

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМ. М.В. ЛОМОНОСОВА

Механико-математический факультет

КУРСОВАЯ РАБОТА
**РЕАЛИЗАЦИЯ ПАРАЛЛЕЛЬНОЙ ЗАПИСИ БОЛЬШИХ ФАЙЛОВ
С ВЕЩЕСТВЕННЫМИ ЧИСЛАМИ В ТЕКСТОВОМ
ПРЕДСТАВЛЕНИИ**

Студент 4 курса:	Нагорных Я.В.
Научный руководитель:	Богачев К.Ю.

Москва

2018

Оглавление

Введение	2
1. Описание алгоритма	3
2. Результаты работы и ускорение	13
3. Заключение	20
Список литературы	21

Введение

Постановка проблемы. Печать больших массивов чисел всегда занимает много времени. Однако, не вся печать упирается в возможности диска, как это может показаться.

Кроме того, у печати данных мало ресурсов для ускорения.

Печать чисел с плавающей запятой также является проблемой, так как само значение числа и его экспоненту нельзя обрабатывать независимо.

Стандартный подход недостаточно точен и в некоторых случаях дает неверные результаты. Кроме того использование функций стандартных библиотек (`printf` и `sprintf`) достаточно затратно по времени.

Цели работы:

1. Ускорить печать больших массивов;
2. Использовать быстрые алгоритмы печати целых чисел и чисел с плавающей точкой.

Обзор предыдущих решений. ????

Возможные варианты улучшений. Мы уже обратили внимание на то, что стандартная функция преобразования буфера в строковый тип `sprintf (char *, const char *, ...)` работает крайне долго. Возникает идея применения более быстрых алгоритмов преобразования чисел в строки. Так например быстрое логарифмирование (разбиение числa на цифры) может заметно ускорить процесс.

Можно уменьшить число обращений к диску. Как известно, данные, отправленные на запись, накапливаются в памяти и записываются тогда, когда получен символ переноса строки или что-то в этом роде. Значит, можно отправлять не по одному числу на печать, а сразу готовым буффером.

Другой идеей для улучшения является использование многопоточного программирования. Из-за того что печать в файл должна быть строго последовательной, кажется, что ресурсов для распараллеливания немного (то есть нельзя разбить исходный массив на равные части и одновременно начать печать). Однако так как большая часть времени уходит на преобразование чисел в строки, то можно распараллелить именно ее. Непосредственно запись в сам файл упирается в возможности диска. Ее ускорить нельзя.

Также можно задуматься над улучшениями и оптимизировать сам алгоритм, несколько модифицировав вид выходного массива. За счет этого можно уменьшить размер полученного файла. Если будет встречаться подряд несколько одинаковых чисел, то можно не записывать в файл их все. Также, немного уменьшит размер файла отбрасывание ненужных нулей в конце записи числа (после точки).

1. Описание алгоритма

Схематично работа алгоритма показана на Рисунке 1.

1.1. Используемые структуры и классы

Структура `reduce_chunk`. В ней находится строковый буффер (готовый для печати) и его порядковый номер (`chunk_id`). Кроме

