НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ МОСКОВСКИЙ ЭНЕРГЕТИЧЕСКИЙ
ИНСТИТУТ

КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И
ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Лабораторная работа №1

По дисциплине «Параллельные вычисления»

«Распараллеливание программ с MPI, OpenMP, POSIX»

Группа: А-14м-24
Выполнил: Романов А.Э.
Преподаватель: Шевченко И.В.
Вариант: 9

Москва 2024

Лабораторное задание

Разработать алгоритм проверки простоты числа (перебором делителей), с использованием библиотек распараллеливания:

1) MPI
2) OpenMP
3) POSIX (Pthreads)

Решение задачи

Входные данные: число a (uint64_t), над которым производится проверка.

Выходные данные: результат проверки, потенциальный делитель, затраченное время (мс).

Алгоритм решения: последовательный перебор делителей в отрезке $[2, \sqrt{a}]$ и проверка наличия остатка от деления. В случае, если остаток найден – алгоритм останавливается, дальнейшая проверка не требуется.

1) Распараллеливание с MPI

Библиотека MPI позволяет создать N параллельных процессов, каждый из которых будет обладать собственным рангом (rank). Таким образом, целесообразно создать две функции:

- int masterProcedure() – функция, выполняемая процессом rank = 0
- int slaveProcedure() – функция, выполняемая остальными процессами

Master-функция позволяет процессам «общаться» между собой, путем передачи сообщений, благодаря функциям:

- MPI_Send(…) – функция передачи сообщения определенному процессу
- MPI_Recv(…) – функция приема сообщения от определенного процесса
- MPI_Bcast(…) – рассылка сообщений между всему процессами

Также, эта функция выводит результаты работы, отладочную информацию в консоль.
При запуске программы, процесс с рангом «0» (далее master-процесс) дожидается ввода числа, а далее рассылает остальным процессам их локальные отрезки, на которых будет производиться проверка.
После рассылки, master-процесс записывает время начала выполнения проверки, при помощи функции:

double MPI_Wtime()

Далее, процесс входит в бесконечный цикл ожидания сообщений от slave-процессов. В случае, если некоторый slave-процесс прислал сообщение об успешно найденном делителе, тогда master-процесс известит остальные процессы о том, что дальнейшие поиски делителя необходимо прекратить.

Slave-функция выполняет алгоритм проверки простоты числа в определенном отрезке, в зависимости от ранга процесса, выполняющего данную функцию.
В случае, если в ходе выполнения функции, процесс обнаружит делитель числа, тот отправит сообщение master-процессу, содержащее делитель и прекратит дальнейшее выполнение функции.
Каждые 1.000.000 итераций производится проверка, не поступало ли сообщение от master-процесса. Если сообщение поступало, что свидетельствует о том, что другой slave-процесс уже обнаружил делитель, то дальнейшее выполнение функции также прекращается.

Завершение master-функции свидетельствует о том, что все остальные slave-процессы завершили свою работу. Тогда master-процесс записывает время окончания выполнения проверки, выводит его и результат проверки.

2) Библиотека OpenMP

Библиотека OpenMP позволяет определить параллельные участки кода, при помощи директивы препроцессора:

```
#pragma omp parallel
```

Таким образом, в данной секции становится возможным использование других директив препроцессора и инструментов распараллеливания.
При запуске программы, та дожидается ввода числа и записывается время начала проверки, посредством функции:

```
double omp_get_wtime()
```

После чего запускается параллельный для N потоков цикл, при помощи директивы препроцессора:

```
#pragma omp for
```

Внутри параллельного цикла для каждого потока определяется свой локальный отрезок, на котором осуществляется проверка простоты введенного числа, после чего запускается цикл проверки, по найденному отрезку.
Каждую итерацию производится проверка, найден ли делитель при помощи глобальной для всех потоков переменной divider. Если её значение ненулевое, что свидетельствует о том, что другой поток обнаружил делитель,

тогда дальнейшее выполнение цикла будет прекращено.
В случае, если поток обнаружит делитель, он запишет его в ранее упомянутую глобальную переменную divider.

