Софийски университет „Св. Климент Охридски“

Факултет по математика и информатика

Пресмятане на e

(v3)

КУРСОВ ПРОЕКТ

ПО

РАЗПРЕДЕЛЕНИ СОФТУЕРНИ АРХИТЕКТУРИ

Изготвил:

Милка Ферезлийска

Фак. номер: 61710

спец. Софтуерно инженерство

Проверили:

проф. В. Георгиев: .....

ас. Хр. Христов: .....

Дата: 19.06.2016г.

1. Увод
    a. Описание

Едно важно за математиката число е *Неперовото число (Ойлеровото число)*, тоест числото *e*. Използвайки сходящи редове, лесно може да се пресметне стойността на *e* с произволно висока точност.

Настоящият документ описва пресмятането на *Неперовото число*, използвайки един от сравнително бързо сходящите към *e* редове:

$$e=\sum (3k)\ 2 + 1\ (3k)!\ ,\ k = 0,\ldots,\infty$$

b. Цели и предназначение

Настоящият проект е предназначен за задължителната за специалност Софтуерно инженерство във Факултета по математика и информатика на СУ "Св. Климент Охридски" дисциплина "Разпределени софтуерни архитектури". Целта му е да изчислява приближение на *Неперовото число*, използвайки алгоритъм, разпределен между много процеси. За целта е използван езика за програмиране Python[1] (използва се C имплементация на езика) като е запазен оригиналния алгоритъм за пресмятане на сума, записан по-горе.

c. Командни параметри

Приложението приема опционални параметри от конзолата, които позволяват управление на процеса по пресмятане и на съхраняването на крайния резултат. Когато параметрите не са упоменати, приложението използва стойности по подразбиране. Пълният набор на параметри е следният:

```
$ python3.4 main.py -h
```

```
usage: main.py [-h] [-q] [-p iterations_count] [-t cpu_count] [-d
digits_precision] [-o filename]
```

Следва подробно описание на командните параметри:

- `-h, --help` - Визуализира съобщение за помощ и излиза от програмата. Съобщението съдържа пълна информация за броя, вида и значението на всички параметри.

[1] Навсякъде в документа под Python се има предвид CPython (имплементацията на езика на C).

- `-q` - Указва "тих" режим на пресмятане, по време на който не се извеждат междинни резултати при пресмятанията, а само крайният резултат и времето, което е отнело изпълнението на програмата. По подразбиране тази опция е изключена.
- `-p iterations_count` - Специфицира броя итерации, които ще се изпълнят. Въведеното число се приема от програмата като целочислен параметър. Ако параметърът не е подаден при изпълнение, се използва 2 000.
- `-t cores_number` - Специфицира броя на използваните ядра на процесора[2] при пресмятането. Ако не е експлицитно зададен параметър, се използва броя на физическите ядра на процесора на машината.
- `-d digits_precision` - Специфицира броя на цифрите след десетичната запетая, използвани при изчисленията. Ако не е експлицитно зададен, се приема 20 000.
- `-o filename` - Специфицира името на файла, в който ще се запази изчисления резултат за е. Стойността по подразбиране на този параметър е "`result.txt`".

За да се изпълни програмата със стойностите по подразбиране, се извиква по следния начин:

```
$ python3.4 main.py
```

Ако искаме да специфицираме броя на използваните ядра да е 3, броя на цифрите след десетичната запетая да е 20, броя на итерациите да е 200 и файлът, в който ще се запише крайния резултат за *e* да се нарича `my_file.txt`, то ще я извикаме по следния начин:

```
$ python3.4 main.py -t 3 -d 20 -p 200 -o my_file.txt
```

d. Извеждани съобщения

По време на изпълнението си, програмата извежда различни пояснящи съобщения:

- `Init Process ForkPoolWorker-1` - Съобщение за стартирането на нов клониран процес. Пренебрегва се в "тих" режим на пресмятане.
- `process <ForkProcess(ForkPoolWorker-1, started daemon)> is calculating calculate_current with index 0` - Съобщение, което пояснява, че процес №1 е стартирал като дъщерен и извършва изчисление на нулевия(текущия) член от сумата. Пренебрегва се в "тих" режим на пресмятане.

[2] Има се предвид също и брой процеси, защото в CPython на 1 ядро на процесор може да се изпълнява само 1 процес.

