МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ АВТОМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

**Расчётно-графическая работа**

**по дисциплине «Параллельное программирование»**

Группа: АВТ-610

Студент: Дунаев Н.Ю.

Преподаватель: Малявко А.А.

Варианты: 7,36,44,55

НОВОСИБИРСК

2018

**Лабораторная работа №1**

**Тема**

Параллельное программирование для систем с общей памятью с использованием основ технологии OpenMP.

**Вариант задачи**

55. Найти минимальное число, большее заданного, которое может быть представлено как сумма степеней 2, 3, 4 и 5 простых чисел (наименьшее такое число есть 60 = 22+23+24+25)

**Алгоритм решения задачи**

У пользователя запрашивается число, больше которого будет производиться поиск. Проверяется корректность введенных данных. Используя «Решето Эратосфена» генерируется массив простых чисел. При помощи четырех вложенных циклов for от числа отнимаются простые числа в степенях 2,3,4,5. Полученный результат сравнивается с нулем и в случае равенства выводится ответ (само число и его разбиение на слагаемые). Если ответ не получен, к числу прибавляется единица и поиск продолжается.

Параллельное выполнение реализовано при помощи *omp* *parallel for* верхнего цикла итераций.

**Результаты**

|  |  |  |
| --- | --- | --- |
| Число N | Последовательное выполнение | Параллельное выполнение |
| 1000 | 0,009 | 0,014 |
| 10000 | 0,323 | 0,060 |
| 100000 | 46.45 | 11.68 |

Подтверждена эффективность распараллеливания циклов технологией OpenMP. При N = 100000 наблюдается значительный выигрыш во времени в 4 раза.

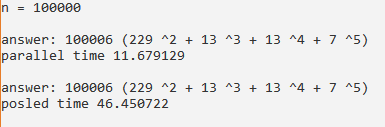


Рисунок 1. Результаты выполнения лабораторной работы №1

**Листинг программы**

//55.Найти минимальное число, большее заданного, которое может быть представлено как

//сумма степеней 2, 3, 4 и 5 простых чисел (наименьшее такое число есть 60 = 2^2+2^3+2^4+2^5)

#include <stdio.h>

#include <malloc.h>

#include <omp.h>

#include <cmath>

int main()

{

int n = 0, size = 0, x = 0, tmp = 0;

printf("n = ");

while (1)

{

char c = getchar();

if (((c < '0') || (c > '9')) && (c != '\n'))

{

printf("Input error\n");

return 0;

}

if (c == '\n')

break;

n = n \* 10 + (c - '0');

}

int \*a = (int\*)malloc(n \* sizeof(int)+10);

int \*simpleBuf = (int\*)malloc(n \* sizeof(int)+10);

for (int i = 0; i < n + 1; i++)

a[i] = i;

for (int p = 2; p < n + 1; p++)

{

if (a[p] != 0)

{

for (int j = p \* 2; j < n + 1; j += p)

a[j] = 0;

}

}

for (int i = 2; i < n && a[i] < sqrt(n); i++)

{

if (a[i] != 0)

{

simpleBuf[size] = a[i];

size++;

}

}

short flag = 1;

tmp = n + 1;

double t1 = omp\_get\_wtime();

while (flag)

{

#pragma omp parallel for private(x)

for (int two = 0; two < size; two++)

for (int three = 0; three < size && flag != 0; three++)

for (int four = 0; four < size; four++)

for (int five = 0; five < size; five++)

{

x = tmp - pow(simpleBuf[two], 2) - pow(simpleBuf[three], 3) - pow(simpleBuf[four], 4) - pow(simpleBuf[five], 5);

if (x == 0)

{

flag = 0;

printf("\nanswer: %d (%d ^2 + %d ^3 + %d ^4 + %d ^5)\n",tmp, simpleBuf[two], simpleBuf[three], simpleBuf[four], simpleBuf[five]);

}

}

tmp++;

}

printf("parallel time %f \n\n", omp\_get\_wtime() - t1);

tmp = n + 1;

flag = 1;

t1 = omp\_get\_wtime();

while (flag)

{

for (int two = 0; two < size; two++)

for (int three = 0; three < size && flag != 0; three++)

for (int four = 0; four < size; four++)

for (int five = 0; five < size; five++)

{

x = tmp - pow(simpleBuf[two], 2) - pow(simpleBuf[three], 3) - pow(simpleBuf[four], 4) - pow(simpleBuf[five], 5);

if (x == 0)

{

printf("answer: %d (%d ^2 + %d ^3 + %d ^4 + %d ^5)\n", tmp, simpleBuf[two], simpleBuf[three], simpleBuf[four], simpleBuf[five]);

flag = 0;

}

}

tmp++;

}

printf("posled time %f \n", omp\_get\_wtime() - t1);

return 1;

}

**Лабораторная работа №2**

**Тема**

Параллельное программирование для систем с общей памятью с использованием расширенных средств технологии OpenMP: редукции, замков, критических секций, атомарных операций.

**Вариант задачи**

1. Пирамидальная сортировка

**Алгоритм решения задачи**

У пользователя запрашивается размер массива, который требуется отсортировать. Массив заполняется случайными числами при помощи *rand()* и сортируется по алгоритму пирамидальной сортировки. Алгоритм параллельной программы несколько сложнее: массив для сортировки разбивается на количество частей, равное количеству потоков, выполняющих сортировку. Каждый поток сортирует свою часть массива. После того, как все потоки исполнили свой код, мы имеем частично отсортированный массив. Используя сортировку слиянием, склеиваем отсортированные части массива.