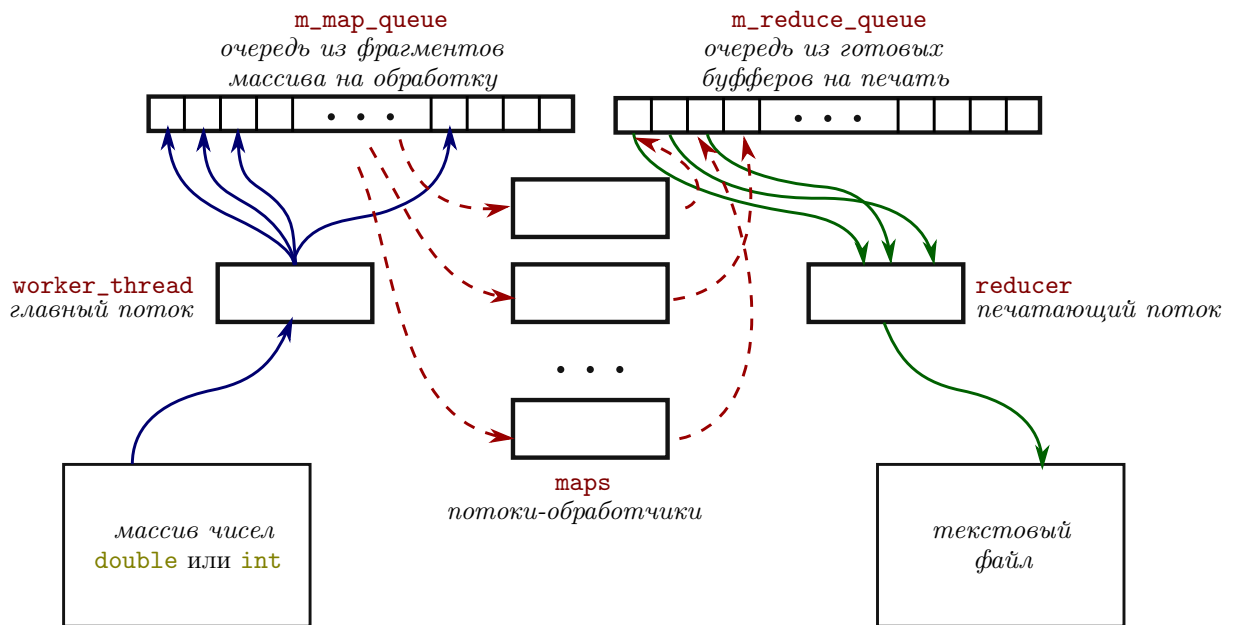


Рисунок 1. Работа потоков.

того, хранится флаг, является ли этот **reduce_chunk** последним.

Структура **map_chunk.** Этот тип состоит из лямбда-функции, которая должна обработать определенный фрагмент массива чисел, и элемента типа **reduce_chunk**, возвращаемый функцией.

Класс **mutex_wait_queue.** Это реализация *блокирующей очереди*, или *мьютексной очереди*. Под ней понимается очередь со следующим свойством: когда поток пытается прочесть что-то из пустой очереди, то он блокируется, до тех пор, пока какой-нибудь другой поток не положит в нее элемент. У этой очереди есть следующие методы:

- **dequeue** – достает верхний элемент из очереди, если очередь непустая. Иначе, поток, вызвавший этот метод блокируется. Также можно передать время блокировки, по истечении кото-

рого, поток разблокируется и вернется ни с чем;

- `dequeue_all` – аналогично `dequeue`, но достает все элементы, находящиеся в очереди, и складывает в указатель вектор из них;
- `enqueue` – кладет элемент в конец очереди.

Класс `worker_thread`. Это главный управляющий поток. Он хранит две блокирующие очереди `m_map_queue` и `m_reduce_queue`, состоящие из `map_chunk` и `reduce_chunk` соответственно. Зачем нужны такие очереди будет сказано позже.

1.2. Распределение задач

Управляющий (главный) поток `worker_thread` вызывает функцию `create_mappers()`, которая создает несколько потоков-обработчиков чисел, и функцию `create_reducer()`, которая создает печатающий поток. Сам управляющий поток будет складывать элементы типа `map_chunk` в очередь `m_map_queue`. Потоки-обработчики будут доставать из этой очереди `map_chunk`-и на обработку и конвертировать числа в буфферы, готовые для печати. Эти готовые буфферы `reduce_chunk` они будут складывать в другую очередь `m_reduce_queue`. Печатающий поток должен забирать все готовые буфферы из этой очереди и записывать их в правильном порядке в файл.

1.3. Преобразование чисел в строковый тип

Используемые обозначения.

Приведем используемые далее обозначения и понятия.

Вообще, число v с плавающей точкой в основании b представляется в памяти как

$$v = f_v \times b^{e_v},$$

где b обычно равно 2. f_v – целое значение или *мантисса*, а e_v – *показатель*.

Любая мантисса f может быть представлена как

$$f = \sum_{i=0}^{p-1} d_i \times b^i,$$

где $0 \leq d_i \leq b$. Числа d_i – *знаки числа*.

Будем называть число *нормированным*, если последний знак d_{p-1} отличен от нуля. Если экспонента имеет неограниченный диапазон, любое ненулевое число может быть нормализовано путем ”сдвига” знака влево при соответствующей корректировке экспоненты.

Так как не все вещественные числа могут быть представлены машиной, поговорим про округления. Как известно, числа с 5 на конце, могут округляться по-разному. Используем следующие обозначения:

- $[x]^\uparrow$ – округление вверх
- $[x]^\square$ – округление до ближайшего четного (то есть число 0.5 округляется до 0, а число 1.5 до 2)
- $[x]^\star$ – используется, когда неважно, как именно округлять.

