Laboratory for Parallel Programming
Prof. Dr. Felix Wolf

TECHNISCHE UNIVERSITÄT DARMSTADT

Aamer Shah, shah@cs.tu-darmstadt.de

Large-Scale Parallel Computing (WS 15/16)

Exercise 2

A solution will be presented on November 10th, 2015. Attendance is optional. The exercise solutions will not be corrected and graded.

Task 1

On HPC systems, many independent compute nodes are connected together through a network to form a single computing system. When a parallel program executes on such system, it executes an independent process on each compute node. As each node then has its own memory, applications use message passing to exchange data and information between the compute nodes.
In this task, we will implement such a message-passing-based application, but will execute it on our own systems. The program will be implemented in C and will use network sockets to pass messages.
The program will consist of a master task and three workers tasks. The objective of the program will be to read two matrices from a file and calculate its sum. The sum will be written back to an output file. The master task will read the two matrices and distribute them among all the tasks (master + workers). Each task will then calculate its partial sum of the matrices. The matrices will then be send back to the master. The master will collect all the partial sums to generate a complete sum matrix. This matrix will be written to an output file.
The attached archive file provides a skeleton for both the master and workers tasks. Use the `Makefile` to compile the programs and the `run_ex.sh` batch script to run the program. Extend the source files to implement matrix addition in the following steps:
The master performs the following steps:

a) Extend the master by first generating and sending ids to each worker task. The master itself has id 0.

b) Come up with a scheme to distribute the matrices among the tasks. This can be either distributing along rows, along columns or both.

c) Inform the workers how many rows and columns they are expected to receive. (Note: the master also gets a share)

d) Send each worker its share of the matrices

e) Calculate the local partial sum

f) Receive the partial sum from each worker

g) Generate the complete sum and write it to the output file

The workers perform the following steps:

