CS5500 HW9

Chase Mortensen A01535275

November 2019

1 Description

Implement Sender-Initiated Distributed Random Dynamic Load Balancing with White/Black Ring Termination.

For this assignment, you are to write a system that performs sender-initiated distributed random dynamic load balancing. Here are some of the particulars, which you should feel free to modify:

The basic unit of work, a task, is represented by an integer i in the range [1-1024], is to increment an integer i^2 times.

The basic unit of work, a task, is represented by an integer i in the range [1-1024], is to increment an integer i^2 times.

Each process has a queue in which it places tasks to perform.

Each process is to perform the following actions:

- check to see if any new work has arrived
- if the number of tasks in the queue exceeds the threshold of 16, then 2 tasks are sent to random destinations
- perform some work
- if the number of tasks generated is less than a random number [1024-2048], generate 1-3 new tasks, and place them at the end of the process task queue

Additionally, your system is to implement a white/black termination algorithm, so that your system terminates normally.

Your report should track discuss you implementation, and should track the number of tasks performed and overall utilization for each process.

My program utilizes MPI_Isends and MPI_Irecvs to implement the load balancing. Essentially, the black/white termination along with the tasks are passed in the same manner. The tasks are integers greater than 0, while the termination algorithm uses 0 to signify work is still being done on the process, -1 to indicate that the process is done, and -2 to terminate the program.

2 Program

```
#include <iostream>
#include <mpi.h>
#include <unistd.h>
#include <stdlib.h>
#include <vector>

//#include "/usr/local/include/mpi.h"
#define MCW MPI_COMM_WORLD

using namespace std;

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

    int rank, size;
    int data;
    int maxTasks = 1024;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MCW, &rank);
    MPI_Comm_size(MCW, &size);
    srand(rank);
    MPI_Status status;
    MPI_Request request;

    vector<int> tasks;
    int task = 0;
    int dest = 0;
    int next = (rank+1)%size;
    int x = 0;
    int tasksGenerated = 0;
    int newTasks = 0;
    int message = 0;
    int messageLength = 0;
    int terminate = -2;
    int done = -1;
    int working = 0;
```

```
// MPI_Send(&workToDo,1,MPI_INT,data,0,MCW);
// MPI_Recv(&workToDo,1,MPI_INT,0,0,MCW,MPI_STATUS_IGNORE);

// MPI_Probe(MPI_ANY_SOURCE,MPI_ANY_TAG,MCW,&mystatus);
// tag = mystatus.MPI_TAG;
// source = mystatus.MPI_SOURCE;
// MPI_Recv(&weight,1,MPI_INT,source,tag,MCW,MPI_STATUS_IGNORE);
// if(tag==0){
//     cout<<"sending a pig of weight "<<weight<<" to farmer "<<source<<".\n";
//     animalType=0;
// }else{
//     cout<<"sending a horse of weight "<<weight<<" to farmer "<<source<<".\n";
//     animalType=1;
// }

// MPI_Send(&animalType,1,MPI_INT,source,0,MCW);

// generate task if process 0
if (!rank){
    tasks.push_back(rand()%1024 + 1);
    MPI_Isend(&done,1,MPI_INT,next,0,MCW,&request);
}

// int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,
//           int tag, MPI_Comm comm, MPI_Request * request)

while (true) {

    // check to see if any new work has arrived
    MPI_Irecv(&message, 1, MPI_INT, MPI_ANY_SOURCE, 0, MCW, &request);

    // After receiving the message, check the status to determine
    // how many numbers were actually received
    MPI_Get_count(&status, MPI_INT, &messageLength);

    if (messageLength){
        if (message > 0)
            tasks.push_back(message);
        else {
            if (message == -1 && tasksGenerated >= maxTasks){
                if (!rank) {// end program
                    MPI_Isend(&terminate,1,MPI_INT,next,0,MCW,&request);
                    // MPI_Send(&terminate,1,MPI_INT,next,0,MCW);
                    break;
                }
                else
```

```
                MPI_Isend(&done,1,MPI_INT,next,0,MCW,&request);
                // MPI_Send(&done,1,MPI_INT,next,0,MCW);
        }
        else if (message == -2){
            if (rank != size - 1)
                MPI_Isend(&terminate,1,MPI_INT,next,0,MCW,&request);
                // MPI_Send(&terminate,1,MPI_INT,next,0,MCW);
            break;
        }
        else
            MPI_Isend(&working,1,MPI_INT,next,0,MCW,&request);
            // MPI_Send(&working,1,MPI_INT,next,0,MCW);
    }
}


// if the number of tasks in the queue exceeds the threshold of 16,
// then 2 tasks are sent to random destinations
while (tasks.size() > 16) {
    task = tasks[tasks.size()-1];
    tasks.pop_back();
    dest = rand()%size;

    MPI_Isend(&task,1,MPI_INT,dest,0,MCW,&request);
    // MPI_Send(&task,1,MPI_INT,next,0,MCW);
}


// perform some work
if (tasks.size()) {
    task = tasks[tasks.size()-1];
    tasks.pop_back();
    x = 0;
    for (int i = 0; i < task; i++)
        x++;
}


// if the number of tasks generated is less than a random number [1024-2048],
// generate 1-3 new tasks, and place them at the end of the process task queue
if (tasksGenerated < maxTasks){
    newTasks = rand()%3 + 1;
    cout << "process " << rank << " generated " << newTasks <<
    " new tasks and has completed " << tasksGenerated << " total\n";
    for (int i = 0; i < newTasks; i++){
        tasksGenerated++;
```

```
                tasks.push_back(rand()%1024 + 1);
            }
        }
        else
            exit(0);
    }

    MPI_Finalize();

    return 0;
}
```

3 Output

```
$ mpic++ load_balancing.cpp
$ mpirun -np 2 ./a.out

process 0 finished 1025 tasks and is exiting
process 1 finished 1026 tasks and is exiting
process 2 finished 1025 tasks and is exiting
process 3 finished 1024 tasks and is exiting
```