МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ

О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ

«Распараллеливание итерационного алгоритма решения системы линейных алгебраических уравнений средствами OpenMP»

студента 2 курса, группы 20205

Муратова Максима Александровича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
доцент
Власенко А. Ю.

Новосибирск, 2022

СОДЕРЖАНИЕ

ЦЕЛИ

1. Распараллелить программу средствами OpenMP;
2. Найти зависимость ускорения программы от числа потоков, исполнявших её;
3. Найти оптимальные параметры для директивы `#pragma omp for schedule (...)`

ЗАДАНИЕ

1. Алгоритм решения СЛАУ — метод сопряжённых градиентов. Пусть нам необходимо решить систему линейных уравнений $Ax=b$, где A – матрица коэффициентов, b – свободные члены, x – искомый вектор решения, $\varepsilon>0$ – погрешность решения.
   Пусть x_0 – первое приближение к решению x, в моей реализации x_0 состоит из единиц. $r_0=b-Ax_0$ – остаток, $z_0=r_0$, α_i,β_i – коэффициенты.
   Одна итерация данного итерационного алгоритма состоит из следующих операций:
   $$r_{n+1}=r_n-\alpha_{n+1}Az_n$$
   $$z_{n+1}=r_{n+1}+\beta_{n+1}z_n$$
   $$\alpha_{n+1}=\frac{(r_n,r_n)}{(Az_n,z_n)}$$
   $$\beta_{n+1}=\frac{(r_{n+1},r_{n+1})}{(r_n,r_n)}$$
   $$x_{n+1}=x_n+a_{n+1}z_n$$
   Условие выхода из цикла – $\frac{\|r_n\|}{\|b\|}<\varepsilon$.
   (a,b) – скалярное произведение в декартовых координатах, $\|a\|=\sqrt{(a,a)}$ – норма вектора.
2. Порядок системы N=2000;
3. Замерить время выполнения программы без балансировки загрузки на 1, 2, 4, 8, 10 и 12 потоках;
4. Замерить время выполнения программы на 10 потоках со статической и динамической балансировками загрузки с размерами блоков от 10 до 200 и шагом 10.

ОПИСАНИЕ РАБОТЫ

Исходный код основной программы содержится единственном файле `parallel.cpp` (приложение 1). Программа собирается утилитой `Makefile`, чья конфигурация описана в приложении 2.

СЛАУ генерируется отдельной программой, чей исходный код приведён в файле `gen_sle.cpp` (приложение 1). Чтобы сгенерировать систему линейных алгебраических уравнений порядка `$k` и записать полученную систему в файл `sle.txt`, необходимо воспользоваться командой

```
$ ./gen-sle $k sle.txt
```

В файле `check.cpp` (приложение 1) описана программа для проверки правильности выполнения программы. Проверка осуществляется перемножением матрицы коэффициентов *A* на полученный вектор *x*. Так, если СЛАУ хранится в файле `sle.txt`, а её решение – в `out.txt`, то сопоставить решения можно командой

```
$ ./check sle.txt out.txt
```

Со сверкой можно ознакомиться в файле `check.txt`. В случае правильной реализации алгоритма содержимое может выглядеть так:

```
Real:

-4.73342 73.2693 28.2871 -99.6021 62.3413 -57.3583
53.2855 -92.8875 60.7129 -95.1852

Evaluated:

-4.73166 73.2666 28.2865 -99.5974 62.3581 -57.3643
53.292 -92.8955 60.7198 -95.1816
```

В приложении 3 описаны скрипты для постановки задач в очередь на выполнение кластером: `run.sh` для замеров времени работы программы при разных количествах потоков, и `scheduler.sh` для замеров времени работы программы на 10 ядрах при разных балансировках загрузки потоков.

Скрипт `run.sh` 4 раза запускает программу при заданном количестве потоков и записывает измерения в `report.txt`. Скрипт требует модификации параметров `ncpus` и `ompthreads` при каждом изменении количестве потоков. Лучшие результаты замеров приведены в приложении 4.

Скрипт `scheduler.sh` согласно заданию делает 4 замера времени при разных видах балансировки. Данный скрипт не требует никаких модификаций и выполняется примерно 45 минут. Результаты записываются в

файл `report-scheduler.txt`, в приложении 5 даны лучшие результаты для каждой банансировки.

ЗАКЛЮЧЕНИЕ

1. При росте числа потоков ускорение программы стремится к квадратному корню из числа потоков;
2. Статическая балансировка загрузки эффективна при большом размере блока итераций, динамическая – при маленькой;
3. Статическая балансировка загрузки потоков ускоряет так же эффективно, как и динамическая, но для этого необходимо правильно подобрать размеры блоков.

