Zadania z PP

Bc. Marián Sabat

November 23, 2019

Contents

Zadanie 1

1.1 Inštalácia

Použitie OpenMPI.

- Inštalácia na sýstéme Linux Manjaro:

  ```
  pacman -S openmpi
  ```

- Kompilácia programu:

  ```
  mpicc -o s1 sample1.c
  ```

- Spustenie programu:

  ```
  mpirun -n 3 --use-hwthread-cpus ./s1
  ```

1.2 Sample 1

```
/*
 * Transmit a message in a 3-process system.
 */
#include <mpi.h>
#include <stdio.h>

#define BUFSIZE 10
int main(int argc, char** argv){
{
// Definicia pomocnych premennych
    ===================================
  int size, rank;
  int slave;
  int buf[BUFSIZE];
  int n, value;
  float rval;
  MPI_Status status;
```

```
// Inicializacia MPI
    ================================================
  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &size);

// Kontrola poctu procesov
    ==========================================
if (size==3) {/* Correct number of processes */}

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// Samotne programy
    ================================================
/*
    * Rank 0 = Master
    * ostatne ranky = Slaves
*/
if (rank==0) { // Master proces
    ====================================
  // Hodnoty pre vypocty
  buf[0]=5; buf[1]=1; buf[2]=8; buf[3]=7; buf[4]=6;
  buf[5]=5; buf[6]=4; buf[7]=2; buf[8]=3; buf[9]=1;

  // Odosielanie hodnot na ostatne procesy
  printf("\n Sending the values {5,1,8,7,6,5,4,2,3,1}");
  printf("\n ----------------------------");
  for (slave=1;slave < size;slave++) {
    printf("\n from master %d to slave %d",rank,slave);
    MPI_Send(buf, 10, MPI_INT, slave, 1, MPI_COMM_WORLD);
  }

  // Spracovanie vysledkov
  printf("\n\n Receiving the results from slaves");
  printf("\n ---------------------------------");
  MPI_Recv(&value, 1, MPI_INT, 1, 11, MPI_COMM_WORLD, &
      status);
  printf("\n Minimum %4d from slave 1",value);
  MPI_Recv(&value, 1, MPI_INT, 2, 21, MPI_COMM_WORLD, &
      status);
  printf("\n Sum     %4d from slave 2",value);
  MPI_Recv(&value, 1, MPI_INT, 1, 12, MPI_COMM_WORLD, &
      status);
  printf("\n Maximum %4d from slave 1",value);
  MPI_Recv(&rval, 1, MPI_FLOAT, 2, 22, MPI_COMM_WORLD, &
      status);
  printf("\n Average %4.2f from slave 2\n",rval);

} else {
  if (rank==1) { // Min/Max proces
```

```
      ===================================
    // Cakanie na hodnoty od mastra
    MPI_Recv(buf, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, &status
        );

    // Vypocet minima
    value=100;
    for (n=0;n<BUFSIZE;n++) {
      if (value>buf[n]) { value=buf[n]; }
    }
    MPI_Send(&value, 1, MPI_INT, 0, 11, MPI_COMM_WORLD);

    // Vypocet maxima
    value=0;
    for (n=0;n<BUFSIZE;n++) {
      if (value<buf[n]) { value=buf[n]; }
    }
    MPI_Send(&value, 1, MPI_INT, 0, 12, MPI_COMM_WORLD);

  } else { // Sum/Avg proces
      ========================================
    // Hodnoty od mastra
    MPI_Recv(buf, 10, MPI_INT, 0, 1, MPI_COMM_WORLD, &status
        );

    // Vypocet Sum
    value=0;
    for (n=0;n<BUFSIZE;n++) {
      value=value+buf[n];
    }
    MPI_Send(&value, 1, MPI_INT, 0, 21, MPI_COMM_WORLD);

    // Vypocet Avg
    rval= (float) value / BUFSIZE;
    MPI_Send(&rval, 1, MPI_FLOAT, 0, 22, MPI_COMM_WORLD);
  }
}
}

// Ukoncenie MPI
    ======================================================
MPI_Finalize();
return(0);
}
```

