HW 8
Load Balancing

Philip Nelson

2018 November 2

Introduction

The purpose of this assignment is to implement sender-initiated distributed random dynamic load balancing with white/black ring termination. I started each process with one task, the tasks were doubles between $[1, 3)$ and processing a task mean sleeping for as many milliseconds as the value of the task times 1000. Each task was assignment a maximum number of tasks to produce between $[1024, 2048]$. If a process ever had more than 16 tasks to process, it would send the next two tasks to a random process. If it ever sent tasks to a process less than it's rank, it would turn it's token black. When process 0 finished it's tasks, it would send a white token to the next process. When other processes finished, they would receive the token and send it on to the next process turning it black if their own token was black. They would then wait to receive an action from process zero. Process zero would tell each process to continue or finalize depending on what color token it received. When a white token returned to process zero, all processes would finalize and report their work.

Code

The code is broken up into two files, main.cpp and random.hpp. The files are included below.

0.1 main.cpp

```
1  #include "random.hpp"
2  #include <chrono>
3  #include <iostream>
4  #include <mpi.h>
5  #include <queue>
6  #include <thread>
7
8  enum TAG
9  {
10   NEW_WORK,
```

```
11    TOKEN,
12    ACTION,
13    CONTINUE,
14    FINALIZE
15  };
16
17  enum TOKEN
18  {
19    WHITE,
20    BLACK
21  };
22
23  void process(double task)
24  {
25    int work = (int)(task*1000);
26    std::this_thread::sleep_for(std::chrono::milliseconds(work));
27  }
28
29  void getNewWork(std::queue<double>& tasks, int& workRecv)
30  {
31    int flag;
32    MPI_Status stat;
33    do
34    {
35      MPI_Iprobe(MPI_ANY_SOURCE, TAG::NEW_WORK, MPI_COMM_WORLD, &flag, &
          stat);
36      if (flag)
37      {
38        double task;
39        MPI_Recv(&task,
40                 1,
41                 MPI_DOUBLE,
42                 stat.MPI_SOURCE,
43                 stat.MPI_TAG,
44                 MPI_COMM_WORLD,
45                 MPI_STATUS_IGNORE);
46        tasks.push(task);
47        ++workRecv;
48      }
49    } while (flag);
50  }
51
52  void makeMoreWork(std::queue<double>& tasks,
53                    int& workMade,
54                    int maxWorkMade,
55                    int workMin,
```

```
56                    int workMax)
57 {
58   if (workMade < maxWorkMade)
59   {
60     auto newWork = random_int(1, 3);
61     for (auto i = 0; i < newWork; ++i)
62     {
63       tasks.push(random_double(workMin, workMax));
64       ++workMade;
65     }
66   }
67 }
68
69 int main(int argc, char** argv)
70 {
71   MPI_Init(&argc, &argv);
72
73   int rank, world_size;
74   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
75   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
76
77   const int workMin = 1, workMax = 3, maxWorkLoad = 16, amntToSend = 2,
78             maxWorkMade = random_int(1024, 2048);
79   int workMade = 0, workDone = 0, workSent = 0, workRecv = 0;
80   enum TOKEN token = TOKEN::WHITE;
81
82   std::queue<double> tasks;
83   if (rank == 0)
84   {
85     tasks.push(random_double(workMin, workMax));
86
87     do // while token == black
88     {
89       token = TOKEN::WHITE;
90       while (!tasks.empty())
91       {
92         process(tasks.front());
93         tasks.pop();
94         ++workDone;
95
96         /* Create more work */
97         makeMoreWork(tasks, workMade, maxWorkMade, workMin, workMax);
98
99         /* Check for incoming work */
100        getNewWork(tasks, workRecv);
101
```

```
102         /* Send work to other processes */
103         if (tasks.size() > maxWorkLoad)
104         {
105           for (auto i = 0; i < amntToSend; ++i)
106           {
107             int dest = random_int(rank + 1, rank + world_size - 1) %
                    world_size;
108             MPI_Send(&tasks.front(),
109                      1,
110                      MPI_DOUBLE,
111                      dest,
112                      TAG::NEW_WORK,
113                      MPI_COMM_WORLD);
114             tasks.pop();
