CSE 4351/5351 Notes 3: Elementary Message-based Programming Using MPI

NO SHARED MEMORY IN MPI!!!

Documentation for MPICH available from `http://reptar.uta.edu/cse4351.html`

> Function reference card (`http://reptar.uta.edu/NOTES4351/functions.pdf`)
> User's guide (`http://reptar.uta.edu/NOTES4351/guide.pdf`)

Sample `.bash_profile` (note that MPI must be added to your $PATH):

```
[cs43bw@mustard cs43bw]$ cat .bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=.:$PATH:/usr/local/mpi/bin:$HOME/bin
ENV=$HOME/.bashrc
USERNAME=""

export USERNAME ENV PATH
```

man pages: `man` <MPI_function_name>

Compiling: Use `mpicc` instead of `cc`

Executing: Use `mpirun -np` <number of processes> `-machinefile` <machinefile name> <executable name> <executable command line options>

> <machinefile name> contains the names of systems to be used, one per line (a name may appear several times)

Note: You will need a `.rhosts` file in your root directory of the following form. Replace ''`bob`'' with your login id. The protection bits for `.rhosts` must be set *exactly* as shown below (use `chmod u-x,o-rx,g-rx .rhosts` to set these).

```
[bob@mustard NOTES03]$ cat ~/.rhosts
ketchup bob
mustard bob
shiva bob
[bob@mustard NOTES03]$ ls -l ~/.rhosts
-rw-------   1 bob      bob             9 Jun 26  1998 /home/bob/.rhosts
```

Example machinefile: Processes with ranks 0 and 1 will be on mustard. Processes with ranks 2 and 3 are on ketchup.

```
[cs43bw@mustard LAYOUT]$ cat machines
mustard
mustard
ketchup
ketchup
```

Note: If there are more processes than lines in the machinefile, then processes are assigned to machines in a cyclic fashion.

Note: Executions using ketchup, mustard and shiva will require the program to be compiled on all three machines under the same directory path. It is convenient to copy files across the network using `rcp` instead of `ftp`. `rcp` cannot be used until `.rhosts` files have been correctly set up

`diameterM.c`: Determines the 2 points in a set that are farthest apart. Based on computing n(n-1)/2 point distances in an interleaved fashion (to "load balance") taking advantage of symmetry of euclidean distances. Assuming that 4 processors are used, each processor computes the respective distances and saves the farthest point distances. These are then sent to processor 0 to compute the overall answer.

Processor 0:
- Point 0 to points 1, 2, 3, . . ., n-1
- Point 4 to points 5, 6, 7, . . ., n-1
- Point 8 to points 9, 10, 11, . . . , n-1
- Point 12 to points 13, 14, 15, . . . , n-1
- . . .

Processor 1:
- Point 1 to points 2, 3, 4, . . ., n-1

Point 5 to points 6, 7, 8, . . ., n-1
Point 9 to points 10, 11, 12, . . ., n-1
Point 13 to points 14, 15, 16, . . ., n-1
. . .

Processor 2: Point 2 to points 3, 4, 5, . . ., n-1
Point 6 to points 7, 8, 9, . . ., n-1
Point 10 to points 11, 12, 13, . . ., n-1
Point 14 to points 15, 16, 17, . . ., n-1
. . .

Processor 3: Point 3 to points 4, 5,6, . . ., n-1
Point 7 to points 8, 9, 10, . . ., n-1
Point 11 to points 12, 13, 14, . . ., n-1
Point 15 to points 16, 17, 18, . . ., n-1
. . .

MPI functions called - in order of first use during execution:

`MPI_Init`:	initializes for MPI and indicates command line arguments
`MPI_Comm_size`:	retrieves number of processes in communicator
`MPI_Comm_rank`:	determines this process's rank within its communicator
`MPI_Bcast`:	broadcasts value(s) to all processes in communicator
`MPI_Send`:	sends value(s) to a particular process using a message with specified integer tag
`MPI_Recv`:	receives values from a particular process (or use `MPI_ANY_SOURCE`) and a particular tag (or use `MPI_ANY_TAG`)
`MPI_Finalize`:	termination for MPI

```
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "mpi.h"
/*Brute force determination of two points giving the diameter
of a random set of 2-d points*/

#define sqr(x) ((x)*(x))

int zero=0;
int numProcesses,rank;
int type,sender;
MPI_Status status;

int x[50000],y[50000];

int generateRandom(minRange,maxRange)
int minRange,maxRange;
{
/* returns integer in range minRange <= x <= maxRange*/
return minRange+abs(rand()) % (maxRange-minRange+1);
}

float CPUtime()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

float elapsedTime()
{
  return (10000.0*times(NULL))/CLOCKS_PER_SEC;
}

main(int argc, char** argv)
{
int n;
int seed,i,j,k,diameterI,diameterJ;
float diameter,dist;
float startCPU,startTime;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numProcesses);
```

```
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (!rank)
{
  printf("Enter n\n");
  scanf("%d",&n);
  printf("Enter seed\n");
  scanf("%d",&seed);
  srand(seed);
  for (i=0;i<n;i++)
  {
    x[i]=generateRandom(0,1000);
    y[i]=generateRandom(0,1000);
  }
}
MPI_Barrier(MPI_COMM_WORLD);
startCPU=CPUtime();
startTime=elapsedTime();
MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(x,n,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(y,n,MPI_INT,0,MPI_COMM_WORLD);
diameter=0.0;
/*Divide work evenly & take advantage of symmetry of distances*/
for (i=rank;i<n;i+=numProcesses)
  for (j=i+1;j<n;j++)
  {
    dist=sqrt((double)(sqr(x[i]-x[j])+sqr(y[i]-y[j])));
    if (dist>diameter)
    {
      diameter=dist;
      diameterI=i;
      diameterJ=j;
    }
  }
type=4;
MPI_Send(&diameter,1,MPI_FLOAT,0,type,MPI_COMM_WORLD);
MPI_Send(&diameterI,1,MPI_INT,0,type,MPI_COMM_WORLD);
MPI_Send(&diameterJ,1,MPI_INT,0,type,MPI_COMM_WORLD);
if (!rank)
{
  MPI_Recv(&diameter,1,MPI_FLOAT,0,type,MPI_COMM_WORLD,&status);
  MPI_Recv(&diameterI,1,MPI_INT,0,type,MPI_COMM_WORLD,&status);
  MPI_Recv(&diameterJ,1,MPI_INT,0,type,MPI_COMM_WORLD,&status);
  for (k=1;k<numProcesses;k++)
  {
    MPI_Recv(&dist,1,MPI_FLOAT,k,type,MPI_COMM_WORLD,&status);
    MPI_Recv(&i,1,MPI_INT,k,type,MPI_COMM_WORLD,&status);
    MPI_Recv(&j,1,MPI_INT,k,type,MPI_COMM_WORLD,&status);
    if (dist>diameter)
    {
      diameter=dist;
      diameterI=i;
      diameterJ=j;
    }
  }
  printf("diameter=%f for (%d,%d) & (%d,%d)\n",diameter,
    x[diameterI],y[diameterI],x[diameterJ],y[diameterJ]);
}
printf("%d: CPU %f\n",rank,CPUtime()-startCPU);
printf("%d: Elapsed %f\n",rank,elapsedTime()-startTime);
MPI_Finalize();
}
```

