Отчет по курсу "Параллельные алгоритмы и программирование" аспиранта 2 курса кафедры СП Одерова Р.С.

Задача

Перемножение двух матриц произвольного размера с использованием OpenMP.

Формула умножения матриц

Даны две матрицы A и B:

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix} \in \mathbb{R}^{n \times m}$$

$$B = \begin{bmatrix} b_{11} & \cdots & b_{1k} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mk} \end{bmatrix} \in \mathbb{R}^{m \times k}$$

Результат умножения A * B – матрица C:

$$C = \begin{bmatrix} c_{11} & \cdots & c_{1k} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nk} \end{bmatrix} \in \mathbb{R}^{n \times k}$$

Элементы матрицы C вычисляются по формуле:

$$c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj}, i \in [1..n], j \in [1..k]$$

Описание программ

Были реализованы две программы, работающие схожим образом:

1. Однопоточное умножение матриц (mult.cpp,код в приложении)
2. Умножение матриц с использованием OpenMP (ompmult.cpp, код в приложении)

Входные данные

4 числа: m1.row, m1.column, m2.row, m2.column – размеры двух матриц, которые необходимо перемножить.

Выходные данные

Время работы основной части программы, перемножающей соответствующие матрицы.

Логика работы

Программы считывают входные параметры и генерируют две матрицы соответствующих размеров, которые заполняются случайными двузначными целыми числами в интервале от 0 до 99.

Затем каждая программа реализует алгоритм умножения матриц (с использованием OpenMP и без него).

Замечание:

Поизводительность алгоритма перемножения матриц сильно зависит от того, в каком порядке идут циклы перемножения. В зависимости от того, как хранятся данные в переменной языка, программа будет работать по-разному.

Так, например, в C/C++ данные двумерного динамического массива хранятся по строкам, из чего следует, что наиболее выгодным алгоритмом будет цикл со следующим порядком перебора: сначала промежуточный индекс, потом по строкам первой матрицы, потом по столбцам второй.

После умножения матриц происходит очистка памяти программы, матрицы удаляются.

Особенности реализации программы с использованием OpenMP

Создание и заполнение двух начальных матриц происходит параллельно. После этого реализуется механизм синхронизации «барьер», гарантирующий консистентное состояние матриц перед умножением.

Непосредственно умножение матриц распараллеливается.

После умножения происходит параллельное удаление матриц.

Описание тестов

Был проведен анализ поведения программ на двух машинах:

Ноутбук Lenovo W540
Характеристики процессора:

Name	Intel Core i7 4800MQ
Architecture	Haswell
Core speed	2.70 GHz
Turbo speed	3.70 GHz
Cores/CPU	4
Threads	8

Сервер на базе процессора Intel Xeon
Характеристики процессора:

Name	Intel Xeon E5 2620
Architecture	Sandy Bridge
Core speed	2.00 GHz
Turbo speed	2.50 GHz
Cores/CPU	12
Threads	24

