Софийски
Университет
Св. Климент Охридски

Проект по Разпределени софтуерни архитектури

Пресмятане на числото Pi - Ramanujan 1

Изготвил:
Любослав Карев
СИ, 3-ти курс, ф.н. 62144

Проверил:

Постановка на задачата

Числото (стойността на) Pi може да бъде изчислено по различни начини. Използвайки сходящи редове, можем да сметнем стойността на Pi с произволно висока точност. Един от бързо сходящите към Pi редове е този, открит от индийския математик Srinivasa Ramanujan през 1910-1914 година. За стойността на Pi имаме:

$$(1)\ \mathit{Ramanujan}\ 1,\ 1914$$

$$\frac{1}{\pi}=\frac{\sqrt{8}}{99^2}\sum_{n=0}^{\infty}\frac{(4n)!}{(4^n n!)^4}\frac{1103+26390n}{99^{4n}}$$

Формула (1)

Вашата задача е да напишете програма за изчисление на числото Pi използвайки цитирания ред, която използва паралелни процеси (нишки) и осигурява пресмятането на Pi със зададена от потребителя точност.

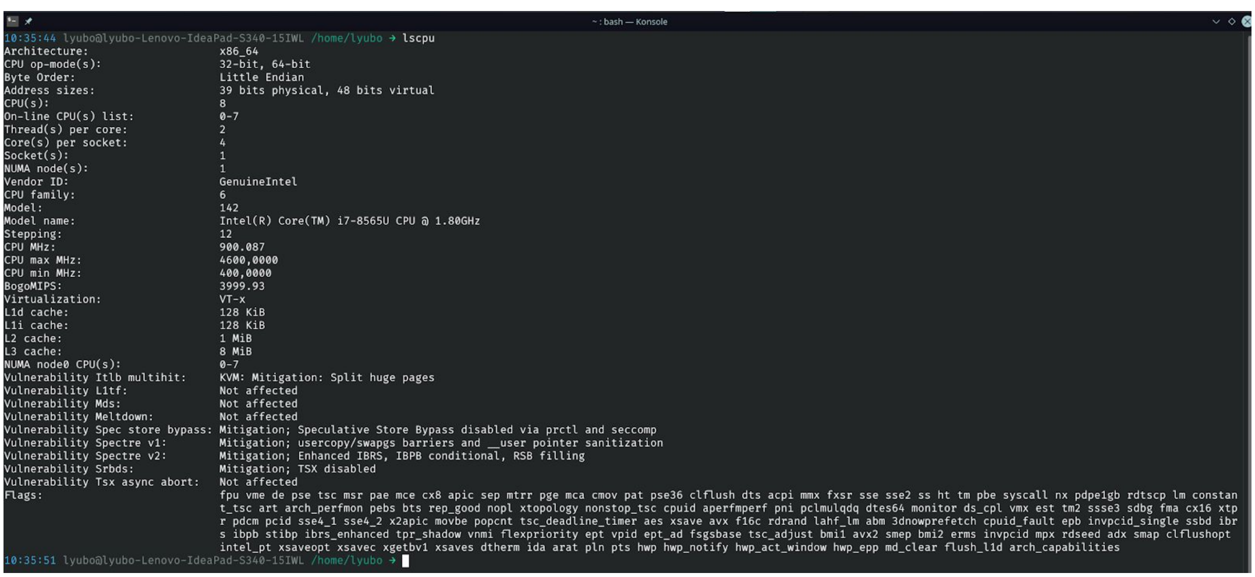
Използван хардуер

За тестването на програмите за използвани две машини. Надолу в анализа и тестовете за различните машини ще бъдат използвани наименованията **Машина 1** и **Машина 2**

Машина 1:

Лаптоп Lenovo IdeaPad S340 със следните характеристики на процесора:

Марка и модел: Intel Core i7-8565U
Брой физически ядра: 4
Скорост по спецификация: 1.8GHz
Multithreading: Да
Размер на кеша: (L1d/L1i/L2/L3): 128KiB/128KiB/1MiB/8MiB
Детайлна информация за процесора е дадена на фиг. 1



```
~ : bash — Konsole
10:35:44 lyubo@lyubo-Lenovo-IdeaPad-S340-15IWL /home/lyubo → lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   39 bits physical, 48 bits virtual
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           142
Model name:                      Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
Stepping:                        12
CPU MHz:                         900.087
CPU max MHz:                     4600,0000
CPU min MHz:                     400,0000
BogoMIPS:                        3999.93
Virtualization:                  VT-x
L1d cache:                       128 KiB
L1i cache:                       128 KiB
L2 cache:                        1 MiB
L3 cache:                        8 MiB
NUMA node0 CPU(s):               0-7
Vulnerability Itlb multihit:     KVM: Mitigation: Split huge pages
Vulnerability L1tf:              Not affected
Vulnerability Mds:               Not affected
Vulnerability Meltdown:          Not affected
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl and seccomp
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Enhanced IBRS, IBPB conditional, RSB filling
Vulnerability Srbds:             Mitigation; TSX disabled
Vulnerability Tsx async abort:   Not affected
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constan
                                 t_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtp
                                 r pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single ssbd ibr
                                 s ibpb stibp ibrs_enhanced tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt
                                 intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp md_clear flush_l1d arch_capabilities
10:35:51 lyubo@lyubo-Lenovo-IdeaPad-S340-15IWL /home/lyubo →
```

(Фиг. 1)

Машина 2:

Машината предоставена ни от ФМИ за упражненията.