- `RESULT: 2.71828182845904530721` - Съобщение за извеждане на екрана на финалния резултат. Пренебрегва се в "тих" режим на пресмятане.
- `Number of started processes: 3` - Съобщение за броя на стартираните процеси. Пренебрегва се в "тих" режим на пресмятане.
- `Total execution time: 0.020379304885864258` - Съобщение за времето, което е отнело пресмятането на *e*. Не се пренебрегва в "тих" режим на пресмятане.

e. Функционални характеристики

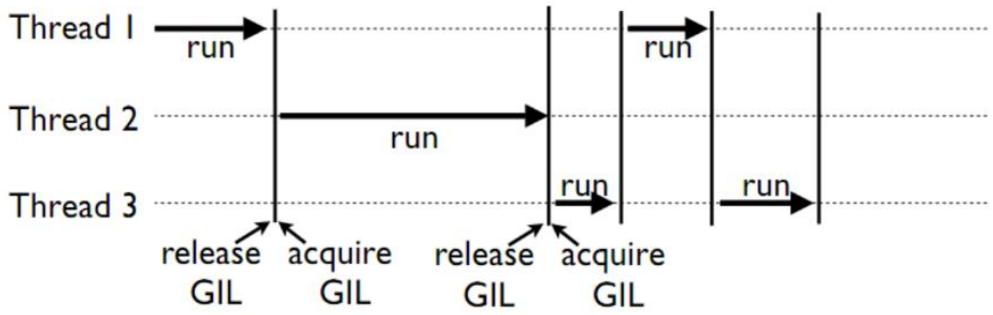
i. Кратко описание на алгоритъма

Програмата е реализирана като 1 основен процес създава pool с клонирани процеси, които ще вършат част от смятанията. Използва се асинхорнно изпълнение на пресмятането, което позволява процесите в pool-а сами да решават кога кой процес кое изчисление под ред ще вземе. Всеки процес взима номера на следващата итерация, пресмята събираемото и го връща. Върнатото число се събира с натрупаната сума до момента и когато се достигне указания от потребителя брой итерации (или стойността по подразбиране, ако не е указан), натрупаната сума се записва в изходен файл и/или се извежда на екрана.

Съществуват 2 версии на програмата – `smart_pool_with_variables.py` и `smart_pool_without_variables.py`. Разликата в тях е видът на променливата, в която се натрупва сумата. В първата версия сумата е имплементирана като споделена променлива между процесите в pool-а, а във втората – глобална променлива, която се обновява от всеки процес.

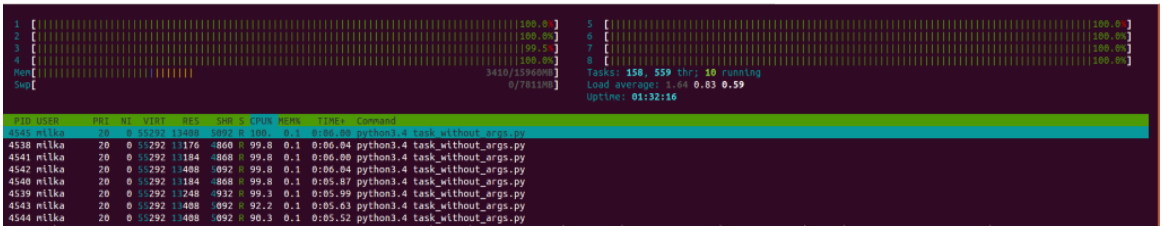
ii. Специфични характеристики на езика, указващи влияние върху имплеметацията на алгоритъма

За разлика от Jython(Python, написан на Java) и IronPython(Python, написан на .NET), CPython използва нишки, които не са симулирани, а са нишки на операционната система. Важна особеност на тази имплементация на езика е наличието на т. нар. Global Interpreter Lock (GIL), който гарантира, че в даден момент само 1 нишка може да общува с обектите в Python. Спецификата на Python определя, че 1 процес работи само върху 1 ядро на процесора. Така, ако се създадат много нишки, те ще се изпълняват само на 1 ядро. Дори и да се създадат много нишки на Python, те няма да работят паралелно, а конкурентно. По определен от операционната система начин нишките ще си предават управлението без да работят паралелно. Така, ако се създаде многонишкова програма на Python не само че няма да се наблюдава ускорение от многото нишки, но и ще се получи забавяне, защото големият брой нишки затруднява още повече процеса по предаване на управлението. Процесът е илюстриран на фигурата по-долу.



Използването на много процеси води до използването на няколко Python интерпретатора паралелно като всеки от тях работи на отделно ядро на процесора. Така всеки процес разполага със своя собствена памет и GIL и се изпълнява самостоятелно докрай. Това е най-лесният начин да се постигне ускорение на задачи, свързани с ядрата на процесора.

