08.cpp

```cpp
#include <cassert>
#include <cmath>
#include <cstdlib>
#include <iostream>

#include "mpi.h"
#define maxNP 32
using namespace std;

// randomly generate a vector
double *genVector(const int N, const int seed)
{
    int i;
    double *A;

    A = new (nothrow) double[N];
    assert(A);

    srand(seed);
    for(i=0;i<N;i++) {
        A[i] = ((double) rand() / RAND_MAX);
    }

    return A;
}

// get a vector size either from command line or from console
int getSize(int argc, char **argv)
{
    int length=0;
    // Determine the vector size
    if(argc>1) {
        length = atoi(argv[1]);
    }
    while(length<=1) {
        cout << "\nPlease enter the length of the vector: ";
        cin >> length;
    }

    return length;
}

// Calculate inner product of two vectors, version 2
int main(int argc, char **argv)
{
    int rank, size;
    double *v1=NULL;
    int length = 0;
    double t1, t2;
    int work;
    int count[maxNP], displ[maxNP];

    int mySize, i;
    double *myData;
    double localSum, vLength;

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

    // Get environment information
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if(size > maxNP) {
        if(rank==0) {
            cout << "\nThis program is not capable of using more than " << maxNP << " processes.";
        }
        MPI_Finalize();
        return 255;
    }

    if(rank==0) {
        length = getSize(argc, argv);
        v1 = genVector(length, 100);

        // determine how work is going to be distributed
        work = length / size;
        displ[0] = 0;
        for(i=0;i<size-1;i++) {
            count[i] = work;
            displ[i+1] = displ[i] + work;
        }
        count[i] = length - (size-1) * work;
```

```
    }

    t1 = MPI_Wtime();

    // let everyone knows how much work they have to do
    MPI_Scatter(count, 1, MPI_INT, &mySize, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // every one allocate memory
    myData = new double[mySize];

    // I am assuming memory allocation will success, which is not good!!!
    MPI_Scatterv(v1, count, displ, MPI_DOUBLE, myData, mySize, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // now every node does computation
    localSum = 0.0;
    for(i=0;i<mySize;i++) {
        localSum += myData[i] * myData[i];
    }

    // sum all squres
    MPI_Allreduce(&localSum, &vLength, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    vLength = sqrt(vLength);

    if(vLength != 0.0) {
        // Now normalize the vector
        for(i=0;i<mySize;i++) {
            myData[i] /= vLength;
        }
    }

    // now send results back to the root
    MPI_Gatherv(myData, mySize, MPI_DOUBLE, v1, count, displ, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    t2 = MPI_Wtime() - t1;
    if(rank==0) {
        cout << "\nNormalization takes " << t2 << " seconds.";
    }

    // Finalize
    MPI_Finalize();

    delete []v1;

    return 0;
}
```

09a_noBarrier.cpp

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

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

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for(i=0;i<10;i++) {
    std::cout << rank << ": " << i << std::endl;
  }
  std::cout << rank << ": " << " done!" << std::endl;

  MPI_Finalize();

  return 0;
}
```

09b_barrier.cpp

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

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

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for(i=0;i<10;i++) {
      std::cout << rank << ": " << i << std::endl;
      MPI_Barrier(MPI_COMM_WORLD);
  }
  std::cout << rank << ": " << " done!" << std::endl;
```

```
  MPI_Finalize();

  return 0;
}
```

10_scan.cpp

```
#include <iostream>
#include "mpi.h"
#define N 3

int main(int argc, char **argv) {
  int i;
  int rank;
  int a[N], b[N];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  for(i=0;i<N;i++) {
    a[i] = i+rank*N;
  }

  std::cout << "\n" << rank << ": a[]=";
  for(i=0;i<N;i++) {
      std::cout << a[i] << ", ";
  }

  MPI_Scan(a, b, N, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

  std::cout << "\n" << rank << ": b[]=";
  for(i=0;i<N;i++) {
      std::cout << b[i] << ", ";
  }

  MPI_Finalize();

  return 0;
}
```

