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 ЛАБОРАТОРНОЙ РАБОТЫ

«Параллельная реализация решения системы линейных алгебраических уравнений с помощью OpenMP»

студента 2 курса, 21204 группы

Осипова Александра Александровича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Кандидат технических наук
А. Ю. Власенко

Новосибирск 2023__

СОДЕРЖАНИЕ

ЦЕЛЬ

Реализовать параллельную программу, вычисляющую приближенное решение СЛАУ с помощью метода сопряженных градиентов, используя стандарт OpenMP.

ЗАДАНИЕ

1. Последовательную программу из практической работы 2, реализующую итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b методом сопряженных градиентов, распараллелить с помощью OpenMP.

2. Замерить время работы программы на кластере НГУ на 1, 2, 4, 8, 12, 16 потоках. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные и параметры задачи подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.

3. Провести исследование на определение оптимальных параметров #pragma omp for schedule(...) при некотором фиксированном размере задачи и количестве потоков.

ОПИСАНИЕ РАБОТЫ

За основу был взят код последовательной программы из Практической работы №2, реализующей итерационный алгоритм решения СЛАУ методом сопряженных градиентов. Полный листинг программы см. Приложение 1

Вектор x инициализируется нулями.

```
double* InitPreSolution(int N) {
	double* x = (double*)malloc(N * sizeof(double));
	for (int i = 0; i < N; ++i) {
		x[i] = 0.0;
	}
	return x;
}
```

Вектор b инициализируется следующим образом: b[i] = i + 1

```
double* InitVectorB(int N) {
	double* b = (double*)malloc(N * sizeof(double));
	for (int i = 0; i < N; ++i) {
		b[i] = i + 1;
	}
	return b;
}
```

Для генерации коэффициентов матрицы A и ее инициализации были написаны следующие функции:

```
void GenerateMatrixA(int N) {
    double* A = (double*)malloc(N * N * sizeof(double));
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            double value = (double)rand() / RAND_MAX + (i == j ?
(double)SIZE / 100 : 0);
            A[i * N + j] = (i > j ? A[j * N + i] : value);
        }
    }

    std::ofstream fileIn;
    fileIn.open("A.txt");
    if (fileIn.is_open()) {
        for (int i = 0; i < N * N; ++i) {
            fileIn << A[i] << std::endl;
        }
        fileIn.close();
    }
}
```

```
double* InitMatrixA(int N) {
    std::ifstream file;
    file.open("A.txt");

    double* A = (double*)malloc(N * N * sizeof(double));
```

```
    for (int i = 0; i < N * N; ++i) {
          file >> A[i];
    }

    file.close();
    return A;
}
```

Функции, реализующие операции над матрицами с помощью OpenMP:

- умножение

```
void MatrixMULT(double* A, double* B, double* C, int L, int M, int N) {
#pragma omp for
     for (int i = 0; i < L; ++i) {
          for (int j = 0; j < N; ++j) {
               C[i * N + j] = 0;
               for (int k = 0; k < M; ++k) {
                    C[i * N + j] += A[i * M + k] * B[k * N + j];
               }
          }
     }
}
```

- вычитание

```
void MatrixSUB(double* A, double* B, double* C, int N) {
#pragma omp for
     for (int i = 0; i < N; ++i) {
          C[i] = A[i] - B[i];
     }
}
```

- сложение

```
void MatrixADD(double* A, double* B, double* C, int N) {
#pragma omp for
     for (int i = 0; i < N; ++i) {
          C[i] = A[i] + B[i];
     }
}
```

- умножение на скаляр

```
void ScalarMULT(double scalar, double* matrix, double* result, int N) {
#pragma omp for
     for (int i = 0; i < N; ++i) {
          result[i] = scalar * matrix[i];
     }
}
```

- скалярное произведение

```
void DotProduct(double* a, double* b, double* dot) {
#pragma omp single
	* dot = 0.0;
#pragma omp for reduction (+:dot[0])
	for (int k = 0; k < SIZE; ++k) {
		dot[0] += (a[k] * b[k]);
	}
}
```

