Московский Государственный Университет
им. М. В. Ломоносова

Механико-математический факультет

Кафедра вычислительной математики

Курсовая работа

«Реализация библиотеки параллельной записи больших файлов с вещественными числами в текстовом представлении»

«Implementation of a parallel record library of large files with floating-point numbers in a text representation»

Выполнила

Я. В. Нагорных

Научный руководитель

д. ф.-м. н., доцент К. Ю. Богачев

Москва

2018

Оглавление

Введение

Постановка проблемы. Печать больших массивов чисел без округления с большой точностью 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 точкой 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 без потери точности;
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 числа 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. Нельзя разбить исходный массив на равные части и параллельно 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 параллельной записи показана на Рисунке 1.

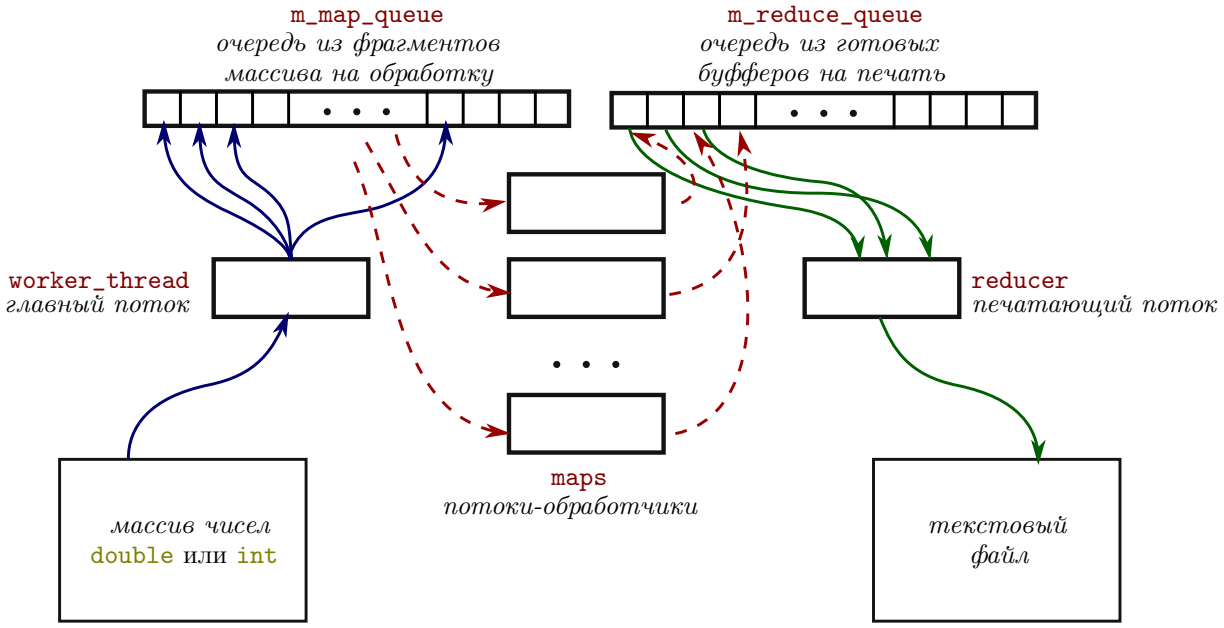

Рисунок 1. Работа потоков.

1.1. Используемые структуры и классы

Структура `reduce_chunk`**.** В ней находится строковый буфер, готовый для печати, и его порядковый номер `chunk_id`. Кроме того, хранится флаг, является ли этот `reduce_chunk` последним.

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

Класс `mutex_wait_queue`**.** Это упрощенная реализация *очереди сообщений*. Под ней понимается очередь со следующим свойством:

когда поток пытается прочитать 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 которого, поток разблокируется и вернется ни с чем;
- `dequeue_all` – аналогично `dequeue`, но достает все элементы, находящиеся в очереди, и складывает в переданный указатель вектор из них;
- `enqueue` – кладет элемент в конец очереди.

Класс `worker_thread`. Это главный управляющий поток. Он хранит две очереди сообщений `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, или главный, поток `worker_thread` вызывает функцию `create_mappers()`, которая создает несколько потоков-обработчиков, которые преобразовывают числа в строки, 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 буфе-

ры типа `reduce_chunk`, готовые для печати. Эти готовые буферы они будут складывать в другую очередь `m_reduce_queue`. Печатающий поток должен забирать все готовые буферы из этой очереди и записывать их в правильном порядке в файл.

