

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И КИБЕРНЕТИКИ
КАФЕДРА СУПЕРКОМПЬЮТЕРОВ И КВАНТОВОЙ ИНФОРМАТИКИ



КАФЕДРАЛЬНЫЙ ПРАКТИКУМ

ЗАДАНИЕ 2: ПАРАЛЛЕЛЬНАЯ ПРОГРАММА НА МРІ, РЕАЛИЗУЮЩАЯ ОДНОКУБИТНОЕ КВАНТОВОЕ ПРЕОБРАЗОВАНИЕ

Выполнил:
Алёшин Н.А.
группа 323

Москва 2020

Постановка задачи.

Разработать схему распределенного хранения данных и параллельный алгоритм для реализации однокубитного квантового преобразования на кластерной системе.

Реализовать параллельную программу на C++ с использованием MPI, которая выполняет однокубитное квантовое преобразование над вектором состояний длины 2^n , где n – количество кубитов, по указанному номеру кубита k .

Протестировать программу на системе Polus. В качестве теста использовать преобразование Адамара по номеру кубита:

- a) 1;
- b) 2 (номер в списке группы + 1);
- c) n .

Начальное состояние вектора должно генерироваться случайным образом (генерация вектора тоже должна быть распараллелена). Заполнить таблицу и построить график зависимости ускорения программы от числа потоков для каждого из случаев а) – с).

Провести тестирование разработанного решения на корректность. Необходимо запустить программу на одном и том же входном векторе на 1, 2, 4 и 8 процессах и показать, что выходной вектор совпадает во всех экспериментах. Тест провести для моделирования 16 кубитов. Входной вектор должен быть сгенерирован случайным образом.

Схема распределенного хранения данных и параллельный алгоритм.

1) Режим чтения.

Запуск:

```
make ; mpirun -np <threads> go read <inputFile> <countQubits> <numQubit> <outputFile>
```

Потоки параллельно читают свою часть вектора размера $\frac{2^{\text{countQubits}}}{\text{threads}}$ (где countQubits – количество кубитов, threads – количество потоков) из файла *inputFile*. Так как количество потоков – степень двойки, то на каждом потоке всегда будут части вектора одинакового размера.

Происходит однокубитное преобразование Адамара по нужному номеру кубита *numQubit*. При необходимости взаимодействия с частью вектора, находящейся на другом потоке происходит обмен данными между двумя потоками с помощью MPI_Isend() и MPI_Recv().

Получившийся после преобразования части вектора потоки параллельно выводят в файл *outputFile*.

2) Режим генерации вектора

Запуск:

```
make ; mpirun -np <threads> go generate <outputFile> <countQubits>
```

Потоки параллельно генерируют свою часть вектора размера $\frac{2^{\text{countQubits}}}{\text{threads}}$ (где countQubits – количество кубитов, threads – количество потоков)

Происходит нормирование вектора с помощью коллективных передач данных – MPI_Gather() и MPI_Bcast().

Получившийся после генерации части вектора потоки параллельно выводят в файл *outputFile*.

3) Режим тестирования

Запуск:

```
make ; mpirun -np <threads> go test <inputFile> <countQubits> <numQubit> <validFile>
```

Аналогично режиму чтения потоки параллельно читают вектор из файла *inputFile* и совершают преобразование по номеру кубита *numQubit*.

Аналогично происходит параллельное считывание вектора из файла *validFile* и происходит сравнение двух полученных векторов.

Запуск готового теста, указанного в постановке задачи:

```
make ; make test
```

На 8 потоках генерируется вектор для 16 кубитов.

Запускается программа в режиме чтения на 1 потоке.

Запускается программа в режиме тестирования на 2, 4 и 8 потоках.

Алгоритм повторяется для преобразований по 1, 2 и 16 кубитам.

Результаты выполнения.

Результаты запусков программы приведены в таблице ниже. В среднем наибольшее ускорение было получено на обработке 27 кубитов.

Исходный код программы: <https://github.com/gtorvald/prac/blob/master/main.cpp>

Количество кубитов	Номер кубита	Количество потоков	Время работы программы, с	Ускорение
25	1	1	4,949540	1,000000
		2	2,559920	1,933474
		4	1,265760	3,910331
		8	0,635988	7,782442
		16	0,347089	14,260147
		32	0,290967	17,010658
		64	0,214062	23,121993
	2	1	4,955900	1,000000
		2	2,456740	2,017267
		4	1,250050	3,964561
		8	0,636201	7,789834
		16	0,333632	14,854390
		32	0,271730	18,238325
		64	0,172711	28,694756
	25	1	5,444940	1,000000
		2	2,753320	1,977591
		4	1,344740	4,049065
		8	0,685816	7,939360
		16	0,354220	15,371633
		32	0,340984	15,968315
		64	0,230730	23,598752
26	1	1	9,813350	1,000000
		2	5,164480	1,900162

