Zeef van Eratosthenes

20/04/2020

Sinem Ertem

Deel 1

1.1 Werking van het algoritme

1. Schrijf getallen van 2 tot 15 op
2. Omcirkel het kleinste getal in de lijst die niet omcirkeld is; **dus 2**
3. Voor elk getal groter dan 2; als het getal een meervoud is van 2, markeer het.
    a. De getallen **4,6,8,10,12,15**
4. Ga terug naar stap 2, nu is **3** het kleinste getal
5. Voor elk getal groter dan 3; als het getal een meervoud is van 3, markeer het.
    a. De getallen **6,9,12,15**
6. Nu is **5** het kleinste getal
7. Voor elk getal groter dan 5; als het getal een meervoud is van 5, markeer het.
    a. De getallen **10,15 → deze zijn al omcirkeld**
8. Nu is **7** het kleinste getal
9. Voor elk getal groter dan 7; als het getal een meervoud is van 7, markeer het.
    a. Het getal **14 → deze is al omcirkeld**
10. Nu is **11** het kleinste getal
11. Voor elk getal groter dan 11; als het getal een meervoud is van 11, markeer het.
    a. Meervoud van 11 staat niet in de lijst
12. Nu is **13** het kleinste getal
13. Voor elk getal groter dan 13; als het getal een meervoud is van 13, markeer het.
    a. Meervoud van 13 staat niet in de lijst

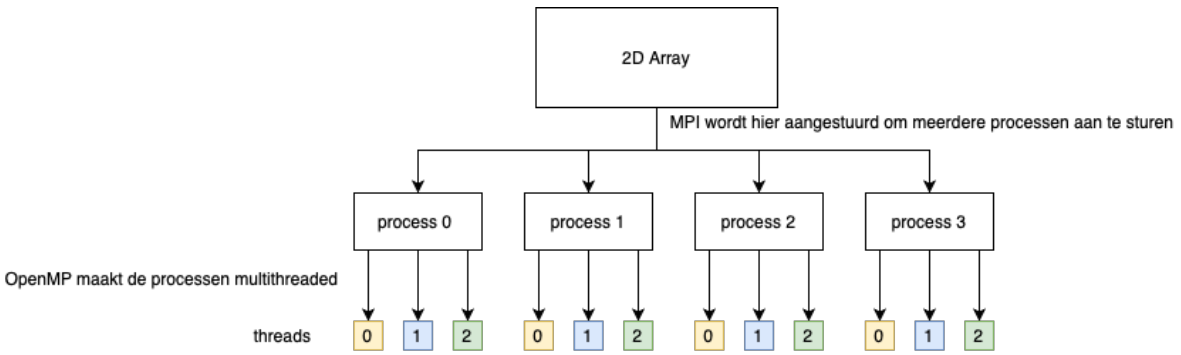
Alle getallen zijn nu omcirkeld of gemarkeerd. De priemgetallen kleiner dan 16 zijn de omcirkelde getallen **2,3,5,7,11 en 13.** De tegenovergestelde getallen zijn de gemarkeerde getallen namelijk; **4,6,8,9,10,12,14 en 15.**

Deel 2

2.1 Conceptueel ontwerp

Maak een conceptueel ontwerp van een parallelle variant waarin MPI wordt gebruikt om meerdere processen aan te sturen, en OpenMP om de processen multithreaded te maken.

Figuur 1: Conceptueel ontwerp



2.2 Analyse performance winsten

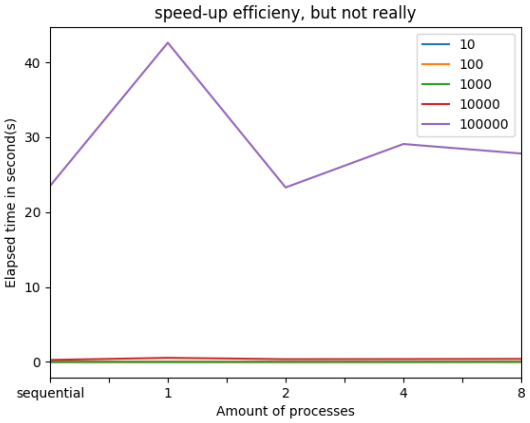
Maak een duidelijke analyse van de performance winsten (speed-up efficiency) van de implementatie, op basis van de verzamelde getallen.