11_ring_1.cpp

```cpp
#include "mpi.h"

int main(int argc, char **argv)
{
    int rank, size;
    int otherRank;
    int tag = 100;
    int to, from;
    MPI_Status status;

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

    // Get environment information
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Pass its rank to the right until it gets it back
    otherRank = rank;
    to = (rank + 1) % size;
    if(rank != 0) {
        from = (rank - 1);
    }
    else {
        from = size - 1;
    }

    do {
        MPI_Send(&otherRank, 1, MPI_INT, to, tag, MPI_COMM_WORLD);
        MPI_Recv(&otherRank, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &status);
    } while(otherRank != rank);

    // Finalize
    MPI_Finalize();

    return 0;
}
```

12_ring_2.cpp

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

int main(int argc, char **argv)
{
    int rank, size;
    int otherRank, tmp;
    int tag = 100;
    int to, from;
    double t1, t2;
    MPI_Status status;

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

    // Get environment information
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Pass its rank to the right until it gets it back
    otherRank = rank;
    to = (rank + 1) % size;
    if(rank != 0) {
        from = (rank - 1);
    }
    else {
        from = size - 1;
    }

    t1 = MPI_Wtime();
    do {
        if(rank % 2 == 0) {
            MPI_Send(&otherRank, 1, MPI_INT, to, tag, MPI_COMM_WORLD);
            MPI_Recv(&otherRank, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &status);
            if(rank==0) std::cout << "[" << otherRank << "]";
        }
        else {
            MPI_Recv(&tmp, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &status);
            MPI_Send(&otherRank, 1, MPI_INT, to, tag, MPI_COMM_WORLD);
            otherRank = tmp;
        }
```

```
    } while(otherRank != rank);
    t2 = MPI_Wtime();

    if(rank==0) {
        std::cout << "\nTime: " << t2-t1 << std::endl;
    }


    // Finalize
    MPI_Finalize();

    return 0;
}
```

13_ring_3.cpp

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

int main(int argc, char **argv)
{
    int rank, size;
    int otherRank, tmp;
    int tag = 100;
    int to, from;
    double t1, t2;
    MPI_Status status;

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

    // Get environment information
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Pass its rank to the right until it travels back
    otherRank = rank;
    to = (rank + 1) % size;
    if(rank != 0) {
        from = (rank - 1);
    }
    else {
        from = size - 1;
    }

    t1 = MPI_Wtime();
    do {
        if(rank % 2 == 0) {
            MPI_Sendrecv_replace(&otherRank, 1, MPI_INT, to, tag, from,
                tag, MPI_COMM_WORLD, &status);
/*
            MPI_Sendrecv(&otherRank, 1, MPI_INT, to, tag,
                &otherRank, 1, MPI_INT, from, tag,
                MPI_COMM_WORLD, &status);
*/
            if(rank == 0) std::cout << "[" << otherRank << "]";
        }
        else {
            MPI_Recv(&tmp, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &status);
            MPI_Send(&otherRank, 1, MPI_INT, to, tag, MPI_COMM_WORLD);
            otherRank = tmp;
        }
    } while(otherRank != rank);
    t2 = MPI_Wtime();
    if(rank==0) {
        std::cout << "\nTime: " << t2-t1 << std::endl;
    }

    // Finalize
    MPI_Finalize();

    return 0;
}
```

14_ring_4.cpp

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

int main(int argc, char **argv)
{
    int rank, size;
    int otherRank, tmp;
    int tag = 100;
    int to, from;
    double t1, t2;
    MPI_Request request[2];
    MPI_Status status[2];