- $\tilde{x} = [x]_p^s$ – округленное число до p знаков после запятой, а s – один из вышеизложенных способов округления.

$x = f \times b^e$ должно быть округлено до ближайшего $|\tilde{x} - x| \leq 0.5 \times b^e$, другими словами, до половины **ulp** (*unit in the last place*).

Для положительного числа $v = f_v \times b^{e_v}$ определим ближайшие числа к нему. v^- – предыдущее число для v , хранящееся в памяти. Аналогично v^+ – следующее число за v . Если v наименьшее, то $v^- = 0$. Если v наибольшее, то $v^+ = v + (v - v^-)$.

Введем понятие *границы* между двумя соседними числами. По определению это просто средние арифметические

$$m^- = \frac{v^- + v}{2} \quad \text{и} \quad m^+ = \frac{v + v^+}{2}.$$

Очевидно, что границы нельзя представить в виде чисел с плавающей точкой, так как они лежат между двумя соседними числами в памяти. Поэтому любое число w , такое что $m^- < w < m^+$ будет округлено до v . Если же w совпало с одной из границ будем округлять его до ближайшего четного.

Будем говорить, что представление R у числа с плавающей точкой v *удовлетворяет требованию*, если при чтении R будет представлено как v .

Определим тип **diy_fp** для x как беззнаковое целое число f_x , состоящее из q битов, и знакового целого числа e_x неограниченного диапазона. Значение x можно вычислить как $x = f_x \times 2^{e_x}$. Очевидно произведение двух таких типов отличается от обычного. Вычислять и обозначать его будем следующим образом:

$$x \otimes y := \left[\frac{f_x \times f_y}{2^q} \right]^\uparrow \times 2^{e_x + e_y + q}$$

В статье [1] описан алгоритм Grisu и его улучшения, также доказана их точность. Опишем кратко эти алгоритмы.

Grisu

Идея алгоритма. Предполагается, не умаляя общности, что у числа с плавающей точкой v отрицательный показатель. Тогда это число можно выразить как

$$v = \frac{f_v}{2^{-e_v}}$$

. Десятичные цифры v могут быть вычислены путем нахождения десятичного показателя t , для которого $1 \leq \frac{f_v \times 10^t}{2^{-e_v}} < 10$.

Первая цифра является целой частью этой дроби. Последующие цифры вычисляются путем повторного использования оставшейся дроби: нужно умножить числитель на 10 и взять целую часть от вновь полученной дроби.

Идея Grisu состоит в том, чтобы кешировать приблизительные значения $\frac{10^t}{2^{e_t}}$. Дорогих операций с большими числами не будет: они заменяются операциями с целыми числами фиксированного размера.

Кэш для всевозможных значений t и e_t может быть дорогостоящим. Из-за этого требования к кэш-памяти в Grisu упрощены. Кэш хранит только нормированные приближения с плавающей точкой всех соответствующих степеней десяти:

$$\tilde{c}_k := [10^k]_q^*,$$

где q – точность кэшированных чисел. Кэшированные числа сокращают большую часть экспоненты v , так что остается только небольшой показатель.

Процесс генерации цифр использует степени десяти с экспонентой $e_{\tilde{c}_t}$, близкой к e_v . Разница между двумя показателями будет небольшой.

Фактически, `Grisu` выбирает степени десяти так, что разница лежит в определенном диапазоне.

Реализация. Алгоритм `Grisu`:

- *Вход*: положительное число с плавающей точкой v точности p .
- *Условие*: точность `diy_fp` удовлетворяет $q \geq p + 2$, а кеш степеней десяти состоит из предварительно вычисленных нормированных округленных `diy_fp` со значениями $\tilde{c}_k := [10^k]_q^*$
- *Вывод*: строковое представление в основании 10 для V такое, что $[V]_p^\square = v$. То есть V должен быть округлен до v при чтении вновь.

Шаги алгоритма:

1. *Преобразование*: определим нормированный `diy_fp` w такой, что $w = v$.
2. *Кэширование степеней*: находим с заданной точностью

$$\tilde{c}_{-k} = f_c \times 2^{e_c}$$

такое, что $\alpha \leq e_c + e_w + q \leq \gamma$. (α и γ заданные заранее параметры, причем $\alpha + 3 \leq \gamma$.)