- вычисление нормы

```
void Norm(double* vector, double* norm) {
#pragma omp single
	* norm = 0.0;
#pragma omp for reduction(+:norm[0])
	for (int k = 0; k < SIZE; ++k) {
		norm[0] += (vector[k] * vector[k]);
	}
#pragma omp single
	* norm = sqrt(*norm);
}
```

- копирование матрицы

```
void CopyMatrix(double* source, double* purpose, int size) {
#pragma omp for
	for (int i = 0; i < size; ++i) {
		purpose[i] = source[i];
	}
}
```

Функция, реализующая алгоритм вычисления приближенного решения СЛАУ с помощью метода сопряженных градиентов. Здесь же создается параллельная секция с numThreads потоками.

```
void ConjugateGradientMethod(double* A, double* x, double* b, int
numThreads) {
	double* r = (double*)malloc(SIZE * sizeof(double));
	double* r_next = (double*)malloc(SIZE * sizeof(double));
	double* z = (double*)malloc(SIZE * sizeof(double));
	double* tmpVector = (double*)malloc(SIZE * sizeof(double));
	double tmpValues[2];

	double alpha = 0.0;
	double beta = 0.0;

	int count = 0;
```

```
#pragma omp parallel num_threads(numThreads)
	{
		InitVectorR(r, A, x, b, tmpVector);
		CopyMatrix(r, z, SIZE);

		for (count = 0; count < 50000; ++count) {
			if (isSolutionReached(r, b, tmpValues)) break;
			CalcNextAlpha(&alpha, A, r, z, tmpVector, tmpValues);
			CalcNextX(x, z, alpha, tmpVector);
			CalcNextR(A, r, z, alpha, r_next, tmpVector);
			CalcNextBeta(&beta, r_next, r, tmpValues);
			CalcNextZ(beta, r_next, z);
			CopyMatrix(r_next, r, SIZE);
		}
	}
	free(r);
	free(r_next);
	free(z);
	free(tmpVector);
}
```

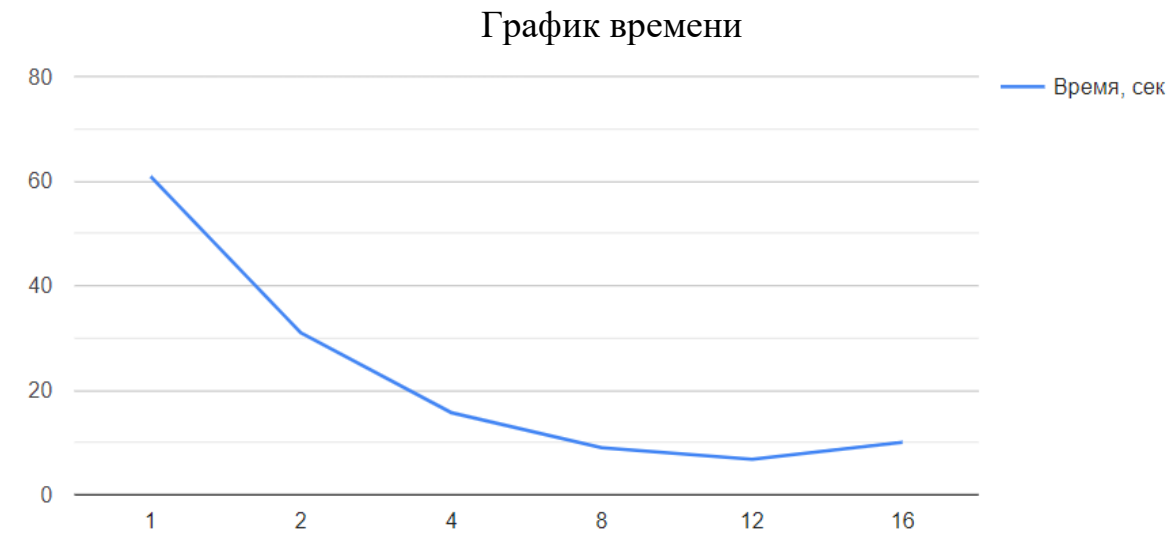
Функция int main(int argc, char** argv). В ней поставлен таймер, определяющий, за какое время программа проинициализирует матрицу A, векторы b и x, а так же найдет приближенное значения вектора решений. Для этих целей был выбран таймер системного времени omp_get_wtime().

