

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ

ОТЧЕТ  
ЗАЩИЩЕН С ОЦЕНКОЙ  
РУКОВОДИТЕЛЬ

доцент, канд. техн. наук.

должность, уч. степень, звание

Д. С. Дехканбаев

подпись, дата

инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ

СОЗДАНИЕ ПАРАЛЛЕЛЬНОГО ПРИЛОЖЕНИЯ С  
ПОМОЩЬЮ MPI

по курсу: ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР.

4645М

номер группы

подпись, дата

Г. В. Гетманенко

инициалы, фамилия

# 1 Цель работы

Приобретение навыков написания параллельных программ. Знакомство с MPI.

# 2 Задание

Вариант 1: Написать программу, генерирующую множество точек, равномерно заполняющих круг. Точки необходимо сохранять в файл. Число точек задается пользователем.

# 3 Теоретические сведения

Генерацию точек, равномерно заполняющих круг, можно провести следующим алгоритмом:

1. Сгенерировать два множества чисел с равномерным распределением от нуля до единицы.

Первое множество  $A$  представляет угол, второе  $R$  - расстояния от центра круга.

2. Заменить каждое значение из  $R$  на квадратный корень из каждого значения:  $r'_i = \sqrt{r_i}$ .

Это равномерно распределит точки по кругу, задающиеся в полярной системе координат парами  $(\alpha_i, r_i)$  из множеств  $A, R$ .

3. Значения случайных углов  $A$  необходимо умножить на необходимый радиус круга, а значения  $R$  умножить на  $2\pi$ , чтобы привести к диапазону  $(0; 2\pi)$ .
4. Вычислить координаты точек в декартовой системе координат:  $x_i = \cos(\alpha_i) \cdot r_i$ ,  $y_i = \sin(\alpha_i) \cdot r_i$ .

# 4 Реализация программы

Поскольку при большом количестве единовременно генерируемых точек (речь о количестве порядка  $10^7$ ) количество потребляемой памяти может достигать нескольких сотен мегабайт, предлагается генерировать точки частями - чанками.

Как вариант реализации, для сохранения очередного чанка процессу необходимо:

1. Отправить чанк процессу с нулевым ранком (или оставить у себя, если используем отдельный доступ к файлу).
2. Создать представление сгенерированных точек в виде строки, которая будет сохранена в файл (форматирование).
3. Записать полученную строку в файл.

Альтернативно, поскольку форматирование тоже требовательный процесс, была добавлена реализация программы, в которой строка форматируется в том же процессе, в кото-

ром она и генерируется. То есть, первые два пункта из списка выше переставлены местами.

## 5 Проверка производительности

Программа написана на языке Python 3.6, поскольку автор работы решил, что знания по работе с MPI на Python в будущем пригодятся ему больше, чем C++. Исходный код программы можно посмотреть на GitHub:

<https://github.com/MrP4p3r/guap-mag-s3/tree/master/mpi>.

Для проверки производительности был написан небольшой скрипт, который можно так же увидеть на GitHub. Были запущены четыре теста на производительность: две реализации \* два режима работы с файлом. В каждом тесте запускалась программа, для каждого количества процессов (1-8), для каждого количества точек из множества (1, 10, ...,  $10^7$ ).

При выполнении программы замерялось время непосредственно с начала генерации точек всеми процессами, до конца работы алгоритма всеми процессами. Для этого была использована синхронизация через `MPI_Barrier`. Результаты тестов можно увидеть ниже.

```
$ # Сначала форматирование, пересылка, потом пишет RANK=0
$ ./performance-test.py ./generate-format-send-write.py
```