Тесты, касающиеся сравнения параллельного/однопоточного исполнения

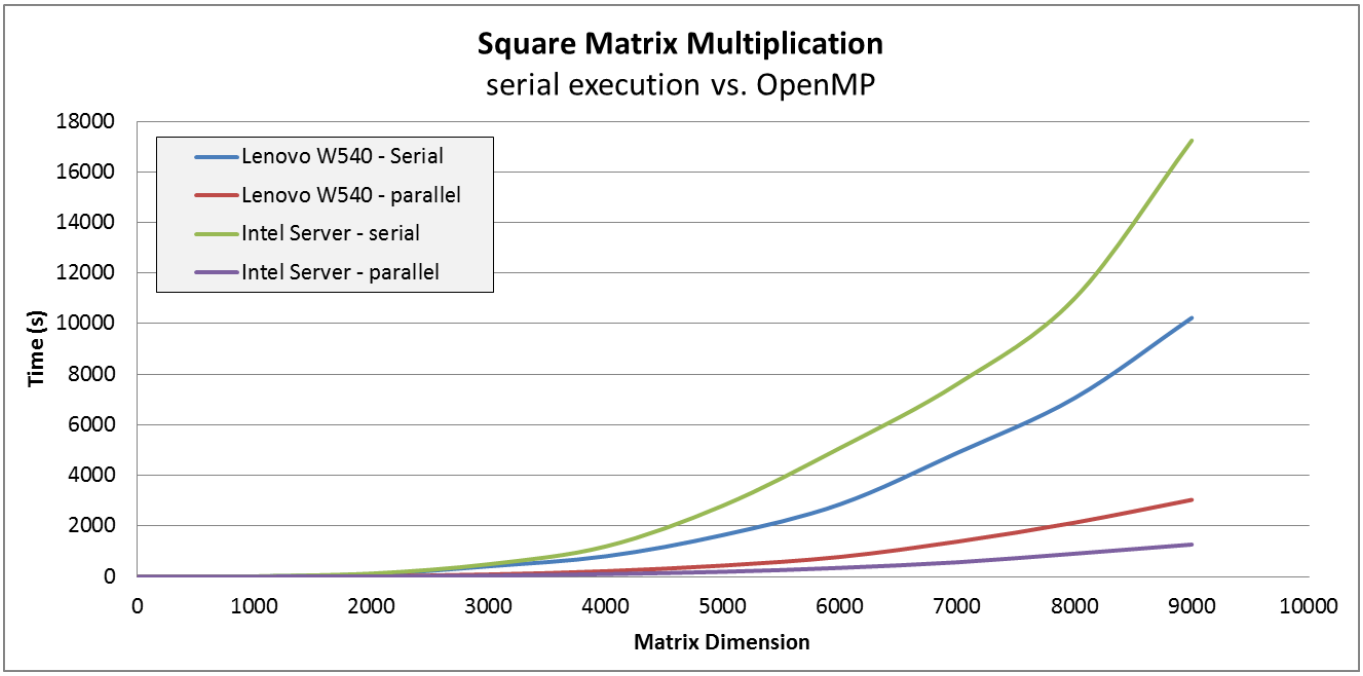
Был проведен набор тестов, направленных на выявление скорости работы программ. Написан скрипт (run.bat, код в приложении) для автоматического запуска программ на тестовых входных данных. В этих тестах последовательность циклов – IJR (в соответствии с нотацией, приведенной в первом разделе отчета о математических основах алгоритма).
Производились умножения *квадратных* матриц различных размеров:

- 10x10 … 90x90 (шаг 10)
- 100x100 … 900x900 (шаг 100)
- 1000x1000 … 9000x9000 (шаг 1000)

Полученные результаты приведены в таблице:

Matrices Multiplication: M1 * M2				**Lenovo W540 Time [s]**		**Xeon Server Time [s]**		**Lenovo W540 / Xeon Server difference**	
M1.rows	M1.cols	M2.rows	M2.cols	serial	parallel	serial	parallel	serial	parallel
10	10	10	10	0.000004	0.000011	0.000006	0.001021	33%	99%
20	20	20	20	0.000029	0.000044	0.000045	0.000081	36%	46%
30	30	30	30	0.000102	0.000064	0.000147	0.000068	31%	6%
40	40	40	40	0.000229	0.000129	0.000375	0.000344	39%	63%
50	50	50	50	0.000444	0.000241	0.000727	0.000293	39%	18%
60	60	60	60	0.00081	0.00026	0.001249	0.000455	35%	43%
70	70	70	70	0.001287	0.000576	0.001944	0.000542	34%	-6%
80	80	80	80	0.001846	0.000601	0.002916	0.000788	37%	24%
90	90	90	90	0.002711	0.001281	0.004356	0.000984	38%	-30%
100	100	100	100	0.003736	0.002115	0.006069	0.001119	38%	-89%
200	200	200	200	0.027996	0.014391	0.045424	0.006319	38%	-128%
300	300	300	300	0.109099	0.057509	0.17606	0.019937	38%	-188%
400	400	400	400	0.278953	0.142871	0.451062	0.051471	38%	-178%
500	500	500	500	0.553804	0.221079	0.928427	0.104153	40%	-112%
600	600	600	600	0.954281	0.352771	1.569393	0.185396	39%	-90%
700	700	700	700	1.553753	0.583441	2.649654	0.306694	41%	-90%
800	800	800	800	2.456212	0.879135	4.592137	0.479049	47%	-84%
900	900	900	900	3.584902	1.392291	8.17568	0.679903	56%	-105%
1000	1000	1000	1000	6.215091	2.16541	11.465421	1.060573	46%	-104%
2000	2000	2000	2000	90.815533	20.141343	122.475968	9.676807	26%	-108%
3000	3000	3000	3000	409.181545	83.788432	491.66925	36.369182	17%	-130%
4000	4000	4000	4000	806.029024	213.328962	1196.893513	100.937073	33%	-111%
5000	5000	5000	5000	1647.023656	437.862508	2811.935392	192.719973	41%	-127%
6000	6000	6000	6000	2865.348739	779.806426	5093.273309	345.879586	44%	-125%
7000	7000	7000	7000	4897.769141	1385.678036	7621.07247	567.635845	36%	-144%
8000	8000	8000	8000	7060.372263	2135.082497	11002.82183	908.553246	36%	-135%
9000	9000	9000	9000	10230.27882	3036.532248	17246.2816	1267.835867	41%	-140%