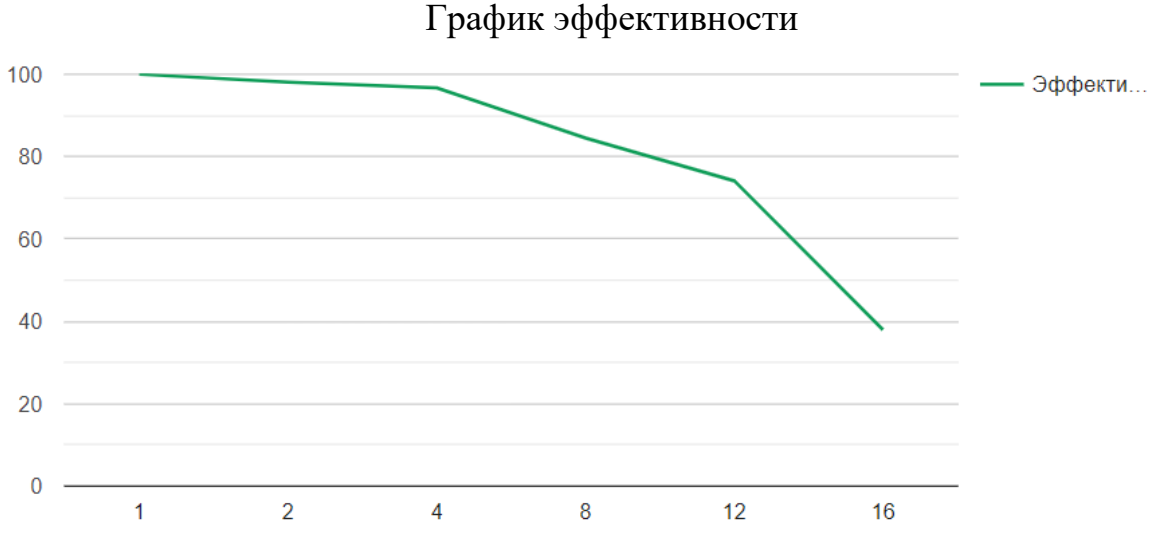
```
int main(int argc, char** argv) {
	int numThreads = (argc > 1 ? atoi(argv[1]) : 1);

	double startTime = omp_get_wtime();

	double* A = InitMatrixA(SIZE);
	double* x = InitPreSolution(SIZE);
	double* b = InitVectorB(SIZE);

	ConjugateGradientMethod(A, x, b, numThreads);

	double endTime = omp_get_wtime();

	PrintResult(numThreads, endTime - startTime);

	free(A);
	free(x);
	free(b);
}
```

Результат работы программы при разном числе потоков (см.Приложение 6):

Число потоков	Время, сек	Ускорение	Эффективность, %
1	60.89	1	100
2	30.99	1.96	98.1
4	15.72	3.87	96.69
8	8.99	6.76	84.51

12	6.83	8.89	74.13
16	10.02	6.07	37.93





Для проведения исследования оптимальных параметров #pragma omp schedule (…) в параллельный цикл функций, реализующих операции над матрицами, было добавлено schedule(runtime) (см. Приложение 2). Значение этого параметра определяется во время исполнения программы, из переменной окружения OMP_SCHEDULE. Значение OMP_SCHEDULE задается в скрипте run3.sh (см. Приложение 5).