0		1	2	3	4	5	6 \
1	1	0.000180	0.000192	0.000208	0.000289	0.009132	0.018648
2	10	0.000201	0.000212	0.000211	0.000497	0.012638	0.019242
3	100	0.000251	0.000246	0.000282	0.000293	0.010503	0.026761
4	1000	0.000681	0.000692	0.000727	0.000738	0.014488	0.021568
5	10000	0.005804	0.003189	0.002575	0.002251	0.012188	0.027320
6	100000	0.057200	0.030123	0.021007	0.019733	0.044065	0.038634
7	1000000	0.555572	0.293506	0.202526	0.166174	0.295985	0.265177
8	10000000	5.858132	2.936803	2.050702	1.618638	2.534332	2.225968
9							
10		7	8				
11	1	0.020893	0.017642				
12	10	0.023246	0.023271				
13	100	0.024871	0.025983				
14	1000	0.013892	0.023676				
15	10000	0.015320	0.023497				
16	100000	0.044337	0.041279				
17	1000000	0.246127	0.212471				
18	10000000	2.424546	1.908181				
19							

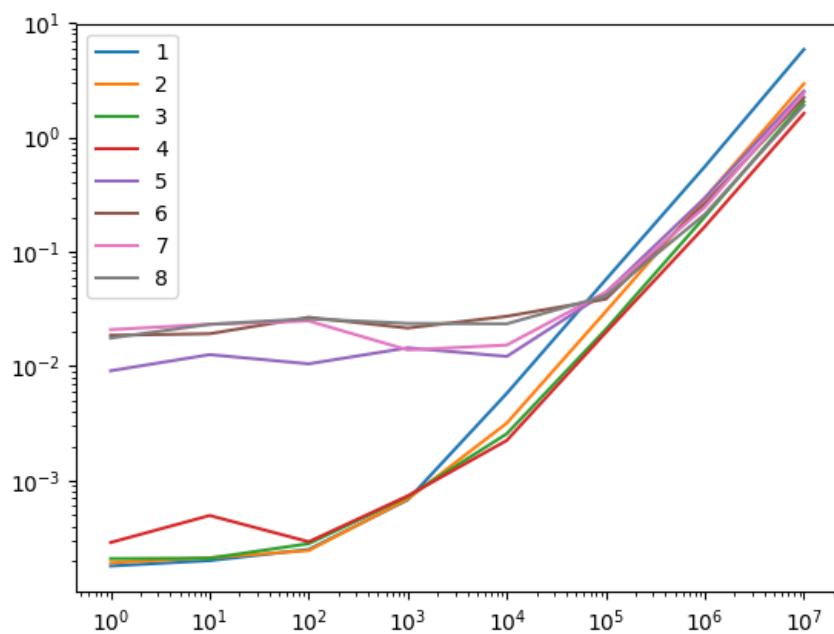


Рисунок 1

```
$ # Сначала форматирование, потом пишет каждый процесс
$ ./performance-test.py ./generate-format-send-write.py -d
```

0		1	2	3	4	5	6 \
1	1	0.000176	0.000218	0.000221	0.000272	0.007106	0.018271
2	10	0.000193	0.000222	0.000226	0.000269	0.006447	0.011831
3	100	0.000242	0.000267	0.000273	0.000353	0.005833	0.007554
4	1000	0.000680	0.000718	0.000713	0.000818	0.007048	0.011560
5	10000	0.005794	0.003190	0.002471	0.002122	0.014790	0.029612
6	100000	0.055500	0.029650	0.020274	0.016863	0.126486	0.196583
7	1000000	0.567695	0.289142	0.197455	0.167175	1.303104	1.996200
8	10000000	6.173603	2.943749	1.989582	1.925692	12.301004	20.944287
9							
10		7	8				
11	1	0.014855	0.018515				
12	10	0.019044	0.016094				
13	100	0.013275	0.014946				
14	1000	0.013600	0.019412				
15	10000	0.037118	0.041461				
16	100000	0.299406	0.260401				
17	1000000	2.918833	3.171790				
18	10000000	29.271140	32.466083				
19							