3. *Произведение*: пусть

$$\tilde{D} = f_D \times 2^{e_D} := w \otimes \tilde{c}_{-k}.$$

4. *Выход*: определим искомое

$$V := \tilde{D} \times 10^k.$$

Вычислим десятичное представление \tilde{D} , за которым следует строка **e** и десятичное представление k .

Поскольку значение `diy_fp` больше, чем значение входного числа, преобразование шага 1 дает точный результат. По определению `diy_fp`-ы имеют бесконечный диапазон экспоненциальности и показатель степени w , следовательно, достаточно велик для нормирования. Заметим, что показатель e_w удовлетворяет $e_w \leq e_v - (q - p)$.

Легко показать, что $\forall i \quad 0 < \tilde{e}_{c_i} - \tilde{e}_{c_{i-1}} \leq 4$, и поскольку кеш неограничен, требуемый \tilde{c}_{-k} должен находиться в кеше. Это является причиной первоначального требования $\gamma \geq \alpha + 3$.

Разумеется, бесконечный кеш не нужен.

Результатом `Grisu` является строка, содержащая десятичное представление \tilde{D} , за которым следуют символ **e** и k знаков. Таким образом, он представляет собой число $V := \tilde{D} \times 10^k$. Утверждается, что представление V у числа v удовлетворяет требованию с точностью p .

Grisu2

Но у `Grisu` есть недостаток: так число 1 будет напечатано в виде 1000000000000000000000000e-19. Поэтому будем использовать `Grisu2`.

Этот алгоритм является усовершенствованием предыдущего и не записывает лишние нули в конец числа. Так если целочисленный тип `diy_fp` содержит более двух дополнительных битов, то эти биты могут использоваться для сокращения выходной строки. В отличие от `Grisu`, `Grisu2` не генерирует полное десятичное представление, а просто возвращает цифры (123) и соответствующий показатель (-2). Затем процедура форматирования объединяет эти данные для получения представления в требуемом формате.

Идея алгоритма. Как описано выше, `Grisu2` использует дополнительные биты для создания более короткой выходной строки. Также `Grisu2` не будет работать с точными числами, а вместо этого будет вычислять аппроксимации m^- и m^+ . Чтобы избежать ошибочных результатов, которые не удовлетворяют требованиям, добавляется "безопасное пространство" (*safety margin*) вокруг приблизительных границ. Как следствие, `Grisu2` иногда может вернуть не самое оптимальное представление: оно может лежать вне нужных границ. Также это *safety-margin* требует изменения начального условия. Действительно, используя только 2 дополнительных бита, вычисление настолько неточно, что `Grisu2` может вернуть пустой интервал (?). Во избежание таких проблем добавляется третий дополнительный бит: $q \geq p + 3$.

Реализация. Алгоритм `Grisu2`:

- *Вход*: положительное число с плавающей точкой v точности p .
- *Условие*: точность `diy_fp` удовлетворяет $q \geq p + 3$, а кеш сте-

пеней десяти состоит из предварительно вычисленных нормированных округленных `diy_fp` значений $\tilde{c}_k := [10^k]_q^\star$

- *Вывод*: десятичные знаки d_i , где $0 \leq i \leq n$ и целочисленное K , такое что $V := d_0 \dots d_n \times 10^K$ удовлетворяет $[V]_p^\square = v$.

Шаги алгоритма:

1. *Границы*: вычисляем границы для v : m^- и m^+ .
2. *Преобразование*: определим `diy_fp` для w^+ так, что $w^+ = m^+$.
Определим также `diy_fp` для w^- так, что $w^- = m^-$ и $e_w^- = e_w^+$.
3. *Кэширование степеней*: находим с заданной точностью

$$\tilde{c}_{-k} = f_c \times 2^{e_c}$$

такое, что $\alpha \leq e_c + e_w + q \leq \gamma$.