1.3. Преобразование чисел в строковый тип

Мы уже говорили о том, что вместо стандартных функций `sprintf, strtod` и подобных им, будем использовать более быстрый алгоритм преобразования чисел, а именно алгоритм Grisu2.

Но прежде чем приступить к самому описанию этого алгоритма, сначала приведем используемые обозначения и кратко опишем его «предшественника» Grisu.

Используемые обозначения.

Приведем используемые далее обозначения и понятия.

Мы рассматриваем стандарт IEEE 754 – формат представления чисел с плавающей точкой. Знак числа хранится отдельно, поэтому далее мы будем рассматривать положительные числа, имея ввиду, что у отрицательных чисел «-» записан в отдельном бите.

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

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

где основание b в стандарте IEEE 754 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 $0 \leqslant d_i \leqslant b$. Числа d_i – *знаки числа.*

Будем называть число *нормированным*, если последний знак d_{p-1} отличен от нуля. Если экспонента может принимать любые неограниченные значения, то любое ненулевое число можно так отнормировать путем сдвига знака влево при соответствующей корректировке экспоненты.

В стандарте IEEE 754 представлены не все вещественные числа. Из-за этого, числа, которые не могут быть записаны через этот стандарт, будем округлять. Как известно, числа с 5 на конце, могут округляться по-разному. Используем следующие обозначения:

- $[x]^{\uparrow}$ – округление вверх;
- $[x]^{\square}$ – округление до ближайшего четного: например, 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.

Количественно определим ошибку $|\tilde{x} - x|$ следующим образом. Представим $\tilde{x}$ в виде $\tilde{x} = f \times b^e$, а f округлим до ближайшего такого, что $|\tilde{x} - x| \leqslant 0.5 \times b^e$, или другими словами, до половины единицы последнего разряда `ulp` (*unit in the last place*).

Для положительного числа $v = f_v \times b^{e_v}$ определим ближайшие в памяти 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* с максимальной точностью без округления, 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 $x = f_x \times 2^{e_x}$. Другими словами, строим отображение:

$$x \to (f_x, e_x), \qquad f_x \in \left(0, 2^{q-1}\right] \cap \mathbb{Z}, \quad e_x \in \mathbb{Z}, \quad \text{т.ч. } x = f_x \times 2^{e_x}$$

Введем произведение двух таких типов. Вычислять и обозначать его будем следующим образом:

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

Алгоритм Grisu

В статье [1] описан алгоритм Grisu и его улучшения, также доказана точность его работы. Опишем кратко этот алгоритм.

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 $1 \leqslant \dfrac{f_v \times 10^t}{2^{-e_v}} < 10$.

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 кешировании приблизительных значений дробей $\dfrac{10^t}{2^{e_t}}$. Дорогостоящих операций с большими числами не будет: они заменяются операциями с целыми числами фиксированного размера.

Кэш для всевозможных значений t и e_t может быть весьма затратным. Из-за этого требования к кеш-памяти в Grisu упрощены: кэш хранит только нормированные приближения с плавающей точкой всех соответствующих степеней десяти, а не самих дробей. Таким образом хранятся

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

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

В процессе вычисления знаков используются 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 $q \geqslant p + 2$, а кеш степеней десяти состоит из предварительно вычисленных нормированных округленных `diy_fp` со значениями $\tilde{c}_k := \left[10^k\right]_q^\star$
- *Вывод:* строковое представление в основании 10 для V такое, что $[V]_p^\square = v$. То есть при чтении числа V, оно должно быть округлено до v.

Шаги алгоритма:

1. *Преобразование:* определим нормированный `diy_fp` w такой, что $w = v$.
2. *Кэширование степеней:* сначала вычисляем

$$k = -\left\lceil \log_{10} 2^{\alpha - e + q - 1} \right\rceil,$$

а затем находим с заданной точностью

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

такое, что $\alpha \leqslant e_c + e_w + q \leqslant \gamma$. ($\alpha$ и γ заданные заранее параметры, причем $\alpha + 3 \leqslant \gamma$. Считаем $\alpha = 0$ и $\gamma = 3$).

