

```

/*
 *  ampi.cc           parallel program file
 *
 *  Author:      Evan Hourigan                               Copyright (c), 2004.
 *                  Hobart & William Smith Colleges
 *                  Geneva, NY 14456
 *
 *                  Email: hourigan@hws.edu
 *  This source code may not be reproduced or modified in any way, shape, or
 *  form without written consent from the author.
 */

#include <ctime>
#include <string>
#include <mpi.h>
#include "Population.h"
#include "Automaton.h"
#include <sys/resource.h>
#include <unistd.h>

#define mpitype MPI_CHAR

#define converttype Population<Automaton,Automaton::Pack>::PopulationPack

using namespace std;

const int POPULATION_SIZE = 100;
const int GENERATIONS = 4000;
const int NUMBER_OF_TRIALS = 100;
const int CATCH_UP_GENERATIONS = 50;
const int MIGRATION_SIZE = 10;
const int PERCENT_CHANCE_OF_MIGRATION = 10;

const int MASTER_PROCESS = 0;

// The following constants are used as additional tag parameters in calls to
// MPI_Iprobe, etc. The values of these constants are arbitrarily chosen,
// and mean nothing
const int MPI_MIGRATION = 1029;           // Used by slaves in messages to
                                           // neighboring slaves signalling an
                                           // incoming migration
const int MPI_DO_MIGRATE = 1035;        // Only used by master in messages to
                                           // slave processes as a command to do
                                           // migrations to both left AND right
                                           // neighbors
const int MPI_STATISTICS = 1050;        // Used by slaves in messages to master
                                           // process signalling an incoming pack
                                           // of statistical data about the
                                           // population

void mb_child(int process_num) {
    srand(time(0)+process_num);

    int commsize; // Size of the population
    MPI_Comm_size(MPI_COMM_WORLD, &commsize); // Get number of processes

    int leftNeighbor; // Process number of left neighbor
    int rightNeighbor; // Process number of right neighbor

    // Assign process numbers of neighboring left and right processes accordingly
    if (process_num == 1) {
        leftNeighbor = commsize-1;
        rightNeighbor = 2;
    } else if (process_num == commsize-1) {
        leftNeighbor = process_num-1;
        rightNeighbor = 1;
    } else {
        leftNeighbor = process_num-1;
        rightNeighbor = process_num+1;
    }
}

```

```

Population<Automaton, Automaton::Pack> P;
P.create(POPULATION_SIZE);

Population<Automaton, Automaton::Pack>::PopulationPack exportPack;
    // will hold individuals we want to send
Population<Automaton, Automaton::Pack>::PopulationPack importPack;
    // will hold individuals coming into program
Population<Automaton, Automaton::Pack>::StatisticsPack statsPack;
    // will hold stats to send to master
int migPackSize = sizeof(importPack);
int statPackSize = sizeof(statsPack);

int genCt = 0;    // Current generation
int statsCount = 0; // Counts each generation. When this is = 4 it's reset

// Variables used for call to MPI_Iprobe()
int masterFlag;
int migrationLeftFlag;
int migrationRightFlag;

MPI_Status status;

while (genCt <= GENERATIONS) {

    // Probe for command from the master to do a migration
    MPI_Iprobe(MASTER_PROCESS,
                MPI_DO_MIGRATE,
                MPI_COMM_WORLD,
                &masterFlag,
                &status);
    if (masterFlag) { // Command from master to do migration, so do it!
        exportPack = P.doEmigration();
        if (randInt(10) < 5) {
            //cout << leftNeighbor << endl;
            MPI_Send(&exportPack,
                    migPackSize,
                    mpitype,
                    leftNeighbor,
                    MPI_MIGRATION,
                    MPI_COMM_WORLD);
        }
        else {
            //cout << rightNeighbor << endl;
            MPI_Send(&exportPack,
                    migPackSize,
                    mpitype,
                    rightNeighbor,
                    MPI_MIGRATION,
                    MPI_COMM_WORLD);
        }
    }

    // Probe for message from either neighbor for incoming migration
    MPI_Iprobe(leftNeighbor,
                MPI_MIGRATION,
                MPI_COMM_WORLD,
                &migrationLeftFlag,
                &status);
    // I found a bug here during last minute review--the message is never
    // to perform a migration is noticed, but received, and so migration
    // occurs more frequently than I expected. Fortunately, this does not
    // change the discussion
    if (migrationLeftFlag) { // Incoming migration from left neighbor,
                            // so process it!
        MPI_Recv(&importPack,
                 migPackSize,
                 mpitype,
                 leftNeighbor,
                 MPI_MIGRATION,