4. *Произведение*: вычисляем

$$\tilde{M}^- := w^- \otimes \tilde{c}_{-k};$$

$$\tilde{M}^+ := w^+ \otimes \tilde{c}_{-k};$$

и пусть также

$$M_{\uparrow}^- := \tilde{M}^- + 1\text{ulp};$$

$$M_{\downarrow}^+ := \tilde{M}^+ - 1\text{ulp};$$

$$\delta := M_{\downarrow}^+ - M_{\uparrow}^-.$$

5. *Количество разрядов*: находим наибольшее κ такое, что M_{\downarrow}^+

$$\bmod 10^\kappa \leq \delta \text{ и определим } P := \left\lfloor \frac{M_{\downarrow}^+}{10^\kappa} \right\rfloor.$$

6. *Выход*: определим $V := P \times 10^{k+\kappa}$. Десятичные знаки d_i и n получены путем вычисления десятичного представления P . Положим $K := k + \kappa$ и возвращаем его с n знаками d_i .

Grisu2 не дает никаких гарантий относительно краткости результата. Его результатом является кратчайшее возможное число в интервале от $M_{\uparrow}^- \times 10^k$ до $M_{\downarrow}^+ \times 10^k$ включительно, где $M_{\uparrow}^- \times 10^k$ и $M_{\downarrow}^+ \times 10^k$ зависят от точности q для `diy_fp`. Чем больше q , тем ближе $M_{\uparrow}^- \times 10^k$ и $M_{\downarrow}^+ \times 10^k$ к фактическим границам m^- и m^+ .

Еще что-то

Будем использовать следующее улучшение. Пусть в массиве есть n подряд идущих одинаковых чисел с заданной точностью, то есть $\|x_i - x_{i+1}\| \leq \varepsilon$. В таком случае будем записывать их как `n*x`. Таким образом, если в нашем массиве много повторяющихся чисел, то выходная строка будет гораздо короче, а значит, можно сэкономить время работы программы и уменьшить размер выходного файла.

Чтобы ускорить преобразование целых чисел используется алгоритм SSE2, о котором подробнее написано в статье [2]. Суть алгоритма заключается в быстром логарифмировании числа по основанию 10.

2. Результаты работы и ускорение

Чтобы проверить эффективность работы вышеописанного алгоритма проведено сравнение с алгоритмом стандартной печати:

```

for (int i = 0; i < n;)
{
    for (int j = 0; j < m; j++, i++)
        fprintf (f, " %f", a[i]);
    fprintf (f, "\n");
}

```

Здесь n – размер массива, а m – количество чисел, записываемых в одну строку. Стандартная печать будет записывать числа с 16 знаками после запятой (машинная точность).

2.1. Тест 1 (массив случайных чисел)

Описанный в предыдущем разделе алгоритм запускался с разным числом потоков. Оба алгоритма запускались на одних и тех же массивах вещественных чисел, сгенерированных случайным образом. Этот тест полезен тем, что в реальных моделях данные могут задаваться каким-либо распределением (нормальным), где все числа являются вещественными и различными.

Время работы в секундах для обоих алгоритмов приведено в следующей таблице. Под числом потоков понимается число потоков-обработчиков. Таким образом реально задействовано на два потока больше (управляющий поток и печатающий).

Размер массива	Число потоков				Стандартная печать
	16	12	4	1	
10^7	0.609	0.550	0.880	3.196	4.256
	0.567	0.500	0.841	3.239	4.176
	0.506	0.473	0.802	3.052	4.188
$5 \cdot 10^7$	2.420	2.528	4.044	15.377	22.476
	2.522	2.446	4.273	16.309	21.116
	2.587	2.339	4.179	15.327	20.893
10^8	5.025	4.665	8.276	32.461	41.712
	4.787	4.630	7.970	30.571	41.785
	4.844	4.544	8.078	30.757	41.961
$5 \cdot 10^8$	21.199	20.074	37.515	148.548	201.941
	21.312	20.297	37.627	148.829	202.333
	21.231	20.171	37.686	149.217	201.692

Заметим, что время работы на шестнадцати потоках несколько больше, чем на двенадцати. Это обусловлено тем, что алгоритм запускался на машине с шестнадцатью ядрами. Таким образом в первом случае число потоков превосходило число ядер, и имели место накладные расходы.

Также стоит заметить, что отношение времени работы при увеличении количества потоков заметно уменьшается, и при увеличении размеров массива стремится к обратному отношению числа потоков. Тот факт, что эти отношения не строго равны объясняется сразу несколькими факторами. Во-первых, часть времени (хоть и небольшую при таких данных) занимала запись на диск. Во-вторых, не все потоки могли быть все время задействованными. Некоторые потоки

могли обращаться к пустой очереди и тем самым тратить время на ожидание.