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 в шаге 1 дает точный результат. По определению `diy_fp`-ы имеют бесконечный диапазон экспоненциальности, и следовательно, показатель степени w достаточно велик для нормирования. Заметим, что показатель e_w удовлетворяет $e_w \leqslant e_v - (q - p)$.

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

Алгоритм Grisu2

Но у Grisu есть недостаток: так число 1 будет напечатано в виде `10000000000000000000e-19` (т.к. согласно алгоритму $k = -19$). 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, или так называемых флагов, то эти флаги можно использовать для сокращения длины 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 флаги для создания более короткой выходной строчки. 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 так называемое безопасное пространство (*safety margin*) вокруг приблизительных границ. То есть увеличили диапазон, в котором, согласно алгоритму, может оказаться полученное число. Как следствие, Grisu2 иногда может вернуть не самое оптимальное представление: оно может лежать вне изначальных границ. Во избежание таких проблем добавляется третий дополнительный флаг: $q \geqslant p+3$.

Реализация. Алгоритм Grisu2:

- *Вход:* положительное число с плавающей точкой v точности p.
- *Условие:* точность `diy_fp` удовлетворяет $q \geqslant p + 3$, а кеш степеней десяти состоит из предварительно вычисленных нормированных округленных `diy_fp` значений $\tilde{c_k} := \left[10^k\right]_q^\star$
- *Вывод:* десятичные знаки d_i, где $0 \leqslant i \leqslant n$ и целочисленное K, такое что $V := d_0 \ldots d_n \times 10^K$ 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 $\alpha \leqslant e_c + e_w + q \leqslant \gamma$.

4. *Произведение:* вычисляем

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

и пусть также

$$M_\uparrow^- := \tilde{M}^- + 1\texttt{ulp};$$
$$M_\downarrow^+ := \tilde{M}^+ - 1\texttt{ulp};$$
$$\delta := M_\downarrow^+ - M_\uparrow^-.$$

5. *Количество разрядов:* находим наибольшее κ такое, что $M_{\downarrow}^{+} \mod 10^{\kappa} \leqslant \delta$ и определим $P := \left\lfloor \dfrac{M_{\downarrow}^{+}}{10^{\kappa}} \right\rfloor$.

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

Grisu2 не дает никаких гарантий относительно краткости длины 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^{+}.

Для наглядности на Рисунке 2 приведен пример работы Grisu2. В частности, видно, что алгоритм отбрасывает лишние нули в конце числа и что выбиарет более короткую запись числа. Иррациональное число Grisu2 напечатал с машинной точностью.

Улучшения для алгоритма

Будем использовать следующее улучшение. (*попробовать сравнивать diy_fp*) Пусть в массиве есть n подряд идущих одинаковых чисел с заданной точностью, то есть $\forall i : 0 \leqslant i \leqslant n$ верно, что $\|x_i - x_{i+1}\| \leqslant \varepsilon$, где ε – машинная точность. В таком случае сократим запись n чисел и вернем строку вида `n*x`. Таким образом, если в нашем массиве много повторяющихся чисел, то выходная строчка будет гораздо короче, а значит, можно ускорить программу и уменьшить размер выходного файла. Для этого также нужно модифицировать алгоритм чтения, чтобы он мог обрабатывать запись вида

