Национальный исследовательский университет «МЭИ»
Институт автоматики и вычислительной техники

Кафедра вычислительных машин, систем и сетей

Лабораторная работа № 2.

Создание параллельной программы с использованием интерфейса MPI и проведение экспериментов с ней на вычислительной системе.

Выполнили: Балашов С.А.
Поздняков Ю.Б.
Кретов Н.В.
Группа: А-08-19
Бригада: №4
Проверил: Филатов А.В.

Москва, 2023

Цель работы:

Знакомство с MPI – средством параллельного программирования многопроцессорных систем с распределенной памятью в модели передачи сообщений. Постановка экспериментов на системе с несколькими процессорными ядрами и исследование изменения коэффициента ускорения выполнения программы в зависимости от числа задействованных процессорных ядер и объема обрабатываемых данных.

1. Задание

Формула для вычисления:

4	$Y_i = (A_i \cdot B_i \cdot C_i)S1 \cdot S2$

$A_i =$ случайное значение от 0 до 20
$B_i =$ случайное значение от 0 до 20
$C_i =$ случайное значение от 0 до 20

Размерности массивов: N = 8, 64, 256, 4096, 16384, 65536, 262144
S1= случайное значение от 0 до 20
S2= случайное значение от 0 до 20

2. Последовательная программа

```
#include <iostream>
#include "mpi.h"
#include <chrono>

// Зерно для генератора псевдорандомных чисел
#define seed 0

// Объявление массивов для задачи
// Переменные для задачи
int *A, *B, *C, *Y;
int S1, S2;

// Типы данных для счёта времени при однопроцессорном режиме
typedef std::chrono::high_resolution_clock Time;
typedef std::chrono::duration<float> secs;

// Функция заполнения массива.
// Возвращает заполненный массив.
// Принимает размер массива.
int *FillVector (int size)
{
    int *X = (int *)(malloc(size * sizeof(int)));
    for (int i = 0; i < size; ++i)
    {
        X[i] = rand() %20;
    }
    return X;
}

// Процедура вывода полученного массива и суммы всех его элементов
// Принимает на вход массив
void PrintResult (int *X)
{
    double sum = 0;
    int size = sizeof(*X)/sizeof(int);
    std::cout<<size<<std::endl;
    for (int i = 0; i < size; ++i)
    {
        std::cout<<"Yi = "<<X[i]<<std::endl;
        sum += X[i];
    }
    std::cout<<"Sum = "<<sum<<std::endl;
    std::cout<<"----------------------------------------------------------------------"<<std::endl;
}

void Init (int vector_size)
```

```
        {
            srand(seed);
            S1 = rand()%20;
            S2 = rand()%20;
            A = FillVector(vector_size);
            B = FillVector(vector_size);
            C = FillVector(vector_size);
            Y = (int *)(malloc(vector_size * sizeof(int)));
        }

        // Процедура однопроцессорного расчёта.
        // Принимает размер массивов
        void ClassicalCalc (int size)
        {
            SolveY(A, B, C, S1, S2, size);
            PrintResult(Y);
        }

        int main () {
            const int v_sizes[] = {8, 64, 256, 1024, 4096, 16384, 65536, 262144};
            for (int v_size : v_sizes){
                Init(v_size);
                std::chrono::time_point<std::chrono::steady_clock> t0 = Time::now();
                ClassicalCalc(v_size);
                std::chrono::time_point<std::chrono::steady_clock> t1 = Time::now();
                secs dif = t1 - t0;
                if (total_rank == 0)
                {
                    std::cout<<"Vector size = "<<v_size<<"\nClocks passed: "<<dif.count()<<"
        seconds"<<std::endl;
                }
                free(Y);
            }
            return 0;
}
```