В даден момент, когато се създадат повече процеси, отколкото са налица ядра на тестваната машина, пак ще трябва да се изчаква някой от процесите да завърши работата си, за да започне следващият. Едновременната работа на 8 процеса върху 8 ядра е визуализирана на снимката по-долу.



f. Качествени характеристики

i. Анализ на източниците

Начинът на управление на клонираните процеси, както и изборът за създаването на общ pool с тях, е повлиян от книгата **'High Performance Python'**. В нея подробно е описано какво е GIL, как влияе на изпълнението на многонишкови процеси и каква е разликата в изпълнението на много процеси спрямо много нишки.

Идеята за използването на pool, в който процесите си разпределят сами работата асинхронно, е породена от документацията на използвания **multiprocessing** модул. В нея се съдържа подробно описание на всички функционалности, които предлага, както и много разнообразни примери за онагледяване на описанието. В програмата са използвани 2 модула от него:

- **Process** - създава дете-копие на текущия процес. Копието (процесът дете) е с нов идентификатор и се изпълнява независимо от родителския процес в операционната система. Процесът дете приема състояние и/или променливи, с които да работи, както и методи, които да изпълнява.
- **Pool** - Обвива **Process** или **threading.Thread API** в pool от изпълняващи процеси / нишки, които споделят части от общата работа, и който връща агрегиран резултат. В програмата е използван за създаването на процеси деца.

  Спицификата на програмата изисква работа с много

2. Проектиране
    a. Функционално проектиране
        i. Архитектура

Архитектурата на приложението се състои от 4 функции - `main`, `init_process`, `calculate_current` и `add_current`, както и 2 глобални променливи, които се използват от всички процеси - `IS_QUIET` и `PREC_COUNT`.

Управлението на програмата е съсредоточено в `main` функцията. В нея се дефинират, приемат и обработват всички параметри от конзолата, създава се pool за клонираните процеси, раздават им се задачи и се изпълняват. След като приключат изчисленията резултатът се записва в специфициран от потребителя файл или в нов файл с име по подразбиране и се извежда времето за изпълнение на пресмятанията.

Функцията `init_process` се извиква при създаването на нов клониран процес. Нейната цел е да уведоми потребителя за създаването на нов процес дете. Когато потребителят използва програмата в "тих" режим, функцията няма да отпечата нищо на конзолата.

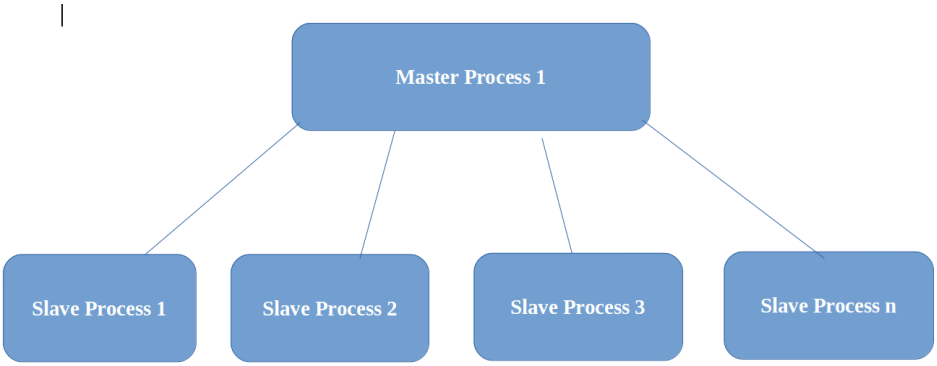
Функцията `calculate_current` се изпълнява многократно от всички процеси, тъй като тя пресмята текущия член от сумата. Пресметната стойност се подава на callback функцията `add_current`, която се грижи да натрупва събираемите в глобалната променлива `final_res`.

Глобалната променлива `IS_QUIET` указва дали програмата ще работи в "тих" режим или не, а `PREC_COUNT` предварително определяне на броя цифри за съхраняване на променливите в програмата.