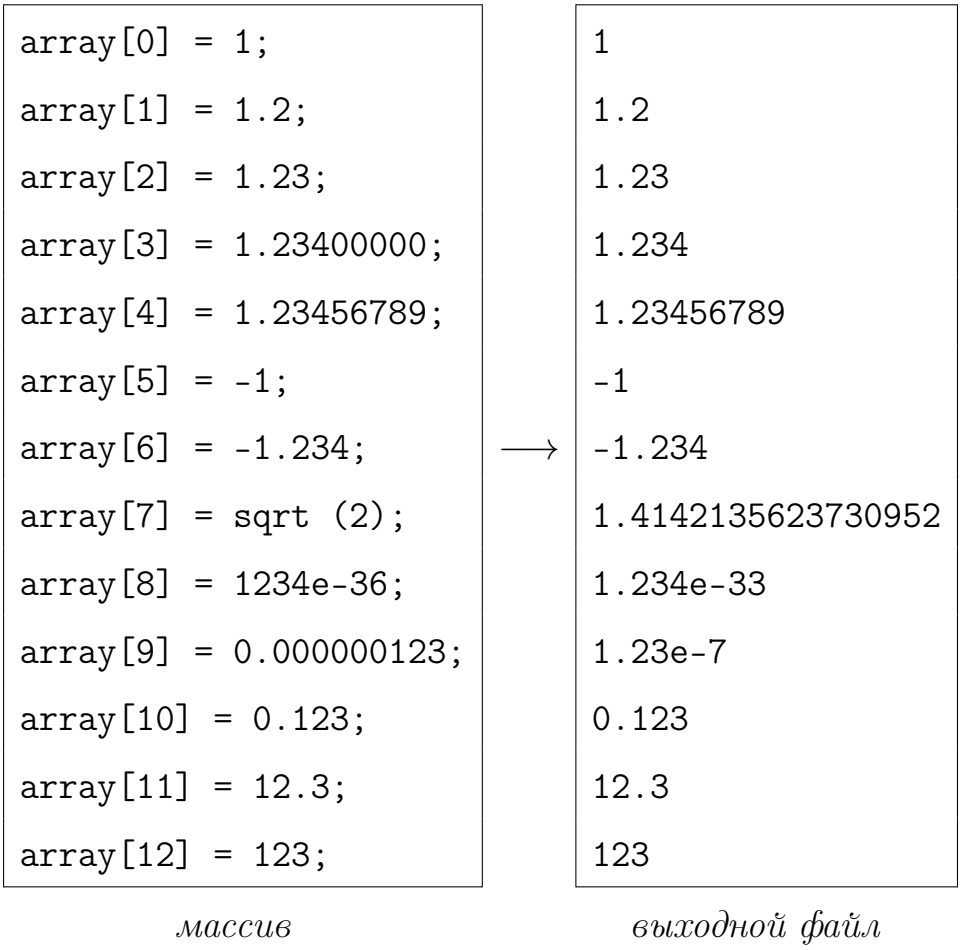

Рисунок 2. Полученный с помощью Grisu2, выходной файл для данного массива.

`n*x`.

Запись целых чисел n, о которых сказано выше, также можно ускорить. Для этого используется алгоритм 2 из статьи [2]. Суть алгоритма заключается в быстром логарифмировании числа по основанию 10.