По результатам можно построить график зависимости времени работы алгоритма перемножения матриц от размеров матрицы:



Выводы

Исходя из проведенных тестов, можно сделать следующие выводы/наблюдения.

- Последовательное исполнение заметно медленнее, чем параллельное.

- Последовательное исполнение напрямую зависит от частоты процессора.
  Казалось бы, промышленный мощный сервер должен справиться с подсчетами быстрее, чем ноутбук, однако из-за различных тактовых частот ноутбук «выиграл» в последовательном однопоточном исполнении.
  Результаты ноутбука лучше серверных примерно в полтора раза, что отражает отношение тактовых частот соответствующих процессоров.

- Параллельное исполнение непосредственно зависит от ресурсов CPU.
  На серверном процессоре, имеющем 24 логических ядра, программы исполнялись в разы быстрее, чем на ноутбуке, на котором доступно только 8 логических ядер.

- На маленьких входных данных (перемножение маленьких матриц) наблюдался проигрыш параллельного алгоритма по сравнению с однопоточным.
  Судя по всему, подобные результаты могут быть связаны с затратами на переключение контекста и управление потоками. Однако когда вычислительная трудоемкость задачи растет, эти сопутствующие потери сглаживаются и становятся пренебрежительно малы.

Тесты, касающиеся сравнения различных последовательностей циклов алгоритма:

Были рассмотрены две реализации алгоритма перемножения матриц с двумя вариантами последовательности основного цикла программы:

