ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ “СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИИ И ИНФОРМАТИКИ”

Кафедра ВС

Лабораторная работа № 4
«РАСПРЕДЕЛЁННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ»

Выполнил:
ст. гр. МГ-165 Марков В.А.
Проверил:
Фульман В.О.

Новосибирск 2016

Задание на лабораторную работу

Базовые задания:

- Продемонстрировать запуск фонового процесса в системах GNU\Linux
- Разработать приложение, порождающее несколько процессов и выводящих информацию о каждом из них. В каждом процессе должны быть выведены значения идентификаторы: PID, PPID, GID, EGID, UID, EUID и т.п.
- Подготовить приложение, реализующее программу, представленную на рисунке 12. Разработать описание задачи, которой требуется для выполнения 3 вычислительных ядра и необходим запуск созданной программы. Запустите задачу на выполнение. Убедитесь, что приложение выдало правильные значения.

Запуск фонового процесса

```
*** daemon1.c file ***
int main(void)
{
    close(STDIN_FILENO);
    close(STDOUT_FILENO);
    close(STDERR_FILENO);

    setsid();   // assing new sid

    for (int i = 0; i < 30; i++) {
        sleep(1);
    }

    return 0;
}

./daemon1 &
```

Листинг приложение, порождающее несколько процессов и выводящих информацию о каждом из них

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void show_info(char *whois)
{
    printf("*** %s ***\n", whois);
    printf("PID  : %u\n", getpid());
    printf("PPID : %u\n", getppid());
    printf("UID  : %u\n", getuid());
    printf("EUID : %u\n", geteuid());
    printf("GID  : %u\n", getgid());
    printf("EGID : %u\n", getegid());
    printf("PG   : %u\n", getpgrp());
    printf("SID  : %u\n", getsid(getpid()));
}
```

```
int main(void)
{
    int pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(1);
    }

    if (pid == 0) {
        sleep(1);
        show_info("Child");
    } else {
        sleep(3);
        show_info("Parent");
    }

    return (0);
}
```

Листинг mpi задание

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

int main(int argc, char** argv)
{
    int world_size, world_rank, name_len;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    /* Initialize the MPI environment */
    MPI_Init(NULL, NULL);

    MPI_Comm_size(MPI_COMM_WORLD, &world_size);            // Get the number of
processes
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);          // Get the rank of the
process
    MPI_Get_processor_name(processor_name, &name_len);   // Get the name of the
processor

    /* Print off a hello world message */
    printf("Hello world from processor %s, rank %d out of %d processors\n",
            processor_name, world_rank, world_size);

    /* Finalize the MPI environment */
    MPI_Finalize();
    return 0;
}

*** Task.job file ***
#PBS -N mpi_task
#PBS -l nodes=3:ppn=1
#PBS -j oe

cd $PBS_O_WORKDIR

mpiexec ./mpi_task
```

Основные задания:

- Доработать программу, выполненную на этапе 3 простого задания так, чтобы выводилась информация о каждом процессе в MPI приложении (информация аналогична пункту 2 простого задания).
- Разработайте программу умножения матриц с использованием библиотеки MPI. Каждая ветвь распределённого приложения должна выполнять умножение матриц (одинаковых). Продемонстрируйте, что результат умножения матриц во всех ветвях получился правильным. Сравните время расчета по всем ветвям.

Листинг mpi-задание

```
void show_info(char *whois)
{
    printf("*** %s ***\n", whois);
    printf("PID  : %u\n", getpid());
    printf("PPID : %u\n", getppid());
    printf("UID  : %u\n", getuid());
    printf("EUID : %u\n", geteuid());
    printf("GID  : %u\n", getgid());
    printf("EGID : %u\n", getegid());
    printf("PG   : %u\n", getpgrp());
    printf("SID  : %u\n", getsid(getpid()));
}

int main(int argc, char** argv)
{
    int world_size, world_rank, name_len;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    /* Initialize the MPI environment */
    MPI_Init(NULL, NULL);

    MPI_Comm_size(MPI_COMM_WORLD, &world_size);            // Get the number of
processes
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);          // Get the rank of the
process
    MPI_Get_processor_name(processor_name, &name_len);   // Get the name of the
processor

    /* Print off a hello world message */
    sleep(world_rank);
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);

    show_info(processor_name);

    /* Finalize the MPI environment */
    MPI_Finalize();
    return 0;
}
```

