УНИВЕРСИТЕТ ИТМО

Лабораторная работа №2 "Исследование эффективности параллельных библиотек для C-программ"

Дисциплина

Параллельные вычисления

Автор

Дмитрий Рачковский

24 ноября 2020 г.

Содержание

1 Введение

1.1 Цель работы

В данной работе необходимо исследовать эффективность распараллеливания программ на языке C с использованием параллельных библиотек. В качестве паралленьной библиотеки была использована AMD Framewave. Для этого одна и та же программа, производящая достаточное количество вычислений с использованием вышеуказанной библиотеки, запускается несколько раз с различным количеством потоков. Производительность полученных выполняемых файлов сравнивается. Также производится контроль результата вычислений, который не должен меняться (в пределах допустимой погрешности) при распараллеливании, гарантируя правильность выполнения.

1.2 Использованное оборудование

Для проведения экспериментов использовалась виртуальная машина с OS Debian. Виртуальная машина получила доступ к 6 ядрам (используется 64-битный процессор Intel Core i7-8750H @ 2.20 GHz с 6 физическими и 12 логическими ядрами) и 6 ГБ оперативной памяти (из 16, доступных в системе). Во время выполнения экспериментов система была подключена к источнику питания.

Версия использованного компилятора GCC: gcc (Debian 6.3.0-18+deb9u1) 6.3.0 20170516

Использованная параллельная библиотека: AMD Framewave 1.3.1

2 Ход работы

2.1 Установка библиотеки

Для эксериментов была использована библиотека AMD Framewave. Библиотека была загружена с официального сайта и распакована в директорию */opt/FW_1.3.1_Lin64*. Была создана переменная среды, указывающая на эту директорию:

```
FW_HOME=/opt/FW_1.3.1_Lin64
```

Для корректного подключения динамических библотек были созданы переменная среды

```
LD_LIBRARY_PATH=$FW_HOME/lib
```

Также для каждого файла в папке *FW_HOME/lib* был создан симлинк в той же папке, отбрасывающий 1.3.1 с конца имени файла. Для корректной работы библиотеки при компиляции (и линковании) были использованы флаги

```
-I ${FW_HOME} -L ${FW_HOME}/lib
```

Были подключены две библиотеки FW, необходимые для корректной работы кода. Финальная команда, использованная для компиляции:

```
gcc -O3 -Wall -Werror -I ${FW_HOME} -L ${FW_HOME}/lib
-lm -lfwBase -lfwSignal -o lab2 lab2.c
```

2.2 Используемая программа

Для проведения экспериментов была программа, использованная в лабораторной работе №1, была модифицирована следующим образом:

- На этапах Map и Merge циклы были заменены вызовами соответствующих функций из библиотеки AMD Framewave.
- В начало программы была добавлена функция, устанавливающая количество используемых потоков $fwSetNumThreads(M)$. Параметр M передаётся при вызове исполняемого файла аргументом командной строки.

Полный текст программы может быть найден в приложении 4.1.

2.3 Получение данных

Полученная программа компилируется без использования автоматического распараллеливания с использованием команды

```
gcc -O3 -Wall -Werror -I ${FW_HOME} -L ${FW_HOME}/lib
-lm -lfwBase -lfwSignal -o lab2 lab2.c
```

Полученная программа запускается со значениями $N = N1, N1 + \Delta, N1 + 2\Delta, N1 + 3\Delta ... N2$, найденными лабораторной работе №1, и значениями $M = 1, 2, 3, 4, 5, 6$, так как тестовая система располагает шестью ядрами.

2.4 Проверка результата

Перед тем, как анализировать полученные данные, нужно убедиться, что распараллеливание прошло правильно и результат не был искажен. Для этого нужно обратить внимание на значения X, которые были получены в ходе работы программы. Эти значения не должны отличаться для одних и тех же значений N при различных значениях M.

Действительно, при разных значениях M значения X не отличаются. На представленном графике видно, что число X остаётся одним и тем же для всех значений M при одном и том же значении N:

	1	2	3	4	5	6
N=400	566,93834	566,93834	566,93834	566,93834	566,93834	566,93834
N=1510	3147,26843	3147,26843	3147,26843	3147,26843	3147,26843	3147,26843
N=2620	4986,88001	4986,88001	4986,88001	4986,88001	4986,88001	4986,88001
N=3730	7375,76884	7375,76884	7375,76884	7375,76884	7375,76884	7375,76884
N=4840	9365,05820	9365,05820	9365,05820	9365,05820	9365,05820	9365,05820
N=5950	11440,20749	11440,20749	11440,20749	11440,20749	11440,20749	11440,20749
N=7060	13472,16641	13472,16641	13472,16641	13472,16641	13472,16641	13472,16641
N=8170	15797,28768	15797,28768	15797,28768	15797,28768	15797,28768	15797,28768
N=9280	17985,46403	17985,46403	17985,46403	17985,46403	17985,46403	17985,46403
N=10390	19518,15739	19518,15739	19518,15739	19518,15739	19518,15739	19518,15739
N=11500	21878,87191	21878,87191	21878,87191	21878,87191	21878,87191	21878,87191

Рис. 1: Значения X при различных значениях N и M

2.5 Анализ результатов

2.5.1 Время выполнения

По выполнению экспериментов я заметил, что график времени выполнения программ имеет почти одну и ту же форму всех значении N. Это может быть замечено при визуальном сравнении графиков:

Рис. 2: Время работы программы при раздичных значениях N и M

Полноразмерные графики, а также таблица значений приведены в приложении 4.3

Как можно заметить, уменьшение скорости выполнения по сравнению с результатами первой лабораторной составляет в среднем 43%. При этом увеличение количества потоков от 1 до 3 не влечёт практически никакого эффекта. Следующее улучшение во времени выполнения, в среднем на 15%, наблюдается при увеличении числа потоков до 5 для всех значений N, кроме $N = 11500$ - улучшение происходит уже при переходе на 4 потока. Улучшение времени выполнения в процентах можно наблюдать в следующей таблице:

	Изменение значения M					
	N/A -> 1	1 -> 2	2 -> 3	3 -> 4	4 -> 5	5 -> 6
N=400	37,50%	0,00%	0,00%	0,00%	20,00%	0,00%
N=1510	46,43%	-6,67%	0,00%	0,00%	18,75%	0,00%
N=2620	41,73%	2,47%	1,27%	1,28%	16,88%	0,00%
N=3730	42,06%	2,74%	0,00%	0,00%	17,61%	0,85%
N=4840	44,31%	-0,44%	0,00%	-0,88%	17,54%	1,06%
N=5950	41,20%	4,65%	0,00%	0,00%	17,38%	0,37%
N=7060	41,31%	5,88%	-0,45%	0,00%	13,33%	5,13%
N=8170	41,75%	5,02%	-0,85%	0,51%	15,11%	2,80%
N=9280	44,80%	0,27%	0,13%	-0,40%	16,84%	0,64%
N=10390	45,77%	0,54%	-0,32%	0,54%	15,93%	1,42%
N=11500	45,93%	0,27%	-0,36%	16,99%	0,11%	0,43%
Ср. знач.	42,98%	1,34%	-0,05%	1,64%	15,41%	1,16%