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

    // Get environment information
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Pass its rank to the right until it gets it back
    otherRank = rank;
    to = (rank + 1) % size;
    if(rank != 0) {
        from = (rank - 1);
    }
    else {
        from = size - 1;
    }

    t1 = MPI_Wtime();
    do {
        MPI_Isend(&otherRank, 1, MPI_INT, to, tag, MPI_COMM_WORLD, &(request[0]));
        MPI_Irecv(&tmp, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &(request[1]));
        MPI_Waitall(2, request, status);
        otherRank = tmp;
        if(rank==0) std::cout << "[" << otherRank << "]";
    } while(otherRank != rank);
    t2 = MPI_Wtime();

    if(rank==0) {
        std::cout << "\nTime: " << t2-t1 << std::endl;
    }
    // Finalize
    MPI_Finalize();

    return 0;
}
```

15_ring_5.cpp

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

int main(int argc, char **argv)
{
    int rank, size;
    int periods;
    MPI_Comm newCom;

    int to, from;

    int otherRank, tmp;
    int tag = 100;
    MPI_Request request[2];
    MPI_Status status[2];

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

    // Get environment information
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // setup ring topology (1: one-dimension,
    //        &size --> number of processes in each dimension
    //        &periods --> whether the grid is periodic or not
    //        0 --> ranking may not be re-ordered
    periods = 1;
```

```
    MPI_Cart_create(MPI_COMM_WORLD, 1, &size, &periods, 1, &newCom);

    // 0: direction, 1: displacement
    // &from, &to
    MPI_Cart_shift(newCom, 0, 1, &from, &to);

    // Initially pass its rank to the right until it gets it back
    otherRank = rank;

    do {
        MPI_Isend(&otherRank, 1, MPI_INT, to, tag, newCom, &(request[0]));
        MPI_Irecv(&tmp, 1, MPI_INT, from, tag, newCom, &(request[1]));
        MPI_Waitall(2, request, status);
        otherRank = tmp;
        if(rank==0) std::cout << "[" << otherRank << "]";
    } while(otherRank != rank);

    // Finalize
    MPI_Finalize();

    return 0;
}
```

16_cart.cpp

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

#define UP 0
#define DOWN 1
#define LEFT 2
#define RIGHT 3

int main(int argc, char **argv)
{
    int myRank1, myRank2;
    int size[2]={3,3};
    int periods[2] = {1,0};
    int reorder=1;
    MPI_Comm newCom;

    int myCoord[2];
    int tmp[2];
    int maxdims=2;

    int who;
    int ngbr[4];

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

    // get my rank in the World communicator
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank1);

    // setup 2D cartesian topology (1: one-dimension,
    MPI_Cart_create(MPI_COMM_WORLD, 2, size, periods, reorder, &newCom);

    // get my rank in the newly created cartesian communicator
    MPI_Comm_rank(newCom, &myRank2);

    // query my coord in the created cartesian topology
    MPI_Cart_coords(newCom, myRank2, maxdims, myCoord);

    cout << "\n" << myRank1 << ": ([" << myRank2 << "], " << myCoord[0] << ", " << myCoord[1] << ")";

    // Query who is at coord (1,2)
    tmp[0] = 1;
    tmp[1] = 2;
    MPI_Cart_rank(newCom, tmp, &who);
    cout << "\n" << who << " is at (1,2)";

    // Find out about neighbors
    MPI_Cart_shift(newCom, 0, 1, &ngbr[LEFT], &ngbr[RIGHT]);
    MPI_Cart_shift(newCom, 1, 1, &ngbr[UP], &ngbr[DOWN]);
    cout << "\n" << myRank2 << "(" << ngbr[LEFT] << ", " << ngbr[RIGHT] <<
        ", " << ngbr[UP] << ", " << ngbr[DOWN] << ") as neighbors.";

    MPI_Finalize();