3. Параллельная программа

В программе инициализация всех массивов производится в процессе 0

Табл.1

Параллельная программа

```
        #include <iostream>
        #include       "/opt/homebrew/Cellar/open-
mpi/4.1.5/include/mpi.h"
        #include <chrono>

        // Зерно для генератора псевдорандомных
чисел
        #define seed 0
        #define debug 0
        #define printres 0
        #define rounds 1000

        // Объявление массивов для задачи
        // Указатели на текущие элементы массивов
        // Массив индексов для частей массивов (при
разбиении)
        // Переменные для задачи
        int *A, *B, *C;
        int *Acur, *Bcur, *Ccur, *Y, *YRes;
        int S1, S2;
        int* VIndex;


        // Типы данных для счёта времени при
однопроцессорном режиме
        typedef std::chrono::high_resolution_clock
Time;
        typedef std::chrono::duration<float> secs;

        // Процедура расчёта заданного уравнения
        //  Принимает на вход 3 массива, 2
переменные и размер массивов
        void SolveY (int *A, int *B, int *C, int
S1, int S2, int size)
        {
            YRes    =    (int    *)(malloc(size    *
sizeof(int)));
            for (int i = 0; i < size; i++)
            {
```

```
        // Процедура многопроцессорного расчёта
        // Принимает размеры многопроцессорной системы и
размер массивов
        void   ParallelCalc   (int   vector_size,   int
total_size, int total_rank)
        {
            // Условие однопроцессорного расчёта
            if (total_size == 1)
            {
                #ifdef debug
                if (debug)
                {
                    std::cout<<"Classical"<<std::endl;
                }
                #endif
                ClassicalCalc(vector_size);
                return;
            }

            // Текущий размер "сегмента"
            int  curBatchSize  =  VIndex[total_rank+1]  -
VIndex[total_rank];

            // Нулевой ранг
            // Расчёт размера сегментов
            // Отправка в память по адресам массивов и
переменных
            // Приёем из памяти по адресам массивов и
переменных
            // Частичные расчёты для каждого сегмента и
отправка их в память
            // Заполнение первого сегмента результата
            // Загрузка из памяти остальных сегментов
            if (total_rank == 0)
            {
                Acur = A;
                Bcur = B;
                Ccur = C;
```

```
                YRes[i] = (A[i] + B[i] + C[i]) *
S1 * S2;
            }
        }

        // Функция заполнения массива.
        // Возвращает заполненный массив.
        // Принимает размер массива.
        int *FillVector (int size)
        {
            int   *X   =   (int   *)(malloc(size   *
sizeof(int)));
            for (int i = 0; i < size; ++i)
            {
                X[i] = rand() %20;
            }
            return X;
        }

        // Функция расчёта количества разделений
        //  Возвращает массив с индексами начала
сегмента
        //  Принимает количество процессоров и
размер
        int* indexes (int cpus, int size)
        {
            int *ind = (int *)(malloc((cpus + 1) *
sizeof(int)));
            ind[cpus] = size;
            for (int i = 0; i < cpus; i++)
            {
                ind[i] = i*size/cpus;
            }
            return ind;
        }

        // Процедура вывода полученного массива и
суммы всех его элементов
        // Принимает на вход массив
        void PrintResult (int *X)
        {
            double sum = 0;
            int size = sizeof(*X)/sizeof(int);
            std::cout<<size<<std::endl;
            for (int i = 0; i < size; ++i)
            {
                std::cout<<"Yi                    =
"<<X[i]<<std::endl;
                sum += X[i];
            }
            std::cout<<"Sum = "<<sum<<std::endl;
            std::cout<<"------------------------
----------------------------------------
"<<std::endl;
        }

        //   Процедура   инициализации   массивов,
переменных и генератора псевдорандомных чисел
        void Init (int vector_size, int total_size,
int total_rank)
        {
            VIndex      =      indexes(total_size,
vector_size);
            srand(seed);
            S1 = rand()%20;
            S2 = rand()%20;
            A = FillVector(vector_size);
            B = FillVector(vector_size);
            C = FillVector(vector_size);
            Y   =   (int   *)(malloc(vector_size   *
sizeof(int)));
            if (total_rank != 0)
            {
                int         curBatchSize         =
VIndex[total_rank+1] - VIndex[total_rank];
                Acur = (int *)(malloc(curBatchSize
* sizeof(int)));
                Bcur = (int *)(malloc(curBatchSize
* sizeof(int)));
                Ccur = (int *)(malloc(curBatchSize
* sizeof(int)));
            }
        }
                #ifdef debug
                if (debug)
                {
                    std::cout<<"Proccess    0    count:
"<<curBatchSize<<std::endl;
                }
                #endif

                for(int i = 1; i< total_size; i++)
                {
                    int  batchSize  =  VIndex[i  +  1]  -
VIndex[i];
                    #ifdef debug
                    if (debug)
                    {
                        std::cout<<"Send   to   proccess
"<<i<<"              from       "<<VIndex[i]<<"       to
"<<VIndex[i]+batchSize<<"  (count:  "<<batchSize<<")  of
"<<vector_size<<std::endl;
                    }
                    #endif
                    MPI_Send(A  +  VIndex[i],  batchSize,
MPI_INT, i, 1, MPI_COMM_WORLD);
                    MPI_Send(B  +  VIndex[i],  batchSize,
MPI_INT, i, 2, MPI_COMM_WORLD);
                    MPI_Send(C  +  VIndex[i],  batchSize,
MPI_INT, i, 3, MPI_COMM_WORLD);
                    MPI_Send(&S1,   1,   MPI_INT,   i,   4,
MPI_COMM_WORLD);
                    MPI_Send(&S2,   1,   MPI_INT,   i,   5,
MPI_COMM_WORLD);
                }
            } else
            {
                MPI_Recv(Acur, curBatchSize, MPI_INT, 0,
1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                MPI_Recv(Bcur, curBatchSize, MPI_INT, 0,
2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                MPI_Recv(Ccur, curBatchSize, MPI_INT, 0,
3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                MPI_Recv(&S1,   1,   MPI_INT,   0,   4,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                MPI_Recv(&S2,   1,   MPI_INT,   0,   5,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            }

            #ifdef debug
            if(debug)
            {
                std::cout<<total_rank<<"  received  data
size: "<<curBatchSize<<std::endl;
            }
            #endif

            SolveY(Acur,    Bcur,    Ccur,    S1,    S2,
curBatchSize);

            if (total_rank == 0){

                for (int i = 0; i < curBatchSize; ++i)
                {
                    Y[i] = YRes[i];
                }

                for(int i = 1; i < total_size; i++){
                    int   batchSize   =   VIndex[i+1]   -
VIndex[i];
                    #ifdef debug
                    if (debug)
                    {
                        std::cout<<"Receive from process
"<<i<<"         from         "<<VIndex[i]<<"         to
"<<VIndex[i]+batchSize<<" of "<<vector_size<<std::endl;
                    }
                    #endif
                    MPI_Recv(Y                          +
VIndex[i],batchSize,MPI_INT,i,6,MPI_COMM_WORLD,MPI_STAT
US_IGNORE);
                }
                #ifdef printres
                if (printres)
                {
                    PrintResult(Y);
                }
```

```
        // Процедура очистки памяти после использования
        void FreeVectors()
        {
            free(Acur);
            free(Bcur);
            free(Ccur);
            free(Y);
            free(YRes);
        }

        // Процедура однопроцессорного расчёта.
        // Принимает размер массивов
        void ClassicalCalc (int size)
        {
            Acur = A;
            Bcur = B;
            Ccur = C;
            SolveY(A, B, C, S1, S2, size);
            Y = YRes;
            YRes = nullptr;
            #ifdef printres
            if (printres)
            {
                PrintResult(Y);
            }
            #endif
        }
                #endif
            }
            else{

MPI_Send(YRes,curBatchSize,MPI_INT,0,6,MPI_COMM_WORLD);
            }
        }

        // Основная процедура
        // Запуск процедуры инициализации
        // Получение информации о системе
        // Расчёт времени выполнения расчётов
        //   Запуск процедуры очистки при каждом прохождении
        int main () {
            MPI_Init(nullptr, nullptr);

            int total_size, total_rank;
            MPI_Comm_size(MPI_COMM_WORLD, &total_size);
            MPI_Comm_rank(MPI_COMM_WORLD, &total_rank);
            std::cout << total_rank << " | " << total_size << std::endl;

            const int v_sizes[] = {8, 64, 256, 1024, 4096, 16384, 65536, 262144};
            for (int v_size : v_sizes){
                Init(v_size,total_size, total_rank);

std::chrono::time_point<std::chrono::steady_clock> t0 = Time::now();
                double tm0 = MPI_Wtime();
                for(int j = 0; j < rounds; j++)
                    ParallelCalc(v_size,    total_size, total_rank);

std::chrono::time_point<std::chrono::steady_clock> t1 = Time::now();
                double tm1 = MPI_Wtime();
                secs dif = t1 - t0;
                double difm = tm1 - tm0;
                if (total_rank == 0)
                {
                    std::cout<<"Vector      size      = "<<v_size<<"\nClocks     passed:     "<<dif.count()<<" seconds"<<std::endl;
                    std::cout<<"Clocks in MPI passed: "<<difm<<" seconds"<<std::endl;
                }
                FreeVectors();
            }
            MPI_Finalize();

            return 0;
}
```