**Результаты**

|  |  |  |
| --- | --- | --- |
| Размер массива | Последовательное выполнение | Параллельное выполнение |
| 10000 | 0.0014 | 0.0041 |
| 100000 | 0.0184 | 0.0089 |
| 10000000 | 3.343 | 0.9967 |

Подтверждена эффективность распараллеливания циклов технологией OpenMP. При размере массива в 10000000 элементов имеем выигрыш во времени в 3.5 раза.



Рисунок 2. Результаты выполнения лабораторной работы №2

**Листинг программы**

// 7. Пирамидальная сортировка

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <omp.h>

#include <math.h>

int min(int x, int y)

{

if (x <= y) return x;

else return y;

}

void push(int \*buf, long pos, long size) {

int tmp;

int child;

tmp = buf[pos];

while (pos <= size / 2)

{

child = pos \* 2;

if (child < size && buf[child] < buf[child + 1])

child++;

if (tmp >= buf[child])

break;

buf[pos] = buf[child];

pos = child;

}

buf[pos] = tmp;

}

void sort(int \*a, int size)

{

int tmp;

for (int i = size / 2; i >= 0; i--)

push(a, i, size - 1);

for (int i = size - 1; i > 0; i--)

{

tmp = a[i];

a[i] = a[0];

a[0] = tmp;

push(a, 0, i - 1);

}

}

void concat(int \*sortedBuf, int \*newBuf, int firstOfFirst, int lastOfFirst, int lastOfLast)

{

int fof = firstOfFirst;

int lof = lastOfFirst + 1;

int nbIter = firstOfFirst;

while ((fof <= lastOfFirst) && (lof <= lastOfLast))

{

if (sortedBuf[fof]<sortedBuf[lof])

{

newBuf[nbIter] = sortedBuf[fof];

fof++;

}

else

{

newBuf[nbIter] = sortedBuf[lof];

lof++;

}

nbIter++;

}

for (; fof <= lastOfFirst; nbIter++, fof++)

newBuf[nbIter] = sortedBuf[fof];

for (; lof <= lastOfLast; nbIter++, lof++)

newBuf[nbIter] = sortedBuf[lof];

}

void parallelSort(int\* sortedBuf, int\* mergeBuf, int bufSize, int threadsNum)

{

int \*sortedTmpBuf = sortedBuf;

int \*mergeTmpBuf = mergeBuf;

int \*tmp;

int i, eoSecPart, eoFirstPart, k, count = 0;

int partSize = (int)ceil((float)bufSize / (float)threadsNum);

#pragma omp parallel for private(i) default(none) shared(threadsNum, sortedTmpBuf, partSize, bufSize)

for (i = 0; i < threadsNum; i++)

sort(sortedTmpBuf + i\*partSize, min(partSize, bufSize - i\*partSize));

while (partSize < (2 \* bufSize))

{

#pragma omp parallel for private(i, eoSecPart, eoFirstPart) default(none) shared(threadsNum, sortedTmpBuf, mergeTmpBuf, partSize, bufSize)

for (i = 0; i < bufSize; i += partSize)

{

eoSecPart = min(i + partSize - 1, bufSize - 1);

eoFirstPart = ((i + i + partSize - 1) / 2);

concat(sortedTmpBuf, mergeTmpBuf, i, eoFirstPart, eoSecPart);

}

#pragma omp critical

tmp = sortedTmpBuf;

sortedTmpBuf = mergeTmpBuf;

mergeTmpBuf = tmp;

partSize \*= 2;

}

#pragma omp parallel for private(k) default(none) shared(sortedBuf, mergeBuf, bufSize)

for (k = 0; k < bufSize; k++)

sortedBuf[k] = mergeBuf[k];

}

int main()

{

printf("size: ");

int size = 0;

while (1)

{

char c = getchar();

if (((c < '0') || (c > '9')) && (c != '\n'))

{

printf("Input error\n");

return 0;

}

if (c == '\n')

break;

size = size \* 10 + (c - '0');

}

int \*bufForSort = (int\*)malloc(size \* sizeof(int)+10);

int \*bufForParallelSort = (int\*)malloc(size \* sizeof(int)+10);

int \*helpBuf = (int\*)malloc(size \* sizeof(int)+10);

for (int i = 0; i < size; i++)

{

int r = rand() % 10000;

bufForSort[i] = r;

bufForParallelSort[i] = r;

}

double start\_time = omp\_get\_wtime();

sort(bufForSort, size);

printf("posled time: %f \n", omp\_get\_wtime() - start\_time);

start\_time = omp\_get\_wtime();

parallelSort(bufForParallelSort, helpBuf, size, 16);

printf("parallel time: %f \n", omp\_get\_wtime() - start\_time);

short flag = 1;

for (int i = 0; i < size; i++)

if (bufForParallelSort[i] != bufForSort[i])

flag = 0;

if (flag)

printf("true\n");

else

printf("false\n");

free(bufForParallelSort);

free(bufForSort);

free(helpBuf);

return 0;

}

**Лабораторная работа №3**

**Тема**

Параллельное программирование для графического процессора в среде NVidia CUDA.

**Вариант задачи**

55. Найти минимальное число, большее заданного, которое может быть представлено как сумма степеней 2, 3, 4 и 5 простых чисел (наименьшее такое число есть 60 = 22+23+24+25).

