Load Balancing

Aaron Morgenegg

11/2/18

OpenMPI Load Balancing

This program implements Sender-Initiated Distributed Random Dynamic Load Balancing with White/Black Ring Termination. Each program has a job queue of jobs it needs to complete. Each job represents sleep time in ms. The main loop consists of the process receiving new jobs from other processes, if any, randomly distributing work to other processes if the job queue is too large, doing the next job on the queue, generating new jobs, handling the white/black ring token, if necessary, and checking whether the terminate signal was receieved.

```
// mpic++ main.cpp
// mpiexec -np 8 --oversubscribe a.out

#include <iostream>
#include <fstream>
#include <time.h>
#include <stdlib.h>
#include <mpi.h>
#include <unistd.h>
#include <algorithm>
#include <math.h>
#include <string>
#include <vector>

#define MCW MPI_COMM_WORLD
#define EMPTY 0
#define WHITE 1
```

```
#define BLACK 2
#define JOB 11
#define TOKEN 12
#define TERMINATE 13
#define ANY MPI_ANY_SOURCE

const int MAX_WORK_SIZE = 1024; // Note: sleep work is in ms
const int MAX_JOB_QUEUE_SIZE = 16;
const int WORK_TO_GENERATE = 1024;
const int JOB_GENERATE_RATE = 2;
const int VERBOSITY = 2; // Set from 0-3
const int MAX_ITERATIONS = 10000;

int getJob(){
return rand()%MAX_WORK_SIZE+1;
}

void initWork(int world_rank){
if(VERBOSITY>0)std::cout<<"p"<<world_rank<<": starting..."<<std::endl;
if(world_rank == 0){
int job = getJob();
MPI_Send(&job,1,MPI_INT,0,JOB,MCW);
int token = BLACK;
MPI_Send(&token,1,MPI_INT,0,TOKEN,MCW);
}
}

void receiveNewJobs(int new_job, MPI_Request my_request, int job_flag, MPI_Status
MPI_Irecv(&new_job,1,MPI_INT,ANY,JOB,MCW,&my_request);
MPI_Test(&my_request,&job_flag,&my_status);
if(!job_flag) return;
job_queue.push_back(new_job);
if(VERBOSITY > 2) std::cout<<"p"<<world_rank<<": received job "<<new_job<<std::end
}

void doWork(std::vector<int> &job_queue, int world_rank, int &jobs_performed){
if(job_queue.size() > 0){
if(VERBOSITY>1) std::cout<<"p"<<world_rank<<": doing job"<<job_queue[0]<<std::endl
```

```
usleep(job_queue[0]);
job_queue.erase(job_queue.begin());
jobs_performed++;
} else {
usleep(10);
return;
}
}

void distributeWork(std::vector<int> &job_queue, int world_size, int world_rank, i
if(job_queue.size()>MAX_JOB_QUEUE_SIZE){
int job1 = job_queue.back();
job_queue.pop_back();
int job2 = job_queue.back();
job_queue.pop_back();
int dest1 = rand()%world_size;
int dest2 = rand()%world_size;

if(VERBOSITY>2) std::cout<<"p"<<world_rank<<": distributing job"<<job1<<" to p"<<d
if(VERBOSITY>2) std::cout<<"p"<<world_rank<<": distributing job"<<job2<<" to p"<<d
MPI_Send(&job1,1,MPI_INT,dest1,JOB,MCW);
MPI_Send(&job2,1,MPI_INT,dest2,JOB,MCW);

if(dest1 < world_rank || dest2 << world_rank){
process_color = BLACK;
}
}
}

void generateNewWork(int jobs_to_spawn, int &spawned_jobs, std::vector<int> &job_q
for(int i = 0; i < JOB_GENERATE_RATE; i++){
if(spawned_jobs < jobs_to_spawn){
if(VERBOSITY>2) std::cout<<"p"<<world_rank<<": spawning additional job"<<std::endl
spawned_jobs++;
job_queue.push_back(getJob());
}
}
}
```

```
void checkTerminate(bool &terminate, int signal, MPI_Request my_request, int termi
MPI_Irecv(&signal,1,MPI_INT,0,TERMINATE,MCW,&my_request);
MPI_Test(&my_request,&terminate_flag,&my_status);
if(!terminate_flag) return;
if(VERBOSITY > 1) std::cout<<"received terminate signal"<<std::endl;
terminate = true;
}

void sendTerminateSignal(int world_rank, int world_size){
int signal = 1;
for(int i = 0; i < world_size; i++){
MPI_Send(&signal,1,MPI_INT,i,TERMINATE,MCW);
}
}

void handleToken(int &token, MPI_Request my_request, int token_flag, MPI_Status my
MPI_Irecv(&token,1,MPI_INT,ANY,TOKEN,MCW,&my_request);
MPI_Test(&my_request,&token_flag,&my_status);
if(!token_flag) return;
if(world_rank == 0){
if (token == WHITE){
sendTerminateSignal(world_rank, world_size);
return;
}
token = WHITE;
}
if(process_color == BLACK){
token = BLACK;
process_color = WHITE;
}
if(job_queue.size() == 0){
MPI_Send(&token,1,MPI_INT,(world_rank+1)%world_size,TOKEN,MCW);
}
}

void loadBalance(int world_rank, int world_size){
int new_job;
```

```
int job;
int job_flag;
int jobs_to_spawn = WORK_TO_GENERATE + rand()%WORK_TO_GENERATE;
int spawned_jobs = 0;
int jobs_performed = 0;
int process_color = WHITE;
std::vector<int> job_queue;
MPI_Request my_request;
MPI_Status my_status;
int token;
int terminate_signal;
int terminate_flag;
int token_flag;
bool terminate = false;

int counter = 0;
while(!terminate and counter++ < MAX_ITERATIONS){
receiveNewJobs(new_job, my_request, job_flag, my_status, world_rank, job_queue);
distributeWork(job_queue, world_size, world_rank, process_color);
doWork(job_queue, world_rank, jobs_performed);
generateNewWork(jobs_to_spawn, spawned_jobs, job_queue, world_rank);
handleToken(token, my_request, token_flag, my_status, process_color, world_rank, w
checkTerminate(terminate, terminate_signal, my_request, terminate_flag, my_status)
}
if(VERBOSITY>0)std::cout<<"p"<<world_rank<<": finished. Jobs completed: "<<jobs_pe
}

int main(int argc, char** argv) {
MPI_Init(&argc, &argv);
srand(time(NULL));
int world_size;
MPI_Comm_size(MCW, &world_size);
int world_rank;
MPI_Comm_rank(MCW, &world_rank);
// MPI_Send(&sendData,1,MPI_INT,dest,0,MCW);
        // MPI_Recv(&recvData,1,MPI_INT,dest,0,MCW,MPI_STATUS_IGNORE);

initWork(world_rank);
```

```
loadBalance(world_rank, world_size);

MPI_Finalize();
return 0;
}
```