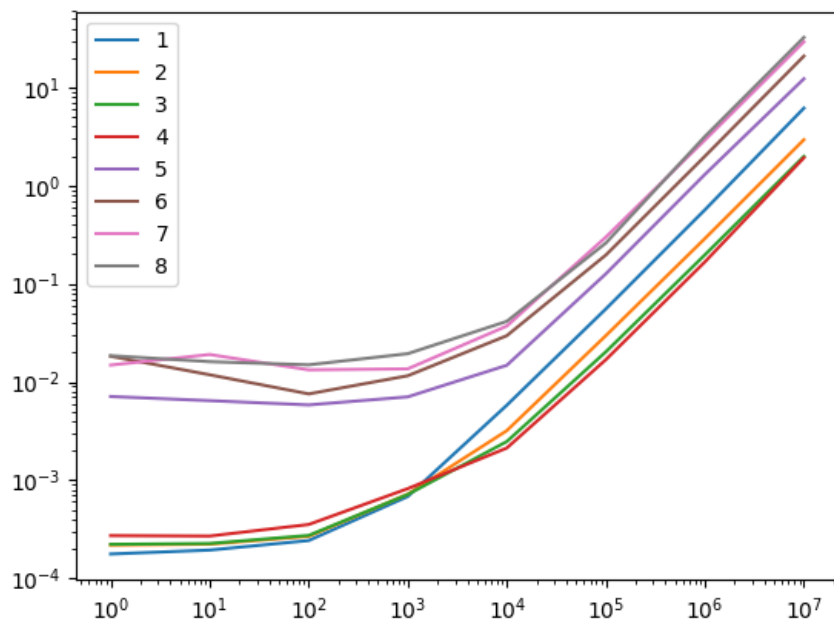


Рисунок 2

```
$ # Сначала пересылка, форматирование, пишет RANK=0
$ ./performance-test.py ./generate-send-format-write.py
```

0		1	2	3	4	5	6 \
1	1	0.000181	0.000206	0.000212	0.000254	0.011728	0.014591
2	10	0.000192	0.000229	0.000223	0.000261	0.009534	0.014517
3	100	0.000238	0.000269	0.000273	0.000315	0.007644	0.019250
4	1000	0.000676	0.000725	0.000737	0.000744	0.011184	0.017825
5	10000	0.005933	0.005069	0.005193	0.005303	0.023553	0.038142
6	100000	0.056163	0.047155	0.048904	0.049403	0.302447	0.320108
7	1000000	0.551898	0.467982	0.475336	0.510450	2.003478	3.809523
8	10000000	5.643379	4.712121	4.977518	5.544168	20.760523	39.713389
9							
10		7	8				
11	1	0.018708	0.019027				
12	10	0.018324	0.023690				
13	100	0.014893	0.020014				
14	1000	0.017196	0.027517				
15	10000	0.046624	0.058060				
16	100000	0.449752	0.575272				
17	1000000	4.293649	5.097090				
18	10000000	48.270933	53.365484				
19							

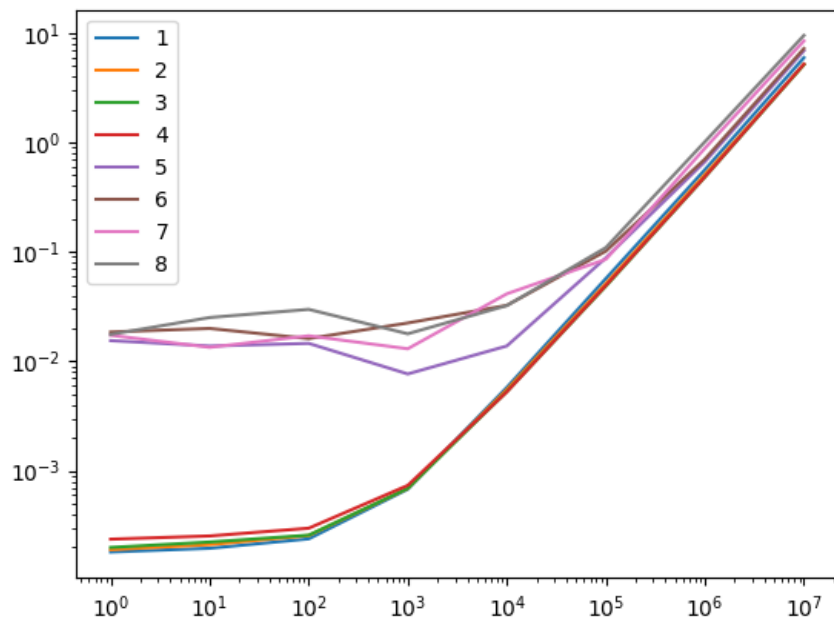


Рисунок 3

```
$ # Форматирование, пишет каждый
```

```
$ ./performance-test.py ./generate-send-format-write.py -d
```