Рис. 3: Улучшение времени выполнения при увеличении числа потоков

Примечательно, что библиотека Framewave не позволила мне установить количество потоков больше, чем 6, то есть больше количества ядер, доступных системе. Это может быть проиллюстрировано добавлением следующих строк в код программы:

```
printf("Setting number of threads to %d\n", num_of_threads);
FwStatus st = fwSetNumThreads(num_of_threads);
printf("%s\n", fwGetStatusString(st));
printf("Current number of threads: %d\n", fwGetNumThreads());
```

Программа позволяет установить от 1 до 6 потоков:

```
vagrant@stretch:~/synced/lab_2$ ./lab2 11500 3
Setting number of threads to 3
No error detected
Current number of threads: 3
```

Рис. 4: Установление трёх потоков успешно

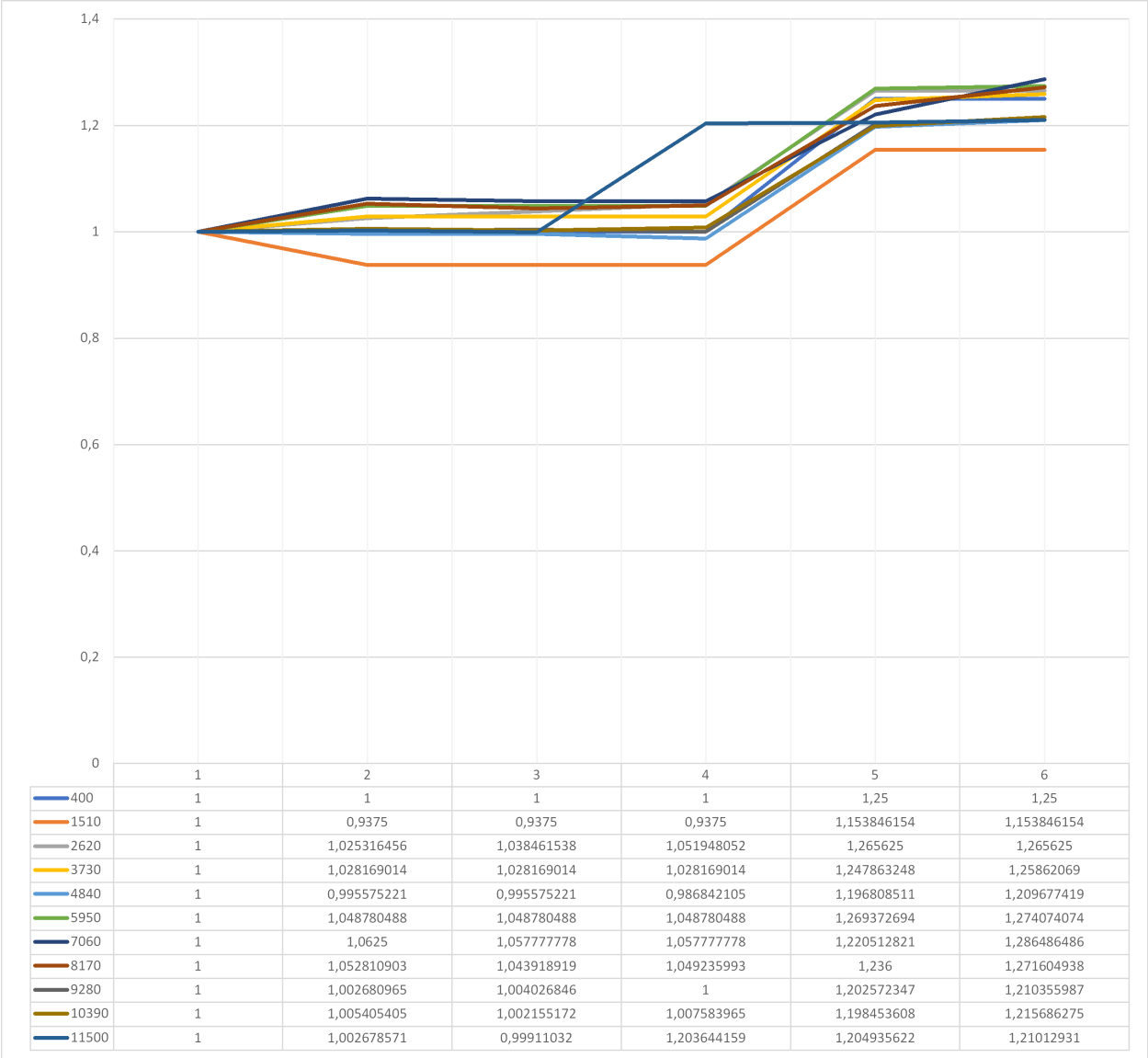
При этом, при попытке установить больше 6 потоков количество потоков остаётся равным 6, и сообщения об ошибке отсутствуют:

```
vagrant@stretch:~/synced/lab_2$ ./lab2 11500 10
Setting number of threads to 10
No error detected
Current number of threads: 6
```

Рис. 5: Попытка установления 10 потоков устанавливает только 6

2.5.2 Параллельное ускорение

Так как количество работы программы при каждом значении N не отличается, величина параллельного ускорения находится по формуле $S(p)_{w=const} = \frac{t(1)}{t(p)}$.



