac{pINt_s}{((I-1)(p-1)+(p-2))pt_w + N(p-1)}}$$

However, we can not find optimum I for Blocking-and-Interleave technique because the derivation of $\frac{dt_{total}(I)}{dI}$ results in a constant as shown below

$$\frac{dt_{total}(I)}{dI} = 0$$

$$(\frac{N}{B} + p - 1)(t_s + Bt_w) = 0$$

Looking at equation of $dt_{total}(I)$, interleave factor only introduce more communication time when sending and receiving shared data. Therefore no optimum interleave level can be derived using this model.

2.3.11 Solution 2: 2-D Mesh Model

As we have discussed in section 2.2.9, 2-D Mesh Model is same with Linear Array model because 2-D Mesh Model only affects the broadcast procedure and solution 2 does not include any broadcast procedure in its implementation.

3 Performance Results

We did performance measurement of both parallel versions in Altix Machine and compare the results against the sequential version.

3.1 Solution 1

3.1.1 Performance of Sequential Code

First we measured the performance of Smith-Waterman algorithm, using sequential code. Figure 6 shows the results.

Sequential Code	
Size	Time
1000	0.135
2000	0.529
3000	1.15
4000	2.061
5000	3.3
6000	4.644
7000	6.212
10000	12.508
20000	50.32

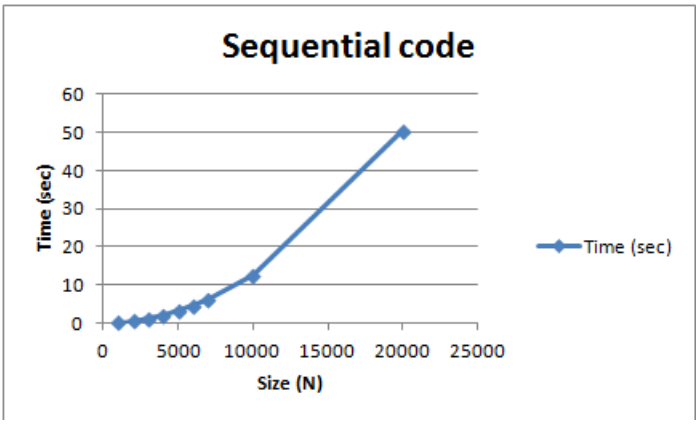

Figure 6: Sequential Code Performance Measurement Result

Figure 6 shows that when N is increased, the time taken to complete filling matrix h is also increased almost linearly.

3.1.2 Find Out Optimum Number of Processor (P)

At first, we observe the performance by fixing number of compared protein(N) to 5000 and 10000, block size (B) to 100 and set the interleave factor (I) to 1. The result is shown in Figure 7.

1. Protein size equals to 5000 (N = 5000) Block size (B) is 100, and Interleave factor (I) is 1

Parallel Different p (N=5000, B=100, I=1)	
p	time (seconds)
2	1.79
4	1.454
8	1.511
16	1.561
32	2.282

Figure 7: Measurement result when N is 5000, B is 100 and I is 1

Plotting the result in diagram in Figure 8

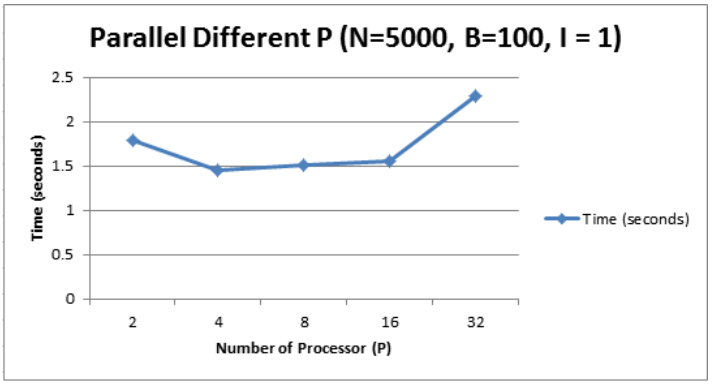

Figure 8: Diagram of measurement result when N is 5000, B is 100, I is 1

When the protein size (N) is 5000 and number of processor (P) is 4, we obtain $\frac{t\_serial}{t\_parallel} = \frac{3.3}{1.454} = 2.26$ times speedup.

