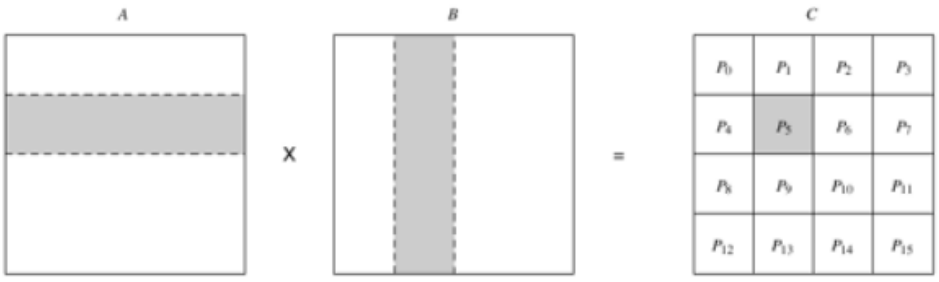
ECE 420 Parallel and Distributed Programming Assignment 3

Instructor: Di Niu
Email: dniu@ualberta.ca
Department of Electrical and Computer Engineering
University of Alberta
Due date: see the schedule posted on course website

Note: this assignment provides some sample questions that are representative of the questions to appear in exams, although questions in exams will be asked in a more formal and rigorous way. The marking of this assignment is largely based on efforts. Solutions will be posted after the due date.

1. Consider the problem of multiplying two $n \times n$ dense, square matrices A and B to yield the product matrix $C = A \times B$. The simple 2-D partitioning to this problem is described as follows. Partition the two $n \times n$ matrices A and B, respectively, into p blocks (submatrices) $A_{i,j}$ and $B_{i,j}$ $(0 \leq i, j < p)$, each of size $(n/p^{0.5}) \times (n/p^{0.5})$. Process $P_{i,j}$ initially stores $A_{i,j}$ and $B_{i,j}$ and computes block $C_{i,j}$ of the result matrix. Computing submatrix $C_{i,j}$ requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $0 \leq k < p^{0.5}$, as shown in the figure below.



Answer the following questions:

1) What's the serial time complexity of this algorithm, i.e., the time to run the algorithm above on a single processor?

2) Assume that t_s is the latency or the startup time for each data transfer, and t_w the per-word transfer time. Derive an expression for the parallel run time of the procedure above on p processes.

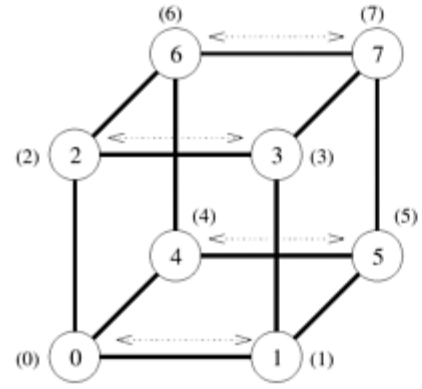
3) Under what values of p is this parallel algorithm cost-optimal?

2. Assume that t_s is the latency or the startup time for each data transfer, and t_w the per-word transfer time. Assume that p processes participate in the operation and the data to be broadcast or reduced contains m words. Analyze the time of one-to-all broadcast and all-to-one reduction.

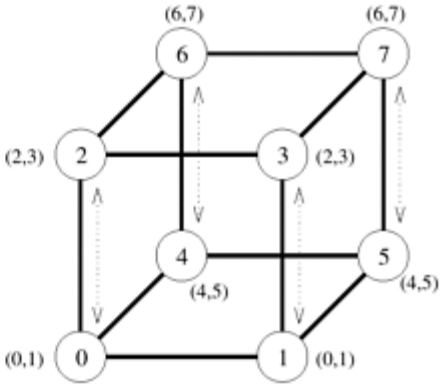
3. The following algorithm performs an all-to-all broadcast on a d-dimensional hypercube.

```
procedure ALL_TO_ALL_BC_HCUBE(my_id, my_msg, d, result)
begin
   result := my_msg;
   for i := 0 to d - 1 do
       partner := my id XOR 2^i;
       send result to partner;
       receive msg from partner;
       result := result ∪ msg;
   endfor;
end ALL_TO_ALL_BC_HCUBE
```

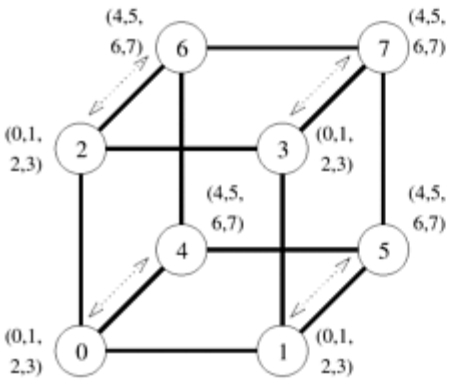
And the following figure illustrates the algorithm procedure on an 8-node (3D) hypercube. On a p-node hypercube, show that the completion time of this algorithm is $t_s \log p + t_w m (p\text{-}1)$.