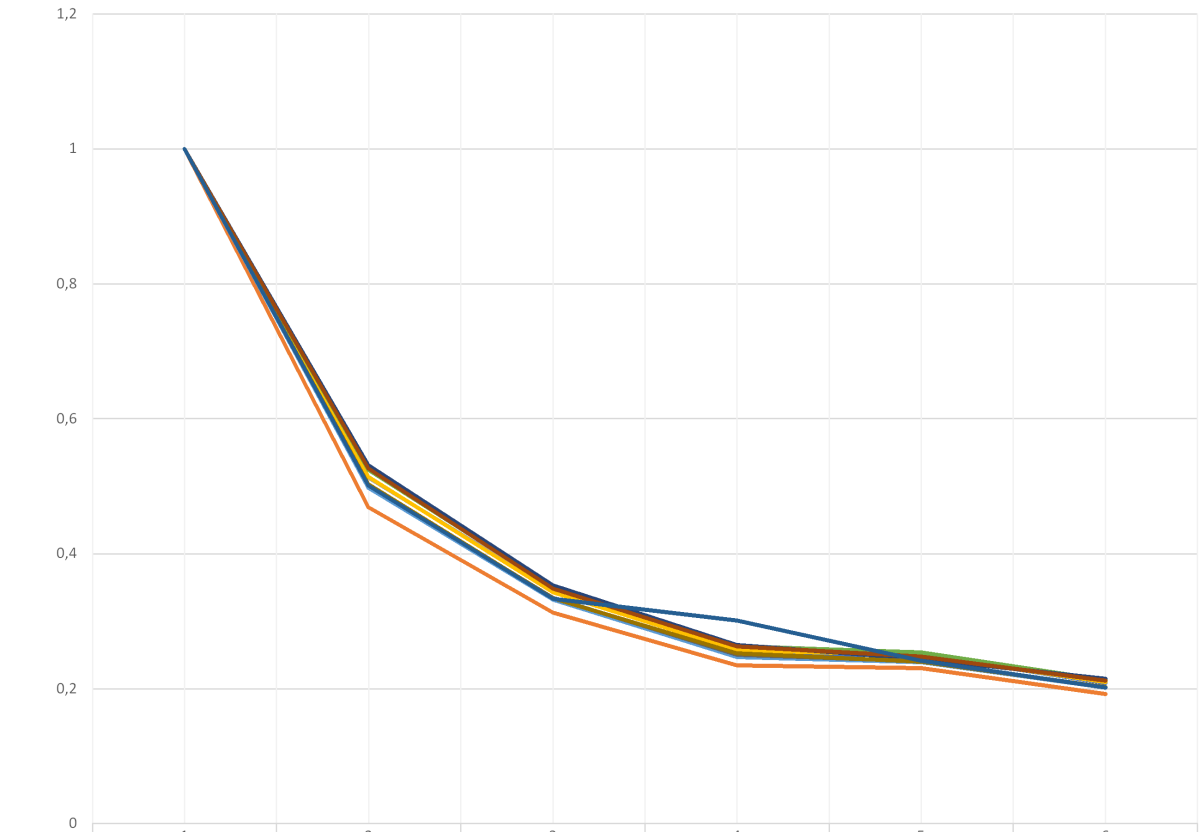
	1	2	3	4	5	6
400	1	1	1	1	1,25	1,25
1510	1	0,9375	0,9375	0,9375	1,153846154	1,153846154
2620	1	1,025316456	1,038461538	1,051948052	1,265625	1,265625
3730	1	1,028169014	1,028169014	1,028169014	1,247863248	1,25862069
4840	1	0,995575221	0,995575221	0,986842105	1,196808511	1,209677419
5950	1	1,048780488	1,048780488	1,048780488	1,269372694	1,274074074
7060	1	1,0625	1,057777778	1,057777778	1,220512821	1,286486486
8170	1	1,052810903	1,043918919	1,049235993	1,236	1,271604938
9280	1	1,002680965	1,004026846	1	1,202572347	1,210355987
10390	1	1,005405405	1,002155172	1,007583965	1,198453608	1,215686275
11500	1	1,002678571	0,99911032	1,203644159	1,204935622	1,21012931

Рис. 6: Параллельное ускорение распараллеленых программ

Можно заметить, что параллельное ускорение увеличивается при переходе от 4 потоков к 5 (или от 3 потоков к 4 в случае $N = 11500$), что также соотносится с уменьшением времени работы программы.

2.5.3 Параллельная эффективность

Параллельная эффективность, как и в первой лабораторной работе, падает с увеличением числа потоков.



	1	2	3	4	5	6
400	1	0,5	0,333333333	0,25	0,25	0,208333333
1510	1	0,46875	0,3125	0,234375	0,230769231	0,192307692
2620	1	0,512658228	0,346153846	0,262987013	0,253125	0,2109375
3730	1	0,514084507	0,342723005	0,257042254	0,24957265	0,209770115
4840	1	0,497787611	0,331858407	0,246710526	0,239361702	0,201612903
5950	1	0,524390244	0,349593496	0,262195122	0,253874539	0,212345679
7060	1	0,53125	0,352592593	0,264444444	0,244102564	0,214414414
8170	1	0,526405451	0,347972973	0,262308998	0,2472	0,211934156
9280	1	0,501340483	0,334675615	0,25	0,240514469	0,201725998
10390	1	0,502702703	0,334051724	0,251895991	0,239690722	0,202614379
11500	1	0,501339286	0,333036773	0,30091104	0,240987124	0,201688218

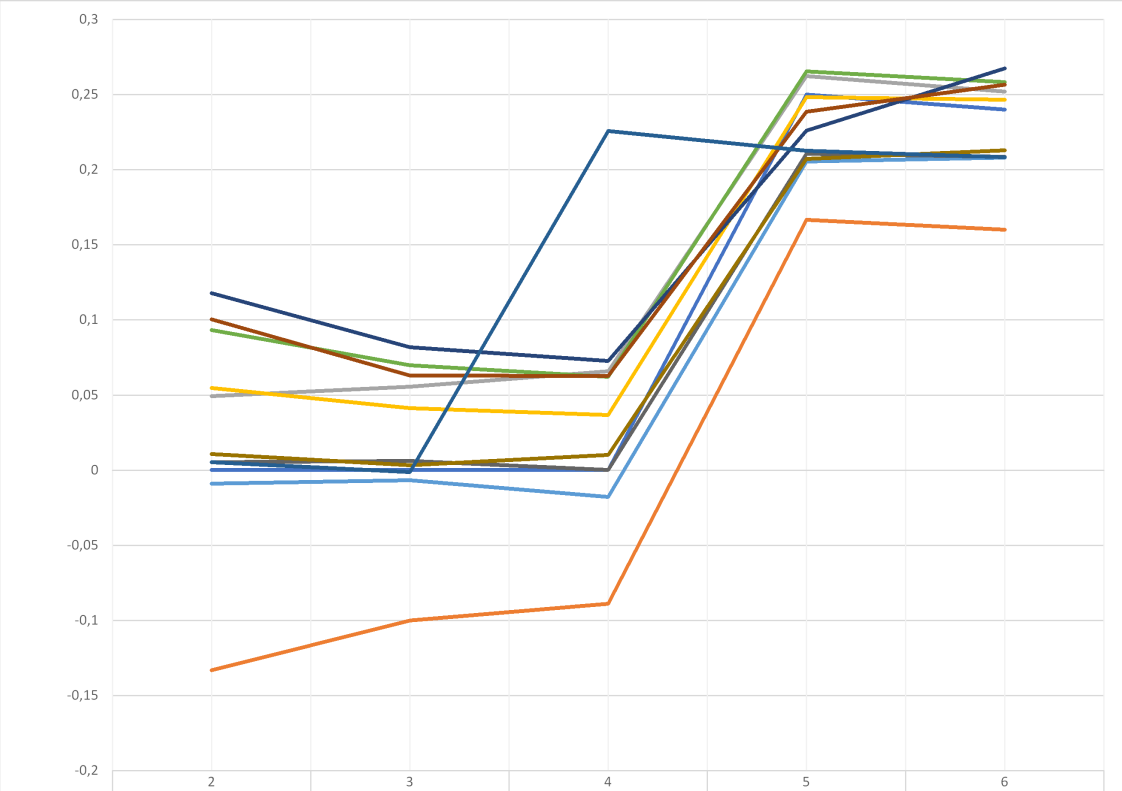
Рис. 7: Параллельная эффективность распараллеленых программ

При переходе на $2, 3, 4, 5$ и 6 потоков величина параллельной эффктивности падает, в среднем, на $49.2\%, 33.3\%, 23.5\%, 5.12\%$ и 15.6% соответственно.

Как можно заметить, при переходе от 4 потоков к 5 (или от 3 потоков к 4 в случае $N = 11500$) параллельная эффективность падает незначительно, что также соотносится с уменьшением времени работы программы и увеличением величины параллельного ускорения.