**Алгоритм решения задачи**

У пользователя запрашивается число, больше которого будет производиться поиск. Используя «Решето Эратосфена» генерируется массив простых чисел. При помощи четырех вложенных циклов for складываются простые числа в степенях 2,3,4,5 и, если сумма таких чисел удовлетворяет условию задачи, выводится на экран. Параллельное выполнение реализовано методом развертки двух вложенных циклов в пространстве XY. В итоге четыре вложенных цикла были заменены на 2. Каждый поток записывает свой результат в массив. Индекс потока определяется по формуле: .

**Результаты**

|  |  |  |
| --- | --- | --- |
| Размер массива | Последовательное выполнение | Параллельное выполнение |
| 10000 | 172 | 15 |
| 100000 | 8027 | 235 |
| 10000000 | 360872 | 4821 |

Подтверждена эффективность распараллеливания циклов технологией NVidia CUDA. При размере массива в 10000000 элементов имеем выигрыш во времени в 75 раз.

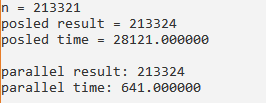


Рисунок 3. Результаты выполнения лабораторной работы №3

**Листинг программы**

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <time.h>

#include <math.h>

#include <malloc.h>

#include <cmath>

\_\_global\_\_ void search(int\* SimpleBuf, int\* resultBuf, int simpleSize, int N) {

int two, three = 0; int four = 0; int five = 0, x = 0, idx, idy;

int iter = simpleSize;

idx = blockIdx.x \* blockDim.x + threadIdx.x;

idy = threadIdx.y + blockIdx.y \* blockDim.y;

int indexInArray = simpleSize\*idy + idx;

for (two = 0; two < simpleSize; two++)

for (three = 0; three < simpleSize; three++)

{

x = pow((double)SimpleBuf[two], 2) + pow((double)SimpleBuf[three], 3) + pow((double)SimpleBuf[idx], 4) + pow((double)SimpleBuf[idy], 5);

if (x > N && x < resultBuf[indexInArray])

resultBuf[indexInArray] = x;

}

}

int main() {

int \*hostResultBuf, \*deviceResultBuf, \*devicehostSimpleBuf, \*hostSimpleBuf;

int n = 0;

int x = 0;

int answer = 2000000000;

int min = answer;

int size = 0;

printf("n = ");

while (1)

{

char c = getchar();

if (((c < '0') || (c > '9')) && (c != '\n'))

{

printf("Input error\n");

return 0;

}

if (c == '\n')

break;

n = n \* 10 + (c - '0');

}

int \*a = (int\*)malloc(n \* sizeof(int)+10);

hostSimpleBuf = (int\*)malloc(n \* sizeof(int)+10);

for (int i = 0; i < n + 1; i++)

a[i] = i;

for (int p = 2; p < n + 1; p++)

{

if (a[p] != 0)

{

for (int j = p \* 2; j < n + 1; j += p)

a[j] = 0;

}

}

for (int i = 2; i < n && a[i] < sqrt(n); i++)

{

if (a[i] != 0)

{

hostSimpleBuf[size] = a[i];

size++;

}

}

free(a);

clock\_t begin = clock();

for (int two = 0; two < size; two++)

for (int three = 0; three < size; three++)

for (int four = 0; four < size; four++)

for (int five = 0; five < size; five++)

{

x = pow(hostSimpleBuf[two], 2) + pow(hostSimpleBuf[three], 3) + pow(hostSimpleBuf[four], 4) + pow(hostSimpleBuf[five], 5);

if (x > n && x < answer)

answer = x;

}

clock\_t end = clock();

printf("posled result = %d\n", answer);

double time\_spent = (double)(end - begin);

printf("posled time = %f\n\n", time\_spent);

hostResultBuf = (int\*)malloc(sizeof(int)\*(size \* 2) \* (size \* 2));

for (int i = 0; i < size \* size; i++)

hostResultBuf[i] = 200000000;

cudaDeviceProp devProp;

cudaGetDeviceProperties(&devProp, 0);

int blocks;

if (size % devProp.maxThreadsPerBlock != 0)

blocks = (size / devProp.maxThreadsPerBlock + 1);

else

blocks = (size / devProp.maxThreadsPerBlock);

cudaMalloc((void\*\*)&devicehostSimpleBuf, sizeof(int)\*(n + 1));

cudaMalloc((void\*\*)&deviceResultBuf, (sizeof(int)\*(size \* 2) \* (size \* 2)));

cudaMemcpy(devicehostSimpleBuf, hostSimpleBuf, sizeof(int)\*(n + 1), cudaMemcpyHostToDevice);

cudaMemcpy(deviceResultBuf, hostResultBuf, (sizeof(int)\*(size \* 2) \* (size \* 2)), cudaMemcpyHostToDevice);

//printf("size = %d\n", size);

//printf("blocks = %d\n", blocks);

clock\_t begin1 = clock();

dim3 blockDim(16, 16, 1);

dim3 gridDim(size / blockDim.x + 1, size / blockDim.y + 1, 1);

search << < gridDim, blockDim >> >(devicehostSimpleBuf, deviceResultBuf, size, n);

cudaMemcpy(hostResultBuf, deviceResultBuf, (sizeof(int)\*(size \* 2) \* (size \* 2)), cudaMemcpyDeviceToHost);

clock\_t end1 = clock();

for (int i = 0; i < size\*size; i++) {

if (hostResultBuf[i] < min)

min = hostResultBuf[i];

//printf("%d\n", hostResultBuf[i]);

}

printf("parallel result: %d\n", min);

double time\_spent1 = (double)(end1 - begin1);

printf("parallel time: %f\n\n", time\_spent1);}

**Лабораторная работа №4**

**Тема**

Параллельное программирование для гетерогенных вычислительных систем с использованием технологии OpenCL.

