Wilson Primes

Student: Brincoveanu Vasile Vlad gr:30241

Numerele prime Wilson sunt acele numere prime p care satsifac p*p divide (p-1)!+1, unde “!” inseamna functia factoriala.

Exista si o teorema a lui Wilson, care spune ca orice numar prim p divide (p-1)! +1.

Singurele numere prime Wilson cunoscute sunt 5, 13, 563. Dar, s-a demonstrate ca exista o infinitate de numere cu propietatea aceasta.

Ps: Chiar si **Numberphile** a facut un video despre aceste numere, pe care il recomand ! [1]

PRECONDITII GENERALE

Algoritmul creat se foloseste de urmatorii pasi:

-pentru intervalul [A,B] si incrementul C verifica care numere sunt prime, folosind tehnici descries mai jos de optimizare.

-din numerele prime verificate mai sus, se verifica care sunt si Wilson, dupa formula

p*p divide (p-1)!+1. (unde p este numarul prim).

-factorialul din formula de mai sus se calculeaza iterativ pentru performanta.

Pentru load balancing pentru threaduri s-a folosit tehnica prezentata in laborator. Pentru **toate testele** am mers pe un interval **[A=1,B=35000].** Unde A variaza in functie de cate threaduri avem.(Max threads 8).

CONFIGURATIE CALCULATOR

PROCESOR: I7 7700HQ 2.8GH (4 CORES, 8 LOGICAL, HYPERTHREAD) MAX GHZ -> 4.0GHZ

RAM: 16 GB DDR4 2144MHZ

STOCARE: M.2 Intel 512GB 1.6GB/s R & W

GPU: NVIDIA GTX 1050TI 4GB RAM

OBSERVATII GENERALE

Nu s-au rulat aplicatiile intr-un mediu optim (s-a lucrat in timpul rularii, antivirus, browsing pe web, deschis multe IDE-uri etc.) De aceea nu vedem un speedup mai mare sau egal cu 4 (adica cate

core-uri are procesorul meu). In mod ideal ar trebui undeva pe la 5,25 speed-up. (4 + 1,25 de la hyperthread).

Se poate observa pe graficele de speedup cat si pe cele de timp de executie, ca cresterea respective scaderea nu este uniforma. Astfel, la **2 threaduri avem aprox acelasi timp ca si cu un thread**.

Explicatia consta in faptul ca fiind 2 threaduri, numerele din intervalul [1,35000] se vor imparti in mod egal la th1 si th2. Astfel, th1 va primii toate numerele impare, iar th2 toate pare. Th2 avand sa gestioneze doar numere pare, el va intra in functia de prim care va returna fals la orice multiplu de doi, fara a mai face alte impartiri costisitoare. S-ar fi putut impartii numerele mai optim, tinand cont de acest lucru, dar per total este ameliorat de cresterea threadurilor, si noi cautam o medie pe crestere, nu individual.

Java.

Prima dificultate a fost implementarea algoritmului serial. Apoi am trecut prin mai multe iteratii de **verificare a unui numar pentru a vedea daca este prim**. Am folosit 10000 de numere.

Prima versiune a fost, impartirea lui cu toate numerele pana la jumatatea sa. -> 8,099 secunde