2. Protein size equals to 10000 (N = 10000) We obtain this following result in Figure 9

Parallel Different p (N=10000, B=100, I=1)	
p	time (seconds)
2	3.717
4	2.823
8	2.47
16	2.575
32	3.332

Figure 9: Measurement result when N is 10000, B is 100 and I is 1

Plotting the result in diagram in Figure 10

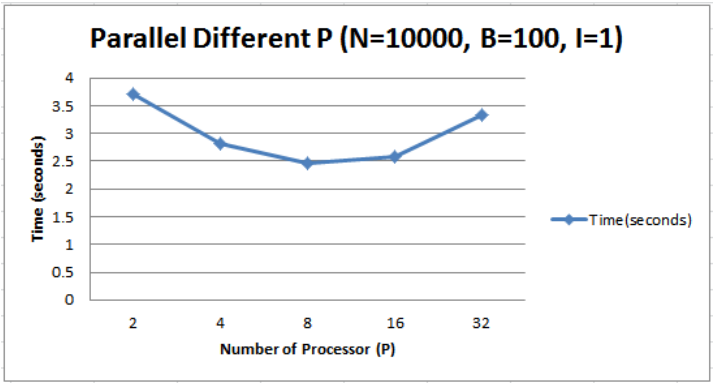

Figure 10: Diagram of measurement result when N is 10000, B is 100, I is 1

When the protein size (N) is 10000 and number of processor (P) is 8, we obtains $\frac{t\_serial}{t\_parallel} = \frac{12.508}{2.47} = 5.06$ times speedup.

Based on the result above, we found that maximum speedup is achieved when number of processor (P) is 8 and protein size (N) is 10000. Therefore, for the subsequent experiment, we will fix the number of processor to 8 and modify other parameters.

3.1.3 Find Out Optimum Blocking Size (B)

In this subsection, we analyze the performance result and find optimum blocking size (B). We fix number of processor (P) to 8, number of protein (N) to 10000 and interleave factor (I) to 1. The results are on Figure 11

Parallel Different B (N=10000, p=8, I=1)	
B	time (seconds)
1	2.861
50	2.592
100	2.401
200	3.273
500	3.287
1000	3.288
10000	4.775

Figure 11: Performance measurement result when N is 10000, P is 8, I is 1

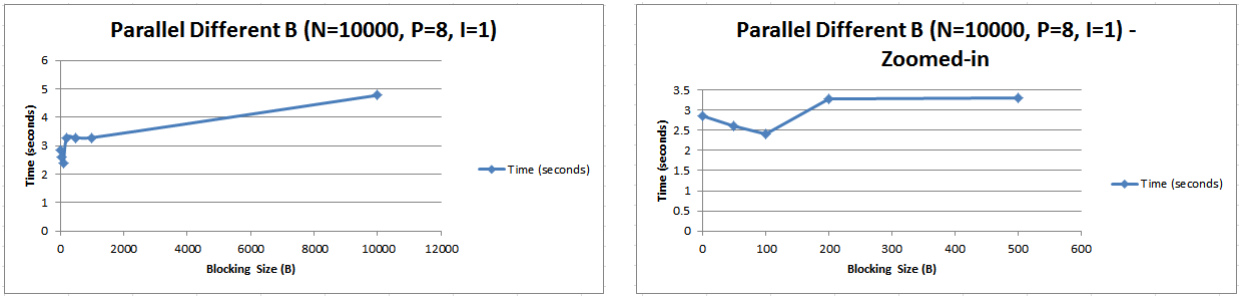

Figure 12: Diagram of measurement result when N is 10000, P is 8, I is 1

Plotting the result into diagram as shown in Figure 12. We zoomed in the diagram in right hand side of Figure 12 so that we have clearer picture on the performance when B is less than or equal to 500.

We found that optimum empirical blocking size (B) in the solution 1 is 100. And this yield in $\frac{t\_serial}{t\_parallel} = \frac{12.508}{2.401} = 5.21$ times speedup.

3.1.4 Find Out Optimum Interleave Factor (I)

Using the result from previous section in finding optimum blocking size (B), we find out most optimum I. The result is shown on Figure 13

Parallel Different I (N=10000, p=8, B=100)	
I	time (seconds)
1	2.625
2	2.906
5	3.99
10	6.84
20	8.464
30	12.94
40	16.75
50	15.16

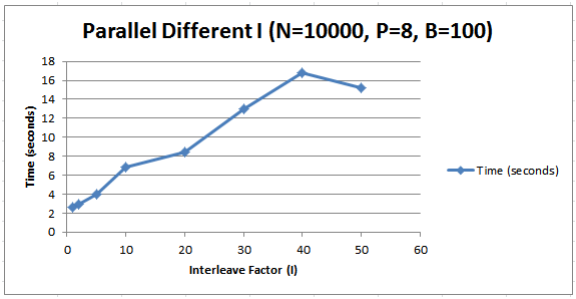

Figure 13: Diagram of measurement result when N is 10000, P is 8, B is 100

We found that optimum I is 1. And using optimum I of 1, we obtain 4.76 times speedup compared to sequential execution.

