

Проект

по Разпределени Софтуерни

Архитектури

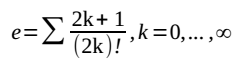
изготвил:

Андрей Ганчев Андреев

Ф.Н: 62173

Тема на проекта.

Темата на проекта е паралелното смятане на числото „e“, използвайки следната формула:



Анализ на решението

Основна терминология:

* + p – прецизност(десетичен знак) за точност при пресмятането на числото „е“.
  + t – брой паралелни процеси, които могат да работят едновременно

Формулата има два основни компонента. Първата е факториела и втората сумата. За големи числа дължината на [n!] е около

от nln(n), на базата на това 1/n! Има прецизност поне n символа за големи n. За това ние можем да подсигурим това, че 1/(2k!) със сигурност има прецизност 2к, за дори и не толкова големи стойности на к, като за достатъчно големи стойности даже

пресмятането на само половината елементи е достатъчно за пресмятане с прецизност n. За по-малките n, може да се получи разминаване за това се презастраховаме като добавим прецизност +5 за сметките и после ги отрежем.

Тъй като целта на алгоритъма е да работи оптимално при нарастването на t, а не толкова константата, която се получава за някое p -> Т(1), най-лесно е ще да вземем условно броят на елементите, които да смятаме за p или поне близост на p //(p+5). Нататък ще реферирам, че ще ни трябват p итерации, въпреки че може и да са в околност на p.

След като вече знаем това, то трябва да изберем подходящ начин за паралелното смятане на тези p събираеми от сумата. Идеята е да можем да разделим този range [0,p], по подходящ начин, така че да минимизираме разликата в работата на всички t нишки. Един начин е това да разделим множеството на “v” подмножества и всяко подмножество да бъде сметнато от една нишка:

[0,p] = [0,x1][x1,x2][x2,x3]...[x(v-1),p](интервали). Важно е тези интервали да са последователи поради начина, по който ще сметнем факториелите(по-късно ще обясня). Ако вземем тези x-ове равномерно дистрибутирани, то някои интервали ще бъдат сметнати по-бързо от други, тъй като в сложността на смятане влиза и големината на числото. Нормалните алгоритми смятат умножение и деление със сложност O([a][b]), тоест при по-големи a,b-та имаме повече сметки. С елементарен анализ можем да видим, че броят операции, които са ни нужни да сметнем p! Е около

O(ln(p)^2 \* p^2). Това можем да се справим с два начина(поне). Единият е v да е по-голям от t. То тогава ще можем да минимизираме работата на всички t нишки, но при прекалено голямо v, нишките започват да се бутат за задачи и работят по-бавно. Също така скоростта за смятане се намалява за една нишка, когато работата е разделена на повече интервали, тъй като сметките се получават по-малки поради факта, че с повече интервали имаме по-къси числа за един интервал. Опита показа, че при избиране на подходящи стойности на “v” позволява бууст в перформънса, но това намалява отношението T(1) и T(32) и за това ще направя паралелна съпоставка, когато v=t и когато v е подходящо избрано.

Вторият начин, който само ще кажа на идейно ниво е да намерим тези x-ве при които са равномерно разпределени на t части, така че всеки интервал [xi,x(i+1)] да има еднаква стойност спрямо функцията f(x) = x^2 ln^2(x).(f([a,b]) = f(b)- f(a)). Ако трябва да съм честен домързя ме да го пробвам, но на хартия се получават добри съотношения, спрямо време на работа на нишките. Реално предполагам, че смесица от двете идеи би било най-добре.

След като имаме установени интервалите, ние трябва да намерим стойностите за всеки един интервал. За целта процесът на пресмятане се разделя на 4 части, като 2 от тях са паралелни и 2 върху една нишка.

Първа фаза:

Всяка нишка хваща интервал [a,b]. Итерира през него, като има две числа. Едното е текущият факториел, другото текущата сума за интервала. Интервалите трябваше да са последователни, защото в тази фаза не се смята реално целият факториел, а само частта, която започва от 2а. След това във втора фаза ще умножим всяка сума с факториела от този в минал интервал и така ще получим сумата без да трябва да извършваме всички операции по v пъти в нишките. Така след първа част имаме стойността:

Sum[a,b] = (2a+1)/2a + … + (2(b-1)+1)/ (2(b-1)+1)\*...\*2a

Втора фаза:

Във втората фаза се допълват факториелите, като от първа фаза вече имаме частичните факториели.