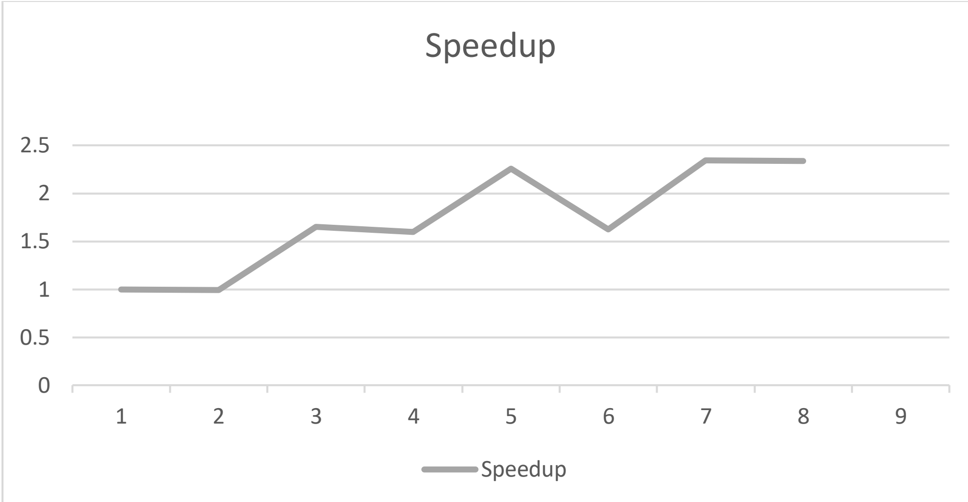
A doua versiune a fost, folosirea tipului BigInteger si metodei **isProbablePrime(**care va returna fals daca e cu siguranta compus, si true daca e probabil prim . Primeste ca si argument posibilitatea de siguranta ca un numar sa fie prim, daca returneaza true probabilitatea de prim este $1 - 1/2^{certainty}$**)** . Am ajustat si iteratia, vom pleca de la i = 2, pana la i * i < numarul, astfel mergem pana la radical din numar, fiind mai eficienta compararea cu divizori. -> 7,917secunde.

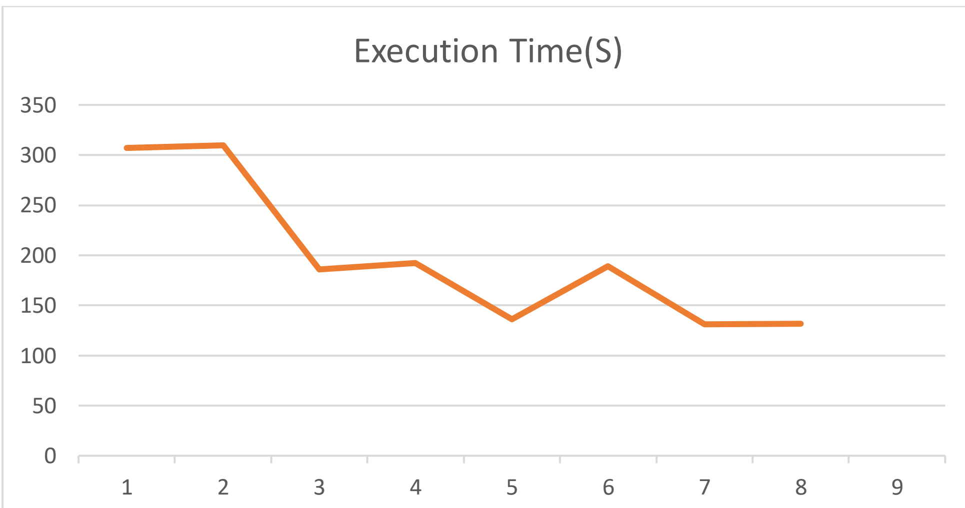
Ultima versiune, verifica pe langa isProbablePrime si daca numarul este par, iar mai apoi iteratia va pleca de la i = 3, cu increment din 2 in 2, verificand doar numerele impare ! -> 7,570

Se va folosi o metoda simpla pentru paralelizarea algoritmului. Pe acelasi interval [A,B] dat, threadurile vor primi numere diferite in functie de un increment dat la inceput…

Vom observa ca cei mai buni timpi sunt atunci cand balansam in mod egal sarcina de lucru pe threaduri. Daca creste numarul de threaduri nu neaparat va scadea si timpul, etc.