Timings:

Single Process (ketchup):

```
[bob@ketchup NOTES03]$ mpirun -np 1 diameterM
Enter n
10000
Enter seed
111
diameter=1405.022827 for (0,997) & (995,5)
0: CPU 20.289999
```

```
0: Elapsed 20.281250
```

Two Processes (ketchup):

```
[bob@ketchup NOTES03]$ mpirun -np 2 diameterM
Enter n
10000
Enter seed
111
diameter=1405.022827 for (0,997) & (995,5)
0: CPU 10.150000
0: Elapsed 10.156250
1: CPU 10.150000
1: Elapsed 10.156250
```

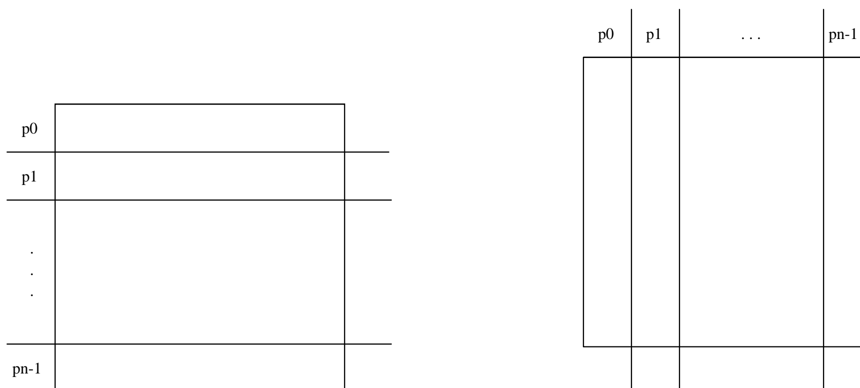
Two Processes (one on ketchup, one on mustard)

```
[bob@ketchup NOTES03]$ mpirun -np 2 -machinefile machine2 diameterM
Enter n
10000
Enter seed
111
diameter=1405.022827 for (0,997) & (995,5)
0: CPU 10.210000
0: Elapsed 10.218750
1: CPU 10.230000
1: Elapsed 10.250000
```

Four Processes (two on ketchup, two on mustard)

```
[bob@ketchup NOTES03]$ mpirun -np 4 -machinefile machine4 diameterM
Enter n
10000
Enter seed
111
diameter=1405.022827 for (0,997) & (995,5)
0: CPU 5.170000
0: Elapsed 5.234375
2: CPU 5.230000
2: Elapsed 5.234375
1: CPU 5.210000
1: Elapsed 5.000000
3: CPU 5.210000
3: Elapsed 5.250000
```

