MONASH University

FIT3143 Tutorial Week 8

Lecturers: ABM Russel (MU Australia) and Vishnu Monn (MU Malaysia)

FAULTS & SIMPLE PARTITIONING

OBJECTIVES

- Review fault detection based on case studies
- Design solutions for a simple parallel partitioning problem

Note: Tutorials are not assessed. Nevertheless, please attempt the questions to improve your unit comprehension in preparation for the labs, assignments, and final assessments.

QUESTIONS

1. In the enclosed case study in Week 7 of Moodle (*The Ariane 5 Launcher Failure*), review this case study. Identify and discuss the type of failure which lead to the Ariane 5 launch failure in 1996.

   This case study content contains the description of the failure, which is software related. However, human error is also contributed to this failure due to improper testing prior to launch.

2. On July 4, 1997, the Mars pathfinder mission landed flawlessly on the Martial surface. However, later its communication failed due to a design flaw in the real-time embedded software kernel VxWorks. The problem was later diagnosed to be caused due to priority inversion. Explain the concept of priority inversion and the type of faulty which would be associated with this condition. Suggest a solution to address this fault.

   – Priority inversion: Low priority task **LP** locks file **F**
   – High priority task **HP** is scheduled next, it also needs to lock file **F**
   – A medium priority **MP** task (with high CPU requirement) becomes ready to run
   – **MP** is the highest priority unblocked task, its allowed to run, consumes all CPU
   – **LP** has no CPU, it stops. **HP 's** priority < **MP's** priority (priority inversion)

   Typically, in real time system kernels, semaphores are used synchronize communication between threads. However, semaphores lack the ability to elevate the priority of a lower task which is currently holding the semaphore. The solution is to use mutex. Using a mutex, a lower priority task holding this mutex will be elevated to the same priority level of the higher priority task. This resolves the issue of a medium priority task running indefinitely.

3. Using the C language with message passing interface (MPI), write a complete parallel program to calculate the definite integral of:

$$f(x,y) = \frac{(x+y)^2}{2y+0.1}$$

from (0,0) to (1,1) using rectangular columns – that is, the 3D version of the numerical integration by rectangles.

Only the process with rank 0 should report the resultant volume; the other processes should only calculate their subtotal and send it to process 0. Include the necessary header files.

You may refer to the following C and MPI subroutines to assist you in this question:

```
FILE *fopen(const char *filename, const char *mode);
int fclose(FILE *stream);
int fscanf(FILE *stream, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int printf(const char *format, ...);
int scanf(const char *format, ...);

void srand48(long int seedval);
double drand48(void);
long int lrand48(void);

int MPI_Init(int *argc, char **argv);
int MPI_Comm_size(MPI_Comm comm, int *size);
int MPI_Comm_rank(MPI_Comm comm, int *rank);
int MPI_Finalize();
int MPI_Send (void *buf,int count, MPI_Datatype datatype, int
dest, int tag, MPI_Comm comm);
int MPI_Recv (void *buf,int count, MPI_Datatype datatype, int
source, int tag, MPI_Comm comm, MPI_Status *status);
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
int MPI_Isend(const void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm, MPI_Request
*request);
```

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype
datatype, int root, MPI_Comm comm);
```

Note: This question is based on a past year exam question. There is no need to compile the code. Focus on writing a logically correct code for this question.

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"

#define STEPSIZE 0.0001

double f(double x, double y)
{
     return ((x+y)*(x+y))/(2*y+0.1);
}


int main(int argc, char *argv[])
{
double x,y;
double total;
int my_rank;
int p;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

double range = 1.0/p;
double start = my_rank*range;
double end = start+range;
double subtotal = 0;

for (x=start; x<end; x+=STEPSIZE)
for(y=0; y<1; y+=STEPSIZE)
subtotal +=
STEPSIZE*STEPSIZE*f(x+0.5*STEPSIZE,y+0.5*STEPSIZE);

MPI_Reduce(&subtotal, &total, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

if (my_rank == 0)
     printf("Total volume = %f\n",total);
MPI_Finalize();
return 0;

}
```

4. Explain the behavior of the following source code. Specifically, describe the behavior of the workload distribution, MPI send and receive functions under non-uniformed data distribution conditions, MPI reduce and the parallel computation section of the code.