**Вариант задачи**

55. Найти минимальное число, большее заданного, которое может быть представлено как сумма степеней 2, 3, 4 и 5 простых чисел (наименьшее такое число есть 60 = 22+23+24+25).

**Алгоритм решения задачи**

У пользователя запрашивается число, больше которого будет производиться поиск. Используя «Решето Эратосфена» генерируется массив простых чисел. При помощи четырех вложенных циклов for складываются простые числа в степенях 2,3,4,5 и, если сумма таких чисел удовлетворяет условию задачи, выводится на экран. Параллельное выполнение реализовано методом развертки двух вложенных циклов в пространстве XY. В итоге четыре вложенных цикла были заменены на 2. Каждый поток записывает свой результат в массив. Индекс потока определяется по формуле: .

**Результаты**

|  |  |  |
| --- | --- | --- |
| Размер массива | Последовательное выполнение | Параллельное выполнение |
| 10000 | 0,110 | 0,016 |
| 100000 | 4,834 | 0,547 |
| 500000 | 68 | 5,06 |

Подтверждена эффективность распараллеливания циклов технологией OpenCL. При размере массива в 500000 элементов имеем выигрыш во времени в 13 раз.

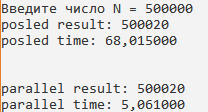


Рисунок 4. Результаты выполнения лабораторной работы №4

**Листинг программы**

#include <iostream>

#include <vector>

#include <string>

#include <stdexcept>

#define \_\_CL\_ENABLE\_EXCEPTIONS

#include <CL/cl.hpp>

#include <ctime>

#include <math.h>

void findSimple(int\* &a, int\* &A, int &n, int &k) {

for (int i = 0; i < n; i++) {

a[i] = i;

}

a[1] = 0;

for (int s = 2; s < n; s++) {

if (a[s] != 0) {

for (int j = s \* 2; j < n; j += s) {

a[j] = 0;

}

}

}

for (int i = 0; i < n; i++) {

if (a[i] != 0 && a[i] < sqrt(n)) {

A[k] = a[i];

k++;

}

}

}

void posled(int\* A, int n, int k, int &X, int x) {

for (int i = 0; i < k; i++)

for (int j = 0; j < k; j++)

for (int m = 0; m < k; m++)

for (int l = 0; l < k; l++)

{

x = pow(A[i], 2) + pow(A[j], 3) + pow(A[m], 4) + pow(A[l], 5);

if (x > n && x < X)

X = x;

}

}

int main() {

setlocale(LC\_ALL, "Russian");

int size = 0;

int number = 0;

int x = 0;

int X = 1000000000;

printf("N = ");

scanf("%d", &number);

int \*simpleTmp = (int\*)malloc(sizeof(int)\*(number + 1));

int \*simpleBuf = (int\*)malloc(sizeof(int)\*(number + 1));

findSimple(simpleTmp, simpleBuf, number, size); // Поиск простых

free(simpleTmp);

int \*resultMas = (int\*)malloc(sizeof(int)\*(size + 1));

for (int i = 0; i < size; i++)

resultMas[i] = 1000000000;

////////////////////////Последовательный алгоритм

clock\_t begin = clock();

posled(simpleBuf, number, size, X, x);

clock\_t end = clock();

printf("posled result: %d\n", X);

double time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;

printf("posled time: %f\n\n", time\_spent);

////////////////////////

std::vector<cl::Platform> platform;

cl::Platform::get(&platform);

if (platform.empty())

throw std::runtime\_error("No OpenCL platforms");

// Инициализация контекста

cl::Context context;

std::vector<cl::Device> device;

for (auto p = platform.begin(); device.empty() && p != platform.end(); p++) {

std::vector<cl::Device> dev;

p->getDevices(CL\_DEVICE\_TYPE\_GPU, &dev);

for (auto d = dev.begin(); device.empty() && d != dev.end(); d++) {

if (!d->getInfo<CL\_DEVICE\_AVAILABLE>())

continue;

device.push\_back(\*d);

try {

context = cl::Context(device);

}

catch (...) {

device.clear();

}

}

}

if (device.empty())

throw std::runtime\_error("No GPUs");

// Создание очереди команд

cl::CommandQueue queue(context, device[0]);

cl::Buffer SimpleBuf(context, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR,

(number + 1) \* sizeof(int), simpleBuf);

cl::Buffer ResultBuf(context, CL\_MEM\_READ\_WRITE | CL\_MEM\_COPY\_HOST\_PTR,

(size + 1) \* sizeof(int), resultMas);

// Код ядра

std::string source = R"(

kernel void add(int k, global int \*simpleBuf, global int \*resultMas, int n) {

int x = 0;

int i = get\_global\_id(0);

int j = get\_global\_id(1);

int indexInArray = j \* k + i;

for (int m = 0; m < k; m++)

for (int l = 0; l < k; l++)

{

x = pow((double)simpleBuf[i], 2) + pow((double)simpleBuf[j], 3) + pow((double)simpleBuf[m], 4) + pow((double)simpleBuf[l], 5);

if (x > n && x < resultMas[indexInArray])

resultMas[indexInArray] = x;

}

})";

cl::Program program(context, source);

try {

program.build(device);

}

catch (const cl::Error&) {

std::cerr

<< "OpenCL compilation error" << std::endl

<< program.getBuildInfo<CL\_PROGRAM\_BUILD\_LOG>(device[0])

<< std::endl;

throw std::runtime\_error("OpenCL build error");