`warshallM.c`: Determines for each pair of vertices in a graph the path of successors connecting those vertices. Two methods of partitioning the array are demonstrated. Both methods take advantage of the fact that row i and column i do not change during round and thus perform a broadcast from the processor posessing the row/column needed by the other processors at the beginning of round i. The first method partitions the rows to the processors in a contiguous fashion. The second method partitions the columns similarly in a contiguous fashion. In both cases, the results are verified against the sequential version.




```
/* Distributed Warshall's algorithm for MPI */
/* Converted from nCUBE code, 9/8/98, BPW */
#include <stdio.h>
#include <sys/time.h>
```

```
#include <sys/resource.h>
#include "mpi.h"

int zero=0;
int numProcesses,rank;
int type,sender;
MPI_Status status;

int **succ,**result,**rowPartition,**colPartition,n;

allocate(size1, size2, matrix)
int size1,size2;
int ***matrix;
{
  int i;

  if (!(*matrix = (int **) malloc(size1*sizeof(int *))))
    {
      printf("Allocate failed\n");
      exit(2);
    }
  for(i = 0; i < size1; i++)
    if (!((*matrix)[i]=(int *) malloc(size2*sizeof(int))))
    {
      printf("Allocate failed\n");
      exit(3);
    }
}

deallocate(size1,matrix)
int size1;
int **matrix;
{
  int i;

  for(i = 0; i<size1; i++)
    free(matrix[i]);
  free(matrix);
}

#define generateRandom(minRange,maxRange) \
  (minRange)+abs(rand()) % ((maxRange)-(minRange)+1)

float CPUtime()
{
  struct rusage rusage;
  getrusage(RUSAGE_SELF,&rusage);
  return rusage.ru_utime.tv_sec+rusage.ru_utime.tv_usec/1000000.0
    + rusage.ru_stime.tv_sec+rusage.ru_stime.tv_usec/1000000.0;
}

float elapsedTime()
{
  return (10000.0*times(NULL))/CLOCKS_PER_SEC;
}

void printPaths(n,succ)
int n,**succ;
{
int i,j,k;

for (i=0;i<n;i++)
  for (j=0;j<n;j++)
    if (succ[i][j]==(-1))
      printf("No path from %d to %d\n",i,j);
    else
    {
      printf("Path %d ",i);
      k=succ[i][j];
      while (k!=j)
      {
        printf("%d ",k);
        k=succ[k][j];
      }
```

```
      printf("%d\n",j);
    }
}

void warshallSeq(n,succ)
int n,**succ;
{
int i,j,k;
float startCPU,startTime;

startCPU=CPUtime();
startTime=elapsedTime();
for (j=0;j<n;j++)
  for (i=0;i<n;i++)
    if (succ[i][j]!=(-1))
      for (k=0;k<n;k++)
        if (succ[i][k]==(-1) && succ[j][k]!=(-1))
          succ[i][k]=succ[i][j];
printf("%d: CPU sequential %f\n",rank,CPUtime()-startCPU);
printf("%d: Elapsed sequential %f\n",rank,elapsedTime()-startTime);
}

void warshallRow(n,succ)
int n,**succ;
{
int i,j,k;
int row[10000];
float startCPU,startTime;

MPI_Barrier(MPI_COMM_WORLD);
startCPU=CPUtime();
startTime=elapsedTime();
if (!rank)
{
  printf("Starting warshallRow()\n",rank);
  k=0;
  for (i=0;i<numProcesses;i++)
    for (j=0;j<n/numProcesses;j++)
    {
      MPI_Send(succ[k],n,MPI_INT,i,1000,MPI_COMM_WORLD);
      k++;
    }
}

for (i=0;i<n/numProcesses;i++)
  MPI_Recv(rowPartition[i],n,MPI_INT,0,1000,MPI_COMM_WORLD,&status);
for (j=0;j<n;j++)
{
  sender=j/(n/numProcesses);
  if (sender==rank)
    for (k=0;k<n;k++)
      row[k]=rowPartition[j%(n/numProcesses)][k];
  MPI_Bcast(row,n,MPI_INT,sender,MPI_COMM_WORLD);

  for (i=0;i<n/numProcesses;i++)
    if (rowPartition[i][j]!=(-1))
      for (k=0;k<n;k++)
        if (rowPartition[i][k]==(-1) && row[k]!=(-1))
          rowPartition[i][k]=rowPartition[i][j];
}
printf("%d: CPU row partitioned %f\n",rank,CPUtime()-startCPU);
printf("%d: Elapsed row partitioned %f\n",rank,elapsedTime()-startTime);

if (!rank)
{
  printf("Verifying row results\n");
  k=0;
  for (i=0;i<numProcesses;i++)
    for (j=0;j<n/numProcesses;j++)
    {
      MPI_Send(result[k],n,MPI_INT,i,2000,MPI_COMM_WORLD);
      k++;
    }
}
```

```
for (i=0;i<n/numProcesses;i++)
{
  MPI_Recv(row,n,MPI_INT,0,2000,MPI_COMM_WORLD,&status);
  for (j=0;j<n;j++)
    if (row[j]!=rowPartition[i][j])
    {
      printf("Processor %d detects error\n",rank);
      return;
    }
}
}

void warshallColumn(n,succ)
int n,**succ;
{
int i,j,k,p;
int col[10000];
float startCPU,startTime;

MPI_Barrier(MPI_COMM_WORLD);
startCPU=CPUtime();
startTime=elapsedTime();
if (!rank)
{
  printf("Starting warshallColumn()\n",rank);
  k=0;
  for (i=0;i<numProcesses;i++)
    for (j=0;j<n/numProcesses;j++)
    {
      for (p=0;p<n;p++)
        col[p]=succ[p][k];
      MPI_Send(col,n,MPI_INT,i,3000,MPI_COMM_WORLD);
      k++;
    }
}

for (i=0;i<n/numProcesses;i++)
{
  MPI_Recv(col,n,MPI_INT,0,3000,MPI_COMM_WORLD,&status);
  for (j=0;j<n;j++)
    colPartition[j][i]=col[j];
}

for (j=0;j<n;j++)
{
  sender=j/(n/numProcesses);
  if (sender==rank)
    for (i=0;i<n;i++)
      col[i]=colPartition[i][j%(n/numProcesses)];
  MPI_Bcast(col,n,MPI_INT,sender,MPI_COMM_WORLD);
  for (i=0;i<n;i++)
    if (col[i]!=(-1))
      for (k=0;k<n/numProcesses;k++)
        if (colPartition[i][k]==(-1) && colPartition[j][k]!=(-1))
          colPartition[i][k]=col[i];
}
printf("%d: CPU column partitioned %f\n",rank,CPUtime()-startCPU);
printf("%d: Elapsed column partitioned %f\n",rank,elapsedTime()-startTime);

if (!rank)
{
  printf("Verifying column results\n");
  k=0;
  for (i=0;i<numProcesses;i++)
    for (j=0;j<n/numProcesses;j++)
    {
      for (p=0;p<n;p++)
        col[p]=result[p][k];
      MPI_Send(col,n,MPI_INT,i,4000,MPI_COMM_WORLD);
      k++;
    }
}
for (i=0;i<n/numProcesses;i++)
{
```

```
  MPI_Recv(col,n,MPI_INT,0,4000,MPI_COMM_WORLD,&status);
  for (j=0;j<n;j++)
    if (col[j]!=colPartition[j][i])
    {
      printf("Processor %d detects error\n",rank);
      return;
    }
}
}
}

main(int argc, char** argv)
{
int i,j,k,seed;
int tail,head;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numProcesses);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

if (!rank)
{
  printf("Enter # of vertices\n");
  scanf("%d",&n);
  if (n%numProcesses)
  {
    printf("Only even distributions are handled\n");
    exit(0);
  }
  allocate(n,n,&succ);
  allocate(n,n,&result);
  for (i=0;i<n;i++)
    for (j=0;j<n;j++)
      succ[i][j]=(-1);
  printf("Enter seed\n");
  scanf("%d",&seed);
  srand(seed);
  for (i=0;i<n/2;i++)
    for (j=0;j<n/2;j++)
    {
      tail=generateRandom(0,n-1);
      head=generateRandom(0,n-1);
      succ[tail][head]=head;
    }
  for (i=0;i<n;i++)
    for (j=0;j<n;j++)
      result[i][j]=succ[i][j];
  warshallSeq(n,result);
  //printPaths(n,result);
}
MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
allocate(n/numProcesses,n,&rowPartition);
allocate(n,n/numProcesses,&colPartition);
warshallRow(n,succ);
warshallColumn(n,succ);

if (!rank)
{
  deallocate(n,succ);
  deallocate(n,result);
}
deallocate(n/numProcesses,rowPartition);
deallocate(n,colPartition);
MPI_Finalize();
}
```