Результаты исследования для 8 потоков:

chunk_size	Время, сек		
	static	dynamic	quided
default	8.9243	43.393299	9.249285
25	9.318743	9.117198	9.298394
50	9.447319	9.767110	9.561208
75	10.442399	9.913324	9.828618
100	9.232287	9.409798	9.547210
125	11.605041	12.040716	11.449481
150	13.382983	15.365792	14.171284
175	11.894821	12.230380	11.418960
200	9.135278	9.199780	9.206766
225	10.479452	10.606715	10.644026
250	11.854896	11.884664	11.583129

Видно, что лучшее время показала программа, при schedule(static). Это обусловлено тем, что итерации статически распределяются между потоками, а так как итерации являются легковесными (умножение и присваивание), то и получается отрыв от dynamic и guided, которые распределяют итерации динамически.

ЗАКЛЮЧЕНИЕ

В рамках данной практической работы был изучен стандарт OpenMP. С помощью него была написана многопоточная программа, реализующая итерационный алгоритм решения СЛАУ методом сопряженных градиентов. По сравнению с MPI, OpenMP имеется такие плюсы, как скорость написания программы, использование общей памяти. Среди минусов можно выделить ограничение только на один вычислительный узел, так как все потоки порождаются в рамках одного процесса. В связи с этим программу с OpenMP нельзя распараллелить на нескольких ЭВМ.

Приложение 1. Листинг программы, без директивы schedule

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fstream>

#define SIZE 1500
#define EPSILON 0.00001
#define T1 60.8

void PrintVector(double* vector, int N) {
	for (int i = 0; i < N; ++i) {
		printf("%f ", vector[i]);
	}
	printf("\n");
}

void PrintMatrix(double* matrix, int N) {
	for (int i = 0; i < N; ++i) {
		for (int j = 0; j < N; ++j) {
			printf("%f ", matrix[i * N + j]);
		}
		printf("\n");
	}
	printf("\n");
}


void CopyMatrix(double* source, double* purpose, int size) {
#pragma omp for
	for (int i = 0; i < size; ++i) {
		purpose[i] = source[i];
	}

}

void MatrixMULT(double* A, double* B, double* C, int L, int M, int N) {
#pragma omp for
	for (int i = 0; i < L; ++i) {
		for (int j = 0; j < N; ++j) {
			C[i * N + j] = 0;
			for (int k = 0; k < M; ++k) {
				C[i * N + j] += A[i * M + k] * B[k * N + j];
			}
		}
	}
}

void MatrixSUB(double* A, double* B, double* C, int N) {
#pragma omp for
	for (int i = 0; i < N; ++i) {
		C[i] = A[i] - B[i];
	}
```

```
}

void MatrixADD(double* A, double* B, double* C, int N) {
#pragma omp for
      for (int i = 0; i < N; ++i) {
            C[i] = A[i] + B[i];
      }
}

void ScalarMULT(double scalar, double* matrix, double* result, int N) {
#pragma omp for
      for (int i = 0; i < N; ++i) {
            result[i] = scalar * matrix[i];
      }
}

void DotProduct(double* a, double* b, double* dot) {
#pragma omp single
      * dot = 0.0;
#pragma omp for reduction(+:dot[0])
      for (int k = 0; k < SIZE; ++k) {
            dot[0] += (a[k] * b[k]);
      }
}

void Norm(double* vector, double* norm) {
#pragma omp single
      * norm = 0.0;
#pragma omp for reduction(+:norm[0])
      for (int k = 0; k < SIZE; ++k) {
            norm[0] += (vector[k] * vector[k]);
      }
#pragma omp single
      * norm = sqrt(*norm);
}

void InitVectorR(double* r, double* A, double* x, double* b, double*
tmp) {
      MatrixMULT(A, x, tmp, SIZE, SIZE, 1);
      MatrixSUB(b, tmp, r, SIZE);
}

void CalcNextX(double* x, double* z, double alpha, double* tmp) {
      ScalarMULT(alpha, z, tmp, SIZE);
      MatrixADD(x, tmp, x, SIZE);
}

void CalcNextR(double* A, double* r, double* z, double alpha, double*
r_next, double* tmpVector) {
      MatrixMULT(A, z, tmpVector, SIZE, SIZE, 1);
      ScalarMULT(alpha, tmpVector, tmpVector, SIZE);
      MatrixSUB(r, tmpVector, r_next, SIZE);
}

void CalcNextZ(double beta, double* r_next, double* z) {
      ScalarMULT(beta, z, z, SIZE);
```