3.2 Solution 1-Improved

We did the same experiment as Solution 1 performance result to obtain necessary data about our improved solution

3.2.1 Find Out Optimum Number of Processor (P)

At first, we observe the performance by fixing number of compared protein(N) to 10000, block size (B) to 100 and set the interleave factor (I) to 1. The result is shown in Figure 14.

We obtain this following result in Figure 14

Parallel Different p (N=10000, B=100, I=1)	
p	time (seconds)
2	4.157
4	3.42
8	2.977
16	3.381

Figure 14: Measurement result when N is 10000, B is 100 and I is 1

Plotting the result in diagram in Figure 15

When the protein size (N) is 10000 and number of processor (P) is 8, we obtains $\frac{t\_serial}{t\_parallel} = \frac{12.508}{2.977} = 4.201$ times speedup.

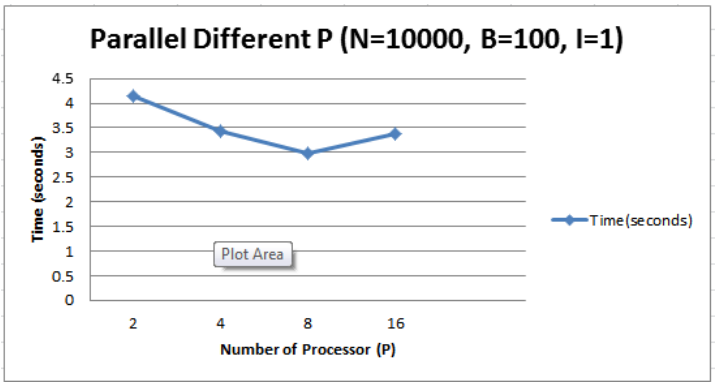

Figure 15: Diagram of measurement result when N is 10000, B is 100, I is 1

Based on the result above, we found that maximum speedup is achieved when number of processor (P) is 8 and protein size (N) is 10000. Therefore, for the subsequent experiment, we will fix the number of processor to 8 and modify other parameters.

3.2.2 Find Out Optimum Blocking Size (B)

In this subsection, we analyze the performance result and find optimum blocking size (B). We fix number of processor (P) to 8, number of protein (N) to 10000 and interleave factor (I) to 1. The results are on Figure 16

Parallel Different B (N=10000, p=8, I=1)	
B	time (seconds)
1	3.651
50	3.438
100	2.768
200	2.464
500	2.709
1000	2.854
10000	5.002

Figure 16: Performance measurement result when N is 10000, P is 8, I is 1

Plotting the result into diagram as shown in Figure 17. We zoomed in the diagram in right hand side of Figure 17 so that we have clearer picture on the performance when B is less than or equal to 500.

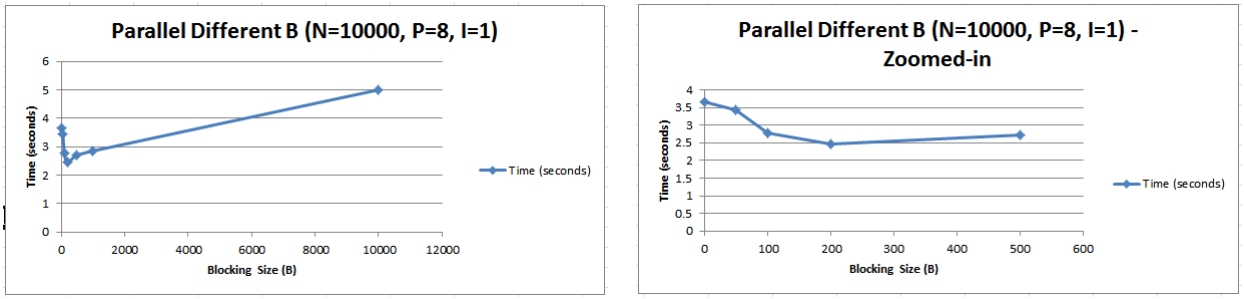

Figure 17: Diagram of measurement result when N is 10000, P is 8, I is 1

We found that optimum empirical blocking size (B) in the solution 1 is 200. And this yield in $\frac{t\_serial}{t\_parallel} = \frac{12.508}{2.464} = 5.08$ times speedup.