Timings:

Single Process (ketchup):

```
Enter # of vertices
500
Enter seed
111
```

```
0: CPU sequential 5.230000
0: Elapsed sequential 5.218750
Starting warshallRow()
0: CPU row partitioned 8.680000
0: Elapsed row partitioned 8.671875
Verifying row results
Starting warshallColumn()
0: CPU column partitioned 15.280001
0: Elapsed column partitioned 15.281250
Verifying column results
```

Two Processes (ketchup);

```
Enter # of vertices
500
Enter seed
111
0: CPU sequential 5.310000
0: Elapsed sequential 5.312500
Starting warshallRow()
0: CPU row partitioned 4.160000
0: Elapsed row partitioned 4.171875
Verifying row results
Starting warshallColumn()
0: CPU column partitioned 7.730000
0: Elapsed column partitioned 7.734375
Verifying column results
1: CPU row partitioned 4.110000
1: Elapsed row partitioned 4.171875
1: CPU column partitioned 7.680000
1: Elapsed column partitioned 7.734375
```

Two Processes (one on ketchup, one on mustard)

```
Enter # of vertices
500
Enter seed
111
0: CPU sequential 5.310000
0: Elapsed sequential 5.312500
Starting warshallRow()
0: CPU row partitioned 4.530000
0: Elapsed row partitioned 4.546875
Verifying row results
Starting warshallColumn()
0: CPU column partitioned 8.080000
0: Elapsed column partitioned 8.078125
Verifying column results
1: CPU row partitioned 4.490000
1: Elapsed row partitioned 4.500000
1: CPU column partitioned 8.000000
1: Elapsed column partitioned 8.000000
```

Four Processes (two on ketchup, two on mustard)

```
Enter # of vertices
500
Enter seed
111
0: CPU sequential 5.320000
0: Elapsed sequential 5.312500
Starting warshallRow()
0: CPU row partitioned 2.890000
0: Elapsed row partitioned 3.031250
Verifying row results
Starting warshallColumn()
0: CPU column partitioned 4.320000
0: Elapsed column partitioned 4.312500
Verifying column results
2: CPU row partitioned 2.370000
2: Elapsed row partitioned 3.046875
2: CPU column partitioned 3.790000
2: Elapsed column partitioned 4.328125
```

```
1: CPU row partitioned 2.530000
1: Elapsed row partitioned 3.000000
1: CPU column partitioned 4.010000
1: Elapsed column partitioned 4.500000
3: CPU row partitioned 2.630000
3: Elapsed row partitioned 3.000000
3: CPU column partitioned 3.970000
3: Elapsed column partitioned 4.500000
```

`hashM.c`: This example uses 2k processes with processes 0 . . . k-1 storing a distributed (‘‘server’’) hash table and processes k . . . 2k-1 generating (‘‘client’’) requests to simulate an application. By changing the MPI machine file, a variety of configurations may be used.

The code is intended only as an example, since the hash structure is very inflexible. The hash structure is based on integer keys that are mapped to processess by using the low-order bits of a key to determine the processor that stores it. The actual hash is a simple double hash. The application processes block to wait for a reply after requesting that a key be inserted or found in the table. To allow clean termination, application processes will inform the servers that they are terminating (i.e. the FINISH message). The general abstract version of this problem, *termination detection*, will be addressed in Notes 4.

