ZEEF VAN ERATOSTHENES

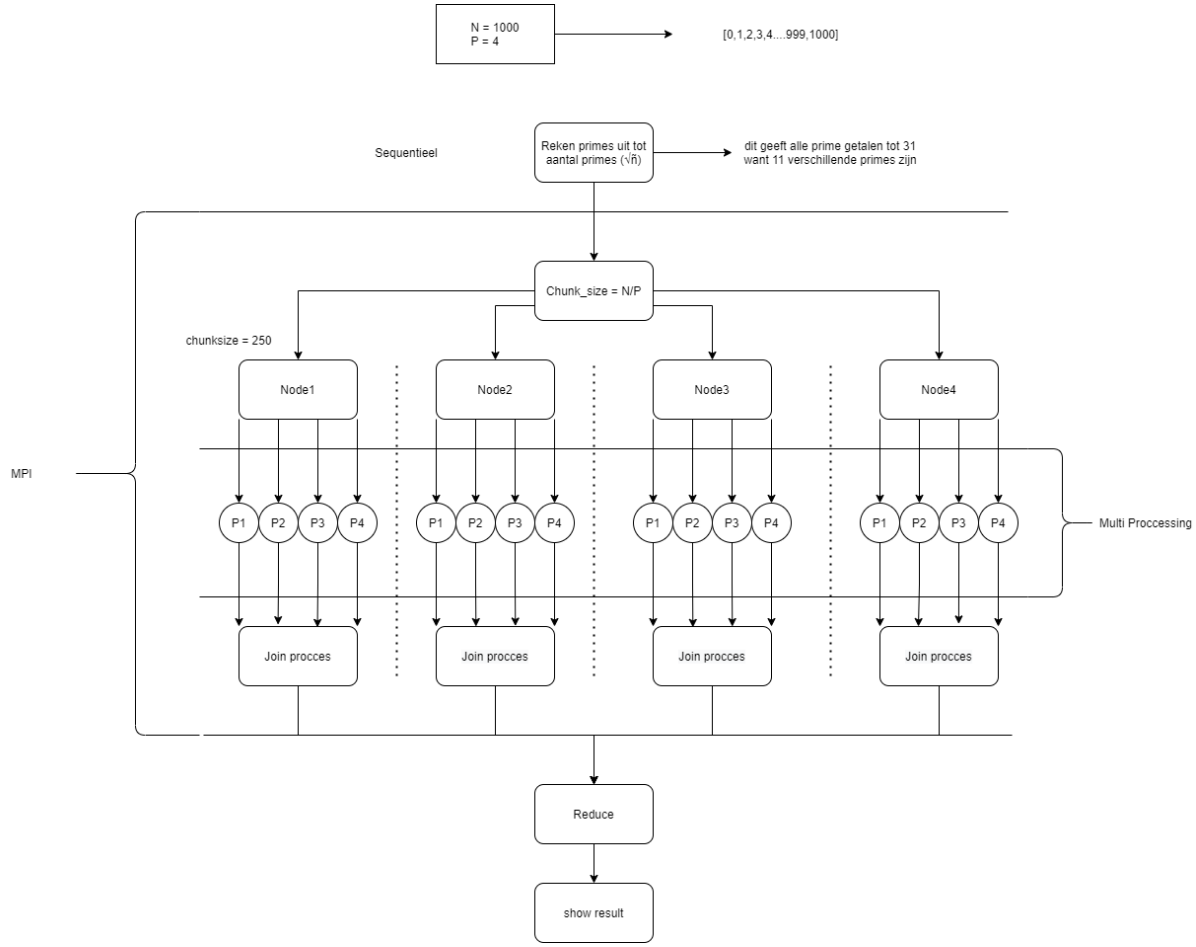
Ontwerp analyse

RICHARD JANSEN

Inhoud

Overzicht model

Hier onder is het overzicht van de structuur in mijn programma. Dit schema is gebouwd op N 1000 als lijst grote en 4 P als Nodes. Op de volgende bladzijde wordt er dieper per onderdeel een toelichting gegeven.