0		1	2	3	4	5	6 \
1	1	0.000181	0.000230	0.000238	0.000268	0.004598	0.010045
2	10	0.000200	0.000237	0.000229	0.000271	0.018990	0.008613
3	100	0.000242	0.000284	0.000274	0.000303	0.003862	0.007525
4	1000	0.000695	0.000718	0.000713	0.000723	0.007551	0.009100
5	10000	0.005720	0.005170	0.005306	0.005598	0.020480	0.037886
6	100000	0.057217	0.048803	0.048638	0.054321	0.224306	0.213112
7	1000000	0.556103	0.484035	0.503715	0.511918	1.739508	2.922164
8	10000000	5.921293	4.716244	4.791783	5.651588	18.206571	27.601820
9							
10		7	8				
11	1	0.016686	0.014382				
12	10	0.005892	0.011865				
13	100	0.007333	0.016730				
14	1000	0.011376	0.017359				
15	10000	0.031934	0.053500				
16	100000	0.348627	0.345944				
17	1000000	3.192401	3.781406				
18	10000000	35.491777	35.149867				
19							

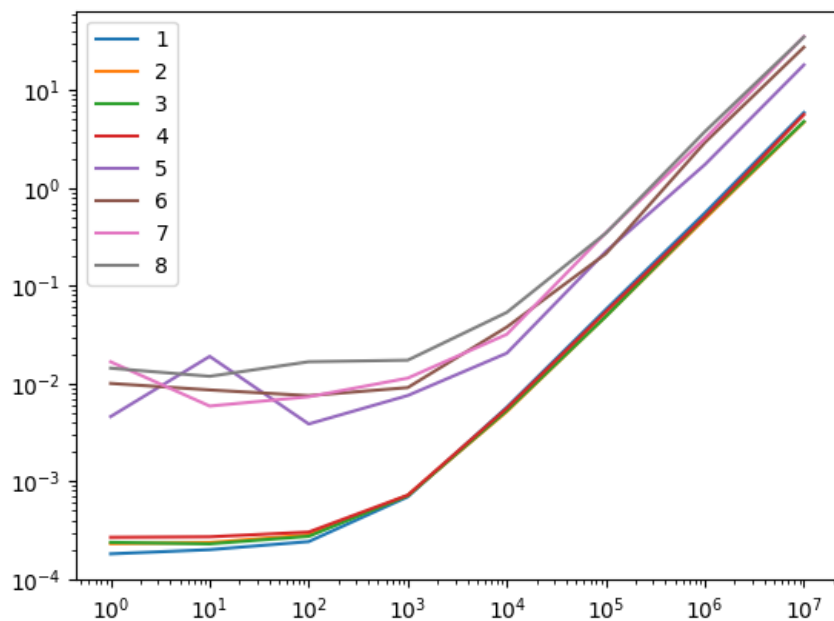


Рисунок 4

## 6 ВЫВОД

Самой эффективной реализацией программы оказался, что не удивительно, вариант с предварительным форматированием. Причем при записи в файл только из процесса

с RANK=0. Вариант с распределенной записью работает приблизительно на 30% дольше. Варианты с форматированием после коммуникации работают значительно медленнее.

Пологая часть графиков связана с размером чанка в 1024 точки. Если уменьшить размер чанка до, скажем, четырех точек, скорость выполнения программы значительно уменьшится, а форма графика станет более прямой. Варьируя размер чанка, можно попробовать найти золотую середину для большей скорости работы программы при незначительном потреблении памяти.