Марка и модел: Intel Xeon E5-2660 x2
Брой физически ядра: 8
Скорост по спецификация: 2.20GHz
Multithreading: Да
Размер на кеша: (L1d/L1i/L2/L3): 32K/32K/256K/20480K

Детайлна информация за процесора е дадена на фиг. 2

```
u62144@t5600:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 45
Model name:            Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz
Stepping:              7
CPU MHz:               2699.914
CPU max MHz:           3000,0000
CPU min MHz:           1200,0000
BogoMIPS:              4389.58
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K
NUMA node0 CPU(s):     0-7,16-23
NUMA node1 CPU(s):     8-15,24-31
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch
_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic popcnt tsc_d
eadline_timer aes xsave avx lahf_lm epb ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid xsaveopt dtherm ida arat pln pts spec_ctrl intel_stibp flush_l1d
u62144@t5600:~$
```

(фиг. 2)

Анализ на проблема и незадоволителни решения

При анализа на зависимостите на данните, забелязваме, че всеки член на реда не зависи единствено от индекса му (в нашия случай n). Можем спокойно да направим програма, която да смята тази сума. Ще наричаме тази програма **SimpleJava.**

SimpleJava

За програмата **SimpleJava** използваме SIMP модела - Single Instruction, Multiple data, заедно с Master Worker [1]. Програмата приема два аргумента - брой цифри след десетичната запетая (ще отбелязваме този параметър с **p**) и брой нишки (ще отбелязваме този параметър с **t**). Стартираме **t**-на брой нишки, като всяка нишка пресмята следващия член да реда, който не е сметнат досега. Грануларността тук е фина (имаме голям брой малки задачи. Една нишка пресмята един член, и приключва работа - след това се стартира нова нишка, с нова задача.

Бяха проведени следните тестове на прогрмата **SimpleJava** върху **Машина 1:**

20 последователни пускания, с параметър p=10240 и вариране на броя нишки между 1, 2 и 4.

На фигура 3 са представени в табличен вариант резултатите, като по редовете е поставен съответното изпълнение, а в колоните е броя на нишките използвани.

Run number	t = 1	t = 2	t = 4
1	3.33	2.552	2.298
2	3.314	2.53	2.495
3	3.214	2.667	2.272
4	3.554	2.914	2.989
5	5.27	3.083	3.542
6	4.235	4.055	3.13
7	4.209	3.695	3.202
8	4.023	4.126	3.854
9	3.905	4.069	4.978
10	4.111	3.657	4.397
11	4.311	3.811	3.214
12	4.241	3.356	3.65
13	4.307	3.523	3.473
14	4.395	3.744	3.219
15	4.47	4.021	3.558
16	4.279	3.57	3.199
17	4.371	3.874	3.288
18	4.24	3.361	3.58
19	4.391	3.932	3.07
20	4.046	4.022	3.12

(фиг. 3)

Вземайки минимумите от тестове за различния брой нишки, получаваме следните стойности:

- t=1: 3.214
- t=2: 2.53

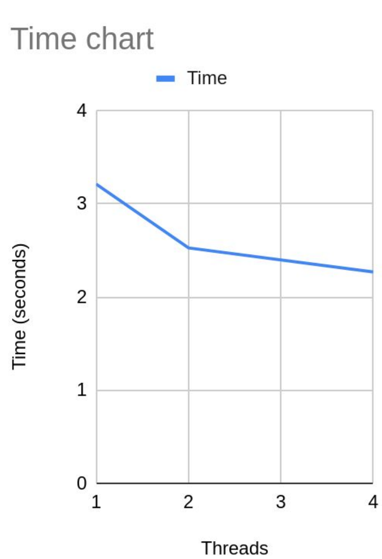
- t=4: 2.272

На фигура 4 са представени и коефициентите **Tt, St и Et,** където:

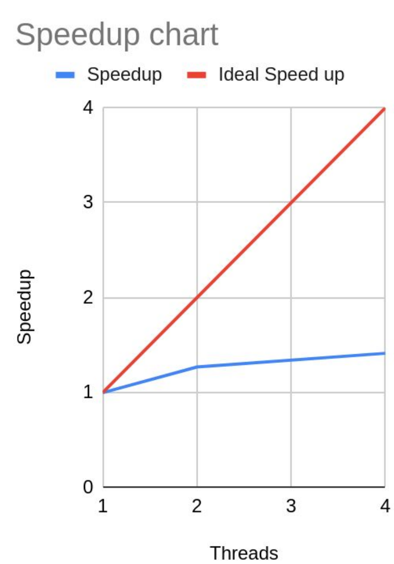
- **Tt -** времето за изпълнение на задачата при t на брой нишки
- **St -** ускорението на програмата при използване на t на брой нишки спрямо T1; St = T1/Tt
- **Et** - ефикасността на програмата - Et = St/t

t	Tt	St	Et
1	3.214	1	1
2	2.53	1.270355731	0.6351778656
4	2.272	1.414612676	0.353653169

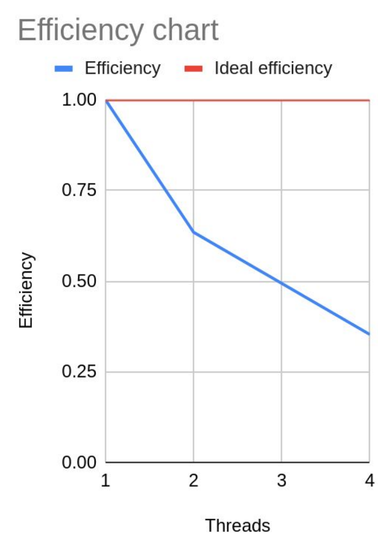
(фиг. 4)



(фиг. 5)



(фиг. 6)



(фиг. 7)

На фигури 5, 6 и 7 виждаме показани трите параметъра графично. Забелязваме, че решението ни не показва почти никакво ускорение, независимо от броя нишки, което използваме. Задаваме си въпроса защо се получава така ? Ако се върнем първоначално на реда, забелязваме че в него се съдържа множител (4*n)!, което би могло да бъде причина за липсата на ускорение, защото макар и да разделяме задачата на повече нишки, всяка следващата нишка има в пъти по-сложна задача, и в крайна сметка, ние трябва да изчакаме приключването ѝ.