- J K I
- K I J

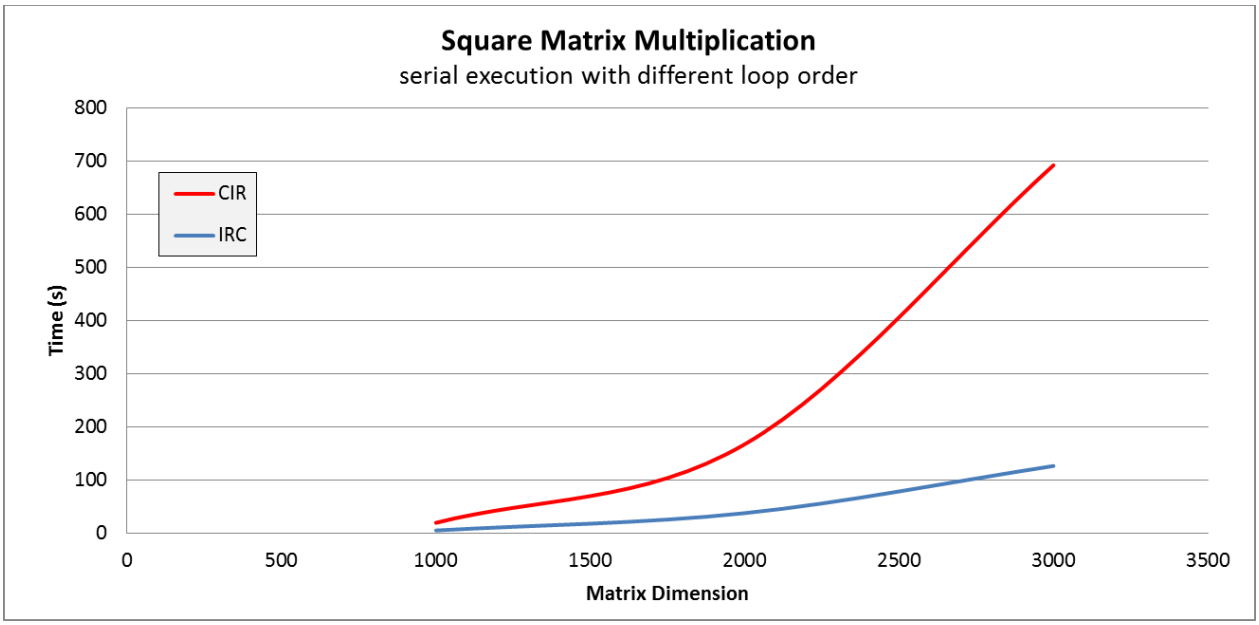
**индексы в нотации, соответствующей первому разделу данного отчета с описанием математических основ алгоритма.*

Однопоточная программа запускалась на матрицах трех размеров:

- 1000x1000
- 2000x2000
- 3000x3000

Результаты представлены ниже:

Matrices Multiplication: M1 * M2				**Lenovo W540**		
				serial execution		
M1.rows	M1.cols	M2.rows	M2.cols	CIR	IRC	delta
1000	1000	1000	1000	19.01911	4.671445	-307%
2000	2000	2000	2000	167.1236	37.26865	-348%
3000	3000	3000	3000	693.1378	126.1778	-449%



Выводы

Очевидно, намного быстрее происходит умножения матриц, когда все данные попадают в кэш, т.е. циклы должны быть организованы в том порядке, чтобы максимально эффективно обходить языковые структуры хранения данных.