3.2.3 Find Out Optimum Interleave Factor (I)

Using the result from previous section in finding optimum blocking size (B), we find out most optimum I. The result is shown on Figure 18

Parallel Different I (N=10000, p=8, B=100)	
I	time (seconds)
1	2.748
2	2.613
5	3.078
10	4.192
20	4.805
30	6.249
40	6.38
50	9.696

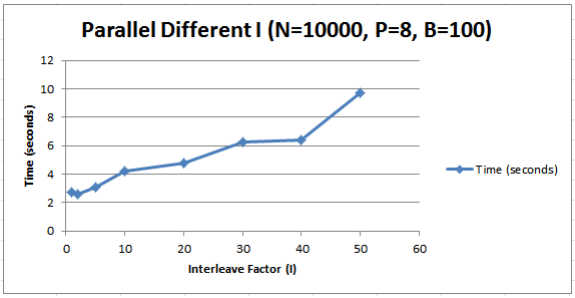

Figure 18: Diagram of measurement result when N is 10000, P is 8, B is 200

We found that optimum I is 2. And using optimum I of 2, we obtain $\frac{t\_serial}{t\_parallel} = \frac{12.508}{2.613} = 5.08$ times speedup.

3.3 Solution 2

Using similar sequential code performance result obtained during Solution 1 evalution, we measured the performance of solution 2.

3.3.1 Find Out Optimum Number of Processor (P)

The first step that we did is to observe the performance by fixing number of compared protein (N), block size (B) and set the interleave factor (I) to 1.

1. Protein size equals to 5000 (N = 5000) Block size (B) is 100, and Interleave factor (I) is 1

Parallel Different p (N=5000, B=100, I=1)	
p	time
2	4.195
4	3.131
8	2.62
16	2.384
32	2.259

Figure 19: Measurement result when N is 5000, B is 100 and I is 1

Plotting the result into diagram

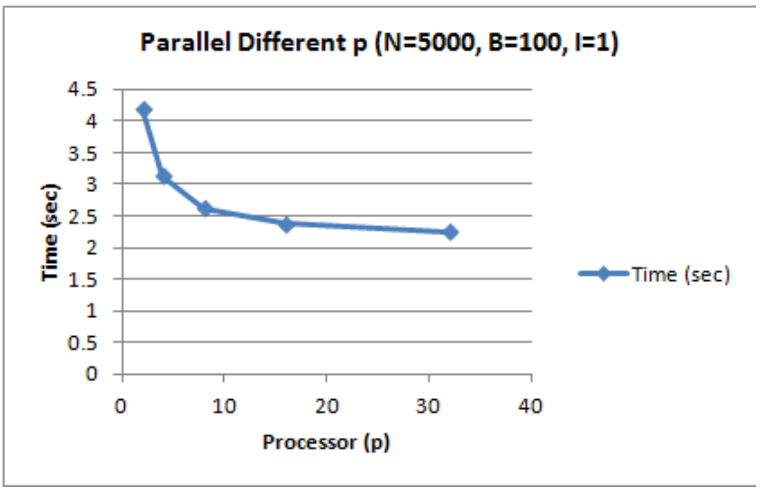

Figure 20: Diagram of measurement result when N is 5000, B is 100, I is 1

Using protein size (N) of 5000 and number of processor (P) is 32, we achieve maximum 31.55% speedup compared to existing sequential code.

2. Protein size equals to 10000 (N = 10000) Block size (B) is 100, and Interleave factor (I) is 1

Parallel Different p (N=10000, B=1000, I=1)	
p	time
2	13.852
4	9.388
8	7.388
16	6.147
32	5.669

Figure 21: Measurement result when N is 10000, B is 100 and I is 1

Plotting the result into Figure 22

Using protein size (N) equals to 10000 and number of processor (P) is 32, we achieve 54.67% speedup compared to existing sequential code.