a) Receive id from the master

b) Receive the number of rows and columns of the matrices

c) Receive the matrices from the master

d) Calculate the sum of the matrices

e) Send the matrices back to the master

Solution

Listing 1: Master

```c
#include <stdio.h>

#include <stdlib.h>
#include <sys/types.h>
#include<arpa/inet.h>
#include <sys/socket.h>
#include <unistd.h>
#include <time.h>

#include "common.h"

#define WORKER_COUNT (3)

int server_socket;
int worker_sockets[WORKER_COUNT];

void create_socket();
void accept_worker_connections();
void send_to_worker(int socket, int *data, int len);
void send_to_workers_same(int val);
void send_to_workers_multi(int *vals);
void recv_from_worker(int socket, int* data, int len);
void close_connections();

FILE* read_rows_cols(int* rows, int* cols);
void read_mat(FILE* fh, int rows, int cols, int mat[rows][cols]);

void write_mat(int rows, int cols, int mat[rows][cols]);
void sum_matric(int rows, int cols, int mat_1[rows][cols], int mat_2[
    rows][cols], int sum[rows][cols], int worker_rows);

void print_mat(int rows, int cols, int mat[rows][cols]);

int main(int argc, char* argv[])
{
  int rows, cols;

  printf("Master: creating socket\n");
  create_socket();
  printf("Master: waiting for workers\n");
  accept_worker_connections();

  printf("Master: reading rows and cols\n");
  FILE* fh = read_rows_cols(&rows, &cols);

  int mat_1[rows][cols];
  int mat_2[rows][cols];
  int sum[rows][cols];

  printf("Master: reading matrices\n");
  read_mat(fh, rows, cols, mat_1);
```

```
51    read_mat(fh, rows, cols, mat_2);
52
53    fclose(fh);
54
55    int worker_ids[WORKER_COUNT];
56
57    printf("Master: sending IDs to workers\n");
58
59    //TODO: Add code to send IDs to client here
60    int i;
61    for(i = 0; i < WORKER_COUNT; i++)
62    {
63      worker_ids[i] = i + 1;
64    }
65    send_to_workers_multi(worker_ids);
66
67    printf("Master: sending to clients the number of rows and columns
          for clients to read\n");
68    //TODO: Add code to send rows and columns to clients here
69    int worker_row = rows / (WORKER_COUNT + 1);
70    int worker_share = worker_row * cols;
71    send_to_workers_same(worker_row);
72    send_to_workers_same(cols);
73
74    printf("Master: sending matrices to clients\n");
75    //TODO: Add code to send the matrices to the clients here
76    for(i = 0; i < WORKER_COUNT; i++)
77    {
78      int worker_ind = worker_row * (i + 1);
79      send_to_worker(worker_sockets[i], &(mat_1[worker_ind][0]),
            worker_share);
80      send_to_worker(worker_sockets[i], &(mat_2[worker_ind][0]),
            worker_share);
81    }
82
83    printf("Master: calculating sum of my share\n");
84    //TODO: Calculate local sum here
85    sum_matric(rows, cols, mat_1, mat_2, sum, worker_row);
86
87    printf("Master: gathering sum\n");
88    //TODO: Gather partial sums from clients here
89    for(i = 0; i < WORKER_COUNT; i++)
90    {
91      int worker_ind = worker_row * (i + 1);
92      recv_from_worker(worker_sockets[i], &(sum[worker_ind][0]),
            worker_share);
93    }
94
95    printf("Master: writing output\n");
96    write_mat(rows, cols, sum);
97
98    printf("Master: closing connection\n");
99    close_connections();
100
101   return 0;
102 }
```

```
103
104
105 void create_socket()
106 {
107   server_socket = socket(AF_INET, SOCK_STREAM, 0);
108
109   if(server_socket == -1)
110   {
111     perror("Master: Can't create socket");
112     exit(-1);
113   }
114
115   struct sockaddr_in sock_addr;
116   sock_addr.sin_family = AF_INET;
117   sock_addr.sin_addr.s_addr = INADDR_ANY;
118   sock_addr.sin_port = htons(PORT_NUM);
119
120   if(bind(server_socket, (struct sockaddr *) &sock_addr, sizeof(
        sock_addr)) == -1)
121   {
122     perror("Master: binding error");
123   }
124
125   if(listen(server_socket, 3) == -1)
126   {
127     perror("Master: listening error");
128   }
129 }
130
131 void accept_worker_connections()
132 {
133   int i;
134
135   socklen_t addrlen;
136   struct sockaddr_in worker_address;
137
138   for(i = 0; i < WORKER_COUNT; i++)
139   {
140     printf("Master: waiting for client %d\n", i);
141     worker_sockets[i] = accept(server_socket, (struct sockaddr *) &
          worker_address, &addrlen);
142   }
143 }
144
145 void send_to_worker(int socket, int *data, int len)
146 {
147   send(socket, data, sizeof(int) * len, 0);
148 }
149
150 void send_to_workers_same(int val)
151 {
152   int i;
153   for(i = 0; i < WORKER_COUNT; i++)
154   {
155     send_to_worker(worker_sockets[i], &val, 1);
156   }
```

```
157 }
158
159 void send_to_workers_multi(int *vals)
160 {
161   int i;
162   for(i = 0; i < WORKER_COUNT; i++)
163   {
164     send_to_worker(worker_sockets[i], &(vals[i]), 1);
165   }
166 }
167
168 void recv_from_worker(int socket, int* data, int len)
169 {
170   recv(socket, data, sizeof(int) * len, 0);
171 }
172
173 FILE* read_rows_cols(int* rows, int* cols)
174 {
175   FILE* fh = fopen("input.dat", "r");
176
177   fscanf(fh, "%d%d", rows, cols);
178
179   return fh;
180 }
181
182 void read_mat(FILE* fh, int rows, int cols, int mat[rows][cols])
183 {
184   int r, c;
185   for(r = 0; r < rows; r++)
186   {
187     for(c = 0; c < cols; c++)
188     {
189       fscanf(fh, "%d", &(mat[r][c]));
190     }
191   }
192 }
193
194 void write_mat(int rows, int cols, int mat[rows][cols])
195 {
196   FILE* fh = fopen("output.dat", "w");
197
198   int r, c;
199   for(r = 0; r < rows; r++)
200   {
201     for(c = 0; c < cols; c++)
202     {
203       fprintf(fh, "%d ", mat[r][c]);
204     }
205     fprintf(fh, "\n");
206   }
207
208   fclose(fh);
209 }
210
211 void close_connections()
212 {
```

```
213   int i;
214   for(i = 0; i < WORKER_COUNT; i++)
215   {
216     close(worker_sockets[i]);
217   }
218   close(server_socket);
219 }
220
221 void sum_matric(int rows, int cols, int mat_1[rows][cols], int mat_2[
      rows][cols], int sum[rows][cols], int worker_rows)
222 {
223   int r, c;
224   for(r = 0; r < worker_rows; r++)
225   {
226     for(c = 0; c < cols; c++)
227     {
228       sum[r][c] = mat_1[r][c] + mat_2[r][c];
229     }
230   }
231 }
232
233 void print_mat(int rows, int cols, int mat[rows][cols])
234 {
235   int r, c;
236   for(r = 0; r < rows; r++)
237   {
238     for(c = 0; c < cols; c++)
239     {
240       printf("%d ", mat[r][c]);
241     }
242     printf("\n");
243   }
244 }
```