}

cl::Kernel add(program, "add");

add.setArg(0, size);

add.setArg(1, SimpleBuf);

add.setArg(2, ResultBuf);

add.setArg(3, number);

begin = clock();

queue.enqueueNDRangeKernel(add, cl::NullRange, cl::NDRange(size, size), cl::NullRange);

queue.enqueueReadBuffer(ResultBuf, CL\_TRUE, 0, (size + 1) \* sizeof(int), resultMas);

int min = 10000000;

for (int i = 0; i < size; i++)

if (resultMas[i] < min)

min = resultMas[i];

std::cout << std::endl << "parallel result: " << min << std::endl;

end = clock();

time\_spent = (double)(end - begin) / CLOCKS\_PER\_SEC;

printf("parallel time: %f\n\n", time\_spent);

}

**Лабораторная работа №5**

**Тема**

Параллельное программирование с использованием основ технологии MPI.

**Вариант задачи**

44. Найти максимальное число, меньшее заданного, которое равно сумме его десятичных цифр, возведенных в степень, большую 1.

**Алгоритм решения задачи**

Разработана программа, решающая поставленную задачу. У пользователя запрашивается число N; Вызывается функция *isSum*, возвращающая 1, если число является ответом, иначе 0. Функция *isSum* разбивает число на цифры и возводит их в степени, меньшие максимально допустимых (т.е. цифра возводится в степень, пока не станет больше числа).

Параллельное выполнение реализовано при помощи обмена данными между процессами и выполнения каждым процессом инструкций на индивидуальных данных. Также в целях ускорения выполнения задачи процессы обменивались информацией о выполнении каждые 100 итераций.

**Результаты**

|  |  |  |
| --- | --- | --- |
| Число N | Последовательное выполнение | Параллельное выполнение |
| 10000 | 0.003 | 0.003 |
| 50000 | 0.33 | 0.11 |
| 123123 | 13.39 | 4.236 |

Подтверждена эффективность распараллеливания циклов технологией MPI. Возможен выигрыш примерно в 3 раза при заданном числе ~100000.

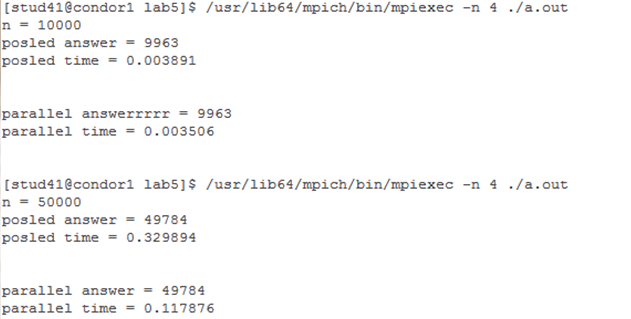
****

Рисунок 5. Результаты выполнения лабораторной работы №5

**Листинг программы**

// 44.Найти максимальное число, меньшее заданного, которое равно сумме его десятичных цифр, возведенных в степень, большую 1

#include <stdio.h>

#include <time.h>

#include <mpi.h>

int isSum(int num)

{

if(num <= 0)

return 0;

int size = 0, tmpNum = num, deg = 1, digits[100], maxDegree[100], degree[100];

while (tmpNum > 0)

{

digits[size++] = tmpNum % 10;

tmpNum /= 10;

}

for (int i = 0; i < size; ++i)

{

degree[i] = 2;

maxDegree[i] = 2;

}

if (size == 1)

return 0;

for (int i = 0; i < size; i++)

{

deg = 1;

int res = digits[i];

if (res == 0 || res == 1)

{

deg = 2;

break;

}

while (res < num)

{

res \*= digits[i];

deg += 1;

}

maxDegree[i] = deg;

}

while (1)

{

degree[0] += 1;

for (int i = 0; i < size - 1; i++)

{

if (degree[i] > maxDegree[i])

{

degree[i] = 2;

degree[i + 1]++;

}

}

if (degree[size - 1] > maxDegree[size - 1])

return 0;

long sum = 0;

for (int i = 0; i < size; i++)

{

sum += pow(digits[i], degree[i]);

}

if (sum == num)

{

//for(int i = 0; i < size; i++)

// printf("%d ",degree[i]);

//printf('\n');

return 1;

}

}

return 0;

}

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

{

int N = 0, max = 0, myid = 0, numprocs = 0, result = 0;

double startwtime = 0.0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &numprocs);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myid);

if (myid == 0)

{

printf("n = ");

while (1)

{

char c = getchar();

if (((c < '0') || (c > '9')) && (c != '\n'))

{

printf("Input error\n");

return 0;

}

if (c == '\n')

break;

N = N \* 10 + (c - '0');

}

startwtime = MPI\_Wtime();

for (int i = N; i > 0; i--)

if (isSum(i))

{

max = i;

break;

}

printf("posled answer = %d\n", max);

printf("posled time = %f\n\n\n", (MPI\_Wtime() - startwtime));

max = 0;

}

MPI\_Bcast(&N, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

startwtime = MPI\_Wtime();

int step = 100;

max = 0;

for (int i = N; i > 0; i -= numprocs)

{

if (isSum(i + myid))

{

max = i + myid;

break;

}

if (i % step == 0)

{

MPI\_Reduce(&max, &result, 1, MPI\_INT, MPI\_MAX, 0, MPI\_COMM\_WORLD);

if (result != 0)

{

if (myid == 0)

{

printf("parallel answer = %d\n", result);

printf("parallel time = %f\n\n\n", (MPI\_Wtime() - startwtime));

}

return;

}

}

}

MPI\_Reduce(&max, &result, 1, MPI\_INT, MPI\_MAX, 0, MPI\_COMM\_WORLD);

if (myid == 0)

{

printf("parallel answer = %d\n", result);

printf("parallel time = %f\n\n\n", (MPI\_Wtime() - startwtime));

}

MPI\_Finalize();

}

**Лабораторная работа №6**

**Тема**

Параллельное программирование с использованием расширенных средств технологии MPI: коллективных операций, производных типов данных.