		4	2,600670	3,773393
		8	1,409970	6,959971
		16	0,791302	12,401523
		32	0,449666	21,823642
		64	0,266268	36,855161
	2	1	9,809360	1,000000
		2	5,042300	1,945414
		4	2,633450	3,724908
		8	1,289260	7,608520
		16	0,684999	14,320254
		32	0,477901	20,525925
		64	0,286531	34,234900
	26	1	10,740900	1,000000
		2	5,478070	1,960709
		4	2,958040	3,631087
		8	1,531000	7,015611
		16	0,958597	11,204813
		32	0,748893	14,342369
		64	0,402161	26,707960
27	1	1	19,593200	1,000000
		2	10,839000	1,807658
		4	5,588600	3,505923
		8	2,758710	7,102305
		16	1,604450	12,211786
		32	0,834076	23,490905
		64	0,481762	40,669874
	2	1	19,592500	1,000000
		2	10,451600	1,874593
		4	5,379400	3,642135
		8	2,732810	7,169360
		16	1,568220	12,493464
		32	1,035890	18,913688
		64	0,522646	37,487133
	27	1	21,515000	1,000000
		2	11,889200	1,809626
		4	6,255900	3,439153
		8	3,144670	6,841735
		16	1,629700	13,201816
		32	1,112760	19,334807
		64	0,741442	29,017779

График ускорения для 25 кубитов

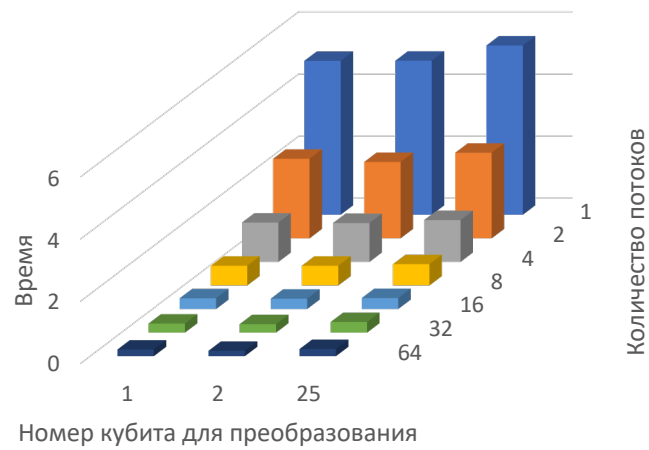


График ускорения для 26 кубитов

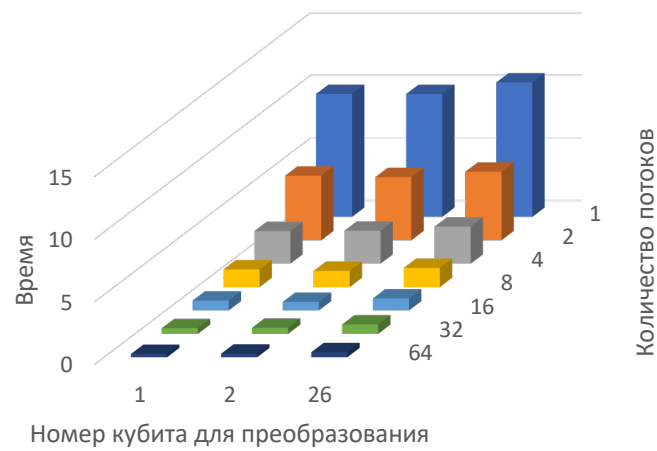
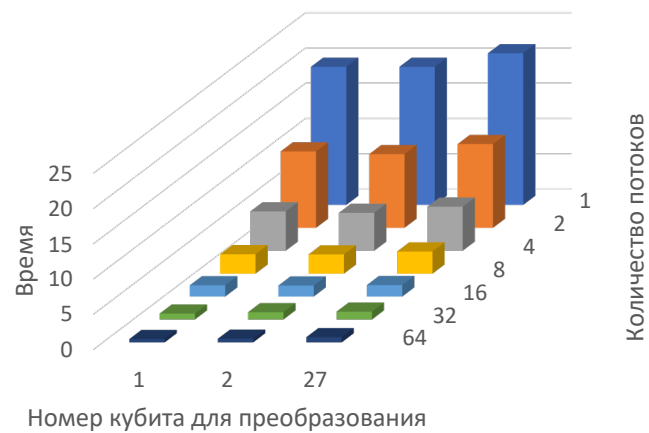


График ускорения для 27 кубитов



Вывод.

При росте числа кубитов наблюдается рост времени выполнения преобразования. Ускорение при $k = 1$ в среднем незначительно выше, чем при $k = 2$, и значительно выше, чем при $k = n$. Это можно объяснить как большим числом пересылок, так и локальностью использующихся каждым потоком данных в памяти.