115             ++workSent;
116           }
117         }
118       }
119
120       /* Send Token */
121       MPI_Send(
122         &token, 1, MPI_INT, rank + 1 % world_size, TAG::TOKEN,
              MPI_COMM_WORLD);
123
124       std::cout << rank << " -- send token -> "
125                 << (token == TOKEN::BLACK ? "BLACK" : "WHITE") << '\n';
126
127       /* Receive Token */
128       MPI_Recv(&token,
129                1,
130                MPI_INT,
131                world_size - 1,
132                TAG::TOKEN,
133                MPI_COMM_WORLD,
134                MPI_STATUS_IGNORE);
135
136       std::cout << rank << " <- recv token -- "
137                 << (token == TOKEN::BLACK ? "BLACK" : "WHITE") << '\n';
138
139       if (token == TOKEN::BLACK)
140       {
141         enum TAG action = TAG::CONTINUE;
142         for (auto i = 1; i < world_size; ++i)
143         {
144           MPI_Send(&action, 1, MPI_INT, i, TAG::ACTION, MPI_COMM_WORLD)
                ;
```

```
145           }
146         }
147
148       } while (token == TOKEN::BLACK);
149
150       /* Kill all processes */
151       enum TAG action = TAG::FINALIZE;
152       for (auto i = 1; i < world_size; ++i)
153       {
154         MPI_Send(&action, 1, MPI_INT, i, TAG::ACTION, MPI_COMM_WORLD);
155       }
156     }
157     else // SLAVE
158     {
159       tasks.push(random_double(workMin, workMax));
160       enum TAG action;
161       do // while action == continue
162       {
163         token = TOKEN::WHITE;
164         while (!tasks.empty())
165         {
166           process(tasks.front());
167           tasks.pop();
168           ++workDone;
169
170           /* Create more work */
171           makeMoreWork(tasks, workMade, maxWorkMade, workMin, workMax);
172
173           /* Check for incoming work */
174           getNewWork(tasks, workRecv);
175
176           /* Send work to other processes */
177           if (tasks.size() > maxWorkLoad)
178           {
179             for (auto i = 0; i < amntToSend; ++i)
180             {
181               int dest = random_int(rank + 1, rank + world_size - 1) %
                      world_size;
182               if (dest < rank) token = TOKEN::BLACK;
183               MPI_Send(&tasks.front(),
184                        1,
185                        MPI_DOUBLE,
186                        dest,
187                        TAG::NEW_WORK,
188                        MPI_COMM_WORLD);
189               tasks.pop();
```

```
190             ++workSent;
191           }
192         }
193       }
194
195       enum TOKEN recvToken;
196       /* Receive Token */
197       MPI_Recv(&recvToken,
198                1,
199                MPI_INT,
200                rank - 1,
201                TAG::TOKEN,
202                MPI_COMM_WORLD,
203                MPI_STATUS_IGNORE);
204
205       if (recvToken == TOKEN::BLACK)
206       {
207         token = TOKEN::BLACK;
208       }
209       std::cout << rank << " <- recv token -- "
210                 << (recvToken == TOKEN::BLACK ? "BLACK" : "WHITE") << '
                      \n';
211
212       /* Send Token */
213       MPI_Send(&token,
214                1,
215                MPI_INT,
216                (rank + 1) % world_size,
217                TAG::TOKEN,
218                MPI_COMM_WORLD);
219
220       std::cout << rank << " -- send token -> "
221                 << (token == TOKEN::BLACK ? "BLACK" : "WHITE") << '\n';
222
223       MPI_Recv(
224         &action, 1, MPI_INT, 0, TAG::ACTION, MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);
225
226     } while (action == TAG::CONTINUE);
227   }
228
229   std::cout << rank << " -- FINALIZE -- " << '\n';
230   std::cout << rank << " -- work done -- " << workDone << '\n';
231
232   int totalWorkDone;
```

```
233   MPI_Allreduce(&workDone, &totalWorkDone, 1, MPI_INT, MPI_SUM,
          MPI_COMM_WORLD);
234
235   double percentWorkDone = (double)workDone / totalWorkDone * 100;
236   std::cout << rank << "-- percent done -- " << percentWorkDone << '\n'
          ;
237
238   if (rank == 0) std::cout << "Total Work Done " << totalWorkDone << '\
          n';
239   MPI_Finalize();
240
241   return EXIT_SUCCESS;
242 }
```