   Note: Use the provided line numbers when describing the code:

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <time.h>
#include <math.h>
#include "mpi.h"

#define BUFF_ELEMENTS 1201

int main(int argc, char* argv[])
{
      int my_rank, p;
      int tag = 0, n = 0, i = 0, offset = 0, temp = 0;
      MPI_Status status;
      int *pSendBuffer = 0, *pRecvBuffer = 0;
      int *pPerfectSquare = 0, *pStoreBuff = 0;
      int chunkDistSize, count = 0, totalCount = 0;
      FILE *pFile;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
      MPI_Comm_size(MPI_COMM_WORLD, &p);

      chunkDistSize = (my_rank == (p-1))? BUFF_ELEMENTS/p +
                              BUFF_ELEMENTS%p : BUFF_ELEMENTS/p;
      pRecvBuffer = (int*)malloc(chunkDistSize * sizeof(int));
      pPerfectSquare = (int*)malloc(chunkDistSize * sizeof(int));

      switch(my_rank){
            case 0:{
                  srand(time(0));
                  pSendBuffer = (int*)malloc(BUFF_ELEMENTS*sizeof(int));
                  for(i = 0; i < BUFF_ELEMENTS; i++){
                        pSendBuffer[i] = rand() % 1501;
                  }
                  memcpy(pRecvBuffer, pSendBuffer, (chunkDistSize *
                                                   sizeof(int)));
                  offset += chunkDistSize;
                  for(i = 1; i < p; i++){
                        MPI_Send((int*)pSendBuffer + offset, chunkDistSize,
                                          MPI_INT, i, 0, MPI_COMM_WORLD);
                        offset += chunkDistSize;
                  }
                  break;
            }
            default:{
                  MPI_Recv(pRecvBuffer, chunkDistSize, MPI_INT, 0, 0,
                                          MPI_COMM_WORLD, &status);
                  break;
            }
      }

      for(i = 0; i < chunkDistSize; i++){
            temp = (int)sqrt((double) pRecvBuffer[i]);
            if(temp*temp == pRecvBuffer[i]){
                  pPerfectSquare[count] = pRecvBuffer[i];
```

```c
                count++;
            }
        }

        MPI_Reduce(&count, &totalCount, 1, MPI_INT, MPI_SUM, 0,
                                                        MPI_COMM_WORLD);

        switch(my_rank){
            case 0:{
                pStoreBuff = (int*)malloc(totalCount*sizeof(int));
                offset = 0;
                memcpy(pStoreBuff, pPerfectSquare, (count*sizeof(int)));
                offset += count;
                for(int i = 1; i < p; i++){
                    MPI_Recv(&count, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
                                                            &status);
                    MPI_Recv((int*)pStoreBuff + offset, count, MPI_INT,
                                    i, 0, MPI_COMM_WORLD, &status);
                    offset += count;
                }

                pFile = fopen("Perfer_Square_Numbers.txt", "w");
                for(i = 0; i < totalCount; i++){
                    fprintf(pFile, "%d\n", pStoreBuff[i]);
                }
                fclose(pFile);
                free(pStoreBuff);
                free(pSendBuffer);
                break;
            }
            default:{
                MPI_Send(&count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
                MPI_Send(pPerfectSquare, count, MPI_INT, 0, 0,
                                                        MPI_COMM_WORLD);
                break;
            }
        }
        free(pRecvBuffer);
        free(pPerfectSquare);
        MPI_Finalize();
        return 0;
}
```

- **Lines 24 – 26: Computes the workload distribution between the processes and heap memory allocation based on the workload for each process. The last node is expected to factor in additional workload if the division returns a remainder.**
- **Lines 31 – 37: Generates a random number into a heap array and as assigns a section of the random numbers to the root node based on its workload.**
- **Lines 39 – 50: The root process uses `MPI_Send` with an offset to contiguously read and send the random numbers to other processes in the group based on the computed workload distribution algorithm. The remaining processes use `MPI_Recv` to receive the sent numbers.**
- **Lines 53 – 60: Each process (including the root) determines if a random number is a perfect square or otherwise. The process saves both the list and number of perfect squares. This process is executed in parallel among the processors.**
- **Line 61: Data reduction of the number of perfect squares into a grand total using `MPI_Reduce`.**
- **Lines 66 – 84: The root process collects the list of perfect squares from itself and other processes into a heap array for storage into a file. An offset is used to ensure that the root process contiguously writes the received list of perfect squares into its array.**

- **Lines 88 – 90: The other processes send the number and list of perfect squares to the root process.**