След изследване на реда, откриваме, че можем да представим всеки член с номер **n+1** като член с номер **n,** умножен по константа, зависеща също от **n.** Така можем да получим рекурентна зависимост за членовете на реда. В новополучената рекурентна редица не присъстват множетелни получени чрез факториели, което теоретично би намалило сложността на задачата. На фигура 8 е показано получаването на общата формула на константата, необходима за рекурентната редица.

$$a_n = \frac{\sqrt{8}}{99^2} \frac{(4n)!}{(4^n n!)^4} \frac{1103 + 26390n}{99^{4n}}$$

$$a_{n+1} = \frac{\sqrt{8}}{99^2} \frac{[4(n+1)]!}{(4^{n+1}(n+1)!)^4} \frac{1103 + 26390(n+1)}{99^{4(n+1)}}$$

$$\frac{a_{n+1}}{a_n} = \frac{1103 + 26390(n+1)}{99^{4n} \cdot 99^4} \frac{\cancel{\sqrt{8}}}{\cancel{99^2}} \frac{(4n+4)!}{[4^n \cdot 4\,(n+1)!]^4} : \frac{\cancel{\sqrt{8}}}{\cancel{99^2}} \frac{(4n)!}{(4^n n!)^4} \frac{1103 + 26390n}{99^{4n}} =$$

$$= \frac{1103 + 26390n + 26390}{\cancel{99^{4n}} \cdot 99^4} \frac{\cancel{(4n)!}(4n+1)(4n+2)(4n+3)(4n+4)}{[\cancel{4^n n!} \cdot 4 \cdot (n+1)]^4} \cdot \frac{\cancel{(4^n n!)^4}}{\cancel{(4n)!}} \frac{\cancel{99^{4n}}}{1103 + 26390n} =$$

$$= \frac{1103 + 26390n + 26390}{99^4} \frac{(4n+1)(4n+2)(4n+3)\cancel{4(n+1)}}{4^{\cancel{4}3}(n+1)^{\cancel{4}3}} \frac{1}{1103 + 26390n} =$$

$$= \frac{(4n+1)(4n+2)(4n+3)}{4^3 (n+1)^3} \frac{1103 + 26390n + 26390}{99^4 (1103 + 26390n)} =$$

$$= \frac{(4n+1)(4n+2)(4n+3)}{4^3 (n+1)^3} \left[\frac{\cancel{1103 + 26390n}}{99^4 (\cancel{1103 + 26390n})} + \frac{26390}{99^4 (1103 + 26390n)} \right] =$$

$$= \frac{(4n+1)(4n+2)(4n+3)}{4^3 (n+1)^3} \left(\frac{1}{99^4} + \frac{26390}{99^4 (1103 + 26390n)} \right)$$

(фиг. 8) - Формула (2)

RecurrentJava

За изпълнението на новата задача, ще бъдат използвани два класа - главен клас, който ще раздава задачите, и клас, който ще изпълнява задачата - пресмятане на даден член на редицата, при подаден предишен член и индекс. Този път, параметъра **p** е броя членове, които да бъдат пресметнати - така знаем цялата област от стойности. Всяка нишка смята различна част от тази област, като на всяка нишка се дава парче от последователни стойности за смятане, с размер: **брой елементи / брой нишки + 1**. Например, ако имаме да сметнем 10 елемента, с 2 нишки, нишка 1 ще смята елементите с индекси от 0 до 4, а нишка 2 - тези от 5 до 9.Тук е използвано статично балансиране.

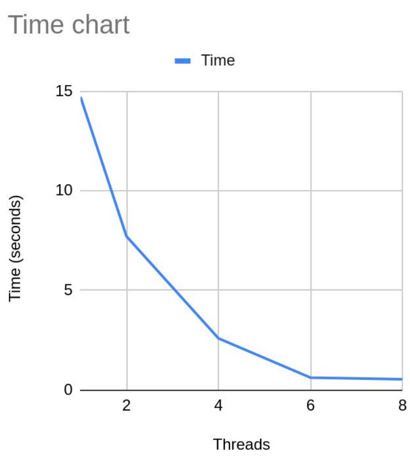
Резултатите тук са фрапантни - на моменти се получава ефективност по-голяма от 1 !

Тестовете отново са извършени на **Машина 1,** като този път, се използват и нишките на процесора с Hyperthreading. Пресметнати са 400 члена на реда, съответно с t=1, t=2, t=4, t=6, t=8. Резултатите са показани на фигура 9

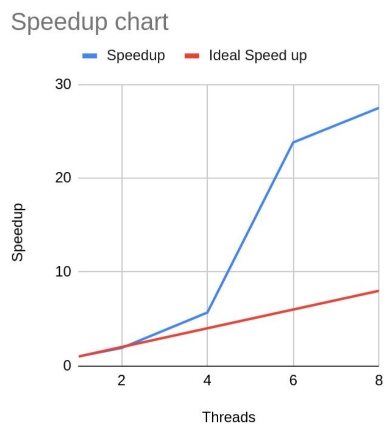
p=400	t = 1	t = 2	t = 4	t = 6	t = 8
1	14.718	8.378	2.667	1.352	0.686
2	15.858	7.713	2.965	1.079	0.716
3	17.204	7.802	3.038	0.627	0.551
4	17.351	9.146	3.227	0.792	0.647
5	17.503	9.717	2.922	0.803	0.622
6	17.179	8.955	3.505	0.737	0.535
7	17.395	9.112	2.811	1.173	0.709
8	17.221	9.268	3.084	0.974	0.801
9	17.115	9.062	3.811	1.21	0.743
10	16.971	9.031	2.592	0.821	0.971
11	17.108	9.005	3.471	0.941	0.722
12	17.278	8.741	3.157	1.062	0.923
13	17.493	9.229	3.072	1.302	0.989
14	17.203	9.472	3.289	1.116	0.84
15	17.077	9.013	2.623	0.618	0.717
16	17.191	9.637	3.279	0.628	0.698
17	17.718	9.303	3.53	0.829	0.713
18	18.375	9.319	3.187	0.745	0.806
19	17.359	9.133	3.544	0.713	0.739
20	17.492	9.096	3.495	0.715	0.989

Threads	Time	Speedup	Efficiency
1	14.718	1	1
2	7.713	1.908206923	0.9541034617
4	2.592	5.678240741	1.419560185
6	0.618	23.81553398	3.969255663
8	0.535	27.51028037	3.438785047

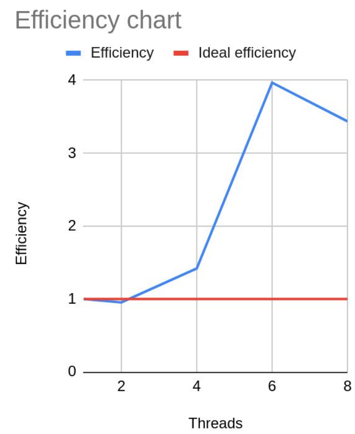
(фиг. 9)



(фиг. 10)



(фиг. 11)



(фиг. 12)

Евентуалните причина за тези резултати, може да е факта, че Java пуска много повече нишки, отколкото е необходимо. Друга възможна причина, е че библиотеката BigDecimal, която бе използвана, също използва нишки, което допринася за тези изкривени резултати.