Приложение 1. mult.cpp

```
#include "stdafx.h"

bool omp_support() {
#ifdef _OPENMP
	return true;
#else
	return false;
#endif
}
int** createMatrix(int rows, int cols) {
	int** matrix = new int*[rows];
	for (int i = 0; i < rows; i++) {
		matrix[i] = new int[cols];
	}
	return matrix;
}

void zeroInitMatrix(int** matrix, int rows, int cols) {
	for (int i = 0; i < rows; i++) {
		for (int j = 0; j < cols; j++) {
			matrix[i][j] = 0;
		}
	}
}
void randFillMatrix(int** matrix, int rows, int cols) {
	for (int i = 0; i < rows; i++) {
		for (int j = 0; j < cols; j++) {
			matrix[i][j] = rand() % 100;
		}
	}
}
void deleteMatrix(int** matrix, int rows) {
	for (int r = 0; r < rows; ++r) {
		delete[] matrix[r];
	}
	delete[] matrix;
}
void printMatrix(int** matrix, int rows, int cols) {
	for (int i = 0; i < rows; i++) {
		for (int j = 0; j < cols; j++) {
			printf("%2d ", matrix[i][j]);
		}
		printf("\n");
	}
}

int main(int argc, char* argv[])
{
	// check the number of arguments
	if (argc != 5) {
		printf("Program needs 4 integer arguments divided by space: M1.rows M1.columns
M2.rows M2.columns\n");
		return 1;
	}
	// check if OpenMP is available
	if (!omp_support()) {
		printf("Unfortunately, OpenMP is not available.\n");
		return 1;
```

```
	}

	//Define matrices
	int rows1 = atoi(argv[1]);
	int cols1 = atoi(argv[2]);
	int rows2 = atoi(argv[3]);
	int cols2 = atoi(argv[4]);
	int **matrix1, **matrix2;

	if (cols1 != rows2) {
		printf("Program needs 4 integer arguments divided by space: M1.rows M1.columns M2.rows M2.columns\nUnfortunately, you've entered wrong matrix size:\n\tM1.columns must be equal to M2.rows!\n");
		return 1;
	}

	int rows3 = rows1;
	int cols3 = cols2;
	int** matrix3;

	srand(time(NULL));

	//create the first matrix
	matrix1 = createMatrix(rows1, cols1);
	//create the second matrix
	matrix2 = createMatrix(rows2, cols2);
	//create the third/multiplication matrix
	matrix3 = createMatrix(rows3, cols3);

	//fill the first matrix
	randFillMatrix(matrix1, rows1, cols1);
	//fill the second matrix
	randFillMatrix(matrix2, rows2, cols2);
	//fill the second matrix
	zeroInitMatrix(matrix3, rows3, cols3);

	auto start_time = omp_get_wtime();
	/*
	IRC
	*/
	for (int i = 0; i < cols1; i++) {
		for (int r = 0; r < rows1; r++) {
			for (int c = 0; c < cols2; c++) {
				matrix3[r][c] = matrix3[r][c] + matrix1[r][i] * matrix2[i][c];
			}
		}
	}
	auto run_time = omp_get_wtime() - start_time;
	printf("parallel - irc,%s,%s,%s,%s,%f\n", argv[1], argv[2], argv[3], argv[4], run_time);


	start_time = omp_get_wtime();
	/*
	CIR
	*/
	for (int c = 0; c < cols2; c++) {
		for (int i = 0; i < cols1; i++) {
			for (int r = 0; r < rows1; r++) {
				matrix3[r][c] = matrix3[r][c] + matrix1[r][i] * matrix2[i][c];
			}
```

```
			}
		}
		run_time = omp_get_wtime() - start_time;
		printf("parallel - cir,%s,%s,%s,%s,%f\n", argv[1], argv[2], argv[3], argv[4],
run_time);


		//delete the first matrix
		deleteMatrix(matrix1, rows1);
		//delete the second matrix
		deleteMatrix(matrix2, rows2);
		//delete the third (multiplication) matrix
		deleteMatrix(matrix3, rows3);

		return 0;
}
```