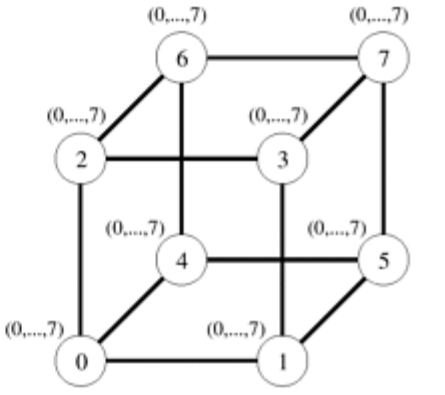
(a) Initial distribution of messages



(b) Distribution before the second step



(c) Distribution before the third step



(d) Final distribution of messages

4. Suppose we use MPI_Send and MPI_Recv to do the send and receive operations in two processes performed in the following order:

Process 0	Process 1
Send Data to Process 1	Send Data to Process 0
Receive from Process 1	Receive from Process 0

Is the program safe? If not, suggest at least three ways to make the program safe.

5. Derive the *exact* cost of the following parallel programs for matrix-vector multiplication $x = A \times b$ using p processes. A is n by n. Assume n is both a multiple of p and a multiple of $p^{0.5}$. Assume that t_s is the latency or the startup time for each data transfer, t_w the per-word transfer time, and t_c is the time to perform each multiplication-addition.

1) 1-D row-wise partitioning. Assume the pointer a points to the locally stored n/p rows of A, and the pointer b points to the corresponding locally stored n/p elements of b.

```
 1   RowMatrixVectorMultiply(int n, double *a, double *b, double *x,
 2                           MPI_Comm comm)
 3   {
 4     int i, j;
 5     int nlocal;         /* Number of locally stored rows of A */
 6     double *fb;         /* Will point to a buffer that stores the entire vector b
 7     int npes, myrank;
 8     MPI_Status status;
 9
10     /* Get information about the communicator */
11     MPI_Comm_size(comm, &npes);
12     MPI_Comm_rank(comm, &myrank);
13
14     /* Allocate the memory that will store the entire vector b */
15     fb = (double *)malloc(n*sizeof(double));
16
17     nlocal = n/npes;
18
19     /* Gather the entire vector b on each processor using MPI's ALLGATHER operat
20     MPI_Allgather(b, nlocal, MPI_DOUBLE, fb, nlocal, MPI_DOUBLE,
21         comm);
22
23     /* Perform the matrix-vector multiplication involving the locally stored sub
24     for (i=0; i<nlocal; i++) {
25       x[i] = 0.0;
26       for (j=0; j<n; j++)
27         x[i] += a[i*n+j]*fb[j];
28     }
29
30     free(fb);
31   }
```

2) 1-D column-wise partitioning. Assume the pointer a points to the locally stored n/p columns of A, and the pointer b points to the corresponding locally stored n/p elements of b.

```
 1   ColMatrixVectorMultiply(int n, double *a, double *b, double *x,
 2                           MPI_Comm comm)
 3   {
 4     int i, j;
 5     int nlocal;
 6     double *px;
 7     double *fx;
 8     int npes, myrank;
14
15     nlocal = n/npes;
16
17     /* Allocate memory for arrays storing intermediate results. */
18     px = (double *)malloc(n*sizeof(double));
19     fx = (double *)malloc(n*sizeof(double));
20
21     /* Compute the partial-dot products that correspond to the local columns of
22     for (i=0; i<n; i++) {
23       px[i] = 0.0;
24       for (j=0; j<nlocal; j++)
25         px[i] += a[i*nlocal+j]*b[j];
26     }
27
28     /* Sum-up the results by performing an element-wise reduction operation */
29     MPI_Reduce(px, fx, n, MPI_DOUBLE, MPI_SUM, 0, comm);
30
31     /* Redistribute fx in a fashion similar to that of vector b */
32     MPI_Scatter(fx, nlocal, MPI_DOUBLE, x, nlocal, MPI_DOUBLE, 0,
33         comm);
34
35     free(px); free(fx);
36   }
```