**Вариант задачи**

36. Число называется укорачиваемым простым слева, если последовательное удаление его десятичных цифр, начиная со старшей, дает простые числа (пример: 3797, 797, 97, 7). Это же число является укорачиваемым простым справа (3797, 379, 37, 3). Найти максимальные укорачиваемые простые слева, справа, с обеих сторон числа, меньшие заданного N.

**Алгоритм решения задачи**

Разработана программа, решающая поставленную задачу. У пользователя запрашивается число N, после чего происходит проверка введенных данных. При помощи «решета Эратосфена» генерируется массив простых чисел до N. После чего для каждого числа вызывается функция *find*. Данная функция отделяет от числа и его палиндрома младшие разряды, пока числа продолжают быть простыми или не равны 0. После чего происходит проверка: если число равняется нулю, т.е. после всех отделений младших разрядов оно осталось простым, помещаем его в соответствующую ячейку в массиве из трех элементов (0 – число, укорачиваемое слева, 1 – справа, 2 – с обеих сторон) при условии, что эта ячейка уже не хранит ответ. Параллельное выполнение реализовано при помощи обмена данными между процессами и выполнения каждым процессом инструкций на индивидуальных данных. Обмен данными между процессами осуществляется функциями *MPI\_Reduce* и *MPI\_Bcast.*

**Результаты**

|  |  |  |
| --- | --- | --- |
| Число N | Последовательное выполнение | Параллельное выполнение |
| 10000 | 0.011 | 0.009 |
| 100000 | 0.43 | 0.21 |
| 1234567 | 5.87 | 1.50 |

Подтверждена эффективность распараллеливания циклов технологией MPI. Возможен выигрыш примерно в 4 раза при заданном числе ~1000000.

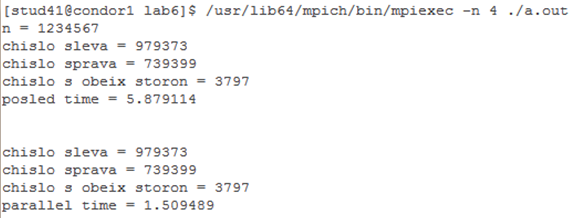
****

Рисунок 6. Результаты выполнения лабораторной работы №6

**Листинг программы**

//36. Число называется укорачиваемым простым слева, если последовательное удаление его десятичных цифр,

// начиная со старшей, дает простые числа (пример: 3797, 797, 97, 7). Это же число является

// укорачиваемым простым справа (3797, 379, 37, 3). Найти максимальные укорачиваемые простые

// слева, справа, с обеих сторон числа, меньшие заданного N

#include <stdio.h>

#include <mpi.h>

#include <malloc.h>

int isSimple(int \*simpleBuf, int pos, int value)

{

for (int i = pos; i > 0; i--)

if (simpleBuf[i] == value)

return 1;

return 0;

}

int reverse(int value)

{

int newValue = 0;

while (value > 0)

{

newValue = newValue \* 10 + (value % 10);

value /= 10;

}

return newValue;

}

void check(int i, int \*answerBuf, int \*simpleBuf, int size)

{

if(i < 0)

return;

int copyForRightCheck = simpleBuf[i];

int copyForLeftCheck = reverse(simpleBuf[i]);

while (copyForRightCheck > 0)

{

copyForRightCheck /= 10;

if (!isSimple(simpleBuf, i, copyForRightCheck))

break;

}

while (copyForLeftCheck > 0)

{

copyForLeftCheck /= 10;

if (!isSimple(simpleBuf, i, copyForLeftCheck))

break;

}

if (copyForLeftCheck == 0 && answerBuf[0] == 0)

answerBuf[0] = simpleBuf[i];

if (copyForRightCheck == 0 && answerBuf[1] == 0)

answerBuf[1] = simpleBuf[i];

if (copyForRightCheck == 0 && copyForLeftCheck == 0 && answerBuf[2] == 0)

answerBuf[2] = simpleBuf[i];

}

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

{

int N = 0, size = 0, copyForRightCheck = 0, copyForLeftCheck = 0, myid = 0, numprocs = 0, rMax = 0, lMax = 0, lrMax = 0;

double startwtime = 0.0;

int answerBuf[4] = { 0, 0, 0 }; // 0 - слева, 1 - справа, 2 - с обеих

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &numprocs);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myid);

if (myid == 0)

{

printf("n = ");

while (1)

{

char c = getchar();

if (((c < '0') || (c > '9')) && (c != '\n'))

{

printf("Input error\n");

return 0;

}

if (c == '\n')

break;

N = N \* 10 + (c - '0');

}

}

MPI\_Bcast(&N, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

int \*simpleBuf = (int\*)malloc(N \* sizeof(int) + 10);

int \*a = (int\*)malloc(N \* sizeof(int) + 10);

for (int i = 0; i < N + 1; i++)

a[i] = i;

for (int p = 2; p < N + 1; p++)

{

if (a[p] != 0)

{

for (int j = p \* 2; j < N + 1; j += p)

a[j] = 0;

}

}

for (int i = 2; i < N && a[i] < N; i++)