3) Библиотека Pthreads

Библиотека Pthreads позволяет создавать поток, который будет выполнять передаваемую ему функцию. Таким образом, необходимо создать функцию, которая будет осуществлять проверку числа:

```
void* find_divider(void* param)
```

Функция получает в качестве параметра void-указатель, содержащий в себе номер потока, исполняющего данную функцию. Далее, функция, в зависимости от полученного номера потока, определяет локальный отрезок, на котором выполняется проверка простоты числа. Аналогично алгоритму, используемом ранее, цикл проверки прекращается, если глобальная переменная divider окажется ненулевой.
В случае, если функция обнаружит делитель, она закроет глобальный мьютекс, что позволит избежать одновременной записи данных в переменную divider, после чего запишет найденный делитель и откроет мьютекс.
В основной функции инициализируется вышеупомянутый мьютекс, создается N потоков исполнения, записывается время начала проверки. Эти действия осуществляются при помощи функций:

- pthread_create(…) – создание потока и привязка функции исполнения
- pthread_mutex_init(…) – инициализация мьютекса
- clock() – запись текущего процессорного времени

Проверка работы программ

Для проверки работы и анализа результатов был написан алгоритм при помощи Python. Тесты проводились для разного количества исполняемых потоков (процессов), разных простых чисел:

Таблица 1. Входные данные

Кол-во потоков	Порядок числа	Число
2	10	1073676287
4	13	4398042316799
8	16	1125899839733759
12	19	2305843009213693951

Тестирование осуществлялось на компьютере под управлением ЦП и

оперативной памятью:

- AMD Ryzen 5 5600X
- Kingston DDR4-3200

Таблица 2. Характеристики компьютера

Кол-во ядер	Кол-во потоков	Тактовая частота	Объем RAM	Скорость RAM
6	12	3.7 ГГц	32 ГБ	3200 МГц

Также, при тестировании программ необходимо учесть, что запуск программы на MPI осуществляется с помощью команды:

mpiexec -n N, где N – кол-во процессов

Запустим алгоритм проверки, для всех входных данных:

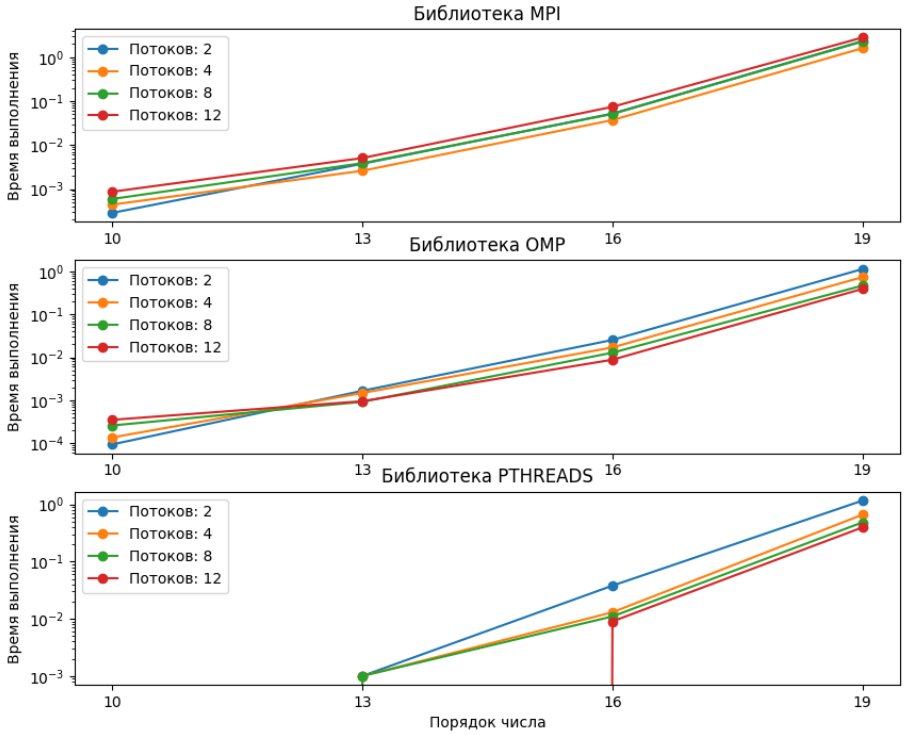

Рисунок 1. График результатов выполнения программ

	MPI	OMP	PTHREADS
2	[0.000287, 0.003762, 0.052012, 2.32081]	[9.2e-05, 0.00165, 0.025506, 1.160393]	[0.0, 0.001, 0.038, 1.168]
4	[0.000442, 0.002605, 0.037125, 1.605156]	[0.000134, 0.001468, 0.017044, 0.750891]	[0.0, 0.001, 0.013, 0.662]
8	[0.000595, 0.003875, 0.051058, 2.278252]	[0.000255, 0.000913, 0.012832, 0.475824]	[0.0, 0.001, 0.011, 0.486]
12	[0.000869, 0.005073, 0.074257, 2.808207]	[0.000346, 0.000942, 0.008809, 0.398353]	[0.0, 0.0, 0.009, 0.397]