Listing 2: Worker

```
1  #include <stdio.h>
2
3  #include <stdlib.h>
4  #include <sys/types.h>
5  #include<arpa/inet.h>
6  #include <sys/socket.h>
7  #include <unistd.h>
8  #include <time.h>
9
10 #include "common.h"
11
12 int worker_socket;
13
14 void create_socket();
15 void send_to_master(int socket, int *data, int len);
16 void recv_from_master(int socket, int* data, int len);
17 void close_worker_connections();
18
19 void sum_matric(int rows, int cols, int mat_1[rows][cols], int mat_2[
      rows][cols], int sum[rows][cols]);
20 void print_mat(int rows, int cols, int mat[rows][cols]);
```

```
21
22 int main(int argc, char* argv[])
23 {
24   int rows, cols;
25   int my_id;
26
27   printf("Client: creating socket\n");
28   create_socket();
29
30   printf("Client: receiving my id from master\n");
31   //TODO: Recv id from master
32   recv_from_master(worker_socket, &my_id, 1);
33
34   printf("Client: My id is %d\n", my_id);
35
36   printf("Client %d: receiving my share of rows and columns\n", my_id
       );
37   //TODO: Recv the dimensions of my share of the matric
38   recv_from_master(worker_socket, &rows, 1);
39   recv_from_master(worker_socket, &cols, 1);
40
41   printf("Client %d: rows: %d, cols: %d\n", my_id, rows, cols);
42
43   int mat_1[rows][cols];
44   int mat_2[rows][cols];
45   int sum[rows][cols];
46
47   printf("Client %d: Receiving matrices from master\n", my_id);
48   //TODO: Recv both matrices from master
49   recv_from_master(worker_socket, &(mat_1[0][0]), rows * cols);
50   recv_from_master(worker_socket, &(mat_2[0][0]), rows * cols);
51
52   printf("Client %d: calculating sum of my share\n", my_id);
53   //TODO: Calculate local sum
54   sum_matric(rows, cols, mat_1, mat_2, sum);
55
56   printf("Client %d: Sending sum to master\n", my_id);
57   //TODO: Send sum to master
58   send_to_master(worker_socket, &(sum[0][0]), rows * cols);
59
60   printf("Client %d: closing connection\n", my_id);
61   close(worker_socket);
62
63   return 0;
64 }
65
66
67 void create_socket()
68 {
69   worker_socket = socket(AF_INET, SOCK_STREAM, 0);
70
71   if(worker_socket == -1)
72   {
73     perror("Client: Can't create socket");
74     exit(-1);
75   }
```

```
76
77    struct sockaddr_in sock_addr;
78    sock_addr.sin_family = AF_INET;
79    sock_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
80    sock_addr.sin_port = htons(PORT_NUM);
81
82    if(connect(worker_socket, (struct sockaddr *) &sock_addr, sizeof(
          sock_addr)) == -1)
83    {
84      perror("Client: connecting error");
85      exit(-1);
86    }
87  }
88
89  void send_to_master(int socket, int *data, int len)
90  {
91    send(socket, data, sizeof(int) * len, 0);
92  }
93
94  void recv_from_master(int socket, int* data, int len)
95  {
96    recv(socket, data, sizeof(int) * len, 0);
97  }
98
99  void sum_matric(int rows, int cols, int mat_1[rows][cols], int mat_2[
       rows][cols], int sum[rows][cols])
100 {
101   int r, c;
102   for(r = 0; r < rows; r++)
103   {
104     for(c = 0; c < cols; c++)
105     {
106       sum[r][c] = mat_1[r][c] + mat_2[r][c];
107     }
108   }
109 }
110
111 void print_mat(int rows, int cols, int mat[rows][cols])
112 {
113   int r, c;
114   for(r = 0; r < rows; r++)
115   {
116     for(c = 0; c < cols; c++)
117     {
118       printf("%d ", mat[r][c]);
119     }
120     printf("\n");
121   }
122 }
```

Task 2

In image processing, a median filter is used to reduce noise from an image. The median filter works by using a 2D window of a certain size. Then for each element of the image, the value of the element is replaced by calculating the median of the window at that point. This way, sudden change in values is

smoothed out and noise is reduced.
In this task, no implementation work is required. The task is to come up with a design of a message-passing-based application consisting of master and workers that apply median filter on an image. Perform the task in the following steps:

a) Identify how this task is different from matrix addition

b) Come up with a design of a median filter application by extending the design of the matrix multiplication program

c) What will happen if we want to apply the filter twice on the same image (of course without running the program twice)?

Task 3

Based on the above experiences, if you want to make a utility library for message passing applications, what kind of functions will you provide to the user?