Табл.2.

Результаты работы программы.

Размер массивов/Число процессоров	1	2	3	4	5	6	7	8
8	0.000163083	0.000863917	0.00178096	0.00264488	0.00586808	0.00753167	0.00902429	0.0102868
64	0.000162542	0.000616875	0.000827833	0.000936708	0.00292563	0.00689908	0.00374525	0.00854008
256	0.000577458	0.00159163	0.00245904	0.00118288	0.00453587	0.00514938	0.00376871	0.00855404
1024	0.0021935	0.00317733	0.00390058	0.00448825	0.0155369	0.01426	0.0166497	0.0232134
4096	0.00867975	0.0103071	0.0115886	0.0133701	0.0284227	0.0191186	0.0300892	0.0323681
16384	0.0361671	0.0354096	0.0303568	0.0312963	0.0511433	0.0627533	0.0702528	0.0869327
65536	0.142962	0.133055	0.108975	0.0993926	0.162539	0.182193	0.236768	0.371563
262144	0.614706	0.562529	0.470668	0.488576	0.729442	0.949862	0.918918	1.04473
Коэффициенты ускорения								
8	1	0.18877	0.09157	0.06166	0.02779	0.02165	0.01807	0.01585
64	1	0.96322	0.19635	0.17353	0.05556	0.02356	0.04339	0.01903
256	1	0.36281	0.23483	0.48818	0.12731	0.11214	0.15322	0.06751
1024	1	0.69036	0.56235	0.48872	0.14118	0.15382	0.13174	0.09449
4096	1	0.84211	0.74899	0.64919	0.39538	0.45399	0.28847	0.26816
16384	1	1.02139	1.19140	1.15563	0.70717	0.57634	0.51481	0.41604
65536	1	1.07446	1.31188	1.43836	0.87956	0.78467	0.60381	0.38476
262144	1	1.09275	1.30603	1.25816	0.84271	0.64715	0.66895	0.58839