Additional MPI functionality used:

`MPI_Recv` in hashServer() uses wildcards for both the sender (`MPI_ANY_SOURCE`) and tag (`MPI_ANY_TAG`). The actual source and tag are stored in the structure named status. `insertServer()` and `findServer()` use the field `status.MPI_SOURCE` when returning a success/failure flag (using `MPI_Send`) to the client that issued the request.

```
/* MPI code for distributed hashing - converted from nCUBE code */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include "mpi.h"

#define INSERT 1
#define FIND 2
#define FINISH 3

int rank,numProcesses;
int type,sender,zero=0,one=1;
MPI_Status status;

int generateRandom(minRange,maxRange)
int minRange,maxRange;
{
/* returns integer in range minRange <= x <= maxRange*/
return minRange+abs(random()) % (maxRange-minRange+1);
}

int keyCount=0;  /* Number of keys in hash structure if process is a hashServer */

void insertServer(request,hashTable)
int request,hashTable[2311];
{
int i,rehash;

i=request%2311;
rehash=(request/2311)%2311;
if (!rehash)
  rehash=1;
while (hashTable[i]!=(-1))
  if (hashTable[i]==request)
  {
    MPI_Send(&zero,1,MPI_INT,status.MPI_SOURCE,one,MPI_COMM_WORLD);
    return;
  }
  else
    i=(i+rehash)%2311;
hashTable[i]=request;
keyCount++;
MPI_Send(&one,1,MPI_INT,status.MPI_SOURCE,one,MPI_COMM_WORLD);
}

void findServer(request,hashTable)
int request,hashTable[2311];
{
int i,rehash;
```

```
i=request%2311;
rehash=(request/2311)%2311;
if (!rehash)
  rehash=1;
while (hashTable[i]!=(-1))
  if (hashTable[i]==request)
  {
    MPI_Send(&one,1,MPI_INT,status.MPI_SOURCE,one,MPI_COMM_WORLD);
    return;
  }
  else
    i=(i+rehash)%2311;
MPI_Send(&zero,1,MPI_INT,status.MPI_SOURCE,one,MPI_COMM_WORLD);
}

void hashServer()
{
int hashTable[2311];
int i,insertRequests=0,findRequests=0,finishCount=0,request;

for (i=0;i<2311;i++)
  hashTable[i]=(-1);
while (finishCount<numProcesses/2)
{
  MPI_Recv(&request,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
    MPI_COMM_WORLD,&status);
  switch (status.MPI_TAG)
  {
    case (INSERT):
      insertServer(request,hashTable);
      insertRequests++;
      break;
    case (FIND):
      findServer(request,hashTable);
      findRequests++;
      break;
    case (FINISH):
      finishCount++;
      break;
    default:
      printf("bad hashServer request\n");
      exit();
  }
}
printf("Process %3d: %4d keys %4d inserts %4d finds\n",
  rank,keyCount,insertRequests,findRequests);
}

int insertHash(x)
int x;
{
int server,result;

server=x%(numProcesses/2);
MPI_Send(&x,1,MPI_INT,server,INSERT,MPI_COMM_WORLD);
MPI_Recv(&result,1,MPI_INT,server,one,MPI_COMM_WORLD,&status);
return result;
}

int findHash(x)
int x;
{
int server,result;

server=x%(numProcesses/2);
MPI_Send(&x,1,MPI_INT,server,FIND,MPI_COMM_WORLD);
MPI_Recv(&result,1,MPI_INT,server,one,MPI_COMM_WORLD,&status);
return result;
}

void finishHash()
{
int i;
```

```
for (i=0;i<numProcesses/2;i++)
  MPI_Send(&i,0,MPI_INT,i,FINISH,MPI_COMM_WORLD);
}

void hashClient()
{
int i;

srandom(rank);
for (i=0;i<1500;i++)
  insertHash(generateRandom(1,1000000));
srandom(rank);
for (i=0;i<1500;i++)
  if (!findHash(generateRandom(1,1000000)))
  {
    printf("key missing!\n");
    abort();
  }
for (i=0;i<100;i++)
  if (findHash(generateRandom(1000001,2000000)))
  {
    printf("extra key!\n");
    abort();
  }
finishHash();
}

main(int argc, char** argv)
{
int child;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numProcesses);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (rank==0 && numProcesses%2==1)
{
  printf("number of processes must be even\n");
  exit(0);
}

if (rank>=numProcesses/2)
  hashClient();
else
  hashServer();
MPI_Finalize();
}
```

Timings: four servers and four clients, divided evenly.

Two Processors (mustard)

```
[bob@mustard NOTES03]$ time mpirun -np 8 a.out
Process   0: 1495 keys 1502 inserts 1596 finds
Process   1: 1432 keys 1437 inserts 1540 finds
Process   2: 1559 keys 1562 inserts 1660 finds
Process   3: 1495 keys 1499 inserts 1604 finds
2.48user 67.69system 5:02.52elapsed 23%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (8031major+6555minor)pagefaults 0swaps
```

Four Processors (four processes on mustard, four processes on ketchup)

```
[bob@mustard NOTES03]$ cat machines
mustard
ketchup
mustard
ketchup
mustard
ketchup
mustard
ketchup
[bob@mustard NOTES03]$ time mpirun -machinefile machines -np 8 a.out
Process   0: 1495 keys 1502 inserts 1596 finds
```

```
Process   2: 1559 keys 1562 inserts 1660 finds
Process   3: 1495 keys 1499 inserts 1604 finds
Process   1: 1432 keys 1437 inserts 1540 finds
0.43user 4.63system 0:19.81elapsed 25%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (7948major+6390minor)pagefaults 0swaps
```

Summer 2000 Lab - Merged Clients & Servers

Requirements:

1. Modify the client/server hashing program `/home/CODE/hash2M.c` to merge the client and server functionality, i.e. each process will be both a client and a server. Submit a hardcopy listing of your program. Also, send a copy of your program to `weems@uta.edu`.

2. Execute your program using various configurations, i.e. different numbers of processes and different machinefiles Submit hardcopy of your executions. You will also need to run some executions of the original program.