3) 2-D Partitioning. Assume the pointer *a* points to the locally stored sub-matrix of *A*. Initially, each process in the first column of the process grid holds a part of *b*. And for these processes, the pointer *b* points to the locally stored portions of the vector *b*. For other processes, the pointer *b* points to empty memory cells initially.

```
void MatrixVectorMultiply_2D(int n, double *a, double *b, double *x,MPI_Comm
comm)
{
    int ROW=0, COL=1; /* Improve readability */
    int i, j, nlocal;
    double *px; /* Will store partial dot products */

    int npes, dims[2], periods[2], keep_dims[2];
    int myrank, my2drank, mycoords[2];
    int other_rank, coords[2];

    MPI_Status status;
    MPI_Comm comm_2d, comm_row, comm_col;

    MPI_Comm_size(comm, &npes);
    MPI_Comm_rank(comm, &myrank);

    dims[ROW]= dims[COL] = sqrt(npes);

    nlocal = n/dims[ROW];

    px = malloc(nlocal*sizeof(double));
    periods[ROW] = periods[COL] = 1;

    //Create a 2D Cartesian topology and get the rank and coordinates of the
process
    MPI_Cart_create(comm, 2, dims, periods, 1, &comm_2d);
    MPI_Comm_rank(comm_2d, &my2drank); /* Get my rank in the new topology */
    MPI_Cart_coords(comm_2d, my2drank, 2, mycoords); /* Get my coordinates */

    /* Create the row-based sub-topology*/
    keep_dims[ROW] = 0;
    keep_dims[COL] = 1; /* Column is still connected*/
    MPI_Cart_sub(comm_2d, keep_dims, &comm_row);
    /* Create the column-based sub-topology*/
    keep_dims[ROW] = 1;
    keep_dims[COL] = 0;
    MPI_Cart_sub(comm_2d, keep_dims, &comm_col);

    /* Redistribute the b vector. The vector b is in the first column*/
    /* Step 1. The processes along the 0th column send their data to the
diagonal processes.
    if (mycoords[COL] == 0 && mycoords[ROW] != 0) {
        coords[ROW] = mycoords[ROW];
        coords[COL] = mycoords[ROW];
        MPI_Cart_rank(comm_2d, coords, &other_rank);
        MPI_Send(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d);
    }
    if (mycoords[ROW] == mycoords[COL] && mycoords[ROW] != 0) {
        coords[ROW] = mycoords[ROW];
        coords[COL] = 0;
        MPI_Cart_rank(comm_2d, coords, &other_rank);
        MPI_Recv(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d, &status);
    }
```

```
    /* Step 2. The diagonal processes perform a column-wise broadcast*/
    coords[0] = mycoords[COL];
    MPI_Cart_rank(comm_col, coords, &other_rank);
    MPI_Bcast(b, nlocal, MPI_DOUBLE, other_rank, comm_col);

    for (i=0; i<nlocal; i++) {
        px[i] = 0.0;
        for (j=0; j<nlocal; j++){
            px[i] += a[i*nlocal+j]*b[j];
        }
    }

    /* Perform the sum-reduction along the rows to add up the partial dot
products */
    coords[0] = mycoords[ROW];
    MPI_Cart_rank(comm_row, coords, &other_rank);
    MPI_Reduce(px, x, nlocal, MPI_DOUBLE, MPI_SUM, other_rank, comm_row);

    MPI_Comm_free(&comm_2d); /* Free up communicator */
    MPI_Comm_free(&comm_row); /* Free up communicator */
    MPI_Comm_free(&comm_col); /* Free up communicator */

    free(px);
}
```

4) An alternative implementation to 1-D column-wise partitioning in 2) will be to use `MPI_Allreduce` to perform the required reduction operation and then have each process copy the locally stored elements of vector *x* from the vector `fx`. What will be the cost of this implementation?