С оглед на това, следващите решения са написани на езика Rust, където при пускането на една нишка от Rust, в ядрото на операционната система се пуска точно една нишка, докато при други езици, това съотношение не е 1 към 1 [2].

RecurrentRust

Тази програма има подобна логика като **RecurrentJava,** единствената промяна е езика - тук се използва Rust, вместо Java.

Начинът, по който програмата пресмята този ред е на базата на три променливи - начално **a** (елемент от редицата), начален индекс, и краен индекс. За намирането на всички членове, се използва формулата (2), а за пресмятането на първоначалното a, формула (1).

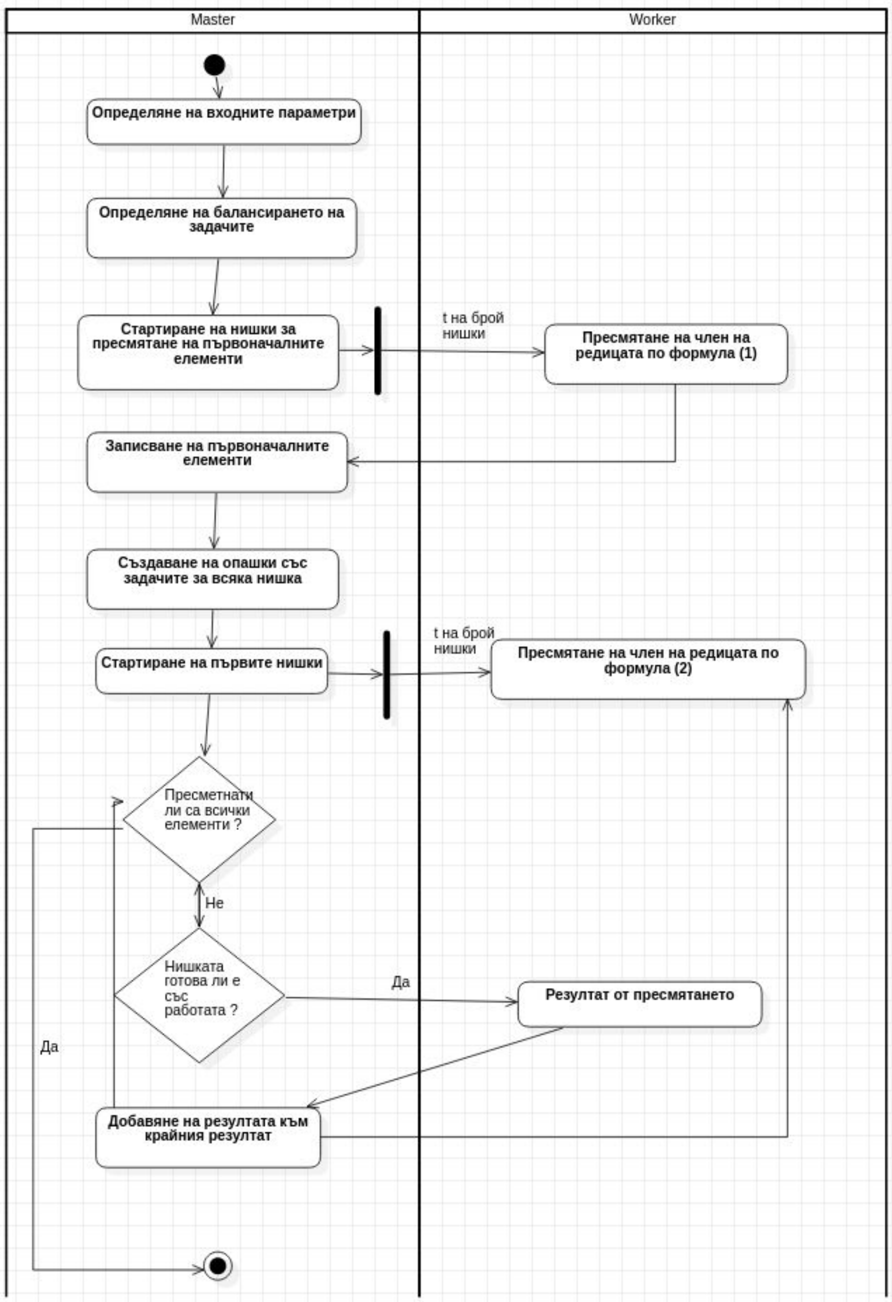
В главната нишка извършваме следните действия:

1. Определяме кои задачи за коя от нишките са
2. Изчисляваме първия елемент на всяка от нишките чрез формула (1)
3. Създаваме опашка за всяка нишка, която държи задачите ѝ
4. Стартираме нишките
5. Изчакваме резултат от някоя нишка
6. При резултат от някоя нишка, добавяме този резултат към променливата за резултат в главната нишка, и стартираме нова нишка със следващия индекс.

7. Ако сме пресметнали всички елементи, спираме

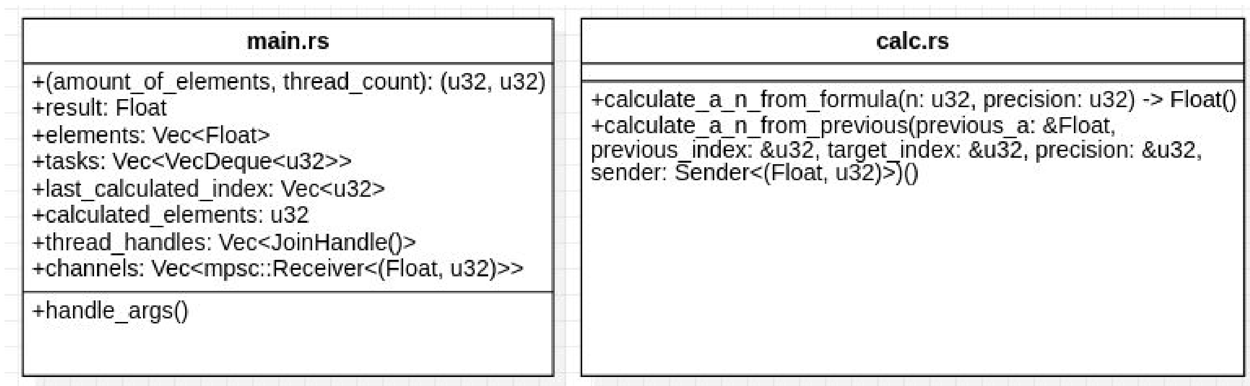
В нишките се извършват едно от следните две действия:
- Пресмятане на член на редица чрез формула (1)
- Пресмятане на член на редица чрез формула (2)

На фигура 13 е представена работата на програмата, във вид на Activity диаграма.