2. Write a brief report discussing the performance improvements gained by your program..

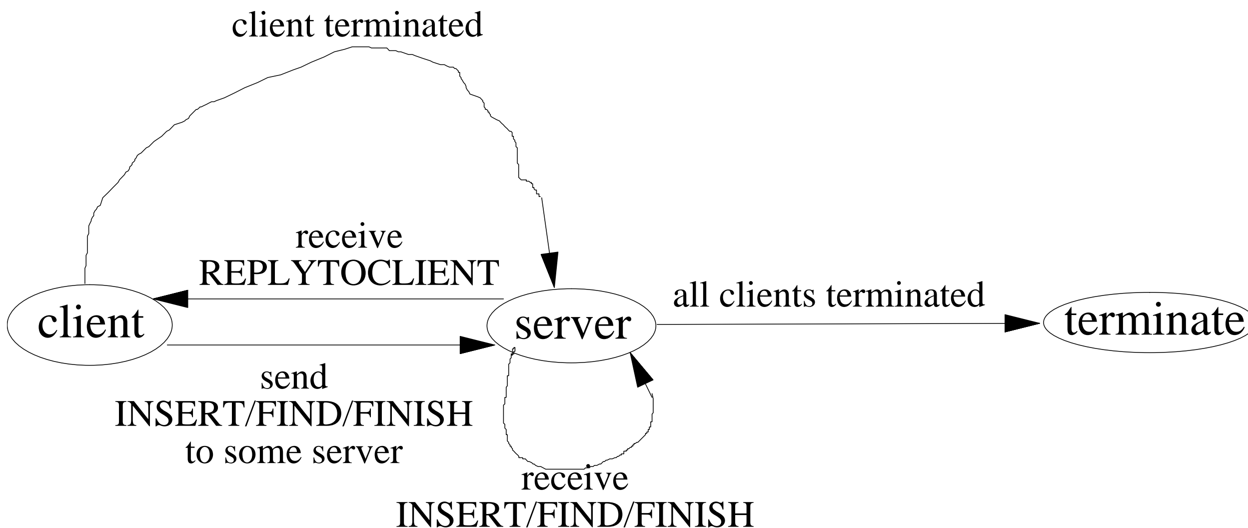
Getting Started:

1. Unlike the original program, your program may work with an odd number of processes.

2. The seeding of the random number generator will need to be changed to allow checking against the original program.

3. Hint: This program may use just one `MPI_Recv()` in `hashServer()`.

4. The output from running the original program with 2n processes should be identical (except for reordering of lines) to your program running with n processes.




```
[weems@va LABS]$ cat hashLabSum00M.c
/* MPI code for distributed hashing - converted from nCUBE code */
// Merged clients & servers together, 6/19/00 BPW
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include "mpi.h"

#define INSERT 1
#define FIND 2
#define FINISH 3
#define REPLYTOCLIENT 4

int rank,numProcesses;
int type,sender,zero=0,one=1;
MPI_Status status;
```

```
int generateRandom(minRange,maxRange)
int minRange,maxRange;
{
/* returns integer in range minRange <= x <= maxRange*/
return minRange+abs(random()) % (maxRange-minRange+1);
}

int keyCount=0;  // Number of keys in hash structure
int hashTable[2311];
int insertRequests=0,findRequests=0,finishCount=0,request;

void insertServer(request,hashTable)
int request,hashTable[2311];
{
int i,rehash;

i=request%2311;
rehash=(request/2311)%2311;
if (!rehash)
  rehash=1;
while (hashTable[i]!=(-1))
  if (hashTable[i]==request)
  {
    MPI_Send(&zero,1,MPI_INT,status.MPI_SOURCE,REPLYTOCLIENT,MPI_COMM_WORLD);
    return;
  }
  else
    i=(i+rehash)%2311;
hashTable[i]=request;
keyCount++;
MPI_Send(&one,1,MPI_INT,status.MPI_SOURCE,REPLYTOCLIENT,MPI_COMM_WORLD);
}

void findServer(request,hashTable)
int request,hashTable[2311];
{
int i,rehash;

i=request%2311;
rehash=(request/2311)%2311;
if (!rehash)
  rehash=1;
while (hashTable[i]!=(-1))
  if (hashTable[i]==request)
  {
    MPI_Send(&one,1,MPI_INT,status.MPI_SOURCE,REPLYTOCLIENT,MPI_COMM_WORLD);
    return;
  }
  else
    i=(i+rehash)%2311;
MPI_Send(&zero,1,MPI_INT,status.MPI_SOURCE,REPLYTOCLIENT,MPI_COMM_WORLD);
}

int hashServer(int clientTerminated)
{
int request;

while (!clientTerminated || finishCount<numProcesses)
{

  MPI_Recv(&request,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
    MPI_COMM_WORLD,&status);
  switch (status.MPI_TAG)
  {
    case (INSERT):
      insertServer(request,hashTable);
      insertRequests++;
      break;
    case (FIND):
      findServer(request,hashTable);
      findRequests++;
      break;
    case (FINISH):
```

```
      finishCount++;
      break;
    case (REPLYTOCLIENT):
      return request;  //Result for local client
    default:
      printf("bad hashServer request\n");
      exit();
  }
}
}

int insertHash(x)
int x;
{
int server,result;

server=x%numProcesses;
MPI_Send(&x,1,MPI_INT,server,INSERT,MPI_COMM_WORLD);
return hashServer(0);
//MPI_Recv(&result,1,MPI_INT,server,REPLYTOCLIENT,MPI_COMM_WORLD,&status);
//return result;
}

int findHash(x)
int x;
{
int server,result;

server=x%numProcesses;
MPI_Send(&x,1,MPI_INT,server,FIND,MPI_COMM_WORLD);
return hashServer(0);
//MPI_Recv(&result,1,MPI_INT,server,REPLYTOCLIENT,MPI_COMM_WORLD,&status);
//return result;
}

void finishHash()
{
int i;

for (i=0;i<numProcesses;i++)
  MPI_Send(&i,0,MPI_INT,i,FINISH,MPI_COMM_WORLD);
}

void hashClient()
{
int i;

srandom(numProcesses+rank);
for (i=0;i<1500;i++)
  insertHash(generateRandom(1,1000000));
srandom(numProcesses+rank);
for (i=0;i<1500;i++)
  if (!findHash(generateRandom(1,1000000)))
  {
    printf("key missing!\n");
    abort();
  }
for (i=0;i<100;i++)
  if (findHash(generateRandom(1000001,2000000)))
  {
    printf("extra key!\n");
    abort();
  }
finishHash();
}

main(int argc, char** argv)
{
int child,i;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numProcesses);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