0.2 random.hpp

```
1  #ifndef RANDOM_HPP
2  #define RANDOM_HPP
3
4  #include <algorithm>
5  #include <functional>
6  #include <random>
7
8  /**
9   * Generate a random number from [low, high]
10  *
11  * @param low  The lower bound
12  * @param high The upper bound
13  * @return     A random number on the range [low, high]
14  */
15 int random_int(int low, int high)
16 {
17   static std::random_device rd;
18   static std::mt19937 mt(rd());
19   std::uniform_int_distribution<> dist(low, high);
20   return dist(mt);
21 }
22
23 /**
24  * Generate a random number from [low, high)
25  *
26  * @param low  The lower bound
27  * @param high The upper bound
28  * @return     A random number on the range [low, high)
29  */
30 double random_double(double low, double high)
31 {
```

```
32    static std::random_device rd;
33    static std::mt19937 mt(rd());
34    std::uniform_real_distribution<> dist(low, high);
35    return dist(mt);
36  }
37
38  /**
39   * Fill a container from [first, last) with random numbers from [low,
         high]
40   *
41   * @param first Iterator to beginning of range to fill
42   * @param last  Iterator to end of range to fill
43   * @param low   The lower bound
44   * @param high  The upper bound
45   */
46  template <typename it>
47  void random_int_fill(it first, it last, const int low, const int high)
48  {
49    static std::random_device rd;
50    static std::mt19937 mt(rd());
51    std::uniform_int_distribution<> dist(low, high);
52    std::generate(first, last, std::bind(dist, mt));
53  }
54
55  /**
56   * Fill a container from [first, last) with random numbers from [low,
         high)
57   *
58   * @param first Iterator to beginning of range to fill
59   * @param last  Iterator to end of range to fill
60   * @param low   The lower bound
61   * @param high  The upper bound
62   */
63  template <typename it>
64  void random_double_fill(it first, it last, const double low, const
       double high)
65  {
66    static std::random_device rd;
67    static std::mt19937 mt(rd());
68    std::uniform_real_distribution<double> dist(low, high);
69    std::generate(first, last, std::bind(dist, mt));
70  }
71
72  #endif
```

Output

```
# mpic++ -O3 main.cpp -o release.out

# mpiexec -n 4 release.out

0 -- send token -> WHITE
1 <- recv token -- WHITE
1 -- send token -> BLACK
2 <- recv token -- BLACK
2 -- send token -> BLACK
3 <- recv token -- BLACK
3 -- send token -> BLACK
0 <- recv token -- BLACK
0 -- send token -> WHITE
1 <- recv token -- WHITE
1 -- send token -> WHITE
2 <- recv token -- WHITE
2 -- send token -> WHITE
3 <- recv token -- WHITE
3 -- send token -> WHITE
0 <- recv token -- WHITE

0 -- FINALIZE --
0 -- work done -- 900

1 -- FINALIZE --
1 -- work done -- 839

2 -- FINALIZE --
2 -- work done -- 974

3 -- FINALIZE --
3 -- work done -- 1202

Total Work Done 3915
0-- percent done -- 22.9885
1-- percent done -- 21.4304
2-- percent done -- 24.8787
3-- percent done -- 30.7024
```

Findings

From the report at the end of the program execution, each process processed between 20% and 30% of the total tasks. This means that despite producing a different amount of tasks, they were able to divide the tasks between themselves fairly well.