- parallel.cpp

```
#include <omp.h>
#include <fstream>
#include <iostream>

std::ostream &print_vector(const double *vector, std::ostream &out, int size) {
        out << vector[0];
        for(int index = 1; index < size; index++)
                out << ' ' << vector[index];
        return out;
}

int main(int argc, char** argv) {
        if(argc != 3) {
                std::cout << "Wrong input\n";
                return 1;
        }

        double epsilon = atof(argv[1]);
        std::ifstream in(argv[2]);
        int size;
        in >> size;
        double start_time = omp_get_wtime();
        double *b = new double[size];
        double *koefs = new double[size * size];
        for(int index = 0; index < size; index++)
                in >> b[index];
        for(int index = 0; index < size * size; index++)
                in >> koefs[index];
        in.close();


        double *result = new double[size];
        double *buffer = new double[size];
        double *rest = new double[size];
        double *new_rest = new double[size];
        double *z = new double[size];
        double *z_1 = new double[size];

        double alpha, beta, control = 0, r2 = 0, nr2 = 0, iter;
        omp_set_nested(false);

        #pragma omp parallel
        {
                #pragma omp for
                for(int index = 0; index < size; index++)
                        result[index] = 1;

                #pragma omp for
                for(int row = 0; row < size; row++) {
                        buffer[row] = 0;
                        for(int column = 0; column < size; column++)
                                buffer[row] += koefs[row * size + column] * result[column];
                }

                #pragma omp for
                for(int index = 0; index < size; index++)
                        rest[index] = b[index] - buffer[index];
```

```
		#pragma omp for
		for(int index = 0; index < size; index++)
			z[index] = rest[index];

		#pragma omp for reduction(+:control)
		for(int index = 0; index < size; index++)
			control += b[index] * b[index];
		#pragma omp single
		control *= epsilon * epsilon;

		#pragma omp for reduction(+:r2)
		for(int index = 0; index < size; index++)
			r2 += rest[index] * b[index];

		for(iter = 0; iter < 10000 && r2 >= control;) {
			#pragma omp for
			for(int row = 0; row < size; row++) {
				z_1[row] = 0;
				for(int column = 0; column < size; column++)
					z_1[row] += koefs[row * size + column] * z[column];
			}

			#pragma omp single
			alpha = 0;
			#pragma omp for reduction(+:alpha)
			for(int index = 0; index < size; index++)
				alpha += z[index] * z_1[index];
			#pragma omp single
			alpha = r2 / alpha;

			#pragma omp for
			for(int index = 0; index < size; index++)
				new_rest[index] = rest[index] - alpha * z_1[index];

			#pragma omp single
			nr2 = 0;
			#pragma omp for reduction(+:nr2)
			for(int index = 0; index < size; index++)
				nr2 += new_rest[index] * new_rest[index];

			#pragma omp single
			beta = nr2 / r2;

			#pragma omp for
			for(int index = 0; index < size; index++)
				result[index] += alpha * z[index];

			#pragma omp for
			for(int index = 0; index < size; index++)
				z[index] = new_rest[index] + beta * z[index];

			#pragma omp single
			{
				std::swap(rest, new_rest);
				r2 = nr2;
				iter++;
			}
		}
	}
```

```
        print_vector(result, std::cout, size) << '\n';
        std::cerr << omp_get_wtime() - start_time << '\n';

        delete[] b;
        delete[] koefs;
        delete[] result;
        delete[] buffer;
        delete[] rest;
        delete[] new_rest;
        delete[] z;
        delete[] z_1;
        return 0;
}
```

- gen_sle.cpp

```
#include <fstream>
#include <iostream>
#include <climits>

double random_double() {
        return rand() / (INT_MAX / 200.) - 100;
}

int main(int argc, char ** argv) {
        if(argc != 3) {
                std::cerr << "Wrong input\n";
                return 1;
        }

        int size = std::stoi(argv[1]);
        if(size <= 0) {
                std::cerr << "Size cannot be less than 1\n";
                return 1;
        }
        std::ofstream out(argv[2]);
        srand(time(0));

        out << size << '\n' << random_double();
        for(int index = 1; index < size; index++) {
                out << '\t' << random_double();
        }
        out << '\n';
        out.flush();

        double *arr = new double[size * size], *pre = new double[size * size];
        for(int row = 0; row < size; row++) {
                for(int column = row + 1; column < size; column++) {
                        arr[row * size + column] = arr[column * size + row]
                                                = random_double();
                }
        }
        for(int index = 0; index < size; index++)
                arr[(size + 1) * index] = random_double() + 150 + size;

        for(int row = 0; row < size; row++) {
                for(int column = 0; column < size; column++) {
                        out << arr[row * size + column] << '\t';
                }
                out << '\n';
                out.flush();
        }
```

```
        delete[] arr;
        delete[] pre;
        return 0;
}
```