2. Результаты работы и ускорение

Чтобы проверить эффективность работы алгоритма параллельной записи, описанного в 1, проведено сравнение с алгоритмом стандартной печати:

```
for (int i = 0; i < n;)
  {
    for (int j = 0; j < m; j++, i++)
      fprintf (f, '' %.16f'', 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.

Для всех дальнейших тестов было сделано следующее. Оба выходных файла, полученные работой параллельного алгоритма и стандартного, считывались вновь. Затем вычислялась разница между считанным числами. Во всех запусках погрешность не превышала машинной точности, что говорит о точности работы реализованного алгоритма.

2.1. Тест 1. Массив случайных чисел

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 (например, нормальным), где все числа

являются вещественными и различными.

Описанный в Разделе 1 алгоритм параллельной записи запускался с разным числом потоков.

Время работы в секундах для обоих алгоритмов приведено в Таблице 1. Под числом потоков понимается число потоков-обработчиков. Таким образом реально задействовано на два потока больше, так как помимо обработчиков есть еще управляющий поток и печатающий.

Размер массива	Число потоков				Стандартная печать	Размер файла
	12	8	4	1		
10^7	0.550	0.496	0.880	3.196	4.256	245 MB
	0.500	0.467	0.841	3.239	4.176	
	0.473	0.457	0.802	3.052	4.188	
$5 \cdot 10^7$	2.420	2.394	4.044	15.377	22.476	1.2 GB
	2.446	2.208	4.273	16.309	21.116	
	2.339	2.096	4.179	15.327	20.893	
10^8	4.665	4.320	8.276	32.461	41.712	2.4 GB
	4.630	4.302	7.970	30.571	41.785	
	4.544	4.362	8.078	30.757	41.961	
$5 \cdot 10^8$	38.074	38.178	49.150	188.914	201.941	12 GB
	38.297	38.562	50.089	189.082	202.333	
	38.171	37.730	49.168	188.521	201.692	

Таблица 1.

Заметим, что время работы на двенадцати потоках несколько больше, чем на восьми. Это обусловлено тем, что в первом случае не все потоки были загружены.

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.

Среднее ускорение работы алгоритма по сравнению со стандартной печатью приведены в следующей Таблице 2:

Размер массива	Число потоков			
	12	8	4	1
10^7	8.20	8.88	5.00	1.33
$5 \cdot 10^7$	8.95	9.79	5.16	1.37
10^8	9.06	9.23	5.16	1.34
$5 \cdot 10^8$	5.29	5.29	4.08	1.06

Таблица 2.

Заметим, что при увеличении массива до определенного размера, ускорение возрастает, а затем спадает. Первое объясняется тем, что с увеличением объема данных, потоки простаивают меньше. Второй факт будет рассмторен подробнее в пункте 2.4.

2.2. Тест 2. Целые числа

Этот тест позволит проверить, как обрабатывает наш алгоритм целые числа. Массив состоит из сгенерированных случайным обра-

зом целых чисел от 0 до 1000.

Размер массива	Число потоков				Стандартная печать
	16	12	4	1	
10^7	0.373	0.225	0.362	1.297	5.500
	0.287	0.215	0.350	1.284	5.445
	0.316	0.213	0.337	1.266	5.512
$5 \cdot 10^7$	1.577	1.052	1.664	6.362	27.934
	1.564	1.069	1.675	6.375	29.461
	1.859	1.071	1.662	6.490	29.431
10^8	3.457	2.165	3.274	12.614	55.044
	3.018	2.245	3.309	12.632	55.703
	3.312	2.229	3.548	13.668	56.312
$5 \cdot 10^8$	15.865	10.824	16.682	62.944	283.614
	17.116	10.910	16.683	64.088	290.702
	16.558	10.155	16.452	64.128	287.546

Таблица 3.