Based on results obtained in this section, we found that parallel implementation of solution 2 achieve most speedup when the number of procesor

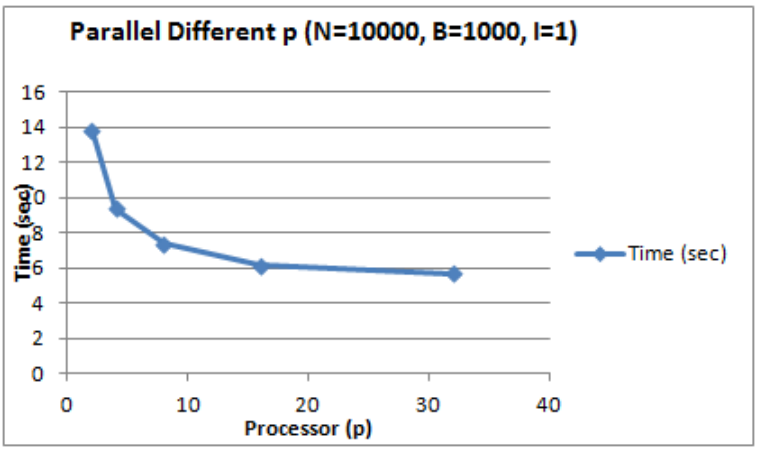

Figure 22: Diagram of measurement result when N is 10000, B is 100, I is 1

is 32. In our subsequent performance evaluation, we will fix the number of processor to 32, and observe most optimum value for other variables.

3.3.2 Find Out Optimum Blocking Size (B)

In this subsection, we analyze the performance result and find optimum blocking size (B). We fix number of processor (P) to 32, number of protein (N) to 10000 and interleave factor (I) to 1. The results are on Figure 23

Parallel Different B (N=10000, p=32, I=1)	
B	time
1	5.932
10	5.882
20	5.765
50	5.763
100	5.779
200	6.085
1000	6.647
10000	12.778

Figure 23: Performance measurement result when N is 10000, P is 32, I is 1

Plotting the result into diagram as shown in Figure 24 below

We found that optimum empirical blocking size (B) in our solution 2 is 50. Interestingly, the performance using optimum B is slightly worse com-

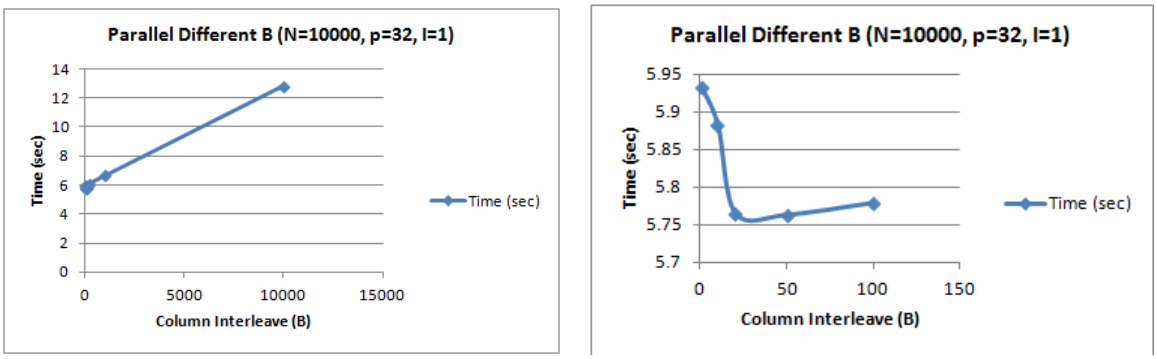

Figure 24: Diagram of measurement result when N is 10000, P is 32, I is 1

pared to the result from section 3.3.1. Using B equals to 50, we achieve 53.91% speedup compared to sequential execution, but 1.69% slower compared to result from section 3.3.1.

3.3.3 Find Out Optimum Interleave Factor (I)

Using the result from previous section in finding optimum blocking size (B), we find out most optimum I. The result is shown on Figure 25 and Figure 26

Parallel Different I (N=10000, p=32, B=50)	
I	Time
1	5.765
2	5.714
5	5.413
10	5.221
20	5.156
30	5.155
40	5.16
50	5.168
100	5.226
1000	6.999

Figure 25: Performance measurement result when N is 10000, P is 32, B is 50

We found that optimum I is 30. Another interesting point is the obtained results shows that the execution times are very close to each other when I is 10 up to 100. That means for existing configuration (N = 10000, P = 32 and B = 50), the value of I does not affect the execution time much when