- check.cpp

```
#include <iostream>
#include <fstream>

std::ostream &print_vector(const double *vector, std::ostream &out, int size) {
        out << vector[0];
        for(int index = 1; index < size; index++)
                out << ' ' << vector[index];
        return out;
}

int main(int argc, char **argv) {
        if(argc != 3) {
                std::cout << "Wrong input\n";
                return 1;
        }

        std::ifstream sle(argv[1]);
        std::ifstream ans(argv[2]);
        std::ofstream out("check.txt");

        int size;
        sle >> size;
        double *b0 = new double[size];
        double *x = new double[size];
        double *matrix = new double[size * size];
        double *b = new double[size];
        for(int index = 0; index < size; index++)
                sle >> b0[index];
        for(int index = 0; index < size * size; index++)
                sle >> matrix[index];
        sle.close();
        for(int index = 0; index < size; index++)
                ans >> x[index];
        ans.close();

        for(int row = 0; row < size; row++) {
                b[row] = 0;
                for(int column = 0; column < size; column++)
                        b[row] += matrix[row * size + column] * x[column];
        }

        out << "Real:\n";
        print_vector(b0, out, size);
        out << "\nEvaluated:\n";
        print_vector(b, out, size) << '\n';
        out.close();

        delete[] b0;
        delete[] b;
        delete[] x;
        delete[] matrix;
        return 0;
}
```

Приложение 2. *Makefile*

```
CXX=g++
CFLAGS=-O3 -c -Wall -std=c++11
CFLAGS_OMP=$(CFLAGS) -fopenmp

PARALLEL_SOURCE=parallel

SLE_GEN_SOURCE=gen_sle
SLE_GEN_EXE=gen-sle

CHECK_FILE=check

all: $(PARALLEL_SOURCE) $(SLE_GEN_EXE) $(CHECK_FILE)


$(PARALLEL_SOURCE): $(PARALLEL_SOURCE).o
	$(CXX) -fopenmp $(PARALLEL_SOURCE).o -o $(PARALLEL_SOURCE)

$(PARALLEL_SOURCE).o: $(PARALLEL_SOURCE).cpp
	$(CXX) $(CFLAGS_OMP) $(PARALLEL_SOURCE).cpp -o $(PARALLEL_SOURCE).o



$(SLE_GEN_EXE): $(SLE_GEN_SOURCE).o
	$(CXX) $(SLE_GEN_SOURCE).o -o $(SLE_GEN_EXE)

$(SLE_GEN_SOURCE).o: $(SLE_GEN_SOURCE).cpp
	$(CXX) $(CFLAGS) $(SLE_GEN_SOURCE).cpp -o $(SLE_GEN_SOURCE).o

$(CHECK_FILE): $(CHECK_FILE).o
	$(CXX) $(CHECK_FILE).o -o $(CHECK_FILE)

$(CHECK_FILE).o: $(CHECK_FILE).cpp $(HEADER)
	$(CXX) $(CFLAGS) $(CHECK_FILE).cpp -o $(CHECK_FILE).o

clean:
	rm -rf *.o
```

- run.sh

```
#!/bin/bash
#PBS -l walltime=00:04:00
#PBS -l select=1:ncpus=12:ompthreads=12:mem=60m

cd $PBS_O_WORKDIR
echo -e "OMP_NUM_THREADS = $OMP_NUM_THREADS\n"
REPORT_FILE=report.txt

for (( iter = 0; iter < 4; iter++ )); do
        echo -ne "$OMP_NUM_THREADS\t" >>$REPORT_FILE
        ./parallel .0001 tests/2000.txt >/dev/null 2>>$REPORT_FILE
done
```