(фиг. 13)

На фигура 14 са представени основните променливи и методи, под формата на Class Diagram.



(фиг. 14)

*(Като странична забележка, бих искал да вметна, че **RecurrentJava** пресмята 400 члена на редица, на една нишка, за около 17 секунди. Както ще видите по-долу, **RecurrentRust** пресмята 300 000 члена, на една нишка, за около 25 секунди.)*

Тестовете отново са извършени на **Машина 1.** Пресметнати са 300000 члена на реда, съответно с t=1, t=2, t=4. Резултатите са показани на фигура 15.

(Отново като забележка, тук вероятно е имало влияние и термалните ограничения на процесора и лаптопа - този факт е игнориран при този и при предишните тестове, защото това не е финалното решение)

p=300000	t = 1	t = 2	t = 4
1	24.202	14.446	10.988
2	25.983	14.495	11.026
3	25.709	14.485	10.942
4	26.087	14.399	10.974
5	25.967	14.419	10.979
6	26.022	14.462	11.043
7	26.29	14.47	11.017
8	26.026	14.5	11.064
9	26.088	14.573	10.99
10	26.138	14.426	10.98
11	25.846	14.448	11.092

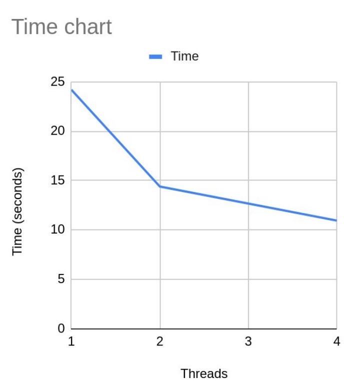
12	25.926	14.464	10.96
13	25.961	14.391	11.5
14	25.742	14.489	11.016
15	26.058	14.426	11.042
16	25.847	14.445	11.341
17	26.101	14.431	11.859
18	25.88	14.414	10.937
19	25.837	14.43	10.972
20	25.756	14.476	10.975

Threads	Time	Speedup	Efficiency
1	24.202	1	1
2	14.391	1.681745535	0.8408727677
4	10.937	2.212855445	0.5532138612

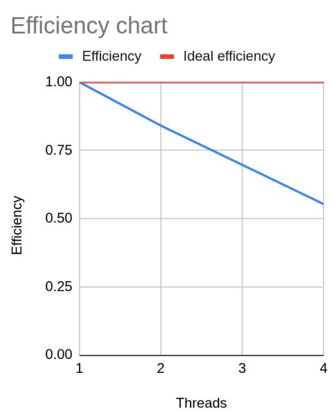
(фиг. 15)

Минималните резултати за различния брой нишки е:

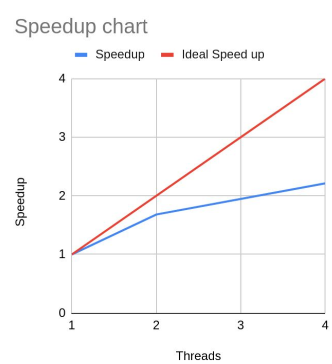
- t=1 - 24.202
- t=2 - 14.391
- t=4 - 10.937



(фиг. 16)



(фиг. 17)



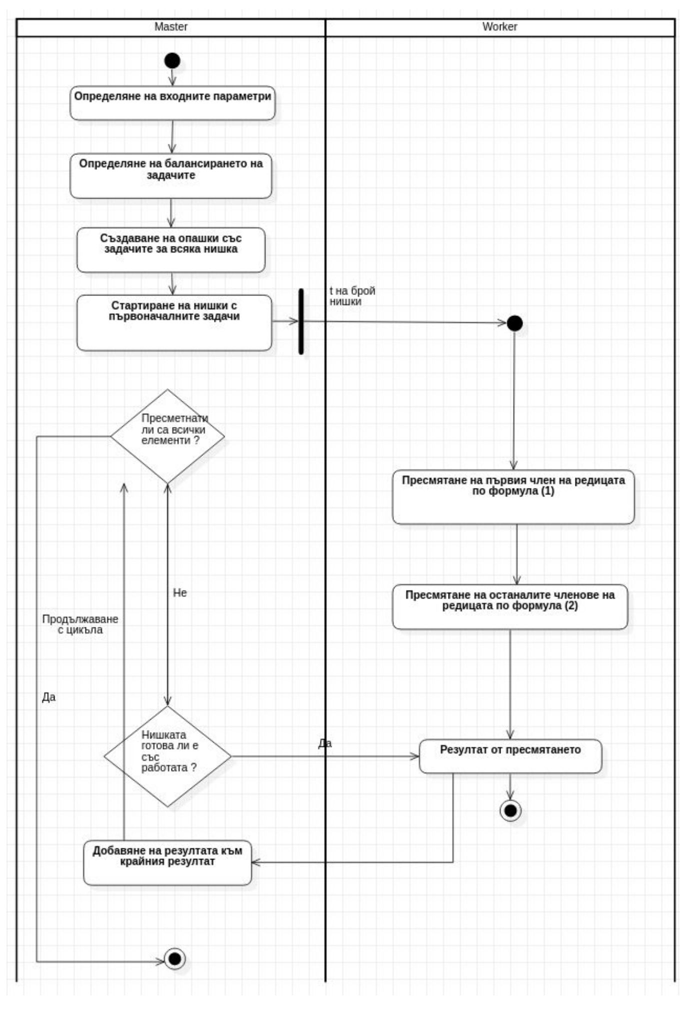
(фиг. 18)

На фигури 16, 17 и 18 са представени коефициентите Tt, Et и St съответно. Тук наблюдаваме по-реална картина, отколкото при **RecurrentJava.** След допълнителен анализ [3], се оказва, че причината за тези резултати е грануларността и начина по който се разпределят заданията.