1.3 Sample 2

```
/*
 * Collective communication in a 4-process system.
```

```
 */
#include <mpi.h>
#include <stdio.h>
#define BUFSIZE 10

int main(int argc, char** argv)
{
// Definicia pomocnych premennych
    ====================================
  int size, rank;
  int buf[BUFSIZE]={0,0,0,0,0,0,0,0,0,0};
  int n, value;
  float rval;
  MPI_Status status;

// Inicializacia MPI
    ================================================
  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &size);

// Kontrola poctu procesov
    ========================================
if (size==4) { /* Correct number of processes */

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// Rozposlanie hodnot na vsetky procesy
    ==============================
if (rank==0) {
  buf[0]=5; buf[1]=1; buf[2]=8; buf[3]=7; buf[4]=6;
  buf[5]=5; buf[6]=4; buf[7]=2; buf[8]=3; buf[9]=1;
  printf("\n Broadcasting {5,1,8,7,6,5,4,2,3,1}");
  printf("\n ---------------------------------------------")
     ;
}

  MPI_Bcast(buf,10,MPI_INT,0,MPI_COMM_WORLD);

// Samotne programy
    ================================================
if (rank==0) { // Master proces
    ====================================
  // Vypocet minima
  printf("\n Computing by master and slaves");
    value=100;
    for (n=0;n<BUFSIZE;n++) {
      if (value>buf[n]) { value=buf[n]; }
    }
```

```
  // Spracovanie vysledkov
  printf("\n Minimum %4d  by  master ",value);
  MPI_Recv(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status
      );
  printf("\n Maximum %4d from slave 1",value);
  MPI_Recv(&value, 1, MPI_INT, 2, 0, MPI_COMM_WORLD, &status
      );
  printf("\n Sum     %4d from slave 2",value);
  MPI_Recv(&rval, 1, MPI_FLOAT, 3, 0, MPI_COMM_WORLD, &
      status);
  printf("\n Average %4.2f from slave 3\n",rval);

} else if (rank==1) { // Vypocet Maxima
    ==============================
    value=0;
    for (n=0;n<BUFSIZE;n++) {
      if (value<buf[n]) { value=buf[n]; }
    }
    MPI_Send(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
} else if (rank==2) { // Vypocet Sum
    ================================
    value=0;
    for (n=0;n<BUFSIZE;n++) {
      value=value+buf[n];
    }
/* send sum to master */
    MPI_Send(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
/* send sum to slave 3 */
    MPI_Send(&value, 1, MPI_INT, 3, 0, MPI_COMM_WORLD);
} else if (rank==3) { // Vypocet Avg
    ================================
    MPI_Recv(&value, 1, MPI_INT, 2, 0, MPI_COMM_WORLD, &
        status);
    rval= (float) value / BUFSIZE;
    MPI_Send(&rval, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
}
}

// Ukoncenie MPI
    ========================================================
MPI_Finalize();
return(0);

}
```

Zadanie 2

2.1 Dekompozícia

Vstupný retazec je upravený tak aby jeho dĺžka bola deliteľná počtom procesov, a zároveň sa nestratil žiaden znak. Následne je retazec rozelený symetricky medzi procesy. Ak je na vstupe 5 procesov, tak reťazec je rozdelený na 4 rovnaké časti. Nakoľko ide o jednoduchú dekompozíciu môžme povedať, že ak je počet procesorov n tak celkový čas výpočtu bude zmenšený $n-1$ násobne.

Príklad:
Vstup: "abcdefg"
Počet procesov: 4
Retazec je upravený na "abcdefg ".
Na proces 1 je poslaný retazec: "abc"
Na proces 2 je poslaný retazec: "def"
Na proces 3 je poslaný retazec: "g "

2.2 Algoritmus

Pozn.: pri jednotlivých krokoch je označený typ procesu, na ktorom sa daná činnosť vykonáva.

1(M) Vstup: hľadaný znak, dĺžka reťazca.

2(M) Vypočet dĺžky reťazca pre jeden proces.

3(M) Vstup: reťazec.

4(M) Rozdelenie reťazca na úseky vypočítanej dĺžky.

5(M) Odoslanie vstupov na Slave procesy.

6(S) Čakanie na vstupy od Mastra

7(S) Prechádzaním každého znaku prijatého prodreťazca, spočítanie výskytu hľdaného znaku

8(S) Odoslanie výsledku na Mastra

9(M) Čakanie na výsledky zo Slave procesov

10(M) Výpis výsledku

2.3 Kód

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <errno.h>

void fillWithSpaces(int, char*);
int countChar(char, int, char*);

int main(int argc, char** argv) {
// Inicializacia premennych
    ========================================
    int size, rank;

    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// Kontrola poctu procesov
    ========================================
    if(size <= 1) {
        perror("Not enought processes");
        return 1;
    }

    if(rank == 0) { // Master
        ====================================
        // Vstup hladaneho znaku
        char search;
        printf("Character to serach for:\n");
        scanf("%c", &search);

        // Vstup velkosti retazca
        int inputLength;
        printf("Input size:\n");
        scanf("%d", &inputLength);
```

```
        int chunkLength = (int)ceil((double)inputLength / (
            size - 1));