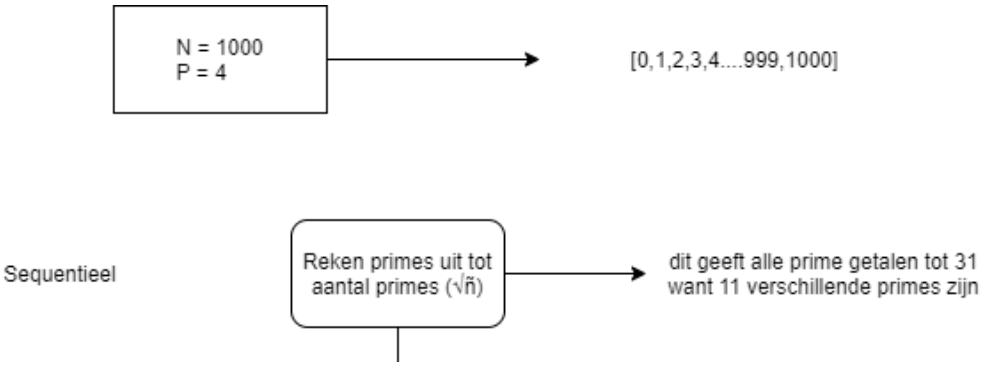


Toelichting

Hieronder wordt toelichting gegeven over de architectuur van het programma

De init

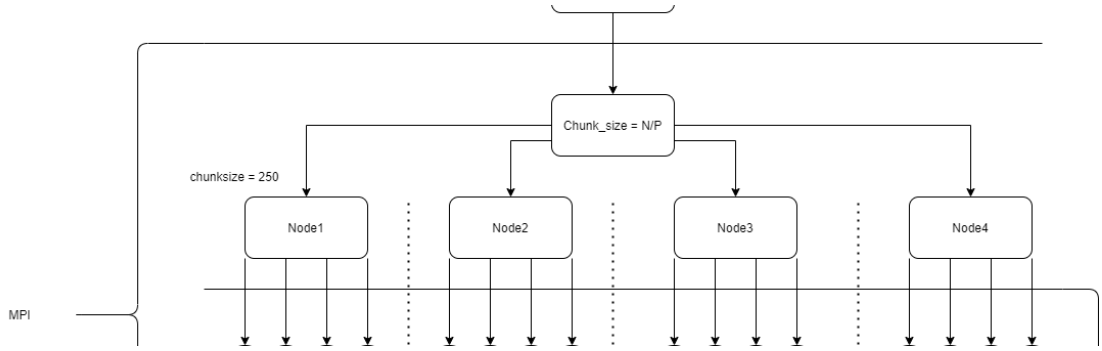
Het programma word gestart met een lijst lengte van 1000 getallen. De range van getallen ziet er als volgt uit: [0,1,2,3,4 … 999,1000]. Het eerste gedeelte van het programma wordt sequentieel uitgevoerd. Als eerste worden alle priem getallen uitgerend in de range van 0 tot √(n) en verwerkt in een lijst. De reden dat ik dit doe is dat ik op internet een uitleg heb gevonden die het beter kan verklaren dan ik(zie link onder het plaatje *). De uitkomst in ons geval is dat we dus van 2 tot 31 kijken wat de prime getallen zijn en dan komen we uit op een lijst van 11 prime getallen. Met deze 11 prime getallen gaan we verder naar de init van MPI en geven we dit met aan elke node.



https://stackoverflow.com/questions/5811151/why-do-we-check-up-to-the-square-root-of-a-prime-number-to-determine-if-it-is-pr

MPI init

Nu we een lijst van prime nummers hebben, moeten we over de beschikbare Nodes van MPI onze hele lijst opdelen. In mijn geval verdeel ik de lengte N, 1000, op over 4 Nodes. Node 1 krijgt alle getallen van 0-249, Node 2 krijgt alle getallen van 250-499, Node 3 krijgt alle getallen van 500-749 en Node 4 krijgt 750-100.