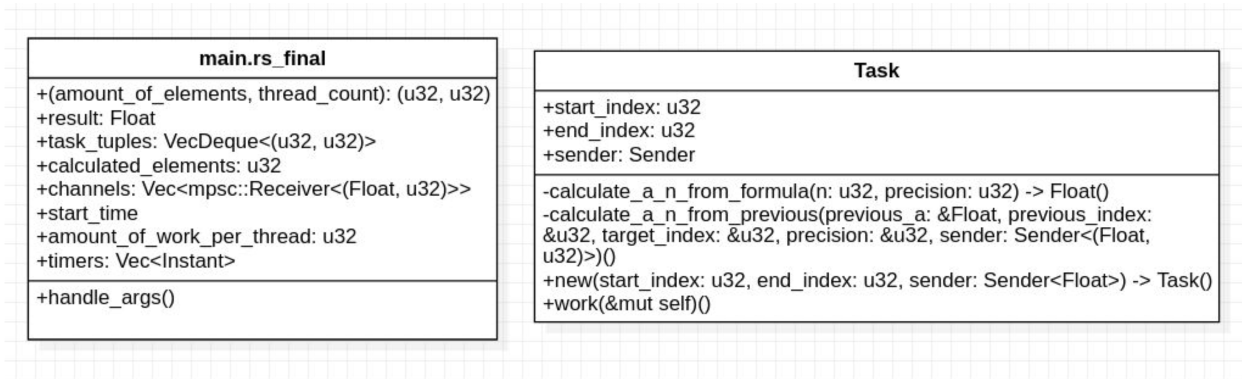
Финалното решение - FinalRust

FinalRust като идея се различава по **RecurrentRust** главно по това, че се използва динамично балансиране на задачите - всяка нишка получава задача с размер (брой-елементи / брой-нишки) / 2. При приключване на задачата, се стартира нова нишка, която получава нова задача - ако съществува такава задача.

Друга основна разлика между двете програми е тяхната архитектура. Разликата спрямо **RecurrentRust** е отделянето на голяма част от логиката в нова структура **Task.** Архитектурата на **FinalRust** е представена отново чрез Activity и Class диаграми, във фигури 19 и 20.



(фиг. 19)



(фиг. 20)

Както виждаме на диаграмите, голяма част от логиката, която бе в Master нишката, вече е изнесена в Task обекта. Task обекта приема три параметъра - начален индекс, краен индекс и комуникационен канал, по който да изпрати резултата. В този случай, Task сумира всички членове на редицата, в даден интервал от индекси, като първо пресмята първия член чрез формула (1), а после рекурентно пресмята и останалите, чрез формула (2).

Това позволява генериране на наредена двойка за задача - начален и краен индекс, като размерът на една такава задача, се изчислява по следната формула: (броя на елементите /броя на нишките) / 2; Това е средна грануларност на задачите. Програмата създава опашка от задачи, като всяка нишка преди да стартира взема задача от тази опашка и я изпълнява.

Програмата стартира първоначално t на брой нишки, които взимат задачи от опашката. При приключване на работата си, нишката се спира, и на нейно място се стартира нова нишка, която получава задачата си от опашката. Ако няма задача в опашката, нова нишка не се стартира.

Това се повтаря, докато не бъдат изчислени необходимия брой елементи.

Резултати

Програмата е тествана на **Машина 1** и **Машина 2** - важно е да се има предвид някои съображения при тестването на всяка от машините. Тези съображения ще бъдат коментирани, заедно с резултатите от тестването.

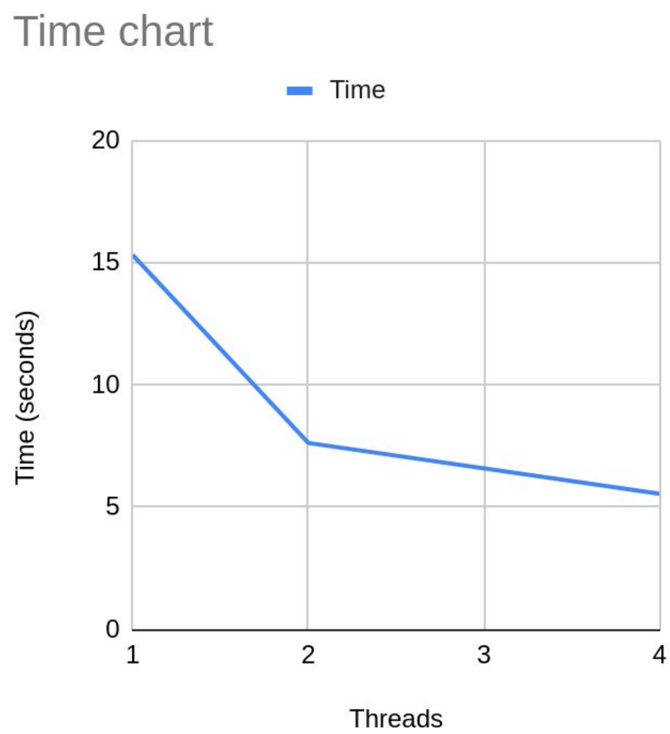
Машина 1. Пресметнати са 300000 члена на реда, съответно с t=1, t=2, t=4. Резултатите са показани на фигура 21, 22, 23 и 24.

(Отново като забележка, тук вероятно е имало влияние и термалните ограничения на процесора и лаптопа - при изчисляване на коефициентите са взети минималните стойности).