```

```

        MPI_COMM_WORLD,
        &status);
    P.doImmigration(importPack);
}
MPI_Iprobe(rightNeighbor,
           MPI_MIGRATION,
           MPI_COMM_WORLD,
           &migrationRightFlag,
           &status);
if (migrationRightFlag) { // Incoming migration from right neighbor,
                           // so process it!
    MPI_Recv(&importPack,
            migPackSize,
            mpitype,
            rightNeighbor,
            MPI_MIGRATION,
            MPI_COMM_WORLD,
            &status);
    P.doImmigration(importPack);
}

double niceOnes = 0.0;
int runningIQ = 0;
// put stats for five generations in statsPack
statsPack.currentGeneration[statsCount] = genCt;
statsPack.populationAvgFitness[statsCount] = P.getAverageFitness();
for (int i = 0; i < POPULATION_SIZE; i++) {
    Automaton *individual = P.getIndividual(i);
    if (individual->isNice())
        niceOnes++;
    runningIQ += individual->nicenessIQ();
}
statsPack.populationNiceness[statsCount]
    = ((niceOnes/static_cast<double>(POPULATION_SIZE))*100.0);
statsPack.populationAvgNiceIQ[statsCount] = runningIQ/POPULATION_SIZE;
statsCount++;
if (statsCount == 5) { // send statsPack to the master (report current
                       // statistics)
    statsCount = 0;
    MPI_Send(&statsPack,
            statPackSize,
            mpitype,
            MASTER_PROCESS,
            MPI_STATISTICS,
            MPI_COMM_WORLD);
}

genCt++;

if (genCt % CATCH_UP_GENERATIONS == 0) { // Generation is a multiple of
                                           // CATCH_UP_GENERATIONS so catch
                                           // up with other slaves
    // First inform master that we are catching up
    //MPI_Send(&genCt,1,MPI_INT,MASTER_PROCESS,MPI_CATCH_UP,MPI_COMM_WORLD);

    // Now catch up... will stay here until all processes have called this
    MPI_Barrier(MPI_COMM_WORLD);
}

P.breed();
}

MPI_Finalize();
}

```

```

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

    struct rlimit lim;
    lim.rlim_cur = RLIM_INFINITY;
    lim.rlim_max = RLIM_INFINITY;
    setrlimit(RLIMIT_CPU,&lim);

    int my_rank; // Rank #0 is the master, anything > 0 is a slave.
    int commsize; // Number of populations???

    MPI_Status status;

    // Pack data to be displayed
    Population<Automaton,Automaton::Pack>::StatisticsPack displayData;

    int statPackSize = sizeof(displayData);

    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == 0) { // Only true for the master process

        int messageCount = 0; // Counts number of messages received

        MPI_Comm_size(MPI_COMM_WORLD, &commsize); // Get number of processes

        cout << "Process | Generation | Niceness % | Avg IQ | Avg Fitness \n\n";

        while (messageCount < ( ((GENERATIONS/5)+1) * (commsize-1) )) {

            MPI_Recv(&displayData,
                    statPackSize,
                    mpitype,
                    MPI_ANY_SOURCE,
                    MPI_STATISTICS,
                    MPI_COMM_WORLD,
                    &status);

            for (int i = 0; i < 5; i++) {
                cout << status.MPI_SOURCE << "\t"
                    << displayData.currentGeneration[i] << "\t"
                    << displayData.populationNiceness[i] << "\t"
                    << displayData.populationAvgNiceIQ[i] << "\t"
                    << displayData.populationAvgFitness[i] << endl;
            }

            messageCount++; // One message has been received and processed

            if (messageCount % (10*(commsize-1)) == 0) {
                cout << "0\t\t\t\t\tMaster catching up at generation "
                    << ((messageCount/(commsize-1))*5) << endl;
                MPI_Barrier(MPI_COMM_WORLD);
            }

            int migprocess = 0;
            while (migprocess == 0)
                migprocess = randInt(commsize);
            int junk = -1;
            if (randInt(100) < PERCENT_CHANCE_OF_MIGRATION) {
                // PERCENT_CHANCE_OF_MIGRATION% chance that migprocess will migrate
                MPI_Send(&junk,1,MPI_INT,migprocess,MPI_DO_MIGRATE,MPI_COMM_WORLD);
            }

        }

        MPI_Finalize();
    }

    else { // For the slave processes
        mb_child(my_rank);
    }
}

```

```
    } return 0;  
}
```