        // Vstup retazca
        inputLength = chunkLength * (size - 1);
        char inputString[inputLength];
        fillWithSpaces(inputLength, inputString);
        printf("Input string:\n");
        for(int i = 0; i < inputLength; i++) {
            inputString[i] = getchar();
        }
        //scanf("%s", inputString);

        // Rozdelenie retazca a odoslanie na ostatne procesy
        for(int i = 1; i < size; i++) {
            char buff[chunkLength];
            memcpy(buff, &inputString[(i - 1) * chunkLength
                ], chunkLength);
            MPI_Send(&chunkLength, 1, MPI_INT, i, 1,
                MPI_COMM_WORLD);
            MPI_Send(&search, 1, MPI_CHAR, i, 2,
                MPI_COMM_WORLD);
            MPI_Send(buff, chunkLength, MPI_CHAR, i, 3,
                MPI_COMM_WORLD);
        }

        // Spracovanie vysledkov z ostatnych procesov
        int count = 0;
        for(int i = 1; i < size; i++) {
            int tempCount;
            MPI_Recv(&tempCount, 1, MPI_INT, i, 0,
                MPI_COMM_WORLD, &status);
            count += tempCount;
        }

        printf("Char %c is in input string %d times\n",
            search, count);

    } else { // Slaves
        ================================================
        // Vstupy
        int strLength;
        MPI_Recv(&strLength, 1, MPI_INT, 0, 1,
            MPI_COMM_WORLD, &status);

        char c;
        MPI_Recv(&c, 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD, &
            status);

        char str[strLength];
```

```
        MPI_Recv(str, strLength, MPI_CHAR, 0, 3,
            MPI_COMM_WORLD, &status);

        // Vypocet a odoslanie na mastra
        int count = countChar(c, strLength, str);
        MPI_Send(&count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}

// Pomocna f-cia na inicializaciu retazca
// Naplni retazec str, velkosti size, znakom medzery
void fillWithSpaces(int size, char* str) {
    for (int i = 0; i < size; i++) {
        str[i] = '␣';
    }
}


// Vypocet vyskytu znaku c v retazci str o velkosit strLen
int countChar(char c, int strLen, char* str) {
    int counter = 0;
    for(int i = 0; i < strLen; i++) {
        if(str[i] == c) counter++;
    }
    return counter;
}
```

Zadanie 3

3.1 Dekompozícia

Opäť ide of jednoduchú dekompozíciu. Pomenujme vstupné matice X a Y (rozmerov $m \times m$, teda výsledkom je matica $X \otimes Y$. Matica X je rozdelená symetricky medzi $n-1$ procesov. Každý proces teda musí maximálne vypočítať $\frac{m^2}{n-1}$ skalárnych súčinov.

Príklad:
Matica X: $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$
Počet procesov: 3
Na jednotlivé procesy sú odoslané súradnice začiatočných a koncových prvkov matice.
Na proces 1 je sú odoslané body $[0, 0]$ a $[1, 1]$.
Na proces 2 je sú odoslané body $[1, 2]$ a $[2, 2]$.
Takto vieme, že proces 1 bude vykonávať výpočty: $1 \times Y, 2 \times Y, 3 \times Y, 4 \times Y, 5 \times Y$.
A proces 2 bude vykonávať výpočty: $6 \times Y, 7 \times Y, 8 \times Y, 9 \times Y$.

3.2 Algoritmus

Pozn.: pri jednotlivých krokoch je označený typ procesu, na ktorom sa daná činnosť vykonáva.

1 Inicializácia matíc X a Y.

2(M) Výpočet prerozdelenia matice X.

3(M) Rozposlanie prerozelení na Slave procesy.

4(S) Čakanie na informácie, ktoré výpočty má proces vykonať.

5(S) Výpočet skalárnych súčinov.

6(S) Odoslanie podmatíc na Mastra.

7(M) Čakanie na výsledky zo Slave procesov

8(M) Skladanie výslednej matice

9(M) Výpis výsledku

3.3 Kód

```
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <errno.h>


void printMatrix(int** mat, int h, int w);
int** scalarProduct(int**, int, int, int);
void fillLoc(int* loc, int start, int end, int val);
int** alloc2DMatrix(int rows, int cols);
void free2DMatrix(int** mat);
void matrixIntegration(int** MAT1, int** mat2, int R, int C,
     int w, int h);

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

// Inicializacia matic
    ================================================

    int n = 3;
    int** X = alloc2DMatrix(n, n);
    int** Y = alloc2DMatrix(n, n);

    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            X[i][j] = i*n + j;
            Y[i][j] = n+1+j;
        }
    }