Размер полученного файла, а также среднее ускорение работы алгоритма по сравнению со стандартной печатью приведены в следующей таблице:

Размер массива	Число потоков				Размер файла
	16	12	4	1	
10^7	7.50	8.20	5.00	1.33	245 Mb
$5 \cdot 10^7$	8.57	8.80	5.16	1.37	1.2 Gb
10^8	10.61	11.23	6.39	1.66	2.4 Gb
$5 \cdot 10^8$	9.51	10.01	5.37	1.36	12 Gb

Заметим, что при увеличении массива до определенного размера, ускорение возрастает, а затем спадает. Этот факт будет рассмотрен подробнее позднее.

2.2. Тест 2 (повторяющиеся числа)

Сгенерируем массив чисел из 0 и 1. В этом случае все последовательности одинаковых подряд идущих чисел будут сворачиваться в короткую строку вида $n \cdot x$.

Также с помощью этого теста во-первых можно проверить работу с целыми числами, а во-вторых убедиться в том, что Grisu2 отбрасывает ненужные нули.

Были сделаны замеры времени аналогично предыдущему тесту. Время работы в секундах приведено в таблице:

Размер массива	Число потоков				Стандартная печать
	16	12	4	1	
10^7	0.318	0.249	0.188	0.652	3.645
	0.334	0.256	0.190	0.629	3.622
	0.307	0.251	0.192	0.661	3.620
$5 \cdot 10^7$	1.657	1.274	0.884	3.183	18.412
	1.505	1.247	0.891	3.167	18.306
	1.522	1.262	0.894	3.175	18.261
10^8	3.105	2.441	1.726	6.306	36.194
	2.983	2.453	1.820	6.329	36.388
	3.246	2.505	1.759	6.339	36.419
$5 \cdot 10^8$	16.194	12.575	8.681	31.505	181.221
	16.414	12.564	8.787	31.291	181.406
	15.815	12.555	8.764	31.690	182.524

Среднее ускорение приведено в следующей таблице. Также приведен размер полученного файла со звездами и без звезд.

Размер массива	Число потоков				Размер файла
	16	12	4	1	
10^7	11.35	14.40	19.10	5.61	24 Mb / 187 Mb
$5 \cdot 10^7$	11.74	14.53	20.60	5.77	123 Mb / 936 Mb
10^8	11.68	14.73	20.55	5.74	245 Mb / 1.8 Gb
$5 \cdot 10^8$	11.26	14.46	20.78	5.77	1.2 Gb / 9.4 Gb

Если сравнить полученные эту таблицу с аналогичной в предыдущем пункте, то можно заметить, что ускорения выросли. Это произошло за счет того, что благодаря записи со звездами, заметно

уменьшился и размер выходного файла (что также видно из таблицы), а как следствие уменьшилось и время работы.

Тест 3 (огромные массивы)

Здесь как и в первом случае числа будут генерироваться случайным образом. Сравнивать будем стандартную печать и алгоритм, запущенный на 12 (+2) потоках.

Помимо обычного запуска, проведем и запуск с записью не на диск, а в *shared-memory*. Ранее говорилось, что скорость диска влияет на печать, но не всегда сильно. Засчет сравнения записи на диск и в *shared-memory* можно оценить, это влияние.

Далее на Рисунке 2 приведена зависимость времени работы от размера массива.