Трета фаза:

В трета фаза пак паралелно се извършват умноженията на факториелите преди интервал [a,b] със сумата от интервал [a,b]. Тук важно уточнение е, че паралелно се разработват само t таска, а не v, като при първа фаза, защото при голямо v бихме имали bottleneck в скороста, тъй като тези паралелни операции са сравнително бързи(само по едно умножение).

Четвърта фаза:

В четвърта фаза се събират различните подсуми в една.

След тези четири фази числото, което се получава е “e” с точност до p-тия знак.

Имплементация

Имплементирал съм идеята си на java 8, използвайки Apfloat за калкулация на стойностите. За паралелно решение съм използвал ForkJoinPool, който отлично се съчетава с parallel stream api-а на Java. Така лесно постигнах бързо и безопасно закачане на таскове. Използвал съм мой ThreadFactory с цел мониторинг на нишките. Също така имам допълнителен мониторинг клас, който позволява по-функционален начин на мониторване на скорост на различните етапи от програмата. Имам и тестване за коректност на отговора спрямо файл с коректната стойност на “e”. Имплементацията на алгоритъма е сходна с написаното горе. Главният клас Context се използва за запазване на параметрите подадени от конзолата. Calculate класа се използва за изчисление на “e”. Range класа абстрахира по-сложната логистика с разбиване на интервал в подинтервали. Чрез него генерирам set от интервали, които обработвам чрез паралелната библиотека, за която споменах.

Забележка: Въпреки, че създавам t нишки и така с главната имам t+1, то тя почива и чака другите t да свършат, тъй че не работят повече от t нишки едновременно.

Параметри

-t <num> = брой на нишките

-p <num> = точност на калкулацията

-o <filename> = име на файл за изпечатване на e

-f <filename> = име на файл за тестване на вярност на решението

-r = още по-детайлна информация за създаването на нишки и коя нишка кой range обработва

-q = mute

-v <num> = броят на таскове, които да бъдат създадени, всъщност множител по който се умножава t за да се разбере колко таска ще има. Например t=2 и v =2 ще се генерират 4 таска. Нормалната стойност е 1ца.

Анализ на скоростта

**При -v = 1 (създаваме t таска):**

*Забележка: анализът е направен за p = 10,000, информацията е изготвена посредством средно аритметично от 5 пускания на програмата.*

*~~~~*

Стойностите са в секунди. 32те нишки са 27пъти по-ефективни от 1на. При малки стойности на -v фаза 1 показва доста по-лоши стойности от фаза 3. Целта е да намерим за дадена двойка (p,t) оптимално -v, такова че сумите на фаза 1 и 3 да са оптимални. При наблюдение може да се види, че оптималност се достига при (-v)\*t ~ p/50

**При -v = v0 оптимално(създаваме v0\*t таска):**

*Забележка: анализът е направен за p = 50,000,v0=p/(50\*t) информацията е изготвена посредством средно аритметично от 5 пускания на програмата.*

~~~~

Вижда се, че алгоритъма работи по-слабо ефективно при нарастване на нишките, но работи с пъти по-ефективно за по-големи p. Също така се забелязва, че фаза 3 е доста по-бавна, тъй като там range-овете се разделят само на t, но ако се разделяха на v, щеше да е по-лошо.

В сравнения с по-гориня анализ:

При -v = 1, и t=32 щеше да отнеме приблизително 120сек. За 50,000 елемента, което означава, че за 1 нишка щеше да отнеме приблизително 3000 секунди, или почти час за същото p, за което алгоритъма го сметна за 226 секунди.

Заключение

Ясно е, че алгоритъма не е максимално оптимален. Има доста имплементационни детайли и различни техники, като тези които дадох като идея по-горе. Все пак алгоритъма е сравнително добре и може в разумно време да смята за стойности на p над 150-200 хиляди.