Рисунок 2. Таблица результатов выполнения программ

Как видно, из рисунка 1 и рисунка 2, с увеличением числа потоков (процессов), алгоритм справляется с задачей быстрее, что и следовало ожидать. Стоит обратить внимание, что программа, написанная на

библиотеке Pthreads для проверяемого числа порядка 10 выполнялась менее 1 мкс, из-за чего время составило выполнения $time = \sim 0$, из-за чего соответствующая точка не отображается на графике с логарифмированной шкалой.
Проведем сравнение выполнения времени программ для числа порядка «19». Для этого рассчитаем время выполнения программ относительно результата для 12 потоков библиотеки MPI:

$$T_{MPI\_12} = 2.808207 \text{ с}$$

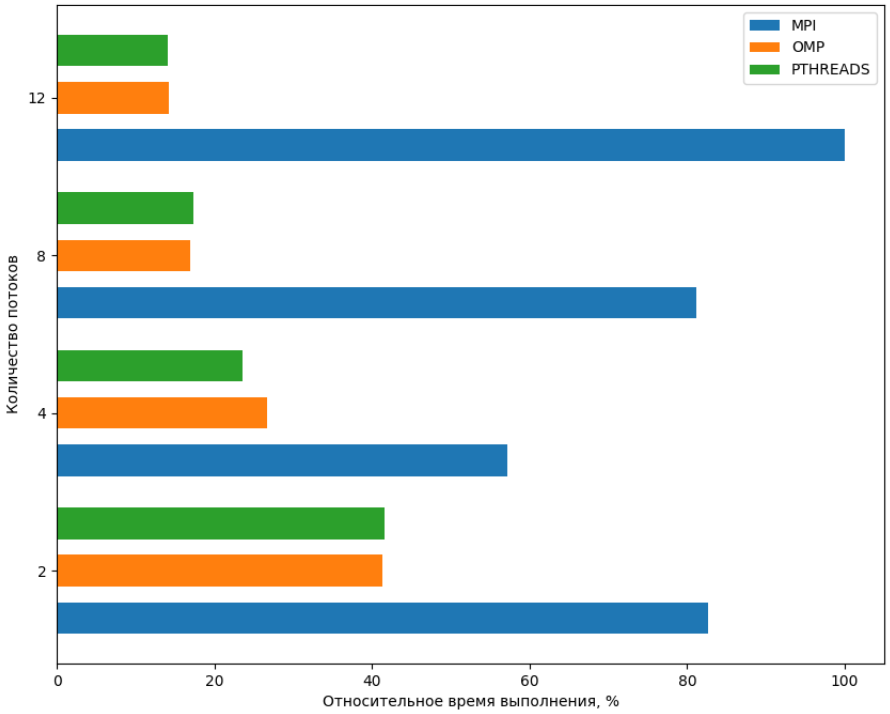

Рисунок 3. График относительных результатов выполнения программ

	MPI	OMP	PTHREADS
2	82.643836	41.321491	41.592375
4	57.159462	26.739161	23.573761
8	81.128350	16.944050	17.306417
12	100.000000	14.185315	14.137134

Рисунок 4. Таблица относительных результатов выполнения программ

Из рисунков 3, 4 видно, что программа, написанная при помощи библиотеки MPI, выполняется медленнее всего, а программа на Pthreads слегка быстрее программы на OpenMP.

Вывод

В ходе выполнения лабораторной работы было написано три программы с использованием разных библиотек для распараллеливания. В результате тестирования и анализа результатов, было выяснено, что библиотека Pthreads оказалась быстрее из всех использованных.

ПРИЛОЖЕНИЕ А. ЛИСТИНГ ПРОГРАММЫ НА MPI

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

#define MASTER 0
#define MAXPROC 128
#define BUFSIZE 256
#define ITERATIONS 1000000

#define TAG_MSG 5
#define TAG_FIN 6
#define TAG_DIV 7
char msg[BUFSIZE];

int rank;
int size;
MPI_Request reqs[MAXPROC];
MPI_Status   stats[MAXPROC];

int simple;
uint64_t number;
uint64_t divider;

uint64_t start;
uint64_t end;
uint64_t segment;

int masterProcedure()
{
	if (number == 0)
	{
		printf_s("Enter number: ");
		fflush(stdout);
		scanf_s("%llu", &number);
	}
	printf_s("Number: %llu\n", number);
	printf_s("Procs num: %d\n", size);

	start = 2;
	end = sqrt(number);
	segment = (end - start) / (size - 1);