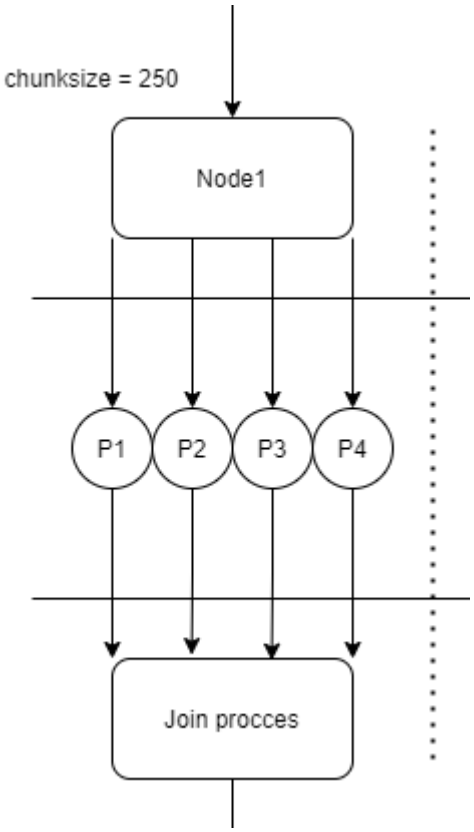
Inhoud van een Node

Elke Node krijgt ook ALLE prime getallen die van tevoren waren uitgerekend in de range van 0 tot √(n). In mijn geval dus een lijst met 11 prime getallen.

In elke node wordt het volgende proces parallel van elkaar uitvoert. Vergelijk de modulo van elk element in de meegegeven chunk van iedere element in de lijst met prime getallen. Als module 0 is, dan is het getal een meervoud van dat prime getal dus ZELF geen prime getal. Bij dit geval, verwijder het element.

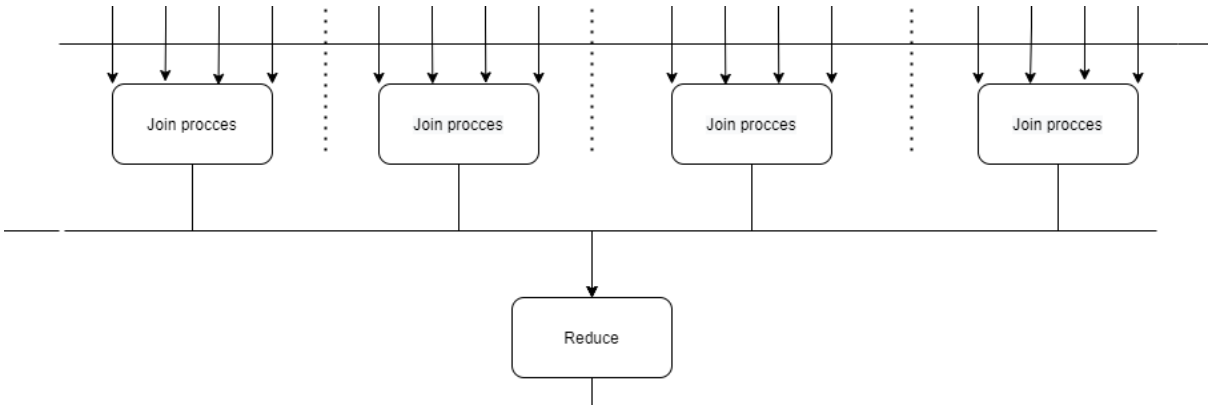
In de Node zelf kan het bovenstaande proces opgedeeld worden op chunks van de meegegeven chunksize (little chunkies 😊). Op deze manier kan je met de beschikbare cores op je Node je chunksize opdelen en zou dus PER node meerdere vergelijking processen gedaan kunnen worden en dus meer load balance zijn van de operaties die gedaan worden zijn zeer tijd waardig.