Във версията `smart_pool_without_variables.py` `final_res` съхранява финалния резултат от всички изчисления, а в `smart_pool_with_variables.py` това се извършва в общата им променлива. За да се осъществи споделянето на променлива, се използва файл, който я инициализира.

ii. Модел на паралелизма

Програмата е реализирана с помощта на техниката SPMD (Single Program Multiple Data) и архитектурата Master – Slave, визуализиран на фигурата по-долу:



Програмата се стартира в главния процес, който от своя страна създава pool още *n* на брой процеси, които да извършват изчисленията и да му доставят нови данни, които да обработва. Pool-ът разпределя работата измежду процесите и те я изпълняват асинхронно и lazy, тоест изпълняват се в момента на извикването на резултата им, до тогава те просто взимат част от работата и знаят, че ще е тяхна задача да я изпълнят.

iii. Модел на декомпозиция на данните

Главният процес създава pool, който съхранява клонираните процеси, които изчисляват на всяко свое извикване текущия член от сумата. Полученият резултат го предават на callback и в него се прибавя стойността му към натрупаната до момента обща сума за стойност на *e*. Общата сума се съхранява в глобалната променлива `final_res = decimal.Decimal(0)`.

iv. Алгоритъм

В `main` функцията се дефинират, приемат и обработват всички параметри от конзолата, създава се pool за клонираните процеси. При всяко създаване на нов процес се извиква функцията `init_process`, която уведомява потребителя, че се стартира нов процес:

```
pool = multiprocessing.Pool(
    cores_number, initializer=init_process)
```

След това всички изисления, които трябва да се направят, се дават на `pool` обекта. От своя страна той ги разпределя измежду процесите деца, които съдържа. Децата ще извикват многократно функцията `calculate_current` с входен параметър номер на текущата итерация. Когато приключи изпълнението си, `calculate_current` връща полученият резултат за сегашния член на сумата. Резултатът се поема и обработва от

функцията `add_current`, която натрупва данни във финалната сума `final_res`. `Всичко това` ще се изпълни асинхронно с реда:

```
jobs = [pool.apply_async(calculate_current, [i],
     callback=add_current) for i in range(iterations_count)]
```

След това се извикват разпределените във времето и между процесите jobs и се получава крайния резултат, запазен в променливата `final_res`:

```
for x in jobs:
    x.get()
```

Следващата стъпка от изпълнението на програмата е записването на крайния резултат във файл с име, въведено от потребителя или с име по подразбиране:

```
with open(filename, 'w') as opened_file:
    opened_file.write(str(final_res))
```

b. Нефункционално проектиране

i. Модел на комуникация между процесите

Програмата използва много клонирани процеси, които се характеризират със своя собствена памет. Когато извършват някакви изчисления, те ги предават по pool-а, чрез който си комуникират. По този начин pool-ът, който е стартиран в главния процес, ги управлява - раздава им задачи и обработва резултатите им.

3. Тестване, преносимост и настройка

a. Измерване и тестване

Програмата е тествана на сървъра `ats24.rmi.yaht.net`, разполагащ с 24 ядра, като проведените тестове са за брой процеси **p** от 1 до 24, брой членове на реда **t** 10 240.

За сравняване на резултатите от проведените измервания, се използва Закона на Амдал, чрез който може да се отчете теоретичното ускорение при изпълнението на една програма, когато се подобрят ресурсите й. За целта ще използваме следните променливи, които са по-лесно измерими спрямо променливите в Закона на Амдал:

- **T1** - времето за изпълнение на програмата, когато се използва само 1 процес, тоест всички инструкции се изпълняват последователно.
- **p** - брой процеси, които извършват пресмятанията в програмата.
- **Tp** - времето за изпълнение на програмата, когато се използват **p** процеса едновременно.

- **Sp = T1 / Tp** - ускорението, което се наблюдава, при изпълнение на програмата с **p** процеса.
- **Ep = Sp / p** - ефективността на програмата, когато се използват **p** процеса за изпълнението ѝ.