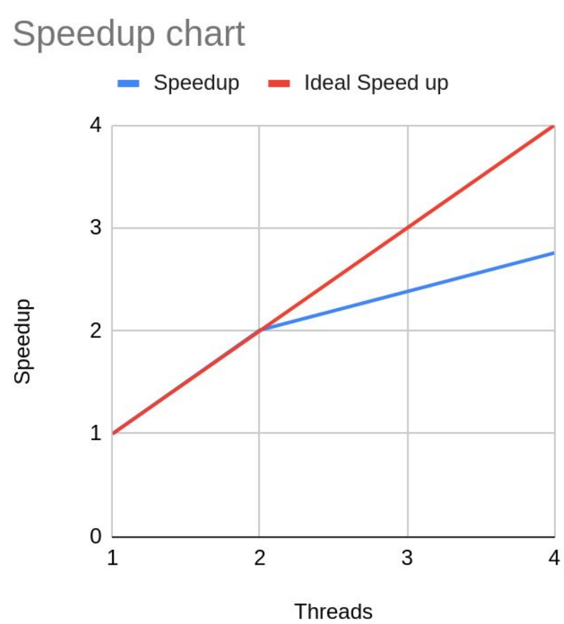
p=300000	t = 1	t = 2	t = 4
1	15.313	7.63	5.553
2	18.667	8.966	6.432
3	18.584	11.068	7.944
4	18.53	11.003	8.265
5	18.545	11.06	9.122
6	18.493	10.981	9.33
7	18.552	11.047	9.211
8	18.525	11.048	8.788
9	18.675	11.021	8.06
10	18.576	11.022	8.8
11	18.53	11.009	7.905
12	18.587	11.019	9.028
13	18.561	11.078	8.206
14	18.587	11.024	8.127
15	18.571	10.991	8.733
16	18.624	11.026	9.323
17	18.683	11.004	8.317
18	18.843	11.005	8.664
19	18.555	11.05	8.654
20	18.509	11.01	8.277

Threads	Time	Speedup	Efficiency
1	15.313	1	1
2	7.63	2.006946265	1.003473132
4	5.553	2.7576085	0.689402125

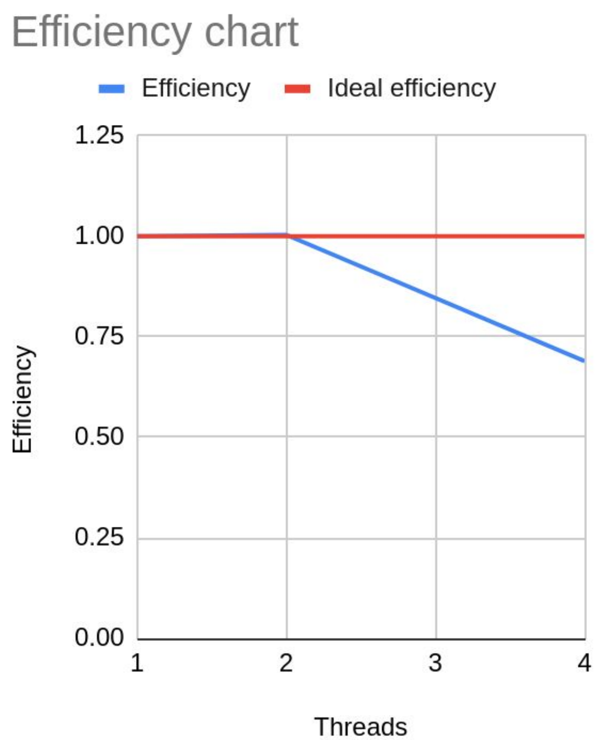
(фиг. 21)


Time chart
Time
20
15
10
5
0
Time (seconds)
1
2
3
4
Threads

(фиг. 22)


Speedup chart
Speedup
Ideal Speed up
4
3
2
1
0
Speedup
1
2
3
4
Threads

(фиг. 23)



(фиг. 24)

От резултатите можем да забележим, че ефектовността на програмата и ускорението ѝ се запазват при скока от една на две нишки, а от две на четири имаме малко забавяне, което би могло да се обясни с достигането на лимита на хардуерния паралелизъм. Независимо от това, можем да твърдим, че в случая на конкретния проблем, динамичното балансиране се справя по-добре от статичното.

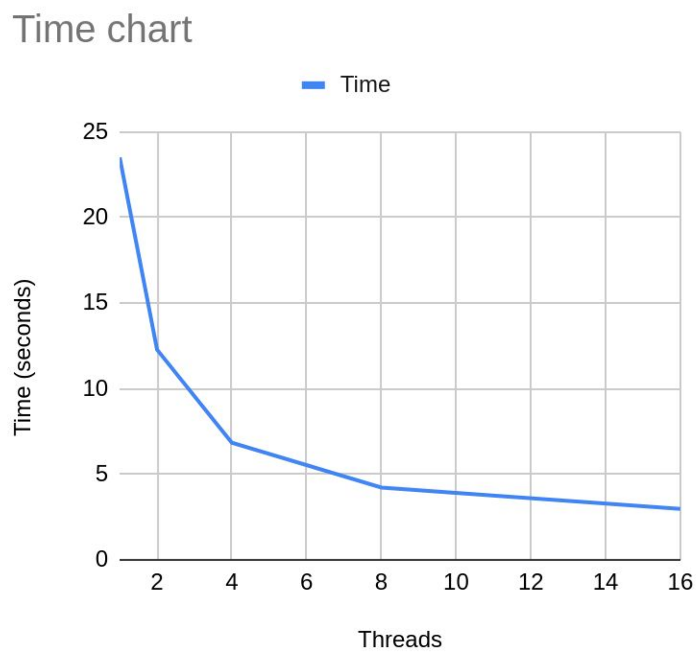
Програмата бе тествана и на **Машина 2,** отново с пресмятане на 300 000 члена, този път обаче с 1, 2, 4, 8 и 16 нишки (16 е лимита на хардуерния паралелизъм при **Машина 2)**. Важно е да отбележа, че тестовите сценарии бяха изпълнение три пъти, но и по време на трите изпълнения имаше доста голямо странично натоварване на машината, затова резултатите могат да не бъдат на 100% коректни. Резултатите могат да бъдат видени на фигури 25, 26, 27 и 28.

p=300000	t = 1	t = 2	t = 4	t=8	t=16
1	24.385	12.467	6.962	4.277	3.24
2	24.537	12.351	6.925	4.254	3.173
3	24.221	12.424	6.919	4.309	3.113
4	24.904	12.588	6.953	4.269	3.342