Размер массива	Число потоков				Размер файла
	16	12	4	1	
10^7	16.86	25.20	15.68	4.27	56 MB / 205 MB
$5 \cdot 10^7$	17.36	27.20	17.36	4.51	295 MB / 1 GB
10^8	17.06	25.16	16.49	4.29	590 MB / 2 GB
$5 \cdot 10^8$	17.39	27.02	17.30	4.51	2.9 GB / 10 GB

Таблица 4.

Заметим, что размер файла, полученного с помощью нового алгоритма гораздо меньше размера файла, полученного стандартной пе-

чатью, так как отброшены лишние нули.

2.3. Тест 3. Повторяющиеся числа

Сгенереруем массив чисел из 0 и 1. В этом случае все последовательности одинаковых подряд идущих чисел будут сворачиваться в короткую строку вида `n*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 3:

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

Таблица 5.

Среднее ускорение приведено в Таблице 6. Также приведен размер файла "со звездами", полученным быстрым алгоритмом, и размер файла "без звезд", полученного алгоритмом стандартной печати.

Размер	Число потоков				Размер
массива	16	12	4	1	файла
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

Таблица 6.

Если сравнить эту таблицу с аналогичной в предыдущем пункте, то можно заметить, что ускорения возросли. Это произошло засчет того, что благодаря записи со звездами, заметно уменьшился и размер выходного файла, а как следствие уменьшилось и время работы.

2.4. Тест 4. 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 разделяемую память *shared-memory*. Как известно, разделяемая память является самым быстром средством обмена данными между проессами.

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 3 приведена зависимость времени работы от размера массива.

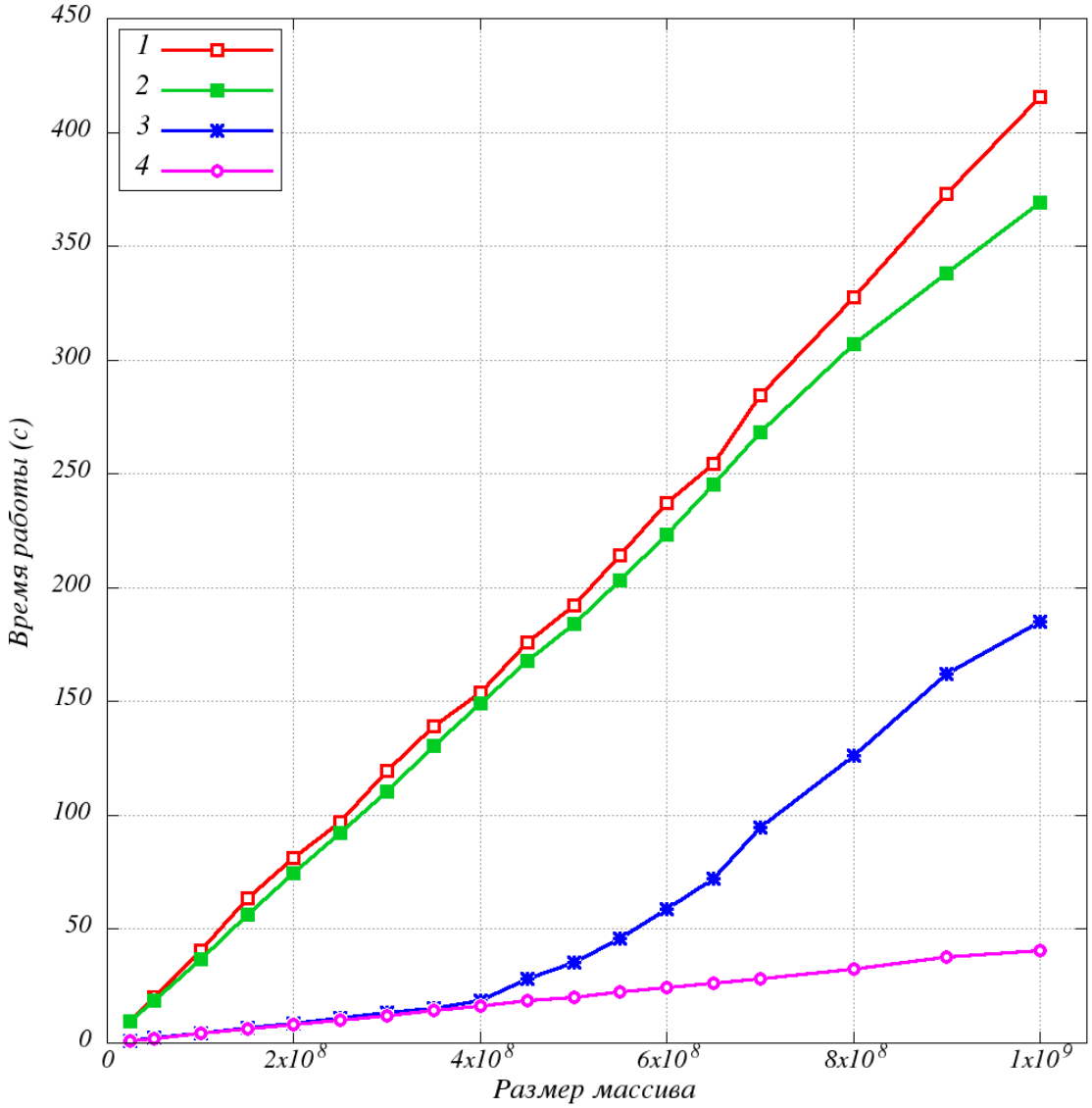

Рисунок 3. 1 – стандартная печать с записью на диск; 2 – стандартная печать с записью в разделяемую память; 3 – алгоритм параллельной печати с записью на диск; 4 – алгоритм параллельной печати с записью в разделяемую память.

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 постоянно, так как оба графика – прямые. Это значит, что ускорение одинакого 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 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. Из-за слишком больших файлов (так файл при размере массива 10^9 достигает 24 GB) создается очередь из буфферов на печать. Потоки-обработчики обрабатывают буфферы быстрее, чем производится сама печать. Разница между третьим и четвертым графиком – накладные расходы на работу диска.

2.5. 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

В результате написания курсовой работы была решена поставленная задача: реализована библиотека параллельной записи массивов вещественных чисел.

В ходе тестирования была проверена точность работы реализованного алгоритма, а также измерено ускорение в сравнении со стандартнйо функцией печати.

Написанная на языке C++ подпрограмма была внедрена в промышленный гидродинамический симулятор tNavigator.

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.
3. Богачев К.Ю.. Основы параллельного программирования. – М.: Бином. Лаборатория знаний, 2010.
4. David Goldberg. What every computer scientist should know about floating-point arithmetic. – ACM Computing Surveys, 23(1): 5–48, 1991.