```
	MatrixADD(r_next, z, z, SIZE);
}

void CalcNextAlpha(double* alpha, double* A, double* r, double* z,
double* tmpVector, double* tmpValues) {
	MatrixMULT(A, z, tmpVector, SIZE, SIZE, 1);
	DotProduct(r, r, &tmpValues[0]);
	DotProduct(tmpVector, z, &tmpValues[1]);
#pragma omp single
	* alpha = tmpValues[0] / tmpValues[1];
}

void CalcNextBeta(double* beta, double* r_next, double* r, double*
tmpValues) {
	DotProduct(r_next, r_next, &tmpValues[0]);
	DotProduct(r, r, &tmpValues[1]);
#pragma omp single
	* beta = tmpValues[0] / tmpValues[1];
}

bool isSolutionReached(double* r, double* b, double* norms) {
	Norm(r, &norms[0]);
	Norm(b, &norms[1]);

	return (norms[0] / norms[1]) < EPSILON;
}

void ConjugateGradientMethod(double* A, double* x, double* b, int
numThreads) {
	double* r = (double*)malloc(SIZE * sizeof(double));
	double* r_next = (double*)malloc(SIZE * sizeof(double));
	double* z = (double*)malloc(SIZE * sizeof(double));
	double* tmpVector = (double*)malloc(SIZE * sizeof(double));
	double tmpValues[2];

	double alpha = 0.0;
	double beta = 0.0;

	int count = 0;

#pragma omp parallel num_threads(numThreads)
	{
		InitVectorR(r, A, x, b, tmpVector);
		CopyMatrix(r, z, SIZE);

		for (count = 0; count < 50000; ++count) {
			if (isSolutionReached(r, b, tmpValues)) break;
			CalcNextAlpha(&alpha, A, r, z, tmpVector, tmpValues);
			CalcNextX(x, z, alpha, tmpVector);
			CalcNextR(A, r, z, alpha, r_next, tmpVector);
			CalcNextBeta(&beta, r_next, r, tmpValues);
			CalcNextZ(beta, r_next, z);
			CopyMatrix(r_next, r, SIZE);
		}
	}
	free(r);
```

```
	free(r_next);
	free(z);
	free(tmpVector);
}

double* InitMatrixA(int N) {
	std::ifstream file;
	file.open("A.txt");

	double* A = (double*)malloc(N * N * sizeof(double));
	for (int i = 0; i < N * N; ++i) {
		file >> A[i];
	}

	file.close();
	return A;
}

double* InitVectorB(int N) {
	double* b = (double*)malloc(N * sizeof(double));
	for (int i = 0; i < N; ++i) {
		b[i] = i + 1;
	}
	return b;
}

double* InitPreSolution(int N) {
	double* x = (double*)malloc(N * sizeof(double));
	for (int i = 0; i < N; ++i) {
		x[i] = 0.0;
	}
	return x;
}

void GenerateMatrixA(int N) {
	double* A = (double*)malloc(N * N * sizeof(double));
	for (int i = 0; i < N; ++i) {
		for (int j = 0; j < N; ++j) {
			double value = (double)rand() / RAND_MAX + (i == j ?
(double)SIZE / 100 : 0);
			A[i * N + j] = (i > j ? A[j * N + i] : value);
		}
	}

	std::ofstream fileIn;
	fileIn.open("A.txt");
	if (fileIn.is_open()) {
		for (int i = 0; i < N * N; ++i) {
			fileIn << A[i] << std::endl;
		}
		fileIn.close();
	}
}

void PrintResult(int numThreads, double totalTime) {
	float boost = T1 / totalTime;
```

