01.cpp

```
#include <iostream>
#include "mpi.h"
using namespace std;

int main(int argc, char **argv)
{
   char name[1024];
   int length=1024, major, minor;

   MPI_Init(&argc, &argv);

   MPI_Get_processor_name(name, &length);
   cout << "\nHello from " << name;

   MPI_Get_version(&major, &minor);
   cout << "\nMPI Version " << major << "." << minor;
   cout << "\nFrom header: Version " << MPI_VERSION << "." << MPI_SUBVERSION;

   MPI_Finalize();

   return 0;
}
```

02.cpp

```
#include <iostream>
#include "mpi.h"

int main(int argc, char **argv)
{
    int rank, size;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    std::cout << "\nSize=" << size << ", MyRank:" << rank;

    MPI_Finalize();
    return 0;
}
```

03.cpp

```
#include <iostream>
#include <cstdlib>
#include "mpi.h"

/* This program uses multiple processors to compute sum of some integers */
int main(int argc, char **argv)
{
    int rank, size, upper=0;
    int nPerNode;
    int begin, end, i;
    unsigned int sum, allSum;
    double t1, t2;

    MPI_Init(&argc, &argv);

    // Get processor configurations
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Interactively enter a number from root when necessary
    if(rank==0) {
        if(argc>1) upper = atoi(argv[1]);
        if(upper<=1) {
            std::cout << "\nThis program computes the sum between 1 and the integer you're going to enter.";
            std::cout << "\nPlease enter an integer: ";
            std::cin >> upper;
        }
    }

    // Record the start time
    t1 = MPI_Wtime();

    // Notify every participating processor the integer
```

```
    MPI_Bcast(&upper, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Now divide the work, we're assuming every processor is equally powerful (or weak)
    // 50 -> 4 nodes --> nPerNode = 12: 1 - 12, 13 - 24, 25 - 36, 37 - 50
    nPerNode = upper / size;
    begin = rank * nPerNode + 1;
    end = begin + nPerNode - 1;

    // special case for the last processor
    if(rank == (size-1)) {
        end = upper;
    }

    // Now every node does its job
    sum = 0;
    for(i=begin; i<=end; i++) {
        sum += i;
    }

    // Now collect results back to root
    MPI_Reduce(&sum, &allSum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Calculate the total wall-clock time
    if(rank==0) {
        t2 = MPI_Wtime() - t1;
        std::cout << "\n" << size << ", sum(1.." << upper << "= " << allSum << ", " << t2 << " seconds.";
    }

    // Clean up
    MPI_Finalize();

    return 0;
}
```

04.cpp

```
#include <iostream>
#include <cstdlib>
#include "mpi.h"
using namespace std;

double intf(double x);
double f(double x);

// This program integrates f(x) between [lower] and [upper] with N divisions.
//    Algorithm: Trapezoidal rule
// function f    is the function to be integrated
// function intf is the analytical solution

// 3x^2 + 2x + 3 --> x^3 + x^2 + 3x
double intf(double x) {
    return x*x*x + x*x + 3*x;
}

// 3x^2 + 2x + 3 --> x^3 + x^2 + 3x
double f(double x) {
    return 3*x*x + 2*x + 3;
}

// Main program
int main(int argc, char **argv)
{
    int size, rank;
    double upper, lower;
    int N;
    int i;
    double myLower, X, L, L2;
    double fLower, fUpper;
    double area, answer, analytical;
    double t1, t2;

    // Initialization
    MPI_Init(&argc, &argv);

    // Get processor configurations
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Get some data
```

```
    if(rank == 0) {
        if(argc==4) {
            lower = atof(argv[1]);
            upper = atof(argv[2]);
            N     = atoi(argv[3]);
        }
        else {
            cout << "\nLower bound: ";
            cin >> lower;
            cout << "\nUpper bound: ";
            cin >> upper;
            cout << "\nDivision per processor: ";
            cin >> N;
        }
    }

    // Timing
    t1 = MPI_Wtime();

    // Broadcast information
    MPI_Bcast(&lower, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(&upper, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Now each processor decides what to do
    L = (upper - lower) / size;
    myLower = rank * L + lower;
    L /= N;
    L2 = L * 0.5;

    // Now calculate area of each division
    area = 0;
    fLower = f(myLower);
    X = myLower;
    for(i=0;i<N;i++) {
        X += L;
        fUpper = f(X);
        area += (fLower + fUpper) * L2;
        fLower = fUpper;
    }

    // Now sum the area up.
    MPI_Reduce(&area, &answer, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    // timing
    t2 = MPI_Wtime();

    // The answer is:
    if(rank == 0) {
        analytical = intf(upper) - intf(lower);
        cout << "\nThe answer is " << answer << "(" << analytical << ")";
        cout << "\nThe error is " << (answer - analytical) / analytical * 100;
        cout << "\nThe solution time: " << t2-t1 << " seconds.";
    }

    // Cleanup
    MPI_Finalize();

    return 0;
}
```

05.cpp

```
#include <iostream>
#include <cstdlib>
#include "mpi.h"

using namespace std;

// This program generates the desired number of random numbers, and
//    calcualte the mean, the max, and the min. of these random numbers
int main(int argc, char **argv)
{
    int rank, size;
    int N;
    int myShare, i;
    double *data;
    double myMax, myMin, mySum;
```

```
    int totalN;
    double allMax, allMin, allSum;

    // Initialization
    MPI_Init(&argc, &argv);

    // Get processor configurations
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Get necessary input data
    if(rank == 0) {
        if(argc == 2) {
            N = atoi(argv[1]);
        }
        while(N <= 1) {
            cout << "\nEnter the number of random numbers you want: (must be larger than 1)";
            cin >> N;
        }
    }

    // Announce the number of data to be generated
    MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Determine how many random numbers I need to generate
    myShare = N / size;

    // Last process may need to generate differnet number of data than others
    if(rank == (size - 1)) {
        myShare = N - (size-1) * myShare;
    }

    // allocate memory
    data = new double[myShare];

    if(data != NULL) {
        // generate data
        // set random number seed, otherwise, all processes use the same default
        //     random seed (1), and produce identical output of random numbers.
        srand(rank);
        for(i=0;i<myShare;i++) {
            data[i] = (double) rand()/RAND_MAX;
            cout << "(" << data[i] << ")";
        }

        // Find the max. and min. of the data, and calculate sum
        myMax = data[0];
        myMin = data[0];
        mySum = data[0];
        for(i=1;i<myShare;i++) {
            mySum += data[i];
            if(data[i] > myMax) myMax = data[i];
            if(data[i] < myMin) myMin = data[i];
        }

        // free memory and discard data
        delete []data;
    }
    else {
        // failed to allocate memory
        myShare = 0;
    }

    // Now, collect data back to root (rank 0) process
    MPI_Reduce(&myShare, &totalN, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&myMax, &allMax, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Reduce(&myMin, &allMin, 1, MPI_DOUBLE, MPI_MIN, 0, MPI_COMM_WORLD);
    MPI_Reduce(&mySum, &allSum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if(rank==0) {
        cout << "\nN:   " << totalN;
        cout << "\nMax: " << allMax;
        cout << "\nMin: " << allMin;
        cout << "\nAvg: " << allSum/totalN;
    }
    MPI_Finalize();

    return 0;
}
```