PROC	EX TIME(S)	Speedup	Efficiency
1	307.11	1	1
2	309.71	0.99160505	0.495802525
3	185.77	1.65317328	0.55105776
4	192.33	1.59678677	0.399196693
5	136.14	2.25583958	0.451167915
6	188.83	1.62638352	0.27106392
7	131.09	2.34274163	0.334677375
8	131.47	2.33597018	0.291996273



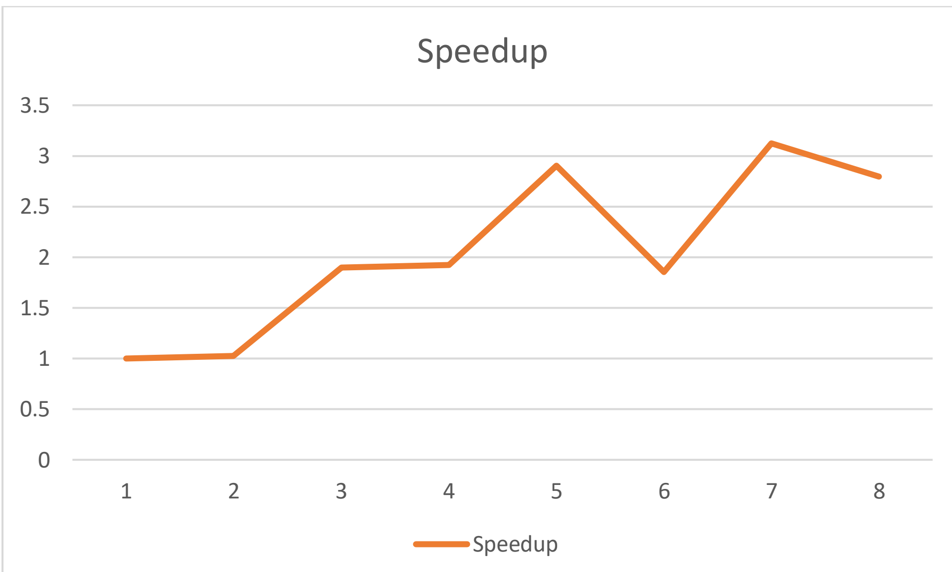


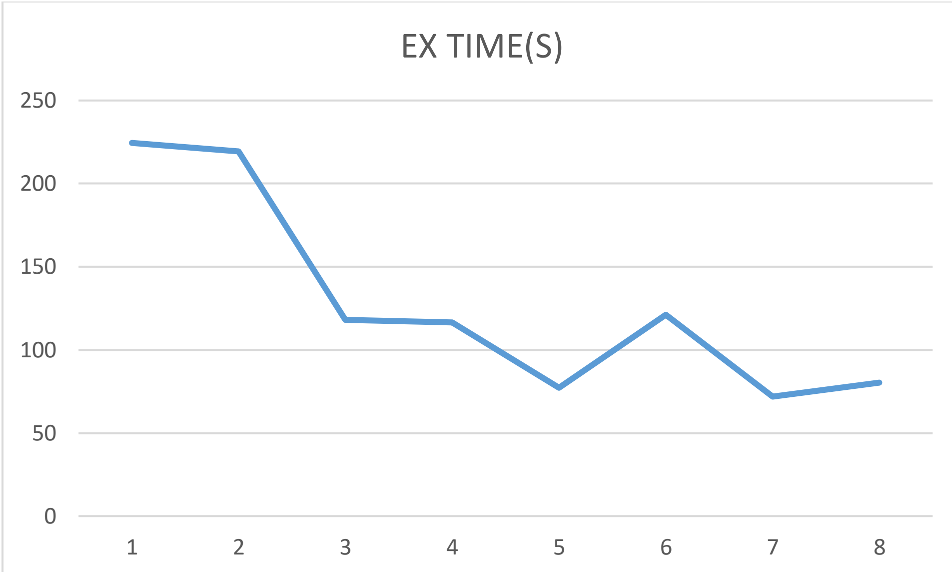
C

Long Long int suporta numere de doar 64 de biti, cea ce este deposit cu mult de un factorial de 100 de exemplu. Astfel am cautat anumite librarii pentru bigDecimals pentru c. Prima librarie gasita a fost InfInt. Principiul acestei librarii este simplu, se tine numarul pe mai multe int-uri intr-o lista inlantuita. **Nu este deloc eficient** ! Ca si comparatie un factorial de 100000 in **java** folosind BigDecimal dureaza aproximativ **2.9** secunde pe masina mea. In aceasta librarie dureaza undeva la **25 de secunde**.