```
	float efficiency = (boost / (float)numThreads) * 100;

	printf("Number of threads: %d\n", numThreads);
	printf("Total time: %f sec\n", totalTime);
	printf("Boost: %f\n", boost);
	printf("Efficiency: %f %\n\n", efficiency);
}


int main(int argc, char** argv) {
	int numThreads = (argc > 1 ? atoi(argv[1]) : 1);

	double startTime = omp_get_wtime();

	double* A = InitMatrixA(SIZE);
	double* x = InitPreSolution(SIZE);
	double* b = InitVectorB(SIZE);

	ConjugateGradientMethod(A, x, b, numThreads);

	double endTime = omp_get_wtime();

	PrintResult(numThreads, endTime - startTime);

	free(A);
	free(x);
	free(b);
    }
```

Приложение 2. Листинг программы с директивой schedule

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fstream>

#define SIZE 1500
#define EPSILON 0.00001
#define T1 60.8

void PrintVector(double* vector, int N) {
	for (int i = 0; i < N; ++i) {
		printf("%f ", vector[i]);
	}
	printf("\n");
}

void PrintMatrix(double* matrix, int N) {
	for (int i = 0; i < N; ++i) {
		for (int j = 0; j < N; ++j) {
			printf("%f ", matrix[i * N + j]);
		}
		printf("\n");
```

```
	}
	printf("\n");
}


void CopyMatrix(double* source, double* purpose, int size) {
#pragma omp for schedule(runtime)
	for (int i = 0; i < size; ++i) {
		purpose[i] = source[i];
	}
}

void MatrixMULT(double* A, double* B, double* C, int L, int M, int N) {
#pragma omp for schedule(runtime)
	for (int i = 0; i < L; ++i) {
		for (int j = 0; j < N; ++j) {
			C[i * N + j] = 0;
			for (int k = 0; k < M; ++k) {
				C[i * N + j] += A[i * M + k] * B[k * N + j];
			}
		}
	}
}

void MatrixSUB(double* A, double* B, double* C, int N) {
#pragma omp for schedule(runtime)
	for (int i = 0; i < N; ++i) {
		C[i] = A[i] - B[i];
	}
}

void MatrixADD(double* A, double* B, double* C, int N) {
#pragma omp for schedule(runtime)
	for (int i = 0; i < N; ++i) {
		C[i] = A[i] + B[i];
	}
}

void ScalarMULT(double scalar, double* matrix, double* result, int N) {
#pragma omp for schedule(runtime)
	for (int i = 0; i < N; ++i) {
		result[i] = scalar * matrix[i];
	}
}

void DotProduct(double* a, double* b, double* dot) {
#pragma omp single
	* dot = 0.0;
#pragma omp for schedule(runtime) reduction(+:dot[0])
	for (int k = 0; k < SIZE; ++k) {
		dot[0] += (a[k] * b[k]);
	}
}

void Norm(double* vector, double* norm) {
#pragma omp single
```

```
      * norm = 0.0;
#pragma omp for schedule(runtime) reduction(+:norm[0])
      for (int k = 0; k < SIZE; ++k) {
            norm[0] += (vector[k] * vector[k]);
      }
#pragma omp single
      * norm = sqrt(*norm);
}

void InitVectorR(double* r, double* A, double* x, double* b, double*
tmp) {
      MatrixMULT(A, x, tmp, SIZE, SIZE, 1);
      MatrixSUB(b, tmp, r, SIZE);
}

void CalcNextX(double* x, double* z, double alpha, double* tmp) {
      ScalarMULT(alpha, z, tmp, SIZE);
      MatrixADD(x, tmp, x, SIZE);
}

void CalcNextR(double* A, double* r, double* z, double alpha, double*
r_next, double* tmpVector) {
      MatrixMULT(A, z, tmpVector, SIZE, SIZE, 1);
      ScalarMULT(alpha, tmpVector, tmpVector, SIZE);
      MatrixSUB(r, tmpVector, r_next, SIZE);
}

void CalcNextZ(double beta, double* r_next, double* z) {
      ScalarMULT(beta, z, z, SIZE);