Приложение 2. ompmult.cpp

```
#include "stdafx.h"

bool omp_support() {
#ifdef _OPENMP
		return true;
#else
		return false;
#endif
}

int** createMatrix(int rows, int cols) {
		int** matrix = new int*[rows];
		for (int i = 0; i < rows; i++) {
			matrix[i] = new int[cols];
		}
		return matrix;
}

void zeroInitMatrix(int** matrix, int rows, int cols) {
		for (int i = 0; i < rows; i++) {
			for (int j = 0; j < cols; j++) {
				matrix[i][j] = 0;
			}
		}
}

void randFillMatrix(int** matrix, int rows, int cols) {
		for (int i = 0; i < rows; i++) {
			for (int j = 0; j < cols; j++) {
				matrix[i][j] = rand() % 100;
			}
		}
}

void deleteMatrix(int** matrix, int rows) {
		for (int r = 0; r < rows; ++r) {
			delete[] matrix[r];
		}
```

```
	delete[] matrix;
}

void printMatrix(int** matrix, int rows, int cols) {
	for (int i = 0; i < rows; i++) {
		for (int j = 0; j < cols; j++) {
			printf("%2d ",matrix[i][j]);
		}
		printf("\n");
	}
}

int main(int argc, char* argv[])
{
	// check the number of arguments
	if (argc != 5) {
		printf("Program needs 4 integer arguments divided by space: M1.rows
M1.columns M2.rows M2.columns\n");
		return 1;
	}
	// check if OpenMP is available
	if (!omp_support()) {
		printf("Unfortunately, OpenMP is not available.\n");
		return 1;
	}

	//Define matrices
	int rows1 = atoi(argv[1]);
	int cols1 = atoi(argv[2]);
	int rows2 = atoi(argv[3]);
	int cols2 = atoi(argv[4]);
	int **matrix1, **matrix2;

	if (cols1 != rows2) {
		printf("Program needs 4 integer arguments divided by space: M1.rows
M1.columns M2.rows M2.columns\nUnfortunately, you've entered wrong matrix
size:\n\tM1.columns must be equal to M2.rows!\n");
		return 1;
	}

	int rows3 = rows1;
	int cols3 = cols2;
	int** matrix3;

	srand(time(NULL));

	//omp_set_nested(1);

#pragma omp parallel
{
#pragma omp sections
	{
	#pragma omp section
		{
			//create the first matrix
			matrix1 = createMatrix(rows1, cols1);
		}
	#pragma omp section
```

```
		{
			//create the second matrix
			matrix2 = createMatrix(rows2, cols2);
		}
	#pragma omp section
		{
			//create the third/multiplication matrix
			matrix3 = createMatrix(rows3, cols3);
		}
	}

#pragma omp sections
	{
	#pragma omp section
		{
			//fill the first matrix
			randFillMatrix(matrix1, rows1, cols1);
		}
	#pragma omp section
		{
			//fill the second matrix
			randFillMatrix(matrix2, rows2, cols2);
		}
	#pragma omp section
		{
			//fill the second matrix
			zeroInitMatrix(matrix3, rows3, cols3);
		}
	}
}

	auto start_time = omp_get_wtime();
#pragma omp parallel for
	/*
	CIR
	*/
	for (int c = 0; c < cols2; c++) {
		for (int i = 0; i < cols1; i++) {
			for (int r = 0; r < rows1; r++) {
				matrix3[r][c] = matrix3[r][c] + matrix1[r][i] * matrix2[i][c];
			}
		}
	}
	auto run_time = omp_get_wtime() - start_time;
	printf("parallel,%s,%s,%s,%s,%f\n", argv[1], argv[2], argv[3], argv[4], run_time);

	start_time = omp_get_wtime();
#pragma omp parallel for
/*
	IRC
	*/
	for (int i = 0; i < cols1; i++) {
		for (int r = 0; r < rows1; r++) {
			for (int c = 0; c < cols2; c++) {
				matrix3[r][c] = matrix3[r][c] + matrix1[r][i] * matrix2[i][c];
			}
		}
	}
	run_time = omp_get_wtime() - start_time;
```

```
	printf("parallel - cir,%s,%s,%s,%s,%f\n", argv[1], argv[2], argv[3], argv[4],
run_time);



//#pragma omp barrier
#pragma omp parallel
{

	#pragma omp sections
	{
		#pragma omp section
		{
			//delete the first matrix
			deleteMatrix(matrix1, rows1);
		}
		#pragma omp section
		{
			//delete the second matrix
			deleteMatrix(matrix2, rows2);
		}
		#pragma omp section
		{
			//delete the third (multiplication) matrix
			deleteMatrix(matrix3, rows3);
		}
	}
}

	return 0;
}
```

Приложение 3. run.bat

```
@echo off

set ser="mult.exe"
set par="ompmult.exe"
set out="out.txt"
set to=30

REM
REM :: SERIAL PROG RUNS ::
REM

for /L %%c in (10,10,90) DO (
echo %%c
cmd /C "%ser% %%c %%c %%c %%c >> %out%"
timeout %to%
)

for /L %%c in (100,100,900) DO (
echo %%c
cmd /C "%ser% %%c %%c %%c %%c >> %out%"
timeout %to%
)
```

```
for /L %%c in (1000,1000,9000) DO (
echo %%c
cmd /C "%ser% %%c %%c %%c %%c >> %out%"
timeout %to%
)

REM
REM :: PARALLEL PROG RUNS ::
REM

for /L %%c in (10,10,90) DO (
echo %%c
cmd /C "%par% %%c %%c %%c %%c >> %out%"
timeout %to%
)

for /L %%c in (100,100,900) DO (
echo %%c
cmd /C "%par% %%c %%c %%c %%c >> %out%"
timeout %to%
)

for /L %%c in (1000,1000,9000) DO (
echo %%c
cmd /C "%par% %%c %%c %%c %%c >> %out%"
timeout %to%
)
```