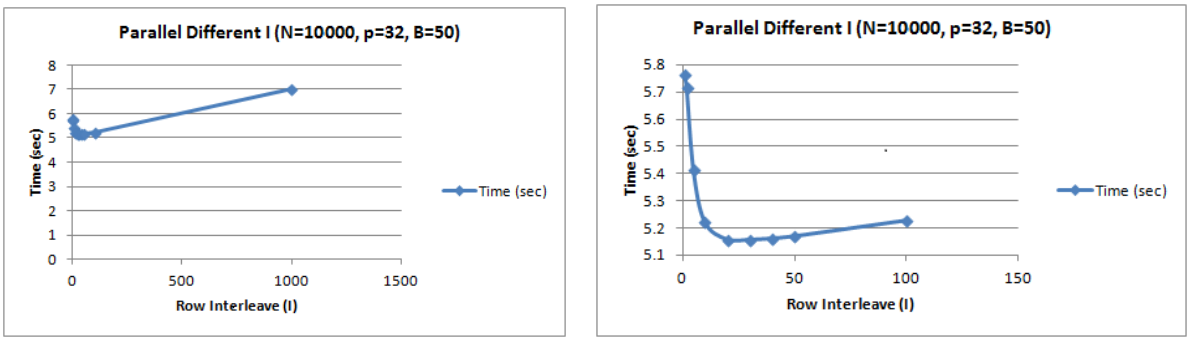

Figure 26: Diagram of measurement result when N is 10000, P is 32, B is 50

it is from 10 to 100. Practically, we can choose any I value from 10 to 100.

Using optimum I of 30, we obtain 58.79% speedup compared to sequential execution, and 10.58% speedup compared to result without using interleaving.

3.4 Putting All the Optimum Values Together

Figure 27 and Figure 28 show the result of comparing all the execution times when optimum parameters are used.

	Execution Time (seconds)		
	Solution1	Improved Solution 1	Solution2
Optimum P	2.47	2.977	5.669
Optimum B	2.401	2.464	5.763
Optimum B & I	2.625	2.613	5.155

Figure 27: Putting all of them together

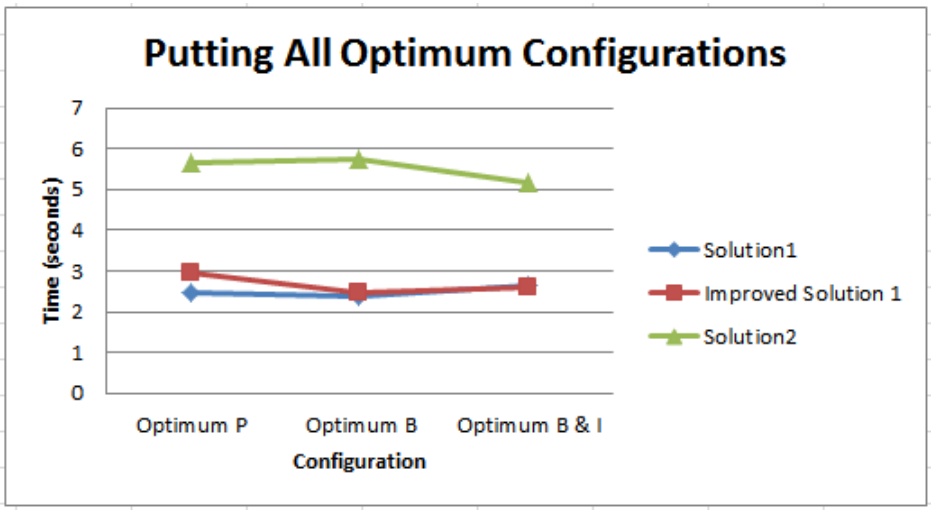

Figure 28: Putting all of them together - the plot

Improved solution 1 has slightly more execution time compared to the original solution 1. The achieved results in improved solution 1 for time performance of the developed models include not only cost of the main part (the interleave loop) but also all piggyback communication like initial broadcast and final gather. Therefore, the result is pretty close to original solution 1.