	MPI_Bcast(&number, 1, MPI_UINT64_T, MASTER, MPI_COMM_WORLD);
	MPI_Bcast(&end, 1, MPI_UINT64_T, MASTER, MPI_COMM_WORLD);
	MPI_Bcast(&segment, 1, MPI_UINT64_T, MASTER, MPI_COMM_WORLD);
	double timeStart = MPI_Wtime();

	int notFinished = size - 1;
	while (notFinished)
	{
		MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &stats[MASTER]);
		switch (stats[MASTER].MPI_TAG)
		{
		case TAG_DIV:
			MPI_Recv(&divider, 1, MPI_UINT64_T, stats[MASTER].MPI_SOURCE,
stats[MASTER].MPI_TAG, MPI_COMM_WORLD, &stats[MASTER]);
			sprintf_s(msg, "Divider: %llu", divider);
			simple = 0;
			for (int i = 1; i < size; i++)
```

```
                        if (i != stats[MASTER].MPI_SOURCE)
                              MPI_Isend(&simple, 1, MPI_INT, i, TAG_MSG,
MPI_COMM_WORLD, &reqs[i]);
                  break;

            case TAG_FIN:
                  notFinished--;
                  MPI_Recv(&msg, BUFSIZE, MPI_CHAR, stats[MASTER].MPI_SOURCE,
stats[MASTER].MPI_TAG, MPI_COMM_WORLD, &stats[MASTER]);
                  break;

            default:
                  MPI_Recv(&msg, BUFSIZE, MPI_CHAR, stats[MASTER].MPI_SOURCE,
stats[MASTER].MPI_TAG, MPI_COMM_WORLD, &stats[MASTER]);
                  break;
            }

            #ifdef DEBUG
            {
                  printf_s("[MASTER]: Message from [%u]: \"%s\"\n",
stats[MASTER].MPI_SOURCE, msg);
                  fflush(stdout);
            }
            #endif

            if (simple == 0)
                  MPI_Waitall(size - 1, reqs, stats);
      }
      double timeEnd = MPI_Wtime();

      if (simple == 0)
      {
            printf_s("\n[MASTER]: %llu is not simple!\n", number);
            printf_s("[MASTER]: Divider is %llu\n", divider);
      }
      else
            printf_s("\n[MASTER]: %llu is simple!\n", number);
      printf_s("Time: %f\n", timeEnd - timeStart);

      return 0;
}

int slaveProcedure()
{
      MPI_Bcast(&number, 1, MPI_UINT64_T, MASTER, MPI_COMM_WORLD);
      MPI_Bcast(&end, 1, MPI_UINT64_T, MASTER, MPI_COMM_WORLD);
      MPI_Bcast(&segment, 1, MPI_UINT64_T, MASTER, MPI_COMM_WORLD);

      start = 2;
      if (rank > 1)                        start = (rank - 1) * segment;
      else if (rank < size - 1) end = rank * segment;

      #ifdef DEBUG
      {
            sprintf_s(msg, "Range: [%llu, %llu]", start, end);
            MPI_Send(&msg, BUFSIZE, MPI_CHAR, MASTER, TAG_MSG, MPI_COMM_WORLD);
      }
      #endif // DEBUG

      int inmsg = 0;
      uint32_t iterator = 0;
      for (divider = start; divider <= end; divider++)
      {
            if (iterator >= ITERATIONS)
            {
```

```
                    iterator = 0;
                    MPI_Iprobe(MASTER, TAG_MSG, MPI_COMM_WORLD, &inmsg,
&stats[rank]);
                    if (inmsg)
                    {
                          MPI_Recv(&simple, 1, MPI_INT, MASTER, TAG_MSG,
MPI_COMM_WORLD, &stats[rank]);
                          #ifdef DEBUG
                          {
                                sprintf_s(msg, "Aborting!");
                                MPI_Send(&msg, BUFSIZE, MPI_CHAR, MASTER, TAG_MSG,
MPI_COMM_WORLD);
                          }
                          #endif // DEBUG
                          break;
                    }
             }

             iterator++;
             if (number % divider == 0)
             {
                    MPI_Send(&divider, 1, MPI_UINT64_T, MASTER, TAG_DIV,
MPI_COMM_WORLD);
                    break;
             }
      }
      MPI_Send(&msg, BUFSIZE, MPI_CHAR, MASTER, TAG_FIN, MPI_COMM_WORLD);

      return 0;
}

int lab1(int argc, char* argv[])
{
      simple = 1;
      number = 0;
      if (argc > 1)
            number = strtoull(argv[1], NULL, 10);