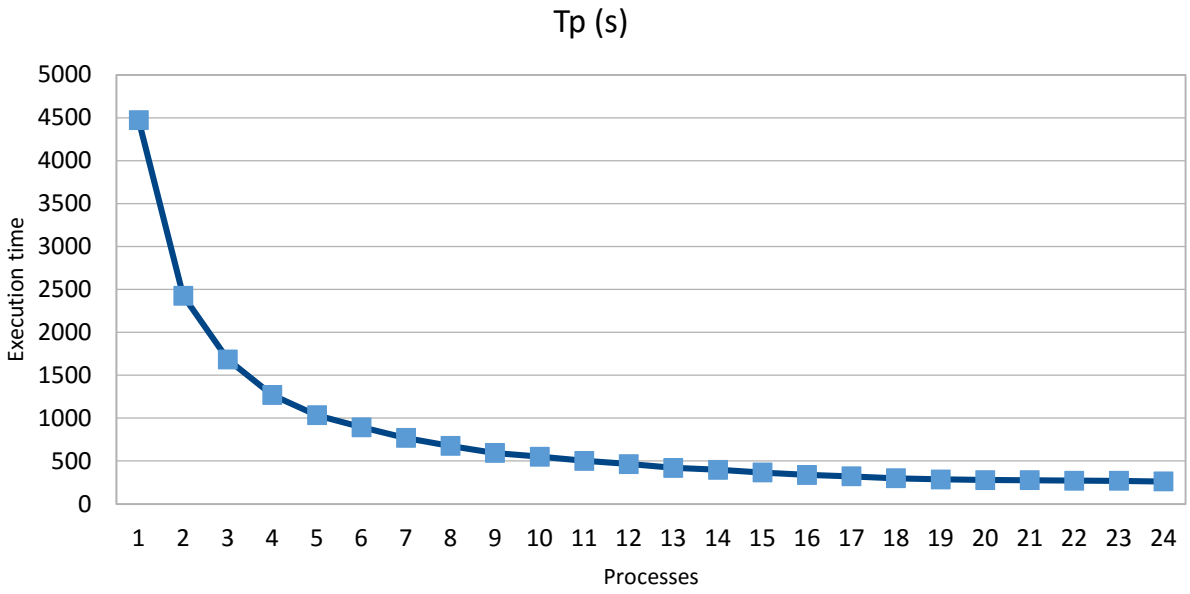
b. Резултати от тестването в табличен вид

В таблицата са използвани описаните по-горе променливи **t**, **p**, **T1**, **Tp**, **Sp** и **Ep** със съответните стойности от проведените тестове:

t	p	T1(s)	Tp(s)	Sp	Ep
10240	1	4472.689517	4472.689517	1	1
10240	2	4472.689517	2427.319326	1.842645699	0.92132285
10240	3	4472.689517	1683.240477	2.657189853	0.885729951
10240	4	4472.689517	1268.875104	3.524924952	0.881231238
10240	5	4472.689517	1034.12126	4.325111269	0.865022254
10240	6	4472.689517	893.3697426	5.006537947	0.834422991
10240	7	4472.689517	768.5554085	5.819605805	0.831372258
10240	8	4472.689517	676.7079649	6.609482597	0.826185325
10240	9	4472.689517	593.2623801	7.53914232	0.83768248
10240	10	4472.689517	550.3036301	8.127675836	0.812767584
10240	11	4472.689517	502.9778976	8.892417615	0.808401601
10240	12	4472.689517	466.0775599	9.596448963	0.79970408
10240	13	4472.689517	420.3012362	10.64162827	0.81858679
10240	14	4472.689517	397.5769746	11.2498706	0.803562186
10240	15	4472.689517	366.2685802	12.21150205	0.814100136
10240	16	4472.689517	338.7446556	13.2037198	0.825232488
10240	17	4472.689517	321.692656	13.90361089	0.817859464
10240	18	4472.689517	298.6559443	14.97606059	0.832003366
10240	19	4472.689517	286.4285495	15.61537607	0.821861898
10240	20	4472.689517	279.4153869	16.00731286	0.800365643
10240	21	4472.689517	276.5888562	16.1708956	0.770042647
10240	22	4472.689517	270.6269407	16.52714066	0.751233667
10240	23	4472.689517	270.1743951	16.55482384	0.71977495