2.5.4 Доля распараллеленных команд

Величина параллельного ускорения может быть рассчитана по закону Амдала с использованием значения k, представляющего долю распараллеленных команд: $S(p)_{w=const} = \frac{t(1)}{t(p)} = \frac{1}{\frac{k}{p}+1-k}$, и так как величина параллельного ускорения нам известна, мы можем найти значение k как $k = \frac{\frac{1}{S(p)}-1}{\frac{1}{p}-1}$. Эта величина для разных значений M и N представлена на следующем графике:



	2	3	4	5	6
400	0	0	0	0,25	0,24
1510	-0,133333333	-0,1	-0,088888889	0,166666667	0,16
2620	0,049382716	0,055555556	0,065843621	0,262345679	0,251851852
3730	0,054794521	0,04109589	0,03652968	0,248287671	0,246575342
4840	-0,008888889	-0,006666667	-0,017777778	0,205555556	0,208
5950	0,093023256	0,069767442	0,062015504	0,265261628	0,258139535
7060	0,117647059	0,081932773	0,072829132	0,225840336	0,267226891
8170	0,100323625	0,063106796	0,062567422	0,238673139	0,25631068
9280	0,005347594	0,006016043	0	0,210561497	0,20855615
10390	0,010752688	0,003225806	0,010035842	0,206989247	0,212903226
11500	0,005342832	-0,001335708	0,225586227	0,212600178	0,208370436

Рис. 8: Доля распараллеленных команд

Как видно на графике, при переходе от 4 потоков к 5 (или от 3 потоков к 4 в случае $N = 11500$) доля распараллеленых команд увеличивается с 0 до в среднем 0.22, показывая, что только при этом количестве потоков параллелизация начинает иметь смысл. Это соотносится с уменьшением времени работы программы, а так же улучшением значений параллельного ускорения и параллельной эффективности.

3 Вывод

На основании данного эксперимента можно сделать два вывода:

- Даже при отсутствии параллелизации использование сильно оптимизированных библиотек помогает значительно уменьшить время выполнения программы при наличии большого количества вычислений. При переходе на использование библиотеки *Framewave* время работы снизилось в среднем на 43%.
- Для данного эксперимента оптимальное количество потоков равняется 5 при сравнительно небольшом количестве вычислений, так как во всех экспериментах, кроме последнего повторное уменьшение времени выполнения программы (в среднем на 15%) было замечено именно при переходе на 5 потоков. С ростом количества вычислений улучшение времены выполнения происходит уже при переходе на 4 потока (улучшение на 17% в последнем эксперименте).

К сожалению, использование библиотеки *Framewave* ограничило меня в установлении большего числа потоков, чем физических ядер, доступных системе, и я не смог провести эксперименты по исследованию чрезмерной параллелизации.

4 Приложения

4.1 Исходный код программы lab2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <fwBase.h>
#include <fwSignal.h>

#define A 700 //

#define NUMBER_OF_ITERATIONS 50

Fw64f random_on_interval(Fw64f min, Fw64f max, unsigned int *seed) {
    return (Fw64f) ((rand_r(seed) % (int)(max + 1 - min)) + min);
}

int main(int argc, char* argv[]) {
    struct timeval T1, T2;
    int N = atoi(argv[1]);
    int num_of_threads = atoi(argv[2]);

    Fw64f M1[N];
    Fw64f M2[N/2];
    Fw64f M2_shifted_copy[N/2];

    Fw64f X = 0;

    fwSetNumThreads(num_of_threads);

    gettimeofday(&T1, NULL);

    int i, j;
    for (i=0; i<NUMBER_OF_ITERATIONS; i++) {
        /******************** GENERATE ********************/
        unsigned int seed = i;
        for (j = 0; j < N; j++) {
            M1[j] = random_on_interval(0, A, &seed);
            // converting to radians for later computations
            M1[j] = (M1[j] * M_PI) / 180.0;
        }
        for (j = 0; j < N/2; j++) {
            M2[j] = random_on_interval(A, A*10, &seed);
            M2_shifted_copy[j] = j == 0 ? 0 : M2[j-1]; // shifted
               copy of M2 needed on the next step
        }

        /******************** MAP ************************/
        // operation #1
        fwsSinh_64f_A50(M1, M1, N);
        fwsPowx_64f_A50(M1, 2, M1, N);

        // operation #3
        fwsAdd_64f_I(M2_shifted_copy, M2, N/2);
```

```
51          fwsTan_64f_A50(M2, M2, N/2);
52          fwsAbs_64f_I(M2, N/2);
53
54          /******************** MERGE ************************/
55          // operation #1
56          fwsPow_64f_A50(M1, M2, M2, N/2);
57
58          /******************** SORT *************************/
59          int k, l, min_k;
60          for (k = 0; k < (N/2)-1; k++)
61          {
62              min_k = k;
63              for (l = k+1; l < N/2; l++)  {
64                  if (M2[l] < M2[min_k]) {
65                      min_k = l;
66                  }
67              }
68              if (min_k != k) {
69                  Fw64f temp = M2[min_k];
70                  M2[min_k] = M2[k];
71                  M2[k] = temp;
72              }
73          }
74
75          /******************** REDUCE ***********************/
76          int min_index;
77          for (min_index = 0; M2[min_index] <= 0; min_index++) {}
78          Fw64f min = M2[min_index];
79
80          for (j = 0; j < N/2; j++) {
81              if (isfinite(M2[j]) && (int)(M2[j] / min) % 2 == 0) {
82                  // remember to convert to radians
83                  Fw64f ans = sin((M2[j] * M_PI) / 180.0);
84                  if (!isnan(ans)) {
85                      X += ans;
86                  }
87              }
88          }
89      }
90
91      gettimeofday(&T2, NULL);
92      long delta_ms = 1000 * (T2.tv_sec - T1.tv_sec) + (T2.tv_usec - T1
            .tv_usec) / 1000;
93      printf("%d\n", N);
94      printf("%ld\n", delta_ms);
95      printf("%.5f\n", X);
96
97      return 0;
98  }
```

4.2 Скрипт для запуска программы lab2.c

```
#!/bin/bash
set -eu

declare -a N_set=("400" "1510" "2620" "3730" "4840" "5950" "7060" "8170" "9280" "10390" "11500")
declare -a M_set=("1" "2" "3" "4" "5" "6")
TIME_TEMPLATE="lab2_template.csv"
TIME_OUTPUT="lab2_output.csv"
X_TEMPLATE="lab2_x_template.csv"
X_OUTPUT="lab2_x_output.csv"
EXECUTABLE="lab2"