    return 0;
}
```

17_group.cpp

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

int main(int argc, char **argv)
{
    int rank, size;
    int newSize;
    int *gRanks;
    MPI_Group gWorld, newGroup;
    MPI_Comm newComm;

    int i, pts;
    int newRank;

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

    // Get environment information
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Get the member of MPI_COMM_WORLD to g1 & g2 group handles
    MPI_Comm_group(MPI_COMM_WORLD, &gWorld);

    // Now we want g1 to have only even ranks
    //   and g2 to have only odd ranks

    newSize = size / 2;
    if(size%2!=0) newSize++;

    gRanks = new int[newSize];

    // assign ranks to be in the different group
    if(rank%2==0) {
      for(i=0,pts=0,newSize=0;i<size;i+=2,pts++,newSize++) {
        gRanks[pts] = i;
      }
    }
    else {
      for(i=1,pts=0,newSize=0;i<size;i+=2,pts++,newSize++) {
        gRanks[pts] = i;
      }
    }

    // Create new groups from gWorld.
    MPI_Group_incl(gWorld, newSize, gRanks, &newGroup);

    // Create group communicator for sub-group communication
    MPI_Comm_create(MPI_COMM_WORLD,newGroup,&newComm);

    MPI_Comm_rank(newComm, &newRank);
    cout << "\nold=" << rank << ", new=" << newRank;

    // Now we can do collective communication with new communicators
    MPI_Bcast(&rank, 1, MPI_INT, 0, newComm);
    cout << "\nbroadcasted rank=" << rank;

    MPI_Comm_free(&newComm);
    delete []gRanks;

    // Finalize
    MPI_Finalize();

    return 0;
}
```

18_Armstrong.cpp

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

int isArmstrong1(unsigned long num){
  return 1==1;
}

int isArmstrong(unsigned long num) {
  int i, j, p, count, product;
  unsigned long n, sum;
  int digits[25]; // in fact, 20 at most...

  count=0;
  n = num;
  while(n>=1) {
    digits[count] = n % 10;
    n /= 10;
    count++;
  }
  p = count;

  sum = 0;
  for(j=0;j<count;j++) {
    product = 1;
    for(i=0;i<p;i++) product *= digits[j];
    sum += product;
  }

  return num==sum;
}

int manager(int size, int count) {
  unsigned long i, out[2], block=100, answer;
  int no=0;
  MPI_Status status;

  // for distributing work
  for(i=100;i<=ULONG_MAX-block;) {
    MPI_Recv(&answer, 1, MPI_LONG, MPI_ANY_SOURCE, 100, MPI_COMM_WORLD, &status);
    if(answer!=0) {
      cout << "\n" << answer;
      no++;
      if(no>=count) break;
    }
    else {
      out[0] = i;
      out[1] = block;
      MPI_Send(out, 2, MPI_LONG, status.MPI_SOURCE, 101, MPI_COMM_WORLD);
      i += block;
    }
  }
  // for terminating workers
  for(i=1;i<size;) {
    MPI_Recv(&answer, 1, MPI_LONG, MPI_ANY_SOURCE, 100, MPI_COMM_WORLD, &status);
    if(answer!=0) {
      // cout << "ans=" << answer << endl;
      continue;
    }
    out[0] = 0;
    out[1] = 0;
    MPI_Send(out, 2, MPI_LONG, status.MPI_SOURCE, 101, MPI_COMM_WORLD);
    i++;
  }

  return no;
}

void worker() {
  unsigned long work[2], i;
  unsigned long answer=0;
  MPI_Status status;

  for(;;) {
    MPI_Send(&answer, 1, MPI_LONG, 0, 100, MPI_COMM_WORLD);
    MPI_Recv(work, 2, MPI_LONG, 0, 101, MPI_COMM_WORLD, &status);
    // cout << work[0] << ", " << work[1] << endl;
    if(work[1]==0) break;
    answer = 0;
    for(i=work[0]; i<work[0]+work[1]; i++) {
```

```
        if(isArmstrong(i)) {
          MPI_Send(&i, 1, MPI_LONG, 0, 100, MPI_COMM_WORLD);
        }
      }
    }
}

int main(int argc, char **argv) {
    int rank, size;
    int count=1;
    double t1,t2;
    // Initialization
    MPI_Init(&argc, &argv);

    // Get environment information
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if(argc>1) {
        count=atoi(argv[1]);
    }
    if(count<1) count=1;

    t1=MPI_Wtime();
    if(rank==0) {
      count=manager(size, count);
      t2=MPI_Wtime();
      cout << "\nFound " << count << " Armstrong numbers";
      cout << ", and took " << t2-t1 << " seconds.";
    }
    else {
      worker();
    }

    // Finalize
    MPI_Finalize();

    return 0;
}
```