Aan het einde van de Node, worden alle processen samen gevoegd met Joins tot weer 1 lijst voor als uitput van de **Node.** Er word dan gejoint op de lijst met overgebleven primes



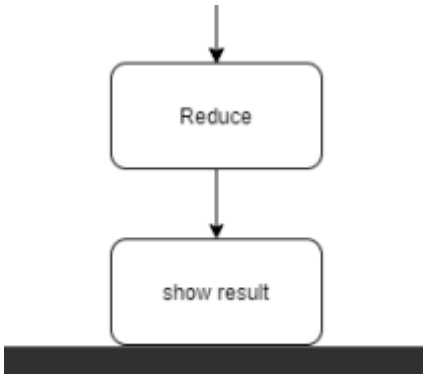
Reduce MPI

Zodra alle Nodes klaar zijn moeten de resultaten samen gevoegd worden. Voor MPI wordt dit gedaan met de reduce functie op de uitgerekende primes van Nodes tot 1 lange lijst.



Show result

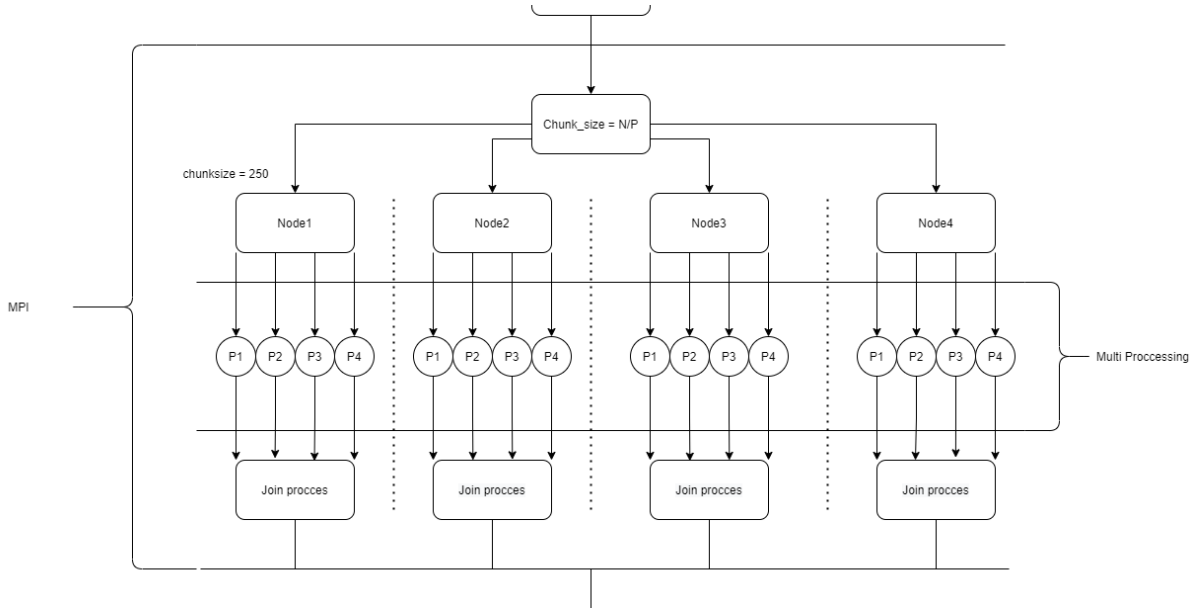
Aan het einde van het programma wordt de lengte van de prime lijst weergegeven. Niet heel spannend



Verwachten uitkomst van MPI in combinatie met multi processing

In mijn ontwerp verwacht ik dat het uitvoeren van dit probleem een versnelling van N (Nodes) * Processen * 100% zou zijn. In mijn voorbeeld dus 16x sneller als we 4 Nodes en 4 processen per Node zouden gebruiken.

Stel sequentieel doet het programma er 100 seconden over, dan met mijn voorbeeld zou het dan ongeveer 6 seconden er over doen. ECHTER, gaat dit nooit zo snel gebeuren omdat er communicatie overhead is en we niet in een perfecte wereld leven waar dit geen rol speelt.



Dubbel werk

Van hoe mijn programma is opgezet, is er een stukje dubbel werk in de eerste Node wat toegelicht moet worden. Omdat ik van tevoren een sequentieel de range van 0 tot √(n) aan primes moet berekenen, is een gedeelte van N1 overbodig maar niet heel erg voor deze opdracht