```
for (i=0;i<2311;i++)
  hashTable[i]=(-1);

hashClient();

hashServer(1);

printf("Process %3d: %4d keys %4d inserts %4d finds\n",
  rank,keyCount,insertRequests,findRequests);
MPI_Finalize();
}
```

Summer 2001 Lab 2 - Stable Marriages

Goals:

1. Understanding of elementary MPI programming.
2. Understanding of termination detection.

Requirements:

1. Write an MPI program to solve small instances of the stable marriages problem. Submit a hardcopy listing of your program.
2. Execute your program on the files `sm1.dat`, `sm2.dat`, and `sm3.dat` in `/home/CODE`.

```
[weems@va NOTES03]$ cat sm3.dat
5
1 4 0 2 3
0 1 2 3 4
1 2 4 3 0
0 2 1 3 4
4 2 1 0 3
4 0 3 1 2
3 4 1 0 2
0 3 1 2 4
2 1 3 0 4
3 1 2 4 0
```

3. Since there will be no (practical) speed-up, your report should simply describe your approach to termination detection.

Getting Started:

1. The point of this exercise is to learn MPI and termination detection. Speed-up and efficiency are irrelevant.

2. The **Stable Marriage Problem (SMP)** is a fundamental combinatorial problem having a number of interesting applications, such as: assigning new MDs to residencies, assigning new lawyers to Federal clerkships, assigning students to universities, registering students for classes, and determining which children play which quarters in YMCA basketball. Even though each of these problems requires significant deviations from SMP, the simple version we will examine provides a great deal of insight and structure for addressing these problems. In SMP, we are provided with n men and n women, each one having a preference list that goes from that person's most preferable spouse down to their least preferable spouse. Our task is to pair the men and women (monogamously) so that the set of marriages is stable. A set of marriages is unstable if there is some man M and some woman W such that they prefer each other to their assigned spouses in the set of marriage pairs.

   At first glance, it may seem that a stable set of marriage pairs is not guaranteed for all instances of SMP. The Gale-Shapley algorithm is constructive proof that a solution always exists:

```
assign each person to be free, i.e. unengaged;
while some man m is free do
begin
   w := first woman on m's preference list to whom m has not yet proposed;
   if w is free then
      assign m and w to be engaged to each other;
   else if w prefers m to her current fiance' m' then
      assign m and w to be engaged and m' to be free;
   else
      w rejects m, thus m remains free
end;
output the stable matching consisting of the n engaged pairs
```

Correctness is established based on the following two observations: 1) at termination, all n women must be engaged and 2) the partial solution (i.e. the engagements) at any time is stable with respect to the marriages (i.e. preference list nodes) that have been considered. In addition, the obtained solution has the remarkable property (man optimality) that 1) all men get the best possible spouse in any stable matching (determining all solutions to SMP is much more involved and is not considered here) and 2) all women get the worst possible spouse in any stable matching . Of course, switching the roles of men and women in the algorithm would reverse the best/worst situation (woman optimality).

Another remarkable property is that the nondeterminism present (in the choice of the ‘‘some man *m*’’ in the while loop) has *no effect* on the solution obtained; it is always *the same*! Thus, using either a stack or queue (or something else) for the free men is quite adequate.

3. A parallel MPI implementation of SMP may use $2n + 1$ processes: n for the men, n for the women, and 1 for the termination detection coordinator. The coordinator also reads the input instance for broadcasting to the other processes.

   The men run the following algorithm:

   ```
   Set pointer to beginning of preference list;
   while TRUE do
   begin
      Send proposal to process of pointer->woman;
      Receive a message;
      Set pointer to next women on preference list;
   end;
   ```

   The women run the following algorithm:

   ```
   Set pointer to end of preference list;
   Receive a message;  // Could be from any of the men
   Scan up the preference list until node of message man is found;
   while TRUE do
   begin
      Receive a message;
      if man that sent message is less preferable than present fiance then
         Send rejection message;
      else
         Send message to break engagement;
         Set pointer at new fiance;
   end;
   ```

   The difficulty with this approach is that all processes will block permanently at a receive when all persons have been paired-up. Your task is to add a termination detection technique that 1) avoids the permanent blocking and 2) allows the results to be output.

4. The input file will have the value of n on the first line, followed by the preference lists for men 0 through n - 1, and finally the preference lists for women 0 through n - 1. Preference lists will be one per line.

5. Your executions must use an MPI machinefile to map processes to at least two of the three available systems. Your code should verify that the number of processes is $2n + 1$.