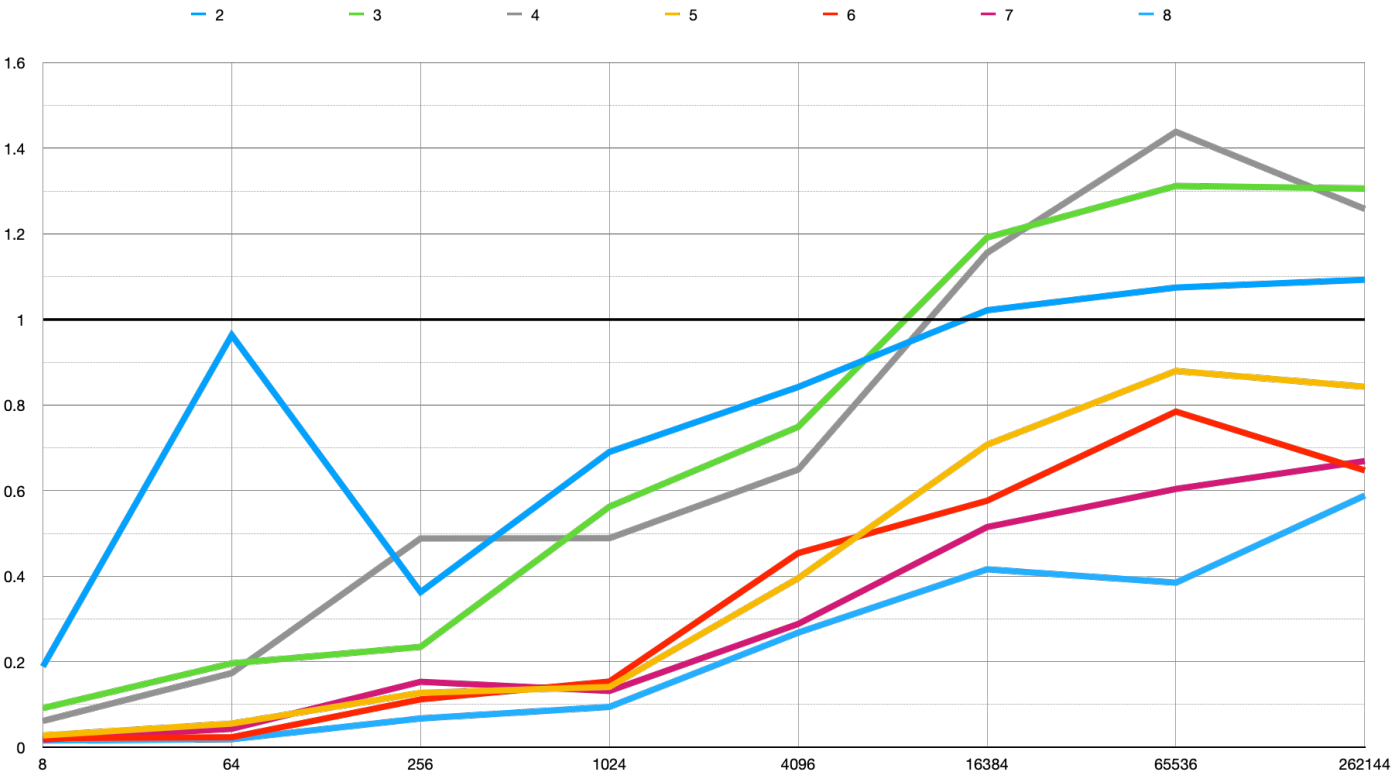

Рис.1. График зависимости значений коэф. ускорения от числа процессов

Вывод: Проанализировав таблицу и график можно сделать вывод, что коэффициент ускорения выше 1 достигается при 2-х, 3-х и 4-х процессорах при размерах массивов более 16384 элементов.

Дальнейшее увеличение числа процессоров приводит к уменьшению коэффициента ускорения и увеличению времени выполнения задач.