Figuur 2: speed-up efficiency

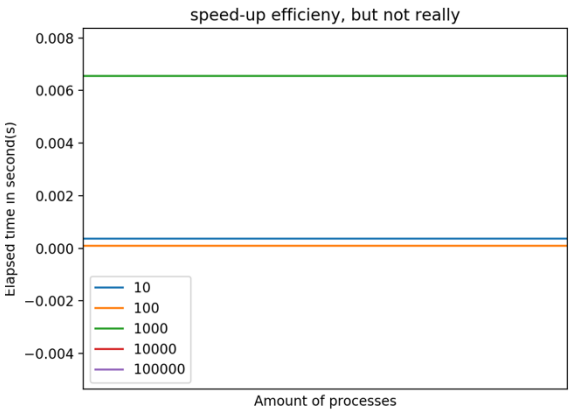
	Sequential	1 process	2 processes	4 processes	8 processes	*Total primes*
10	0.00001 second(s)	0.0015 second(s)	0.00031 second(s)	0.00142 second(s)	0.00854 second(s)	*4*
100	0.00007 second(s)	0.00017 second(s)	0.00040 second(s)	0.00516 second(s)	0.02015 second(s)	*25*
1.000	0.00547	0.01 second(s)	0.00535 second(s)	0.00575 second(s)	0.02063 second(s)	*168*
10.000	0.25710 second(s)	0.54 second(s)	0.36088 second(s)	0.37576 second(s)	0.39939 second(s)	*1229*
100.000	23.49340 second(s)	42.62 second(s)	23.29048 second(s)	29.08724 second(s)	27.81755 second(s)	*9592*

Een feit is dat je alleen goede speed-up efficiency krijgt als je werkt met grote lijsten vanaf een miljoen getallen. Echter wilde ik alsnog kijken wat voor resultaten ik zou krijgen bij kleinere lijsten, opbouwend naar grotere lijsten. Ik merk dat mijn laptop moeite krijgt met lijsten ter grootte van 1 miljoen getallen, waardoor het eigenlijk onmogelijk was om op te bouwen naar 10 tot 100 miljoen getallen om echt het verschil te gaan merken.

Figuur 3: Plot speed-up efficiency



Figuur 4: Ingezoomde versie figuur 3



Zoals nog duidelijker te zien is in figuur 3 heeft het nauwelijks zin om MPI te gebruiken bij aantallen van 10 tot 10.000 getallen. Pas bij het gebruik van 100.000 getallen zie je dat er veel beweging zit in het verbruik van verschillende aantallen processen. Het is daarom jammer dat ik de test niet heb kunnen uitvoeren bij miljoenen getallen, máár dit neemt niet weg dat ik wel het verschil heb kunnen leren tussen een sequentiële variant en een met bijvoorbeeld 8 processen.

2.3 Reflectie

In welke zin vind je dat je geslaagd bent in het maken van de opgave; wat kon er beter, wat was eenvoudig/lastig.

Over het algemeen heeft het vak High Performanced Programming een goede bijdrage geleverd aan het duidelijk overbrengen van hoe je een programma kan laten voldoen aan snelheids- en efficiëntie eisen. Dit door o.a. gebruik te maken van multi-core, multi-processor of zelfs multi-computersystemen. De opgave die we hier kregen heeft dit wederom enigszins goed laten zien omdat dit een redelijke samenvatting was van voorgaande opgaven. Hoewel we zowel programma's hebben geschreven in *C* en *Python* i.p.v. ons goed te verdiepen in een taal. Het heeft wel zijn voordeel omdat je ziet hoe de verschillende talen omgaan met meerdere processen.

Het is interessant dat we aan de hand van deze opgaven een oude algoritme leren kennen en deze vertalen naar een programmeertaal. Met de hand kunnen we dit algoritme uittekenen en priemgetallen selecteren maar zodra je dit doet met een array van 100+ getallen wordt dit natuurlijk lastig en is het leuk om het via deze weg te doen met behulp van meerdere processen om het verloop efficiënter te laten verlopen.

Tijdens het runnen van het programma merk ik al snel dat mijn computer het niet aan kan om bijvoorbeeld priemgetallen te selecteren uit een array van 1 miljoen getallen, terwijl mijn buren gerust 100 miljoen getallen invoeren. Dit zal misschien ook liggen aan mijn gebrek aan kennis op dit gebied van de MPI library en hoe ik deze dus het beste kan laten werken, maar ook het aantal cores wat mijn laptop heeft is relatief weinig. Met getallen minder dan 1 miljoen kon ik al wel duidelijk zien wat de speed-up efficiency is en hiermee nuttige conclusies trekken.

Figuur 5: Specificaties van mijn laptop ***1 processor en 2 cores***