NUM_OF_TRIES=3

function add_to_line() {
  FILE_NAME="${1}"
  LINE_NUM="${2}"
  TEXT="${3}"
  sed -e "${LINE_NUM}s/\$/${TEXT};/" -i "${FILE_NAME}"
}

cp "./${TIME_TEMPLATE}" "./${TIME_OUTPUT}"
cp "./${X_TEMPLATE}" "./${X_OUTPUT}"

for M in "${M_set[@]}"
do
    x_line_num=1
    add_to_line ${X_OUTPUT} ${x_line_num} ${M}
    ((x_line_num=x_line_num+1))
    time_line_num=2
    for N in "${N_set[@]}"
    do
        X=$(./${EXECUTABLE} ${N} ${M})
        y=(${X//$'\n'/ })
        min_time=${y[1]}
        for (( meh=2; meh<=${NUM_OF_TRIES}; meh++ ))
        do
          X=$(./${EXECUTABLE} ${N} ${M})
          y=(${X//$'\n'/ })
          if [ "${y[1]}" -lt "${min_time}" ]
          then
            min_time=${y[1]}
          fi
        done
        add_to_line ${TIME_OUTPUT} ${time_line_num} ${min_time}
        add_to_line ${X_OUTPUT} ${x_line_num} $(sed 's/\./,/g' <<< ${y[2]})
        ((x_line_num=x_line_num+1))
        ((time_line_num=time_line_num+1))
    done
done