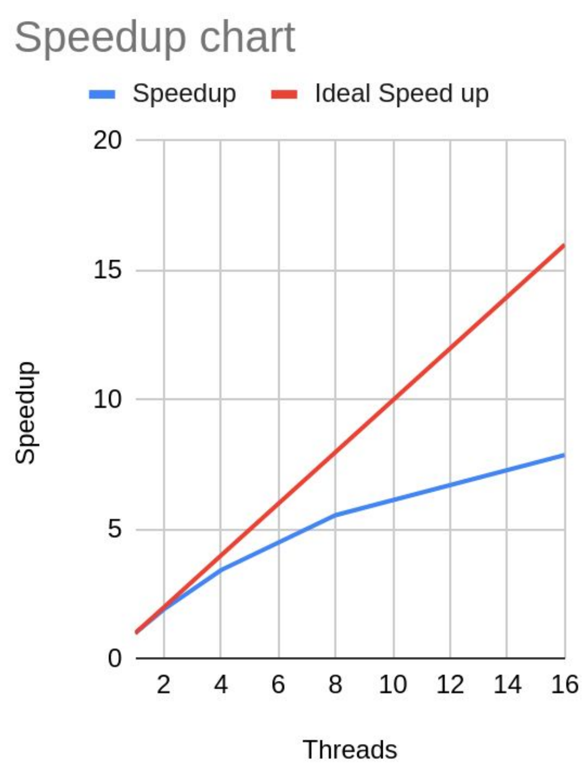
5	24.856	12.509	6.978	4.277	3.265
6	25.579	12.475	7.042	4.289	3.067
7	26.273	12.49	6.912	4.292	2.983
8	28.851	12.457	6.936	4.291	3.349
9	31.126	12.474	6.908	4.229	3.357
10	33.766	12.491	6.879	4.254	3.229
11	25.51	12.449	6.913	4.281	3.225
12	23.615	12.318	7.02	4.244	3.216
13	23.508	12.432	6.845	4.293	3.38
14	24.073	12.588	6.962	4.34	3.243
15	23.759	12.441	6.933	4.281	3.21
16	24.415	12.555	6.981	4.247	3.234
17	23.985	12.516	6.851	4.353	3.232
18	23.871	12.275	6.886	4.257	3.256
19	24.096	12.413	6.969	4.29	3.208
20	23.97	12.612	6.932	4.286	4.5

Threads	Time	Speedup	Efficiency
1	23.508	1	1
2	12.275	1.915112016	0.9575560081
4	6.845	3.434331629	0.8585829072
8	4.229	5.558760936	0.694845117
16	2.983	7.880657057	0.492541066

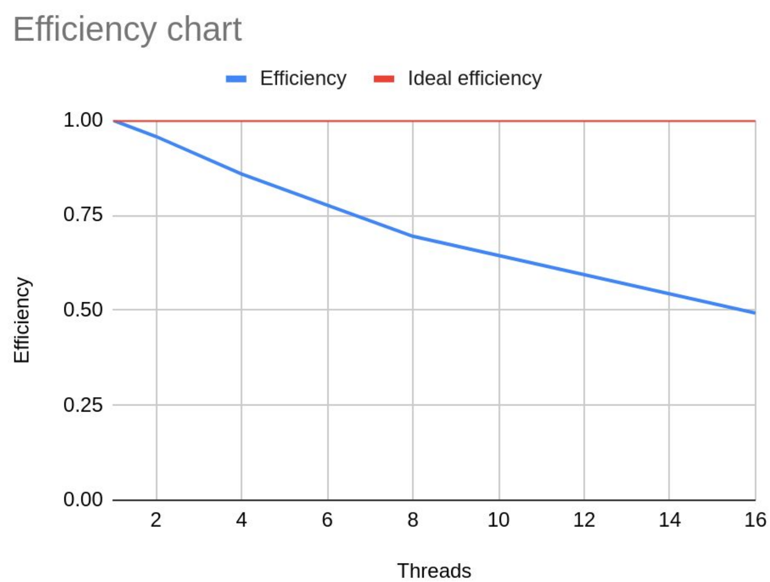
(фиг. 25)


Time chart
Time
25
20
15
10
5
0
Time (seconds)
2
4
6
8
10
12
14
16
Threads

(фиг. 26)


Speedup chart
Speedup
Ideal Speed up
20
15
10
5
0
Speedup
2
4
6
8
10
12
14
16
Threads

(фиг. 27)



(фиг. 28)

Тук виждаме подобна тенденция, както при тестовете на **Машина 1 -** добра ефективност, която спада, при доближаването на хардуерния лимит на системата. Важно е да се отбележи, че тук е възможно да играе роля и балансирането на нишките от самото ядро на операционната система, между двата процесора в системата. Дори и на 8 нишки, ефективността ни е около 70%, което говори за сравнително добро балансиране на задачите.

Посоки за развитие

Има някои неща, които биха могли да се направят, с цел получаване на още по-добри резултати. Едно от тях е промяна на параметъра на грануларност - интересно би било да се изследва дали по-голям или по-малък размер на задачите ще има ефект върху ускорението. Друго нещо, което може да представлява предмет на изследване е друг начин за изчисляване на членове на редицата - не рекурентно, а както са във формула (1) - там все още седи проблемът за сложността при изчисляване на факториелите, но и те могат да бъдат пуснати за паралелно изчисляване, което би увеличило броя на нишките.

Резултати и код:

Кодът на програмите може да бъде намерен на следните адреси:

- SimpeJava: https://github.com/lyubolp/multithreaded-pi-calculation/tree/dev/src/com/company
- RecurrentJava: https://github.com/lyubolp/multithreaded-pi-calculation/tree/master/src/com/company
- RecurrentRust: https://github.com/lyubolp/multithreaded-pi-calculations-rust/tree/recurrent-rust
- FinalRust: https://github.com/lyubolp/multithreaded-pi-calculations-rust/tree/master

Пълните резултати могат да бъдат намерени тук

Източници:

[1] Parallel Design Patterns-L01 - © EPCC, The University of Edinburgh, www.epcc.ed.ac.uk

[2] 16.1. Threads - Carol Nichols и Steve Klabnik - The Rust Programming Language

[3]https://pdfs.semanticscholar.org/44a7/96b9a01c2adc6b7978359f3cdc10356e03ce.pdf - P. M. Kogge - Parallel Solution of Recurrence Problems