      MatrixADD(r_next, z, z, SIZE);
}

void CalcNextAlpha(double* alpha, double* A, double* r, double* z,
double* tmpVector, double* tmpValues) {
      MatrixMULT(A, z, tmpVector, SIZE, SIZE, 1);
      DotProduct(r, r, &tmpValues[0]);
      DotProduct(tmpVector, z, &tmpValues[1]);
#pragma omp single
      * alpha = tmpValues[0] / tmpValues[1];
}

void CalcNextBeta(double* beta, double* r_next, double* r, double*
tmpValues) {
      DotProduct(r_next, r_next, &tmpValues[0]);
      DotProduct(r, r, &tmpValues[1]);
#pragma omp single
      * beta = tmpValues[0] / tmpValues[1];
}

bool isSolutionReached(double* r, double* b, double* norms) {
      Norm(r, &norms[0]);
      Norm(b, &norms[1]);

      return (norms[0] / norms[1]) < EPSILON;
}
```

```
void ConjugateGradientMethod(double* A, double* x, double* b) {
	double* r = (double*)malloc(SIZE * sizeof(double));
	double* r_next = (double*)malloc(SIZE * sizeof(double));
	double* z = (double*)malloc(SIZE * sizeof(double));
	double* tmpVector = (double*)malloc(SIZE * sizeof(double));
	double tmpValues[2];

	double alpha = 0.0;
	double beta = 0.0;

	int count = 0;

#pragma omp parallel
	{
		InitVectorR(r, A, x, b, tmpVector);
		CopyMatrix(r, z, SIZE);

		for (count = 0; count < 50000; ++count) {
			if (isSolutionReached(r, b, tmpValues)) break;
			CalcNextAlpha(&alpha, A, r, z, tmpVector, tmpValues);
			CalcNextX(x, z, alpha, tmpVector);
			CalcNextR(A, r, z, alpha, r_next, tmpVector);
			CalcNextBeta(&beta, r_next, r, tmpValues);
			CalcNextZ(beta, r_next, z);
			CopyMatrix(r_next, r, SIZE);
		}
	}
	free(r);
	free(r_next);
	free(z);
	free(tmpVector);
}

double* InitMatrixA(int N) {
	std::ifstream file;
	file.open("A.txt");

	double* A = (double*)malloc(N * N * sizeof(double));
	for (int i = 0; i < N * N; ++i) {
		file >> A[i];
	}

	file.close();
	return A;
}

double* InitVectorB(int N) {
	double* b = (double*)malloc(N * sizeof(double));
	for (int i = 0; i < N; ++i) {
		b[i] = i + 1;
	}
	return b;
}

double* InitPreSolution(int N) {
	double* x = (double*)malloc(N * sizeof(double));
```

```
	for (int i = 0; i < N; ++i) {
		x[i] = 0.0;
	}
	return x;
}

void GenerateMatrixA(int N) {
	double* A = (double*)malloc(N * N * sizeof(double));
	for (int i = 0; i < N; ++i) {
		for (int j = 0; j < N; ++j) {
			double value = (double)rand() / RAND_MAX + (i == j ?
(double)SIZE / 100 : 0);
			A[i * N + j] = (i > j ? A[j * N + i] : value);
		}
	}

	std::ofstream fileIn;
	fileIn.open("A.txt");
	if (fileIn.is_open()) {
		for (int i = 0; i < N * N; ++i) {
			fileIn << A[i] << std::endl;
		}
		fileIn.close();
	}
}

void PrintResult(double totalTime) {
	printf("Total time: %f sec\n\n", totalTime);
}


int main(int argc, char** argv) {
	double startTime = omp_get_wtime();

	double* A = InitMatrixA(SIZE);
	double* x = InitPreSolution(SIZE);
	double* b = InitVectorB(SIZE);

	ConjugateGradientMethod(A, x, b);

	double endTime = omp_get_wtime();

	PrintResult(endTime - startTime);

	free(A);
	free(x);
	free(b);
}
```

Приложение 3. Скрипт run1.sh

```
#!/bin/bash

#PBS -l walltime=00:05:00
#PBS -l select=1:ncpus=12

cd $PBS_O_WORKDIR

echo "The result with 1 thread:"
./default 1
```