3.5 Testing with different GAP penalties

Using optimum blocking size (B) of 50, optimum interleave factor (I) of 30 and protein size of 10000, we tried to find out the result with different gap penalties. The result is shown on Figure 29

Parallel Different I (N=10000, p=32, B=50, I = 10)	
Delta	Time
-4	5.221
-2	5.24
-1	5.22
0	5.2
1	5.28
2	5.16
4	5.223

Figure 29: Testing with different gap penalties

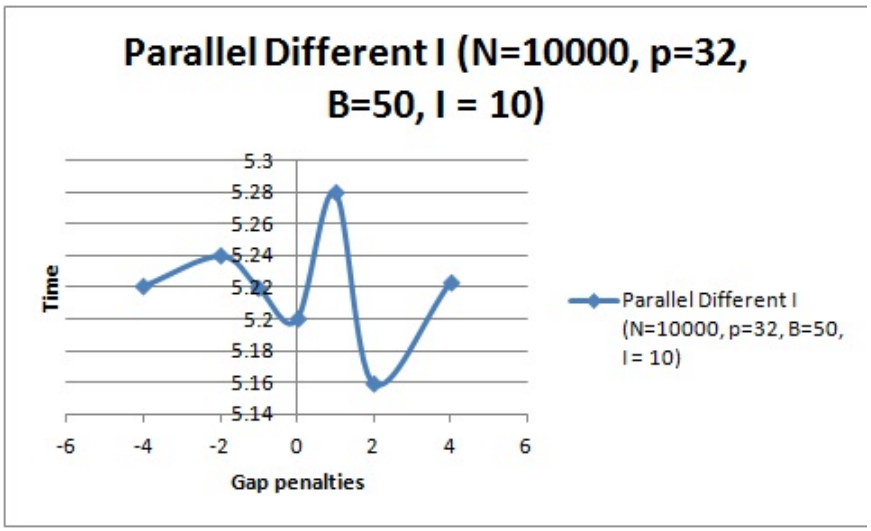

Figure 30: gap penalty vs Time

We found that there was no effect or very minor effect of changing gap penalties on the over all execution time of the implementations.

4 Conclusions

We successfully implemented three different solutions of the Smith-Waterman Algorithm. Initially we provided a solution using Scatter and Gather. We found that first version of solution 1 exhibits MPI_barrier's property of blocking all process at certain point. In general MPI_Gather doesn't have such property but for our pipelining realization, where each processes are dependent from each other, each process waits till master will be able to send the data. d realization was proposed. Therefore we optimized our first implementation so that it does not have the aforementioned MPI_barrier property. In the improved version, each process allocates enough memory for all chunks to store results from interleave stage and final gather will be invoked after all calculation work is completed. The second implementation used primitive Send and Receive method provided by MPI.

For all the implementation we did evaluation and Testing on the Altix machine and empirically find out Optimum B and I. We created performance model for both the implementations using two different interconnection networks i.e. Linear and 2D-Mesh. We also calculated optimum B and I by using derivative.

We tested our implementations for different values of B,I,p and DELTA. Factor p which is related with the processor has the major effect on the execution time. Increasing number of processor decreases the execution time of the problem. Factor B also improves the performance of the code as shown in the result. DELTA has no effect on the execution time of the implementations. We also found that execution time has certain deviation so the choice of optimal parameter is very tricky

APPENDIX

A Source Code Compilation

We created Makefile to automate the compiling process. To compile the source code that we created, we use this command

```
make
```

To remove the executables that created by compilation process, we use this command

```
make clean
```

Here is the content of the makefile

```
CXX      = icc

all: protein_free_par

clean:
        rm protein_free_par

protein_free_par: proteinFree.cpp
        ${CXX} proteinFree.cpp -o protein_free_par -lmpi
```

B Execution on ALTIX

We used Slurm+MOAB utility to submit the job at Altix Machine for execution of the code. Following is the script we used for submitting the job to the Slurm.

```
#!/bin/bash
        # @ job_name = test
        # @ initialdir = .
        # @ output = mpi_%j.out
        # @ error = mpi_%j.err
        # @ total_tasks = 4
        # @ wall_clock_limit = 00:02:00

        time mpirun -np 4 ./protein_free_par a_500k b_500k data.
            score 1 5000 100 1
```

To execute the script we used mnsubmit command.

```
mnsubmit script
```

You can find our script on below mentioned directory.

```
/home/cursos/ampp/ampp03/Documents/AMPP-Final/ProteinFree/script
```

C Timing diagram for Blocking technique in Solution 2

Figure 31: Performance Model Solution 2

D Timing diagram for Blocking-and-Interleave technique in Solution 2

Figure 32: Performance Model with Interleave

References

[1] Jun Zhang, *Chapter 5: Basic Communication Operations*. University of Kentucky, Lexington, 2010.