exit 0
```

4.3 Время выполнения программы lab2.c

	N/A	1	2	3	4	5	6
400	8	5	5	5	5	4	4
1510	56	30	32	32	32	26	26
2620	139	81	79	78	77	64	64
3730	252	146	142	142	142	117	116
4840	404	225	226	226	228	188	186
5950	585	344	328	328	328	271	270
7060	811	476	448	450	450	390	370
8170	1061	618	587	592	589	500	486
9280	1355	748	746	745	748	622	618
10390	1715	930	925	928	923	776	765
11500	2077	1123	1120	1124	933	932	928

Рис. 9: Таблица всех значений времени выполнения программы

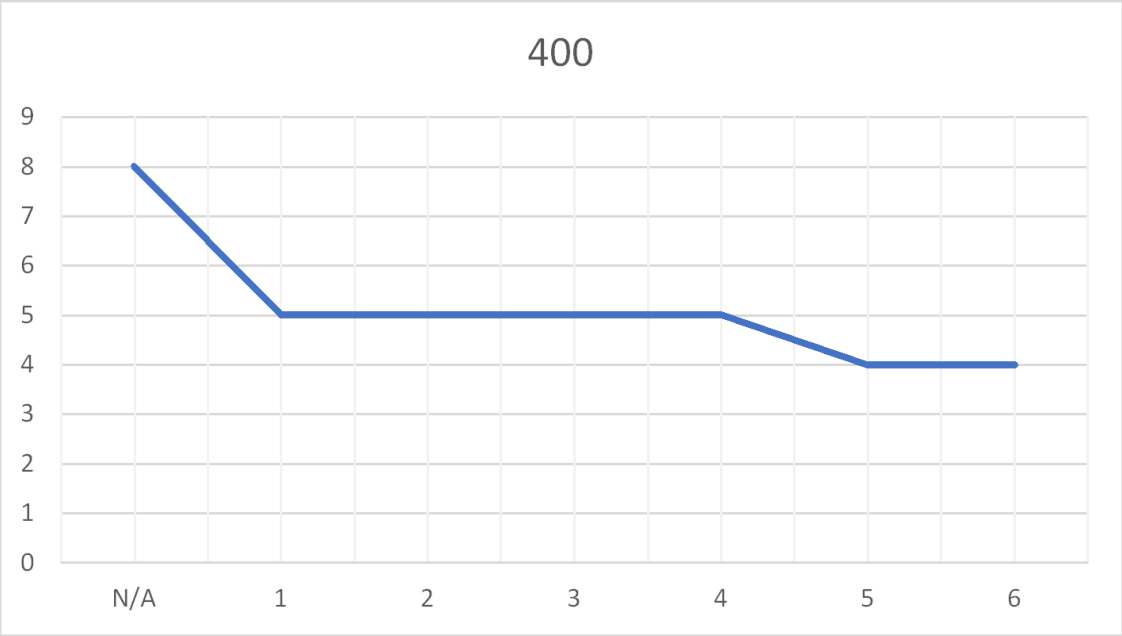


Рис. 10: Время выполнения при $N = 400$

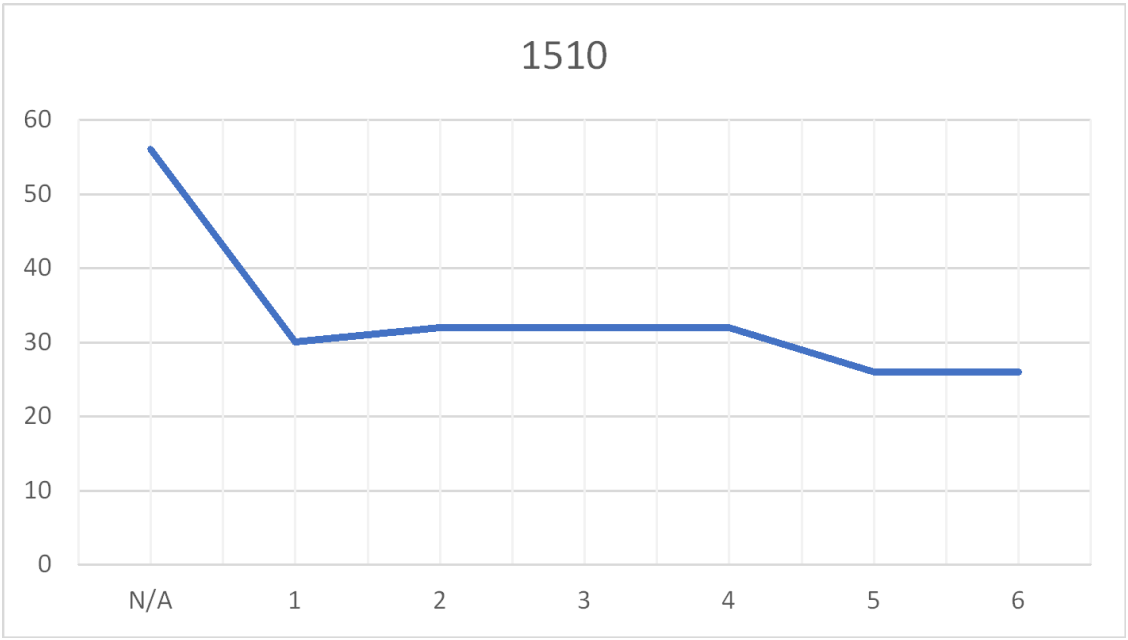


Рис. 11: Время выполнения при $N = 1510$

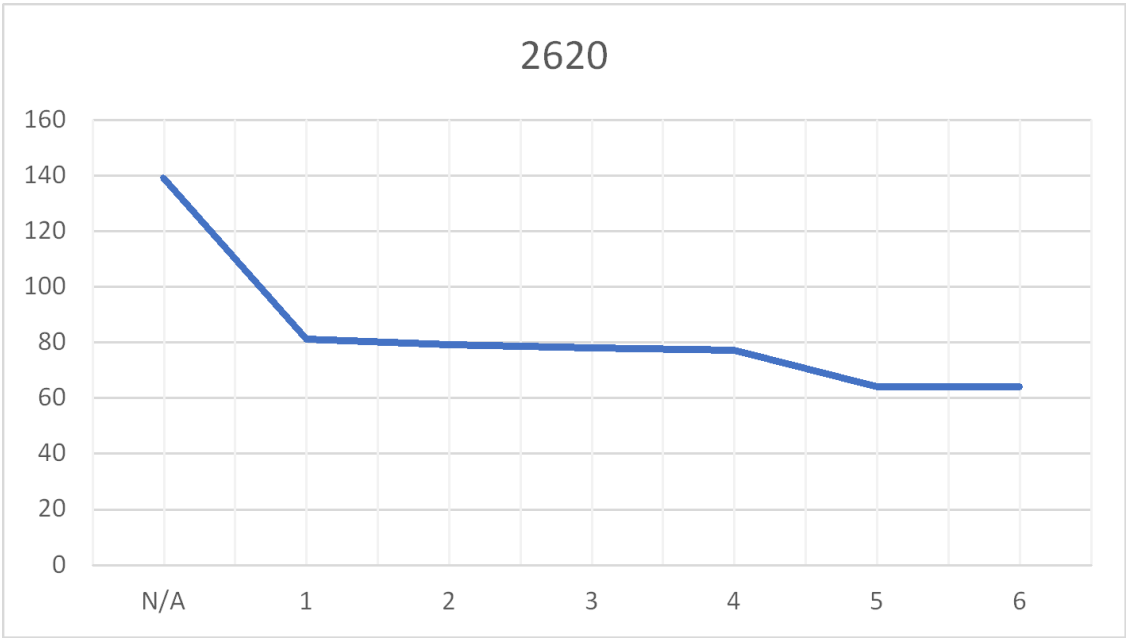


Рис. 12: Время выполнения при $N = 2620$

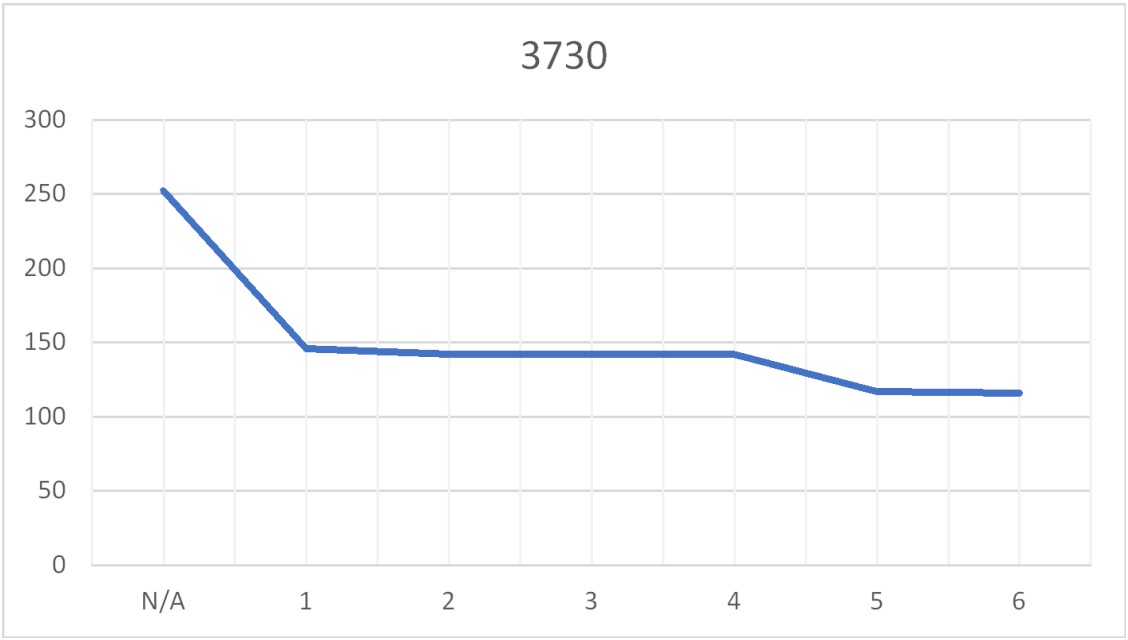


Рис. 13: Время выполнения при $N = 3730$

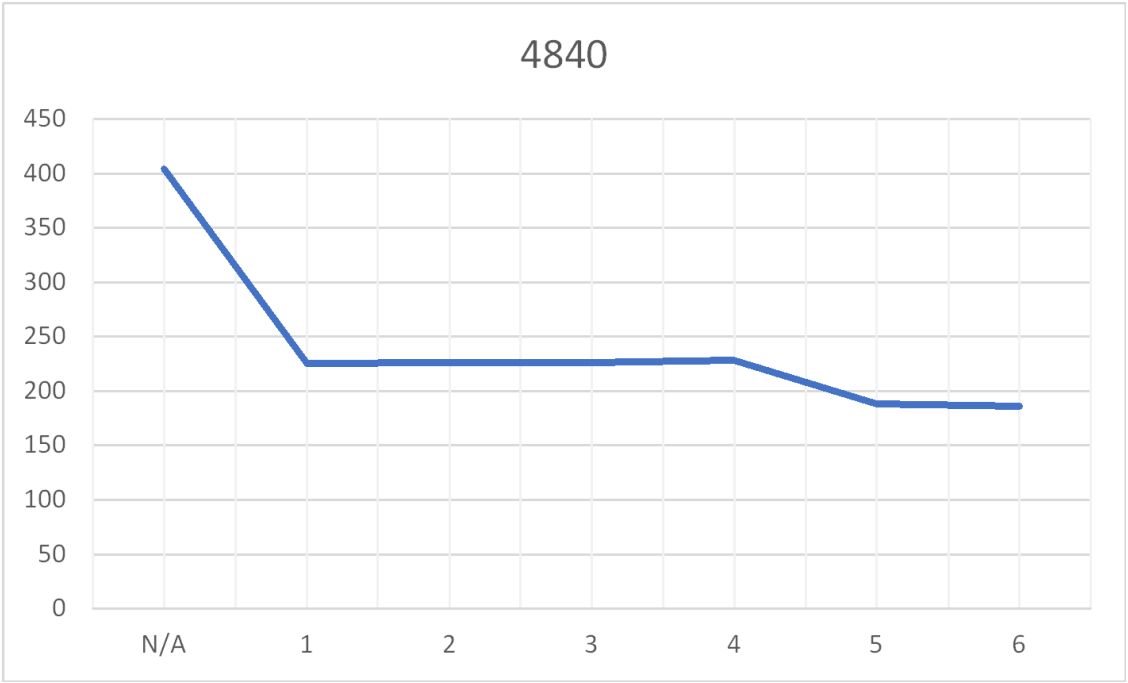


Рис. 14: Время выполнения при $N = 4840$

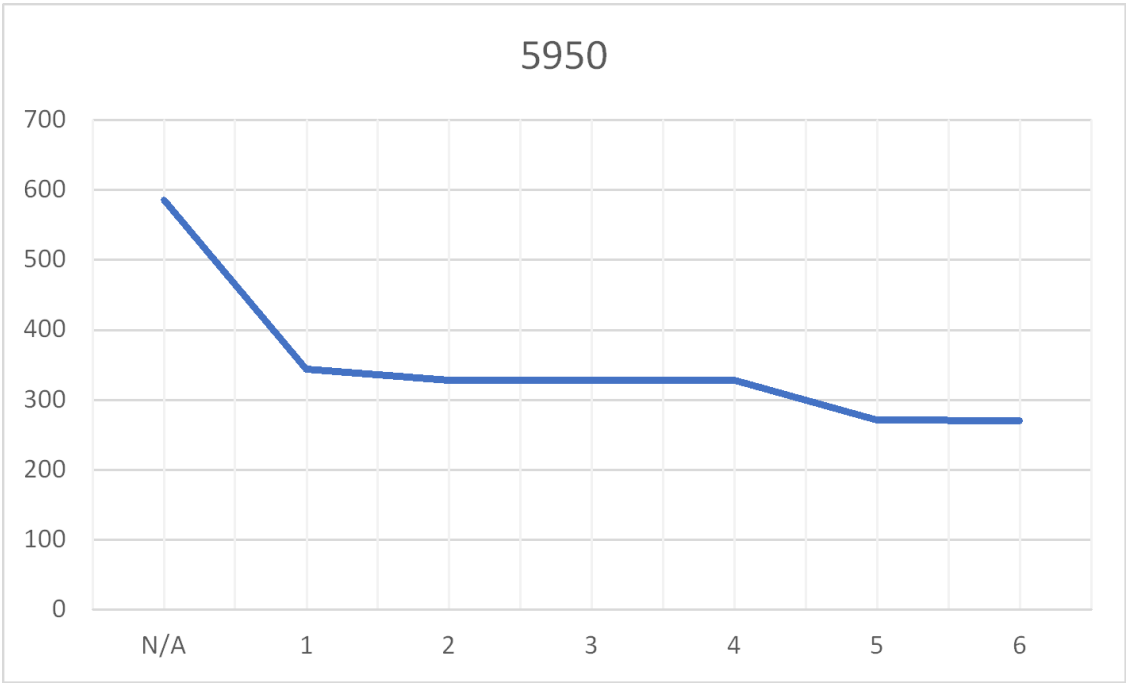