Приложение 4. Скрипт run2.sh

```
#!/bin/bash

#PBS -l walltime=00:10:00
#PBS -l select=1:ncpus=12

cd $PBS_O_WORKDIR

./parallel 2
./parallel 4
./parallel 8
./parallel 12
./parallel 16
```

Приложение 5. Скрипт run3.sh

```
#!/bin/bash

#PBS -l walltime=00:20:00
#PBS -l select=1:ncpus=8:ompthreads=8

cd $PBS_O_WORKDIR

echo "Number of threads: $OMP_NUM_THREADS"

### STATIC SCHEDULE ###

echo "Static default:"
OMP_SCHEDULE="static" ./schedule
echo

for ((i = 1; i <= 10; i++))
do
    echo "Static $((i * 25))"
    OMP_SCHEDULE="static,$((i * 25))" ./schedule
    echo
done
```

```
### DYNAMIC SCHEDULE ###

echo "Dynamic default:"
OMP_SCHEDULE="dynamic" ./schedule
echo

for ((i = 1; i <= 10; i++))
do
    echo "Dynamic $((i * 25))"
    OMP_SCHEDULE="dynamic,$((i * 25))" ./schedule
    echo
done

### GUIDED SCHEDULE ###

echo "Guided default:"
OMP_SCHEDULE="guided" ./schedule
echo

for ((i = 1; i <= 10; i++))
do
    echo "Guided $((i * 25))"
    OMP_SCHEDULE="guided,$((i * 25))" ./schedule
    echo
done
```

Приложение 6. Результат работы программы на разном числе потоков

```
hpcuser60@clu:~/lab2> cat run.sh.o5415652
The result with 1 thread:
Time: 60.889022 sec

hpcuser60@clu:~/lab2> cat run2.sh.o5415753
Number of threads: 2
Total time: 30.989585 sec
Boost: 1.961949
Efficiency: 98.097473

Number of threads: 4
Total time: 15.720759 sec
Boost: 3.867498
Efficiency: 96.687439

Number of threads: 8
Total time: 8.992997 sec
Boost: 6.760816
Efficiency: 84.510201

Number of threads: 12
Total time: 6.834377 sec
Boost: 8.896202
Efficiency: 74.135017

Number of threads: 16
Total time: 10.016880 sec
Boost: 6.069755
Efficiency: 37.935966
```

Приложение 7. Результат работы программы с директивой schedule

```
Number of threads: 8
Static default:
Total time: 8.924300 sec

Static 25
Total time: 9.318743 sec

Static 50
Total time: 9.447319 sec

Static 75
Total time: 10.442399 sec

Static 100
Total time: 9.232287 sec

Static 125
Total time: 11.605041 sec

Static 150
Total time: 13.382983 sec

Static 175
Total time: 11.894821 sec

Static 200
Total time: 9.135278 sec

Static 225
Total time: 10.479452 sec

Static 250
Total time: 11.854896 sec
```

```
Dynamic default:
Total time: 43.393299 sec

Dynamic 25
Total time: 9.117198 sec

Dynamic 50
Total time: 9.767110 sec

Dynamic 75
Total time: 9.913324 sec

Dynamic 100
Total time: 9.409798 sec

Dynamic 125
Total time: 12.040716 sec

Dynamic 150
Total time: 15.365792 sec

Dynamic 175
Total time: 12.230380 sec

Dynamic 200
Total time: 9.199780 sec

Dynamic 225
Total time: 10.606715 sec

Dynamic 250
Total time: 11.884664 sec
```

```
Guided default:
Total time: 9.249285 sec

Guided 25
Total time: 9.298394 sec

Guided 50
Total time: 9.561208 sec

Guided 75
Total time: 9.828618 sec

Guided 100
Total time: 9.547210 sec

Guided 125
Total time: 11.449481 sec

Guided 150
Total time: 14.171284 sec

Guided 175
Total time: 11.418960 sec

Guided 200
Total time: 9.206766 sec

Guided 225
Total time: 10.644026 sec

Guided 250
Total time: 11.583129 sec
```