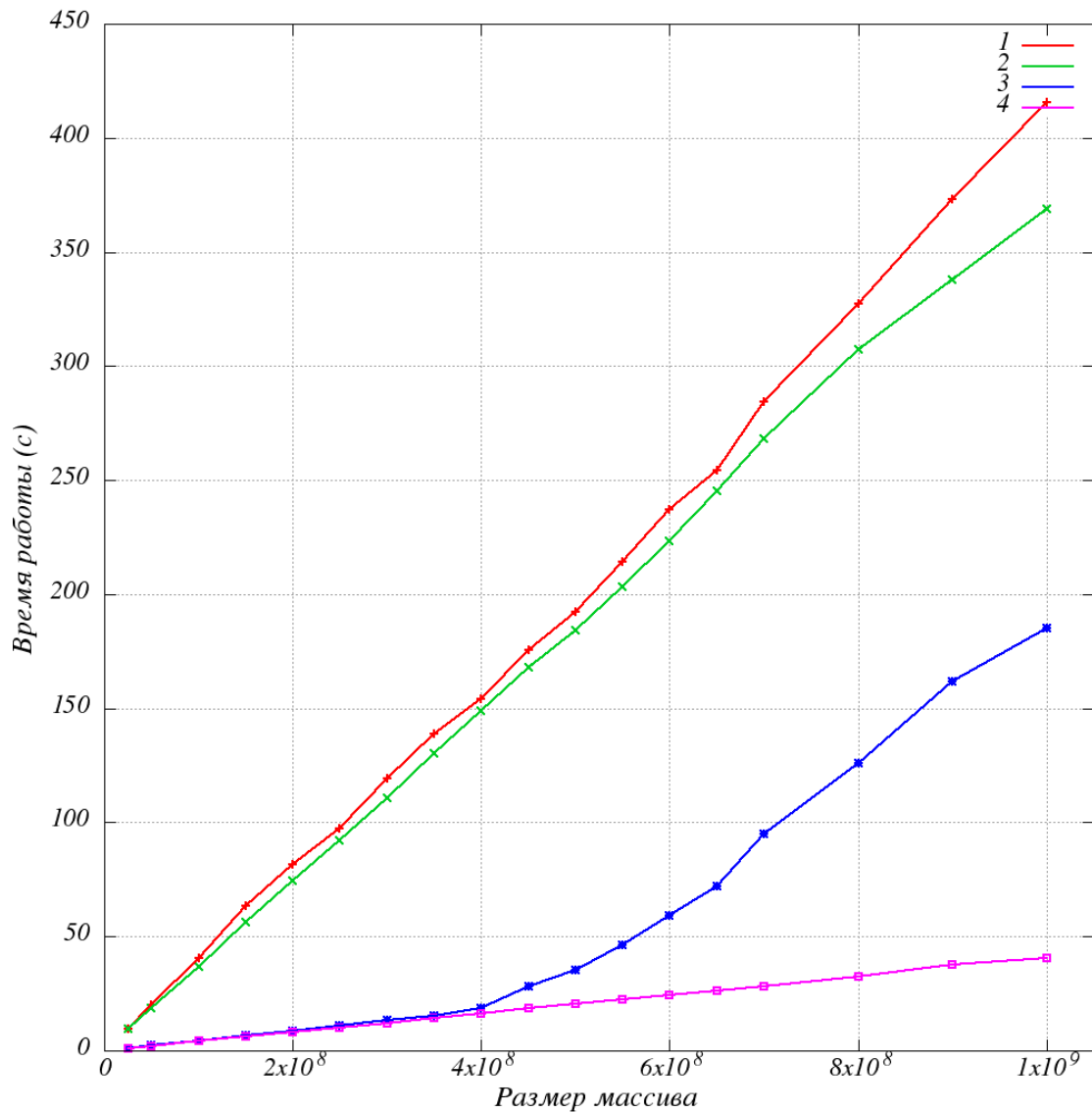


Рисунок 2. 1 – стандартная печать с записью на диск; 2 – стандартная печать с записью в *shared-memory*; 3 – алгоритм(?) с записью на диск; 4 – алгоритм с записью в *shared-memory*.

Сначала хочется заметить, что стандартная печать не сильно замедляется при записи в диск. Все время скорость работы диска была порядка 40–50 Mb/s. Диск фактически не оказывает никакого существенного влияния на работу.

Из графиков также видно, что при записи в *shared-memory* отношение времени работы стандартного алгоритма и ускоренного по-

стоянно (оба графика – прямые). Это значит, что ускорение одинакового на всех данных.

Однако, такого нельзя сказать в случае записи на диск. На графике в точке $4 \cdot 10^8$ (9.6 *Gb*) происходит излом: начинает ощущаться влияние диска (именно это мы и наблюдали и в Тесте 1 – ускорение немного упало при $5 \cdot 10^8$.) С этого момента печать начинает упираться в диск. При запуске тестов было замечено, что скорость записи на диск временами достигает 800-900 *Mb/s*.

2.3. Тестирование на реальных моделях

	Время работы	Время работы	Ускорение
COORD	0.028	0.003	9.33
ZCORN	0.135	0.013	10.38
COORD	0.132	0.010	13.2
ZCORN	12.819	0.982	13.05

3. Заключение

Список литературы

1. FLORIAN LOITSCH. Printing Floating-Point Numbers Quickly and Accurately with Integers, 2004.
2. WOJCIECH MUŁA. SSE: conversion integers to decimal representation, 2011.