//MPI Init
    ============================================================

    int size, rank;

    MPI_Status status;
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(size <= 1) {
        perror("Not enought processes");
        return 1;
    }

//Vypocty
    ============================================================


    if(rank == 0) {
        // velkost vstupnych matic (pocet prvkov)
        int inputL = n * n;
        // pocet nasobeni skalarom na 1 proces
        int chunkL = (int)ceil((double)inputL/ (size - 1));
        // rozdelenie vypoctov medzi procesmi
        int* calcLoc = (int*) malloc(inputL * sizeof(int));

        // rozdelenie a odoslanie na slave procesy
        for(int i = 1; i < size; i++) {
            int b = (i-1) * chunkL;
            int e = i * chunkL - 1;
            e = e < inputL ? e : inputL - 1;

            int info[4] = {b / n, b % n, e / n, e % n};
            fillLoc(calcLoc, info[0] * n + info[1], info[2]
                * n + info[3], i);
            MPI_Send(info, 4, MPI_INT, i, 1, MPI_COMM_WORLD)
                ;
        }

        // skladanie vyslednej matice
        int** resultMatrix = alloc2DMatrix(inputL, inputL);
        for(int i = 0; i < inputL; i++) {
            int** recvMat = alloc2DMatrix(n,n);
            MPI_Recv(&(recvMat[0][0]), inputL, MPI_INT,
                calcLoc[i], i, MPI_COMM_WORLD, &status);
            matrixIntegration(resultMatrix, recvMat, (i / n)
                 * n, (i % n) * n, n, n);
            free2DMatrix(recvMat);
        }

        // vypis vysledku
        printMatrix(X, n, n);
        printf("\n----------------------\n");
        printMatrix(Y, n, n);
        printf("\n----------------------\n");
        printMatrix(resultMatrix, inputL, inputL);
```

```
        printf("\n");

        free2DMatrix(resultMatrix);
        free(calcLoc);

    } else {
        // info o tom ktore vypocty sa maju vykonat
        int info[4];
        MPI_Recv(info, 4, MPI_INT, 0, 1, MPI_COMM_WORLD, &
            status);

        // vypocet matic a odoslanie na mastra
        for(int i = info[0]; i <= info[2]; i++) {
            int j = i == info[0] ? info[1] : 0;
            int jEnd = i == info[2] ? info[3] : n - 1;
            for(j; j <= jEnd; j++) {
                int** sProduct = scalarProduct(Y, n, n, X[i
                    ][j]);
                int flag = i * n + j;
                MPI_Send(&(sProduct[0][0]), n*n, MPI_INT, 0,
                     flag, MPI_COMM_WORLD);
                free2DMatrix(sProduct);
            }
        }

    }

    free2DMatrix(X);
    free2DMatrix(Y);

    MPI_Finalize();

    return 0;
}

// Nasobenie matice mat, o vyske matH a sirke matW, skalarom
     scal
int** scalarProduct(int** mat, int matH, int matW, int scal)
     {
    int** retMat = alloc2DMatrix(matH, matW);
    for(int i = 0; i < matH; i++) {
        for(int j = 0; j < matW; j++) {
            retMat[i][j] = mat[i][j] * scal;
        }
    }

    return retMat;
}

// Naplnenie pola loc hodnotou val od indexu start po end
```

```
void fillLoc(int* loc, int start, int end, int val) {
    for(int i = start; i <= end; i++) {
        loc[i] = val;
    }
}

// Alokovanie pamate pre maticu vysky rows a sirky cols
int** alloc2DMatrix(int rows, int cols) {
    int* data = (int*) malloc(rows * cols * sizeof(int));
    int** array = (int**) malloc(rows * sizeof(int*));
    for(int i = 0; i < rows; i++) {
        array[i] = &(data[cols * i]);
    }
    return array;
}

// Uvolnenie matice alokovanej pomocou alloc2DMatix
void free2DMatrix(int** mat) {
    free(mat[0]);
    free(mat);
}

// Vypis matice na stdout
void printMatrix(int** mat, int h, int w) {
    printf("[");
    for(int i = 0; i < h; i++) {
        printf("[");
        for(int j = 0; j < w; j++) {
            printf("%d,", mat[i][j]);
        }
        printf("],");
    }
    printf("]");
}

// Vlozenie matice mat2 (rozmerov w, h) do matice MAT1 na
   poziciu [R, C]
void matrixIntegration(int** MAT1, int** mat2, int R, int C,
     int w, int h) {
    for(int i = 0; i < w; i++) {
        for(int j = 0; j < h; j++) {
            MAT1[i + R][j + C] = mat2[i][j];
        }
    }
}
```