Dupa lungi cautari si testari de librari de pe [2], am ajuns sa folosesc boost, ce foloseste ca si variabila big number -> cpp_int iar operatiile sunt suprascrise peste cele de baza (c++ style), dar nu ne suparam, folosim in continuare doar C. Performanta a crescut considerabil. De exemplu calcularea factorialului de 100000 cu libraria boost a durat **1.8 secunde** !

PROC	EX TIME(S)	Speedup	Efficiency
1	224.48	1	1
2	219.43	1.0230142	0.511507
3	118.21	1.8989933	0.632998
4	116.62	1.9248842	0.481221
5	77.29	2.9043861	0.580877
6	121.15	1.8529096	0.308818
7	71.86	3.1238519	0.446265
8	80.29	2.795865	0.349483





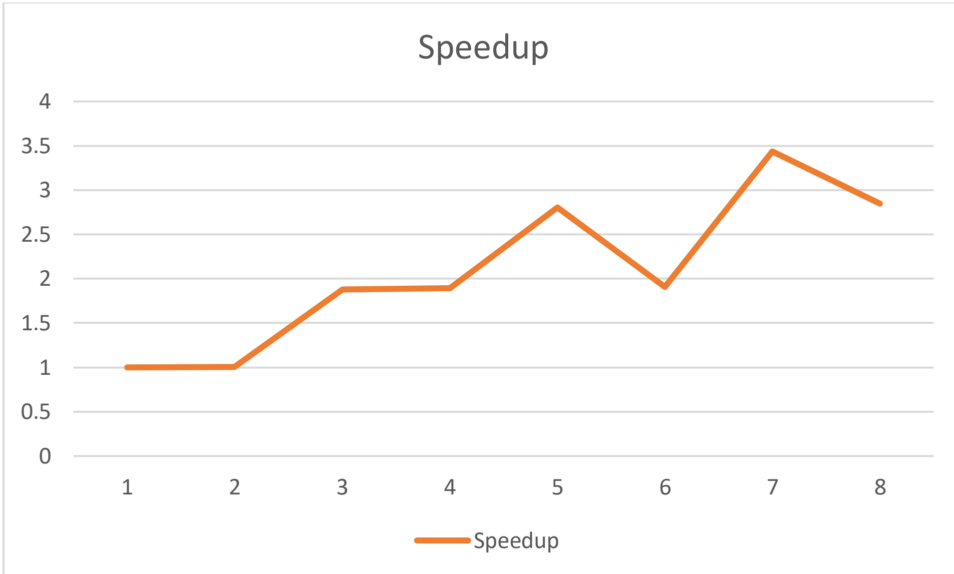
OpenMP

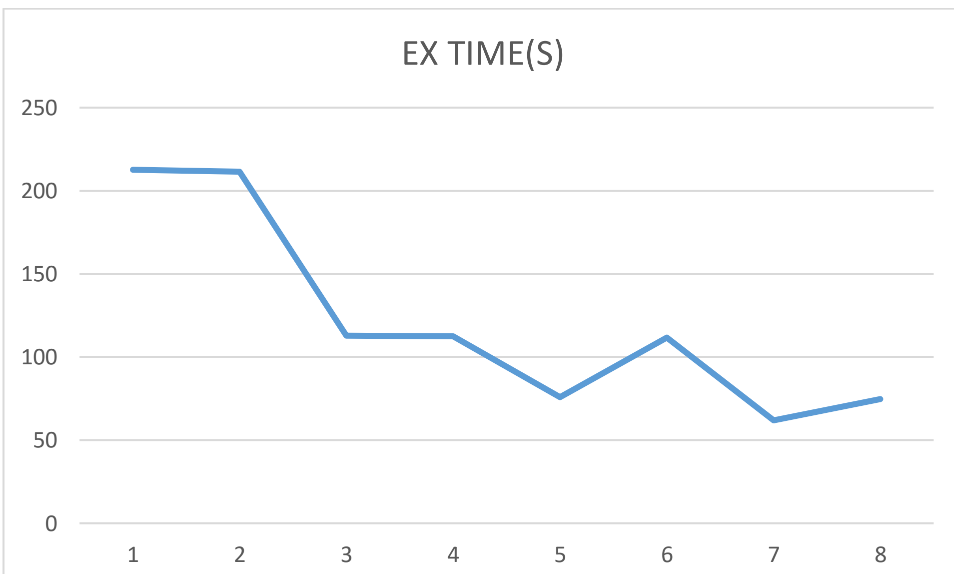
S-a folosit omp_set_num_threads pentru a seta numarul de threaduri.