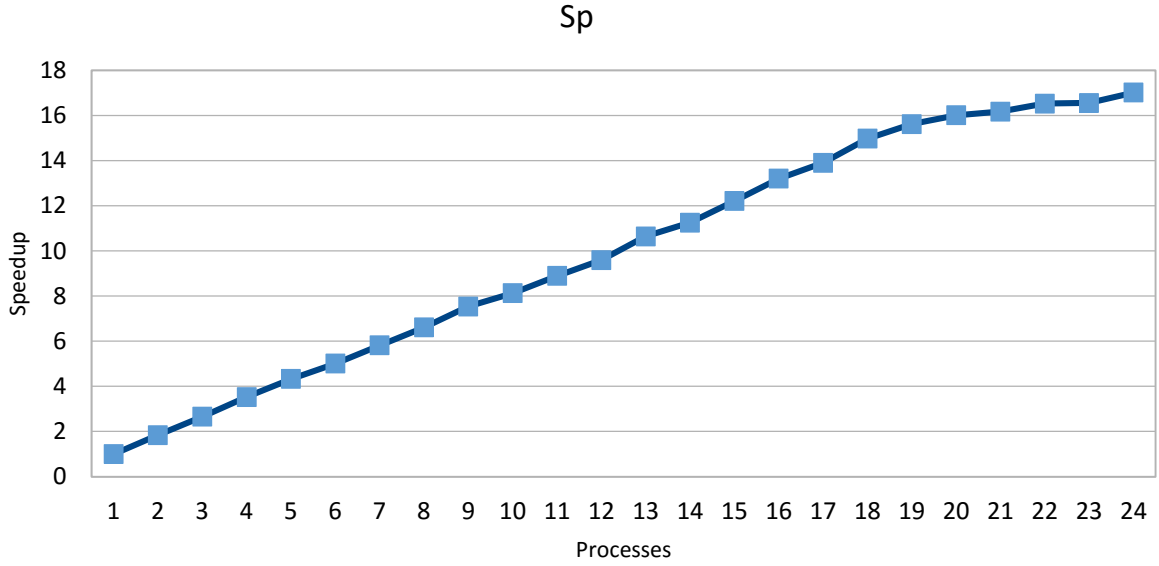
c. Диаграма на времето за изпълнение на паралелната програма - **Tp**

Данните от таблицата за времето за изпълнение на програмата при различен брой използвани процеси са визуализирани на графика от тип Line Chart:



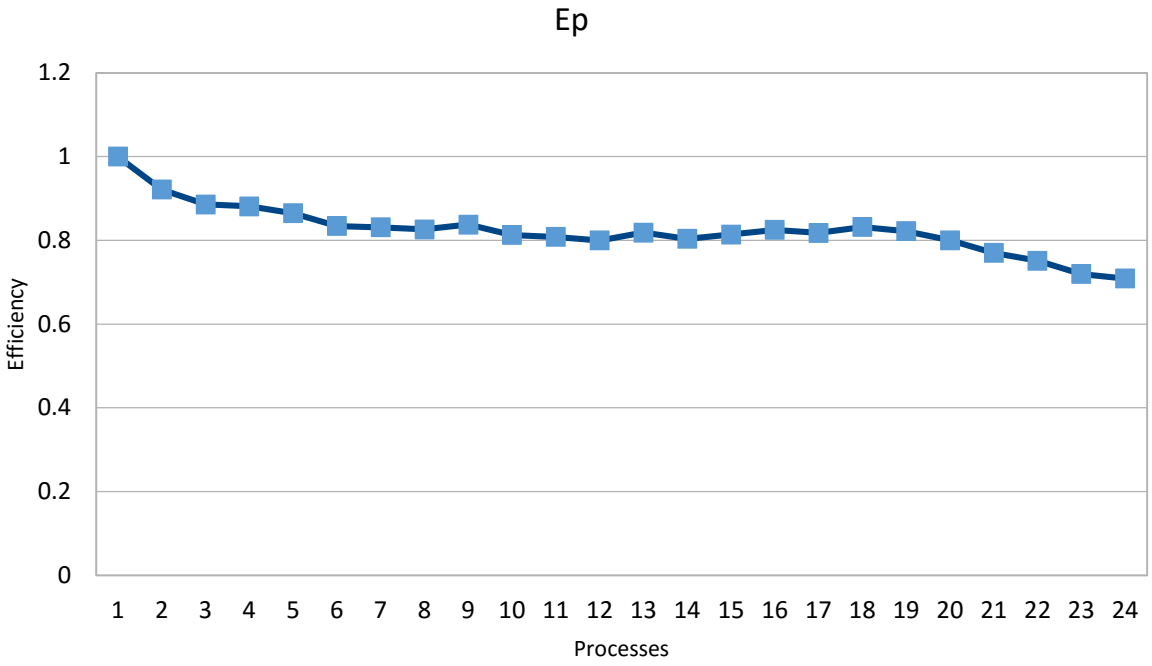
d. Диаграма на ускорението – **Sp**

Данните от таблицата за наблюдаваното ускориние при изпълнение на програмата с различен брой използвани процеси са визуализирани на графика от тип Line Chart:



e. Диаграма на ефективността – **Ep**

Данните от таблицата за ефективността на програмата при изпълнението ѝ при различен брой използвани процеси са визуализирани на графика от тип Line Chart:



4. Източници

- Книга High Performance Python (2014г.): **http://shop.oreilly.com/product/0636920028963.do**
- Документация на библиотека **multiprocessing**:

  https://docs.python.org/3.4/library/multiprocessing.html