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

      if (rank == MASTER)      masterProcedure();
      else                           slaveProcedure();

      MPI_Finalize();

      return 0;
}
```

ПРИЛОЖЕНИЕ Б. ЛИСТИНГ ПРОГРАММЫ НА OpenMP

```
#include <omp.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <math.h>
#include <time.h>

int lab2(int argc, char* argv[])
{
	uint64_t num, end;
	uint64_t divider = 0;
	int64_t segment;

	size_t threadCount = 4;
	double timeStart, timeEnd;
	double timeProcedure;

	if (argc > 2)
	{
		threadCount = strtoull(argv[1], NULL, 10);
		num = strtoull(argv[2], NULL, 10);
	}
	else
	{
		printf_s("Enter number: ");
		fflush(stdout);
		scanf_s("%llu", &num);
	}
	printf_s("Threads num: %llu\n", threadCount);
	printf_s("Number: %llu\n", num);

	end = sqrt(num);
	segment = (end - 2) / threadCount;

	omp_set_num_threads(threadCount);
	timeStart = omp_get_wtime();

	#pragma omp parallel
	{
		int thread = omp_get_thread_num();
		#pragma omp for
		for (int i = 0; i < threadCount; i++)
		{
			uint64_t start = i * segment + 2;
			uint64_t end = start + segment;
			double timeProcStart = omp_get_wtime();

			for (uint64_t div = start; div < end && !divider; div++)
			{
				if (!(num % div) != divider)
				{
					divider = div;
					#pragma omp flush
				}
			}
		}
	}
	timeEnd = omp_get_wtime();

	if (divider != 0)  printf_s("%llu is not simple!\nDivider: %llu\n", num,
divider);
	else			printf_s("%llu is simple!\n", num);
```

```
        timeProcedure = timeEnd - timeStart;
        printf_s("Time: %f", timeProcedure);

        return 0;
}
```

ПРИЛОЖЕНИЕ В. ЛИСТИНГ ПРОГРАММЫ НА Pthreads

```
#define HAVE_STRUCT_TIMESPEC
#include <pthread.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <math.h>
#include <time.h>

uint64_t number, segment;
uint64_t divider = 0;

clock_t globalStart;
pthread_mutex_t mutex;

void* find_divider(void* param)
{
	size_t local = *static_cast<size_t*>(param);

	uint64_t start = segment * local + 2;
	uint64_t end = start + segment;

	for (uint64_t div = start; div < end && !divider; div++)
		if (number % div == 0)
		{
			pthread_mutex_lock(&mutex);
			divider = div;
			pthread_mutex_unlock(&mutex);
			break;
		}
	return nullptr;
}

int lab3(int argc, char* argv[])
{
	number = 0;
	size_t size = 4;
	pthread_t* threads;
	size_t* params;

	if (argc > 2)
	{
		size = strtoull(argv[1], NULL, 10);
		number = strtoull(argv[2], NULL, 10);
	}
	else
	{
		printf_s("Enter number: ");
		fflush(stdout);
		scanf_s("%llu", &number);
	}
	printf_s("Threads num: %llu\n", size);
	printf_s("Number: %llu\n", number);

	threads = new pthread_t[size];
	params = new size_t[size];

	uint64_t start = 2;
	uint64_t end = sqrt(number);
	segment = (end - start) / (size);

	pthread_mutex_init(&mutex, NULL);
```

```
	clock_t timeStart = clock();
	globalStart = timeStart;


	for (size_t i = 0; i < size; i++)
	{
		params[i] = i;
		pthread_create(&threads[i], NULL, find_divider,
static_cast<void*>(&params[i]));
	}
	for (size_t i = 0; i < size; i++)
		pthread_join(threads[i], NULL);

	clock_t timeEnd = clock();

	pthread_mutex_destroy(&mutex);
	delete[] threads;
	delete[] params;

	if (divider != 0)  printf_s("%llu is not simple!\nDivider: %llu\n", number,
divider);
	else                        printf_s("%llu is simple!\n", number);
	printf_s("Time: %f\n", double(timeEnd - timeStart) / CLOCKS_PER_SEC);

	return 0;
}
```