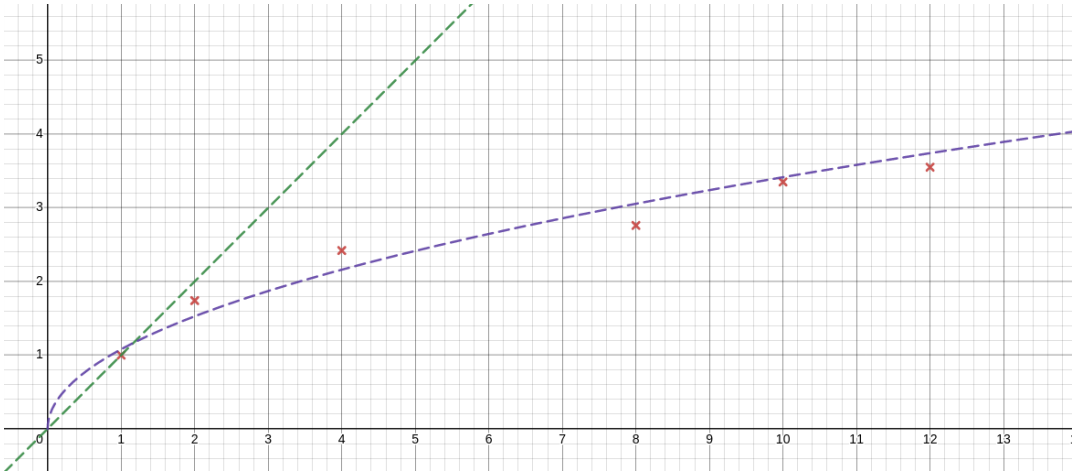
- scheduler.sh

```
#!/bin/bash
#PBS -l walltime=00:55:00
#PBS -l select=1:ncpus=10:ompthreads=10:mem=60m

cd $PBS_O_WORKDIR
echo -e "OMP_NUM_THREADS = $OMP_NUM_THREADS\n"
REPORT_FILE=report-schedule.txt

SCHEDULER_TYPE=static
for (( CHUNK = 10; CHUNK <= 200; CHUNK = $CHUNK + 10 )); do
        export OMP_SCHEDULE="$SCHEDULER_TYPE, $CHUNK"
        for (( iter = 0; iter < 4; iter++ )); do
                echo -ne "$SCHEDULER_TYPE\t$CHUNK\t" >>$REPORT_FILE
                ./parallel .0001 tests/2000.txt >/dev/null $CHUNK 2>>$REPORT_FILE
        done
done

SCHEDULER_TYPE=dynamic
for (( CHUNK = 10; CHUNK <= 200; CHUNK = $CHUNK + 10 )); do
        export OMP_SCHEDULE="$SCHEDULER_TYPE, $CHUNK"
        for (( iter = 0; iter < 4; iter++ )); do
                echo -ne "$SCHEDULER_TYPE\t$CHUNK\t" >>$REPORT_FILE
                ./parallel .0001 tests/2000.txt >/dev/null $CHUNK 2>>$REPORT_FILE
        done
done
```

Приложение 4. *Зависимость времени выполнения от числа потоков*

Число потоков	Время, с	Ускорение	Эффективность, %
1	51.52	1.00	100.00
2	29.69	1.74	87.00
4	21.29	2.42	60.50
8	18.65	2.76	34.50
10	15.38	3.35	33.50
12	14.52	3.55	29.58

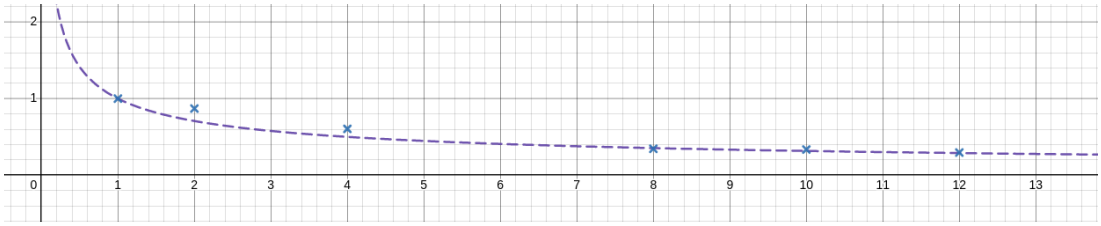
График зависимости ускорения программы от числа потоков



Легенда:

- Ox – число задействованных потоков t ;
- Oy – ускорение при данном числе процессов n ;
- Красные крестики – реальные ускорения программы при заданных количествах потоков;
- Зелёная линия – график фунции $n=t$;
- Фиолетовая кривая – аппроксимация ускорения функцией $n=1.08\sqrt{t}$

График зависимости эффективности потоков от их числа:



Легенда:

- Ox – число задействованных потоков t ;
- Oy – эффективность потоков k ;
- Красные крестики – реальные эффективности потоков при заданных количествах потоков;
- Фиолетовая кривая – аппроксимация эффективност функцией $k = 1/\sqrt{t}$

Приложение 5. *Зависимость времени выполнения от размера блоков*

Размер блока	Время при статической загрузке, с	Время при динамической загрузке, с
10	21.4016	20.3846
20	20.1958	16.4881
30	20.3322	15.8869
40	19.7951	15.8077
50	18.8751	16.3059
60	18.8091	16.8579
70	18.2475	16.6933
80	18.4681	17.5751
90	18.3762	17.9635
100	17.0047	16.4097
110	17.2035	16.5062
120	17.5084	18.2037
130	17.9624	18.8808
140	17.6127	18.6596
150	18.2223	19.0219
160	18.1690	19.4215
170	18.0360	19.7309
180	18.8877	19.9315
190	17.1530	18.2837
200	15.3596	17.7643