6. A popular serial algorithm for sorting an array of n elements whose values are uniformly distributed over an interval [a, b] is the bucket sort algorithm. In this algorithm, the interval [a, b] is divided into p equal-sized subintervals referred to as buckets. The algorithm first places each element in an appropriate bucket. Since the n elements are uniformly distributed over the interval [a, b], the number of elements in each bucket is roughly n/p. The algorithm then sorts the elements in each bucket, yielding a sorted sequence. The following is an MPI program implementing a parallel bucket sort algorithm:

```
/* parallel_bucket_sort.c -- sort a list of evenly distributed numbers.
 * The numbers are randomly generated and are evenly distributed in
 * the range between 0 and 2*n
 *
 * Input: n, the number of numbers
 *
 * Output: the sorted list of numbers
 *
 * Note:  Arrays are allocated using malloc.  A single large array
 *        is used.
 *
 * Uses the bucket sort and selection sort algorithms.
 */
#include <stdio.h>
#include <stdlib.h>    /* for random function */
#include "mpi.h"

void Make_numbers(long int [], int, int, int);
void Sequential_sort(long int [], int);
int  Get_minpos(long int [], int);
void Put_numbers_in_bucket(long int [], long int [], int, int, int, int);

main(int argc, char* argv[]) {
    long int  * big_array;
    long int  * local_array;
    int       n=80;   /* default is 80 elements to sort */
    int       n_bar;  /* = n/p */
    long int  number;
    int       p;
    int       my_rank;
    int       i;
    double    start, stop;   /* for timing */

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    n = atoi(argv[1]);  /* first parameter is the number of numbers */

    if (my_rank == 0) {

      /* check if parameters are valid */
```

```
      if (n%p != 0) {
            fprintf(stderr,"The number of processes must evenly divide total
                  elements.\n");
            MPI_Abort( MPI_COMM_WORLD, 2 );
            exit(1);
      }

      /* make a big input array */
      big_array = malloc(n*sizeof(long int));
      printf("\nTotal elements = %d; Each process sorts %d elements.\n",
           n, n/p);
      Make_numbers(big_array, n, n/p, p);
      start = MPI_Wtime();  /* Time measurement */
    }

    n_bar = n/p;

    local_array = malloc(n_bar*sizeof(long int));

    Put_numbers_in_bucket(big_array, local_array, n, n_bar, p,
        my_rank);

    Sequential_sort(local_array, n_bar);

    MPI_Gather(local_array, n_bar, MPI_LONG,
             big_array,   n_bar, MPI_LONG, 0, MPI_COMM_WORLD);

    stop = MPI_Wtime();

    if (my_rank==0) {
      printf("\nAfter sorting:\n");
      for(i=0; i<n; i++) printf("%7ld %c", big_array[i],
            i%8==7 ? '\n'  : ' ');
      printf("\n\nTime to sort using %d processes = %lf msecs\n",
           p, (stop - start)/0.001);
    }

    free(local_array);
    if (my_rank==0) free(big_array);
    MPI_Finalize();

    }  /* main */


/*************************************************************/
void Make_numbers(long int  big_array[]  /* out */,
              int       n            /* in  */,
              int       n_bar        /* in  */,
              int       p            /* in  */)
{
  /* Puts numbers in "buckets" but we can treat it otherwise  */
    int       i, q;
    MPI_Status status;
```

```
    printf("Before sorting:\n");
    for (q = 0; q < p; q++) {
      printf("\nP%d: ", q);
      for (i = 0; i < n_bar; i++) {
        big_array[q*n_bar+i] =  random() % (2*n/p) + (q*2*n/p); /* Assuming
            the range of elements in big_array is [0, 2n]*/

        printf("%7ld %s", big_array[q*n_bar+i], i%8==7 ? "\n    " :
            " ");
      }
      printf("\n");
    }
    printf("\n");
}  /* Make_numbers */



/*****************************************************************/
void Sequential_sort(long int array[]   /* in/out */,
                 int  size          /* in     */)
{
  /* Use selection sort to sort a list from smallest to largest */
  int      eff_size, minpos;
  long int temp;

  for(eff_size = size; eff_size > 1; eff_size--) {
    minpos = Get_minpos(array, eff_size);
    temp = array[minpos];
    array[minpos] = array[eff_size-1];
    array[eff_size-1] = temp;
  }
}

/* Return the index of the smallest element left */
int Get_minpos(long int array[], int eff_size)
{
  int i, minpos = 0;

  for (i=0; i<eff_size; i++)
    minpos = array[i] > array[minpos] ? i: minpos;
  return minpos;
}


/*****************************************************************/
void Put_numbers_in_bucket(long int  big_array[]   /* in */,
                     long int  local_array[] /* out */,
                     int       n             /* in  */,
                     int       n_bar         /* in  */,
                     int       p             /* in  */,
                     int       my_rank       /* in  */)
{
  /* Assume that numbers in big_array are evenly distributed at root,
```

```
     but are unsorted.  Send numbers to the process (bucket that should have
      them. This version uses unsafe messaging and may fail in some cases!!
      */

  int       i, q, bucket;
  MPI_Status status;

  if (my_rank == 0) {
    for (i=0; i<n; i++) {
      bucket = big_array[i]/(2*n_bar);  /* Assuming the range of the
            elements in big_array is [0, 2n] */
      MPI_Send(&big_array[i], 1, MPI_LONG, bucket,
             0, MPI_COMM_WORLD);
      /*      printf("P%d:%ld ", bucket, big_array[i]);     */
    }
    /*    printf("\n"); */
  }
  for (i=0; i<n_bar; i++) {
    MPI_Recv(&local_array[i], 1, MPI_LONG, MPI_ANY_SOURCE,
           0, MPI_COMM_WORLD, &status);
  }
}
```

1) Assume the number of processes can evenly divide the total number of elements in `big_array`. Given n elements and p processes, each handling a different bucket, analyze the parallel running time complexity and cost of the above MPI program, between the time `start` and `stop`, ignoring the time of generating `big_array`. Note that selection sort has $O(n^2)$ time complexity when sorting n numbers.

2) Under what value of p will the above program be cost-optimal? Assume that the best sequential algorithm you can use is the serial bucket sort with selection sort applied in each bucket.