Рис. 15: Время выполнения при $N = 5950$

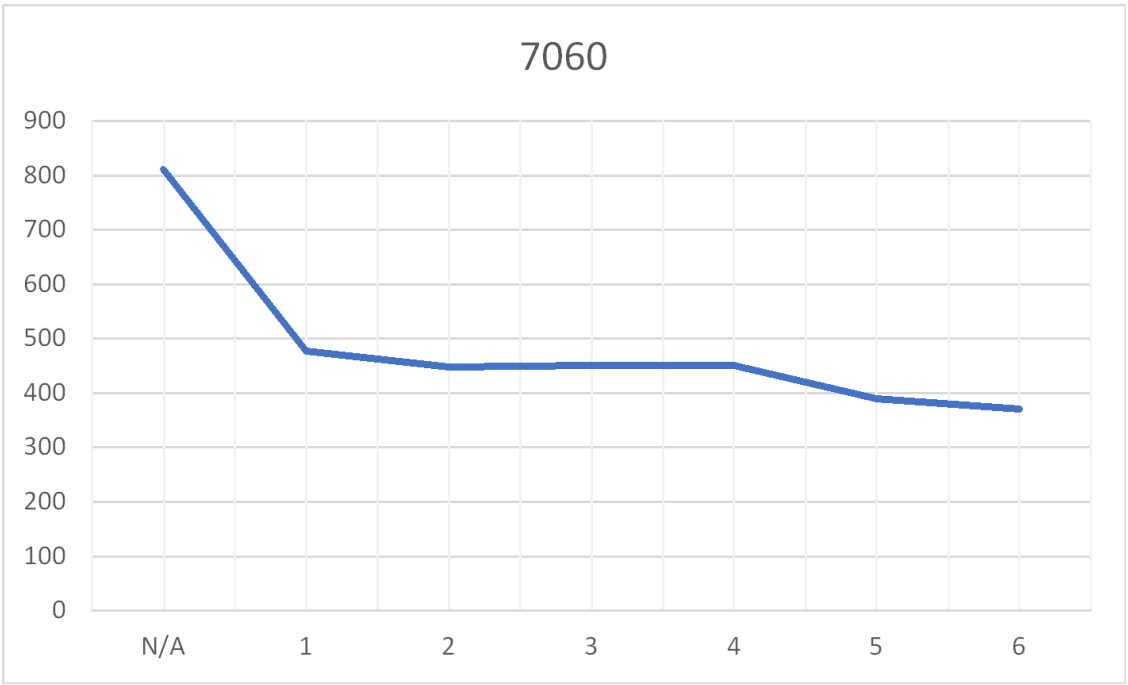


Рис. 16: Время выполнения при $N = 7060$

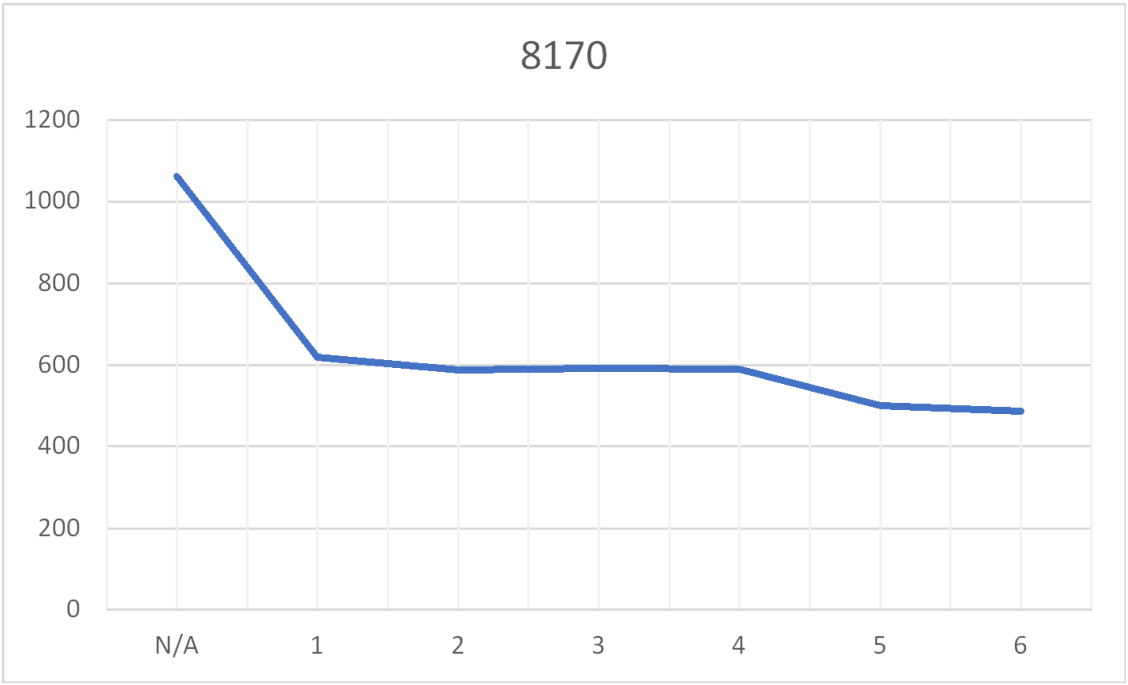


Рис. 17: Время выполнения при $N = 8170$

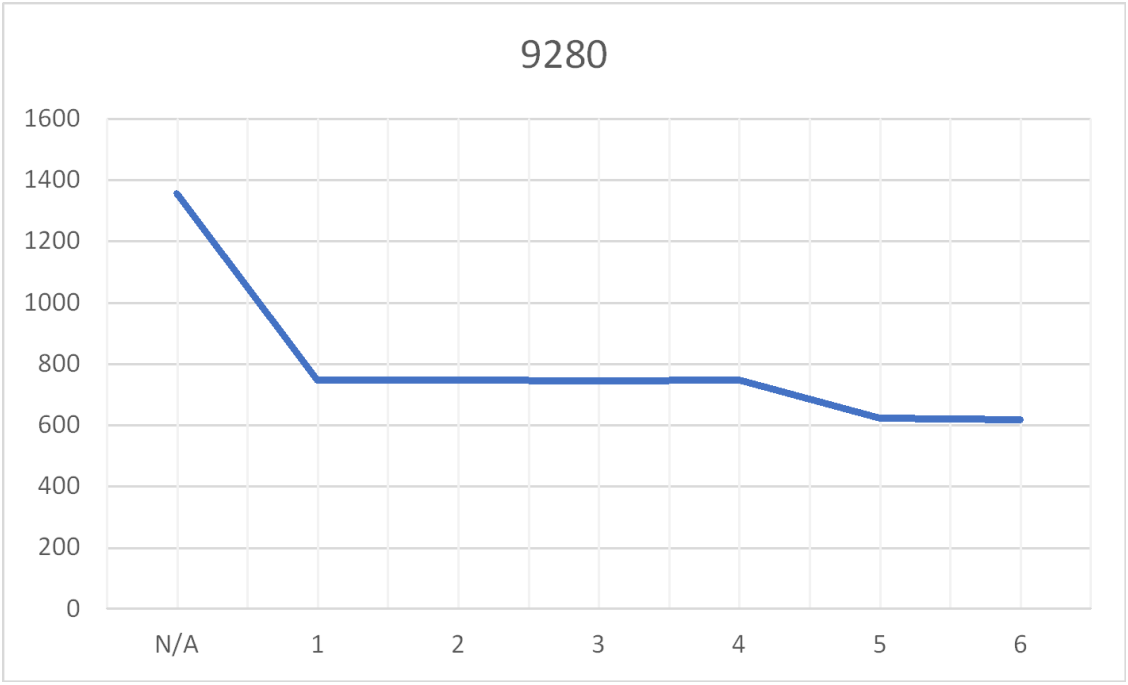


Рис. 18: Время выполнения при $N = 9280$

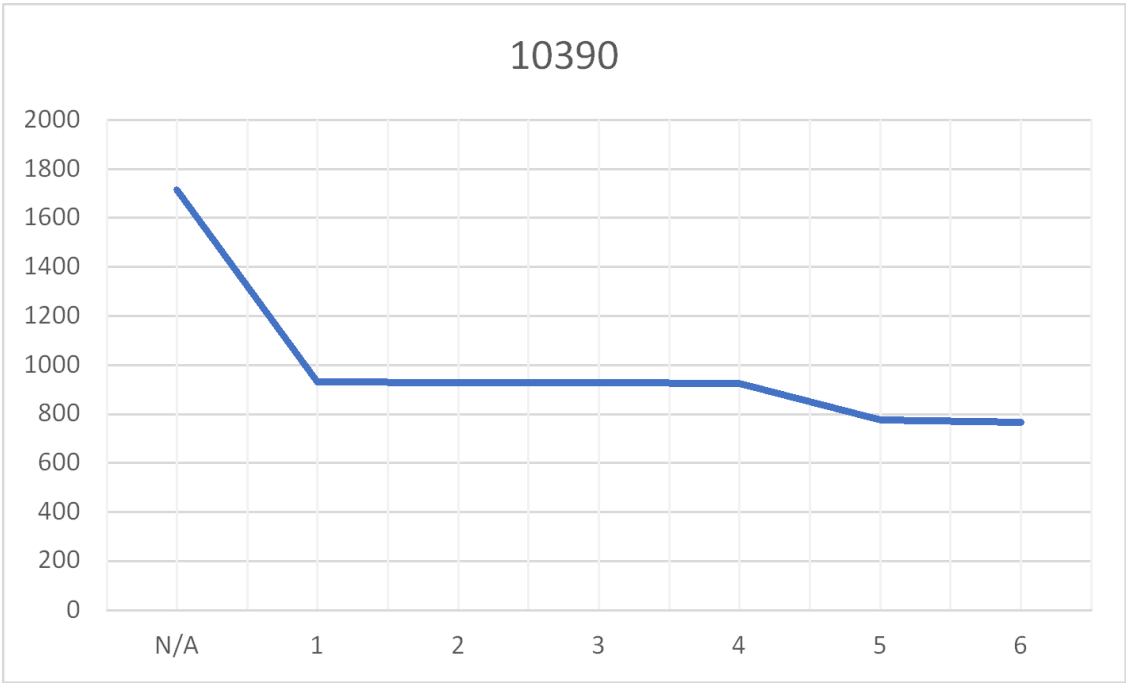


Рис. 19: Время выполнения при $N = 10390$

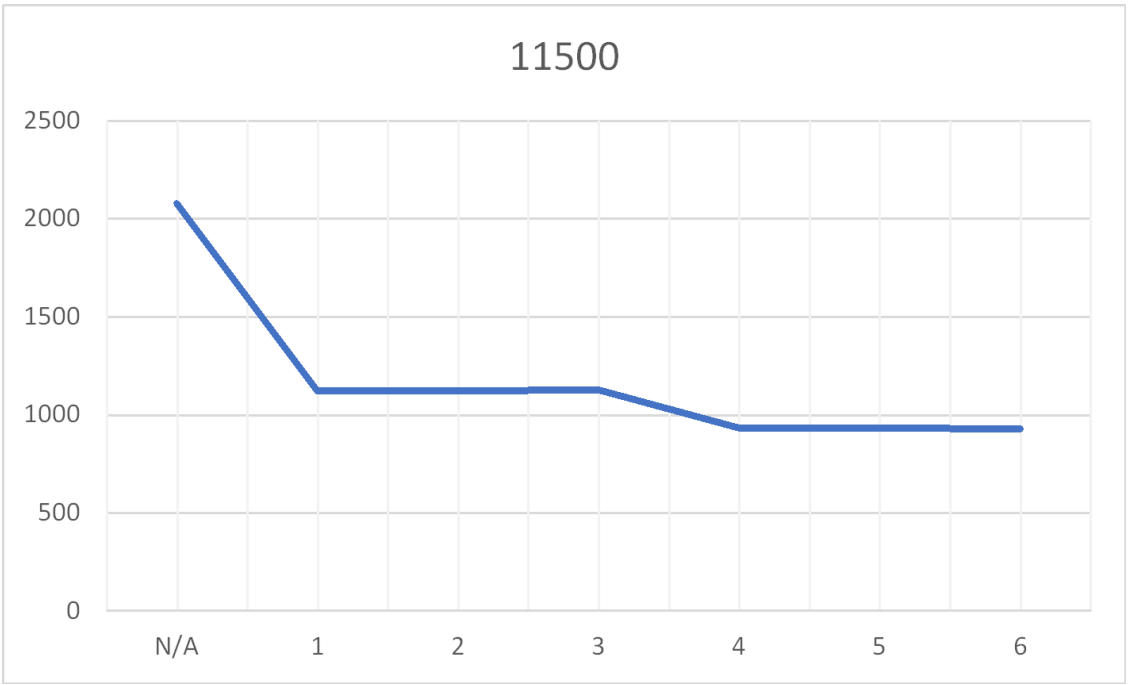


Рис. 20: Время выполнения при $N = 11500$