Pragma omp parallel incadreaza codul ce va fi rulat paralel.

Avem 2 variabile private fiecarui thread. Unul este id-ul sau dupa care se face incrementul, numele, si se salveaza rezultatele in matricea de rezultate, pe linia data de indicele threadului. Values se foloseste pentru a stabili coloana din matricea rezultat, pentru salvarea valorilor, trebuie sa fie private fiecarui thread!

PROC	EX TIME(S)	Speedup	Efficiency
1	212.63	1	1
2	211.58	1.00496266	0.502481331
3	113.01	1.88151491	0.627171637
4	112.48	1.89038051	0.472595128
5	75.87	2.80255701	0.560511401
6	111.62	1.90494535	0.317490892
7	61.88	3.43616677	0.490880968
8	74.72	2.84569058	0.355711322





PROLOG

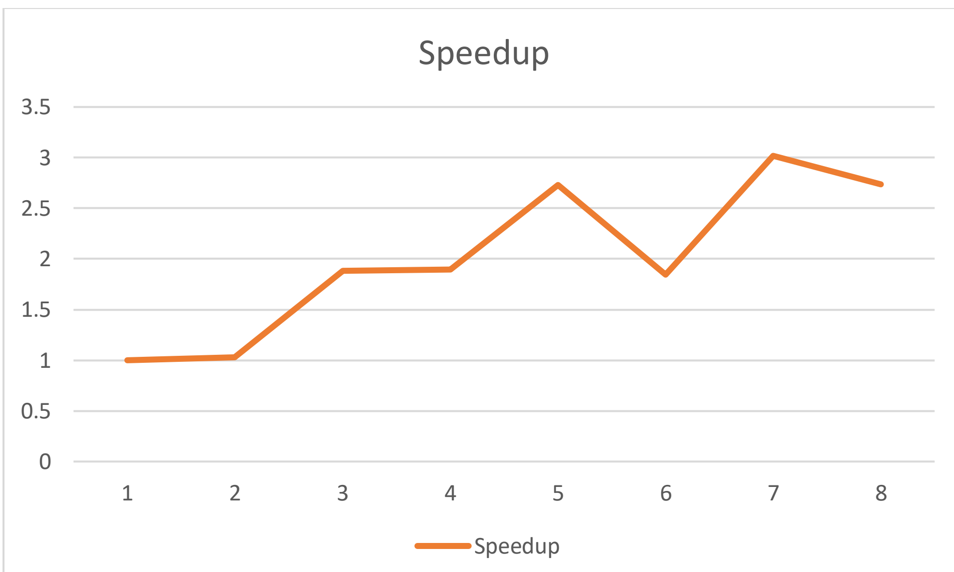
Nu multe optimizari s-au adus aici, fata de logica codului deja optimizata din implementarile predecesoare.