{

if (a[i] != 0)

{

simpleBuf[size] = a[i];

size++;

}

}

free(a);

startwtime = MPI\_Wtime();

if(myid == 0)

{

for (int i = size - 1; i > 0; i--)

{

check(i, answerBuf, simpleBuf, size);

}

printf("chislo sleva = %d \n", answerBuf[0]);

printf("chislo sprava = %d \n", answerBuf[1]);

printf("chislo s obeix storon = %d \n", answerBuf[2]);

printf("posled time = %f\n\n\n", MPI\_Wtime() - startwtime);

}

startwtime = MPI\_Wtime();

for (int i = size; i > 0; i -= numprocs)

check(i + myid, answerBuf, simpleBuf, size);

MPI\_Reduce(&answerBuf[0], &lMax, 1, MPI\_INT, MPI\_MAX, 0, MPI\_COMM\_WORLD);

MPI\_Reduce(&answerBuf[1], &rMax, 1, MPI\_INT, MPI\_MAX, 0, MPI\_COMM\_WORLD);

MPI\_Reduce(&answerBuf[2], &lrMax, 1, MPI\_INT, MPI\_MAX, 0, MPI\_COMM\_WORLD);

if (myid == 0)

{

if (lMax > answerBuf[0])

answerBuf[0] = lMax;

if (rMax > answerBuf[1])

answerBuf[1] = rMax;

if (lrMax > answerBuf[2])

answerBuf[2] = lrMax;

printf("chislo sleva = %d \n", answerBuf[0]);

printf("chislo sprava = %d \n", answerBuf[1]);

printf("chislo s obeix storon = %d \n", answerBuf[2]);

printf("parallel time = %f\n\n\n", MPI\_Wtime() - startwtime);

}

free(simpleBuf);

MPI\_Finalize();

}

**Лабораторная работа №7**

**Тема**

Параллельное MPI-программирование с использованием библиотеки MPE для анализа процессов взаимодействия ветвей программы.

**Вариант задачи**

55. Найти минимальное число, большее заданного, которое может быть представлено как сумма степеней 2, 3, 4 и 5 простых чисел (наименьшее такое число есть 60 = 22+23+24+25).

**Алгоритм решения задачи**

Из программы лабораторной работы №8 был удален фрагмент (1).

При помощи команд:

1. */usr/src/mpe2-2.4.9b/bin/mpecc –mpilog –o a.out l8.c*
2. */usr/lib64/mpich/bin/mpiexec –n 4 ./a.out*

был создан файл *a.out.slog2* и запущен при помощи *jumpshot\_launcher.jar*.

**Результаты**

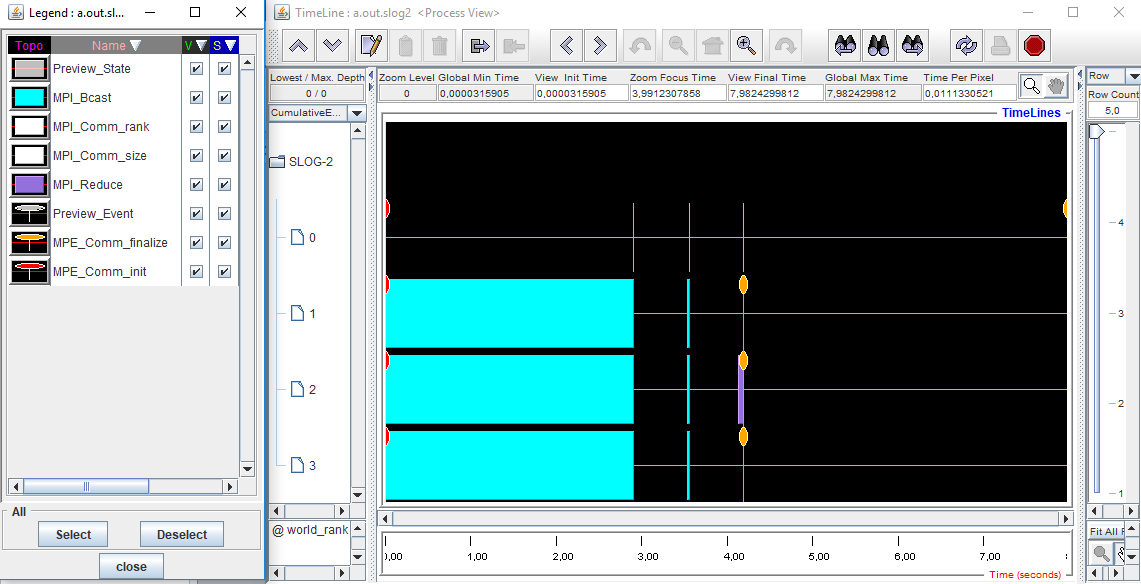


Рисунок 7. Результаты работы *jumpshot.*

На рис. 8 видно, как процессы производят обмен флагами. Такой обмен совершается после каждой проверки одного числа процессом. Методом *MPI\_Reduce* нулевой процесс собирает флаги, после чего рассылает всем флаг – информацию о том, была ли выполнена задача методом *MPI\_Bcast.*

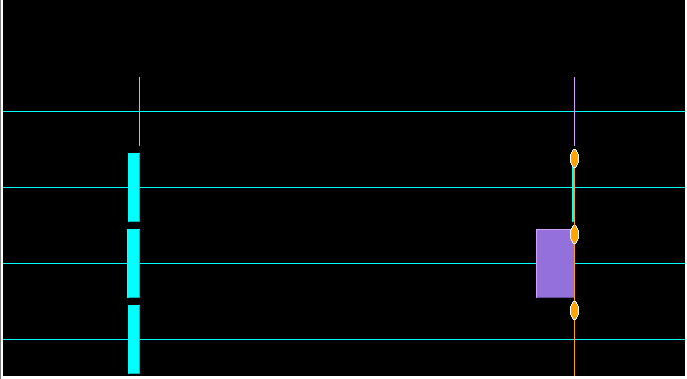


Рисунок 8. Демонстрация операций *MPI\_Reduce* и *MPI\_Bcast.*

**Листинг программы**

См. листинг программы 8.