Лисмтинг mpi-перемножение матриц

```
void generatingMatrix(vector<vector<int>> &a, int row, int col)
{
```

```
    //cout << "Generating matrix:" << endl;

    for (auto i = 0; i < row; ++i) {
        vector<int> v(col);
        a.push_back(v);
    }

    for (auto i = 0; i < row; ++i) {
        for (auto j = 0; j < col; ++j) {
            a[i][j] = j;
        }
    }
}

void multiplyMatrix(vector<vector<int>> &a, vector<vector<int>> &b, int m, int n,
int q)
{
    vector<vector<int>> c;

    for (auto i = 0; i < m; ++i) {
        vector<int> v(q);
        c.push_back(v);
    }

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    sleep(rank);

    printf("Rank %d:\n", rank);

    for (auto i = 0; i < m; ++i) {
        for (auto j = 0; j < q; ++j) {
            c[i][j] = 0;

            for (auto k = 0; k < n; ++k) {
                c[i][j] = c[i][j] + (a[i][k]*b[k][j]);
            }
            cout << c[i][j] << " ";
        }
        cout << endl;
    }
}

double wtime()
{
    struct timeval t;
    gettimeofday(&t, NULL);
    return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}

void run_matrix()
{
    vector<vector<int>> matrix1;
    vector<vector<int>> matrix2;

    generatingMatrix(matrix1, 10, 10);
    generatingMatrix(matrix2, 10, 10);

    multiplyMatrix(matrix1, matrix2, 10, 10, 10);
}

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

```
    int world_size, world_rank;

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    double t = wtime();
    run_matrix();
    printf("Rank %d elapsed time %f sec\n", world_rank, wtime() - t);

    MPI_Finalize();
    return 0;
}
```

Задания повышенной сложности:

- Разработайте гибридное приложение (MPI+OpenMP), реализующее алгоритм умножения матриц. Программа должна работать аналогично той, что разработана в п. 2 основных заданий и плюс к тому, использовать параллельную версию для каждой ветви. Сравните результаты с полученными при выполнении п. 2 основного задания.

```
#include <mpi.h>
#include <iostream>
#include <vector>
#include <random>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>
#include <omp.h>

using namespace std;

void generatingMatrix(vector<vector<int>> &a, int row, int col)
{
    for (auto i = 0; i < row; ++i) {
        vector<int> v(col);
        a.push_back(v);
    }

    for (auto i = 0; i < row; ++i) {
        for (auto j = 0; j < col; ++j) {
            a[i][j] = j;
        }
    }
}

void multiplyMatrix(vector<vector<int>> &a, vector<vector<int>> &b, int m, int n,
int q)
{
    vector<vector<int>> c;

    for (auto i = 0; i < m; ++i) {
        vector<int> v(q);
        c.push_back(v);
    }
```

```
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {


        for (auto i = 0; i < m; ++i) {
            for (auto j = 0; j < q; ++j) {
                c[i][j] = 0;
                for (auto k = 0; k < n; ++k) {
                    c[i][j] += a[i][k]*b[k][j];
                }
            }
        }
    } else {
        #pragma omp parallel
        {
            printf("omp_get_num_threads() %d\n", omp_get_num_threads());

            int n_threads        = omp_get_num_threads();
            int thread_id        = omp_get_thread_num();
            int items_per_thread = m / n_threads;

            int low_b = thread_id * items_per_thread;
            int upr_b = (thread_id == n_threads - 1) ? (m - 1) : (low_b +
items_per_thread - 1);

            for (int i = low_b; i <= upr_b; i++) {
                for (auto j = 0; j < q; ++j) {
                    c[i][j] = 0;
                    for (auto k = 0; k < n; ++k) {
                        c[i][j] += a[i][k]*b[k][j];
                    }
                }
            }
        }
    }
}

double wtime()
{
    struct timeval t;
    gettimeofday(&t, NULL);
    return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}

void run_matrix()
{
    vector<vector<int>> matrix1;
    vector<vector<int>> matrix2;

    int size = 1024;

    generatingMatrix(matrix1, 1024, 1024);
    generatingMatrix(matrix2, 1024, 1024);

    multiplyMatrix(matrix1, matrix2, 1024, 1024, 1024);
}

int main(int argc, char** argv)
{
    int world_size, world_rank;
    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    double t = wtime();
    run_matrix();
    printf("Rank %d elapsed time %f sec\n", world_rank, wtime() - t);
    MPI_Finalize();
    return 0;
}
```