```
[weems@va NOTES03]$ cat smM.c
// Stable marriages by distributed Gale-Shapley w/ termination detection
// by credit recovery.  BPW 6/2001
#include <stdio.h>
#include "mpi.h"

#define DISTRIBUTE 1
#define PROPOSAL 2
#define REJECT 3
#define CREDIT 4
#define TERMINATE 5

int numProcesses,rank;
MPI_Status status;

main(int argc, char** argv)
{
int n;
int preference[50],i,j,k;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numProcesses);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

```
if (rank==0)
{
  scanf("%d",&n);
  if (numProcesses!=2*n+1)
  {
    printf("0(%d): Wrong number of processes\n",__LINE__);
    MPI_Abort(MPI_COMM_WORLD,__LINE__);
  }
  MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
  for (i=1;i<=2*n;i++)
  {
    for (j=0;j<n;j++)
      scanf("%d",&preference[j]);
    MPI_Send(preference,n,MPI_INT,i,DISTRIBUTE,MPI_COMM_WORLD);
  }
  for (i=0;i<n;i++)
    MPI_Recv(&j,1,MPI_INT,MPI_ANY_SOURCE,CREDIT,MPI_COMM_WORLD,&status);
  for (i=1;i<=2*n;i++)
    MPI_Send(&i,0,MPI_INT,i,TERMINATE,MPI_COMM_WORLD);
}
else
{
  MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
  MPI_Recv(preference,n,MPI_INT,0,DISTRIBUTE,MPI_COMM_WORLD,&status);
  if (rank<=n)
  { // I'm a guy
    i=0;
    MPI_Send(&i,0,MPI_INT,preference[0]+n+1,PROPOSAL,MPI_COMM_WORLD);
    while (1)
    {
      MPI_Recv(&j,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
      if (status.MPI_TAG==REJECT)
      {
        i++;
        MPI_Send(&i,0,MPI_INT,preference[i]+n+1,PROPOSAL,MPI_COMM_WORLD);
      }
      else
        break;
    }
    if (status.MPI_TAG==TERMINATE)
      printf("%d: (%d,%d)\n",rank,rank-1,preference[i]);
    else
    {
      printf("%d(%d): Bad tag %d\n",rank,__LINE__,status.MPI_TAG);
      MPI_Abort(MPI_COMM_WORLD,__LINE__);
    }
  }
  else
  { // I'm a girl
    MPI_Recv(&i,1,MPI_INT,MPI_ANY_SOURCE,PROPOSAL,MPI_COMM_WORLD,&status);
    MPI_Send(&i,0,MPI_INT,0,CREDIT,MPI_COMM_WORLD);
    for (i=n-1;i>=0;i--)
      if (preference[i]==status.MPI_SOURCE-1)
        break;
    while (1)
    {
      MPI_Recv(&i,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
      if (status.MPI_TAG==PROPOSAL)
      {
        for (j=n-1;j>=0;j--)
          if (preference[j]==status.MPI_SOURCE-1)
            break;
        if (j<i)
        { // Break engagement
          MPI_Send(&i,0,MPI_INT,preference[i]+1,REJECT,MPI_COMM_WORLD);
          i=j;
        }
        else // Reject proposal
          MPI_Send(&i,0,MPI_INT,status.MPI_SOURCE,REJECT,MPI_COMM_WORLD);
      }
      else
        break;
    }
    if (status.MPI_TAG!=TERMINATE)
```

```
    {
      printf("%d(%d): Bad tag %d\n",rank,__LINE__,status.MPI_TAG);
      MPI_Abort(MPI_COMM_WORLD,__LINE__);
    }
  }
}

MPI_Finalize();
}

[weems@va NOTES03]$ cat sm2.dat
3
0 1 2
0 1 2
0 1 2
2 1 0
2 1 0
2 1 0
[weems@va NOTES03]$ mpicc smM.c
[weems@va NOTES03]$ mpirun -np 7 a.out<sm2.dat
1: (0,2)
3: (2,0)
2: (1,1)
```

Basic Collective Communication:

```
[weems@va NOTES03]$ cat collectiveM.c
#include <stdio.h>
#include <math.h>
#include "mpi.h"

int numProcesses,rank;
MPI_Status status;

main(int argc, char** argv)
{
int i,*myValues,*result;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numProcesses);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
myValues=(int*) malloc(numProcesses*sizeof(int));
result=(int*) malloc(numProcesses*sizeof(int));
for (i=0;i<numProcesses;i++)
  myValues[i]=numProcesses*rank+i;
printf("%d: myValues: ",rank);
for (i=0;i<numProcesses;i++)
  printf("%d ",myValues[i]);
printf("\n");
MPI_Reduce(myValues,result,numProcesses,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
if (rank==0)
{
  printf("%d: result(sum) ",rank);
  for (i=0;i<numProcesses;i++)
    printf("%d ",result[i]);
  printf("\n");
}
MPI_Allreduce(myValues,result,numProcesses,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
printf("%d: result(sum) ",rank);
for (i=0;i<numProcesses;i++)
  printf("%d ",result[i]);
printf("\n");
MPI_Scatter(myValues,1,MPI_INT,result,numProcesses,MPI_INT,0,MPI_COMM_WORLD);
printf("%d: result(scatter) ",rank);
for (i=0;i<numProcesses;i++)
  printf("%d ",result[i]);
printf("\n");
MPI_Gather(myValues,1,MPI_INT,result,1,MPI_INT,0,MPI_COMM_WORLD);
if (rank==0)
{
  printf("%d: result(gather) ",rank);
  for (i=0;i<numProcesses;i++)
    printf("%d ",result[i]);
  printf("\n");
```

```
}
MPI_Finalize();
}

[weems@va NOTES03]$ mpirun -np 5 a.out
0: myValues: 0 1 2 3 4
0: result(sum) 50 55 60 65 70
0: result(sum) 50 55 60 65 70
0: result(scatter) 0 55 60 65 70
1: myValues: 5 6 7 8 9
1: result(sum) 50 55 60 65 70
1: result(scatter) 1 55 60 65 70
4: myValues: 20 21 22 23 24
4: result(sum) 50 55 60 65 70
4: result(scatter) 4 55 60 65 70
2: myValues: 10 11 12 13 14
2: result(sum) 50 55 60 65 70
2: result(scatter) 2 55 60 65 70
3: myValues: 15 16 17 18 19
3: result(sum) 50 55 60 65 70
3: result(scatter) 3 55 60 65 70
0: result(gather) 0 5 10 15 20
```