**Лабораторная работа №8**

**Тема**

Параллельное программирование с одновременным использованием технологий OpenMP и MPI.

**Вариант задачи**

55. Найти минимальное число, большее заданного, которое может быть представлено как сумма степеней 2, 3, 4 и 5 простых чисел (наименьшее такое число есть 60 = 22+23+24+25).

**Алгоритм решения задачи**

У пользователя запрашивается число, больше которого будет производиться поиск. Проверяется корректность введенных данных. Используя «Решето Эратосфена» генерируется массив простых чисел. При помощи четырех вложенных циклов for от числа отнимаются простые числа в степенях 2,3,4,5. Полученные результат сравнивается с нулем и в случае равенства выводится ответ (само число и его разбиение на слагаемые). Если ответ не получен, к числу прибавляется единица и поиск продолжается.

Обмен данными между процессами осуществляется функциями *MPI\_Reduce* и *MPI\_Bcast.* После каждой проверки процессы обмениваются флагами, отвечающими за наличие решения. Если хотя бы один процесс нашел ответ, все остальные перестают работать.

**Результаты**

|  |  |  |
| --- | --- | --- |
| Число N | Последовательное выполнение | Параллельное выполнение |
| 10000 | 2.7 | 1.23 |
| 50000 | 11.3 | 4.21 |
| 123456 | 21.1 | 6.95 |

Подтверждена эффективность распараллеливания циклов технологиями OMP и MPI. Возможен выигрыш примерно в 3 раза при заданном числе ~100000.

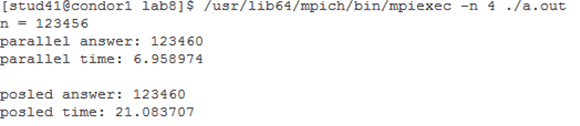
****

Рисунок 9. Результаты выполнения лабораторной работы №8

**Листинг программы**

#include <stdio.h>

#include <malloc.h>

#include <mpi.h>

#include <omp.h>

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

{

double t1, t2;

int n = 0, size = 0, x = 0, tmp = 0, answer = 0, numprocs = 0, myid = 0, res = 0, tmpFlag = 1;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &numprocs);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myid);

if (myid == 0)

{

printf("n = ");

while (1)

{

char c = getchar();

if (((c < '0') || (c > '9')) && (c != '\n'))

{

printf("Input error\n");

return 0;

}

if (c == '\n')

break;

n = n \* 10 + (c - '0');

}

}

MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

int \*a = (int\*)malloc(n \* sizeof(int) + 10);

int \*simpleBuf = (int\*)malloc(n \* sizeof(int) + 10);

for (int i = 0; i < n + 1; i++)

a[i] = i;

for (int p = 2; p < n + 1; p++)

{

if (a[p] != 0)

{

for (int j = p \* 2; j < n + 1; j += p)

a[j] = 0;

}

}

for (int i = 2; i < n && a[i] < sqrt(n); i++)

{

if (a[i] != 0)

{

simpleBuf[size] = a[i];

size++;

}

}

int flag = 1;

tmp = n + 1;

t1 = omp\_get\_wtime();

while (flag)

{

#pragma omp parallel for private(x) // (1)

for (int two = 0; two < size; two++)

for (int three = 0; three < size && flag != 0; three++)

for (int four = 0; four < size; four++)

for (int five = 0; five < size; five++)

{

x = tmp + myid;

x = x - pow(simpleBuf[two], 2) - pow(simpleBuf[three], 3) - pow(simpleBuf[four], 4) - pow(simpleBuf[five], 5);

if (x == 0)

flag = 0;

}

MPI\_Reduce(&flag, &tmpFlag, 1, MPI\_INT, MPI\_MIN, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(&tmpFlag, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

if (!tmpFlag)

{

if (flag == 1)

tmp = 2000000000;

flag = 0;

}

tmp += numprocs;

}

tmp = tmp + myid - numprocs;

MPI\_Reduce(&tmp, &res, 1, MPI\_INT, MPI\_MIN, 0, MPI\_COMM\_WORLD);

t2 = omp\_get\_wtime();

if (myid == 0)

{

printf("parallel answer: %d\n",res);

printf("parallel time: %f\n\n",(t2 - t1));

tmp = n + 1;

flag = 1;

t1 = omp\_get\_wtime();

while (flag)

{

for (int two = 0; two < size; two++)

for (int three = 0; three < size && flag != 0; three++)

for (int four = 0; four < size; four++)

for (int five = 0; five < size; five++)

{

x = tmp - pow(simpleBuf[two], 2) - pow(simpleBuf[three], 3) - pow(simpleBuf[four], 4) - pow(simpleBuf[five], 5);

if (x == 0)

{

flag = 0;

}

}

tmp++;

}

t2 = omp\_get\_wtime();

printf("posled answer: %d\n",(tmp - 1));

printf("posled time: %f\n\n",(t2 - t1));

}

MPI\_Finalize();

}