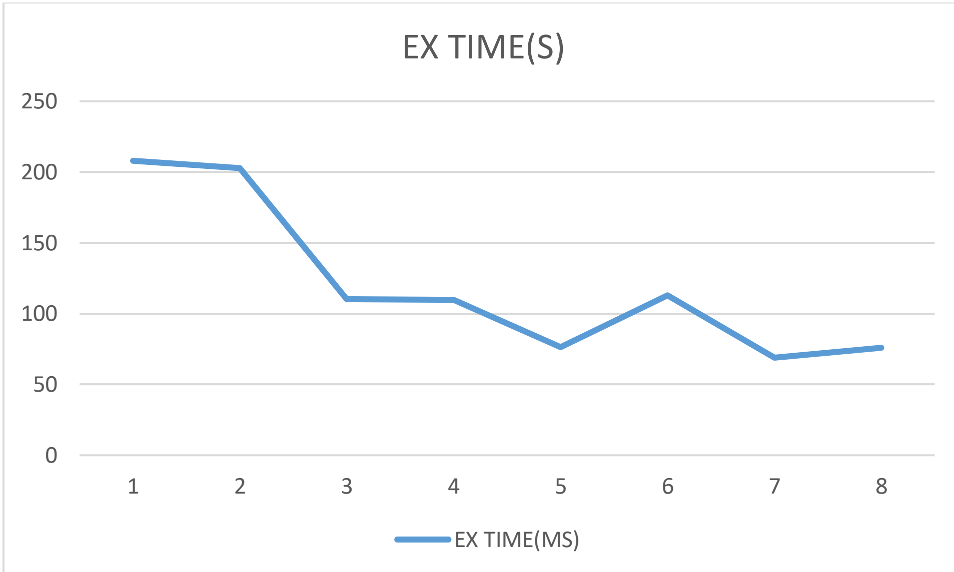
Poate ceva tipic prologului, taierea de backtrack este necesara pentru performanta programului. Ea face ca algoritmul sa se opreasca din a mai cauta solutii alternative, odata ce a dat rezultatul dorit.

Alta optimizare ar fi recursivitatea cu acumulator, mult mai rapida, in cazul nostru, folosita la calcularea factorialului (de ce ? valoarea se calculeaza pe loc, se tot adauga o alta noua, iar la final, se adauga lista vida si avem rezultatul, fata de recursivitatea fara, care punea pe stiva, in waiting valorile rezultatului, pana ajungeam la ultimul apel, iar mai apoi revenea construind lista de rezultate, foarte ineficient).

OBSERVATIE: Am facut tot posibilul sa nu rulez si altceva in background.

PROC	EX TIME(MS)	Speedup	Efficiency
1	208	1	1
2	202.6	1.026654	0.513327
3	110.39	1.884229	0.628076
4	109.84	1.893664	0.473416
5	76.17	2.730734	0.546147
6	112.92	1.842012	0.307002
7	68.92	3.017992	0.431142
8	76	2.736842	0.342105





OBSERVATII GENERALE

Nu s-au rulat aplicatiile intr-un mediu optim (s-a lucrat in timpul rularii, antivirus, browsing pe web, deschis multe IDE-uri etc.) De aceea nu vedem un speedup mai mare sau egal cu 4 (adica cate core-uri are procesorul meu). In mod ideal ar trebui undeva pe la 5,25 speed-up. (4 + 1,25 de la hyperthread).

Se poate observa pe graficele de speedup cat si pe cele de timp de executie, ca cresterea respective scaderea nu este uniforma. Astfel, la 2 threaduri avem aprox acelasi timp ca si cu un thread.

Explicatia consta in faptul ca fiind 2 threaduri, numerele din intervalul [1,35000] se vor imparti in mod egal la th1 si th2. Astfel, th1 va primii toate numerele impare, iar th2 toate pare. Th2 avand sa gestioneze doar numere pare, el va intra in functia de prim care va returna fals la orice multiplu de doi, fara a mai face alte impartiri costisitoare.

[1]- http://awesci.com/wilson-primes/

[2]- https://en.wikipedia.org/wiki/List_of_C%2B%2B_multiple_precision_arithmetic_libraries

[3]- https://www.boost.org/