2022

Verslag – Zeef van Eratosthenes

CEYHUN ÇAKIR | 1784480

Table of Contents

Inleiding *2*

Ontwerp *2*

MPI met OpenMP (2 processen) (2 threads) **2**

MPI met OpenMP (4 processen) (2 threads) **2**

Realisatie *3*

Sequentiële **3**

MPI **5**

Uitkomsten *6*

Sequentiële **8**

MPI **9**

spreadsheet 9

MPI (1 tot 16 processen) (n = 1m) 10

MPI (1 tot 16 processen) (n = 10m) 10

MPI (1 tot 16 processen) (n = 100m) 11

MPI (1 tot 16 processen) (n = 1b) 11

Zelfreflectie *12*

Wat ging goed? **12**

Wat vond ik lastig? **12**

Wat kon beter? **12**

Wat heb ik geleerd? **12**

Conclusie *12*

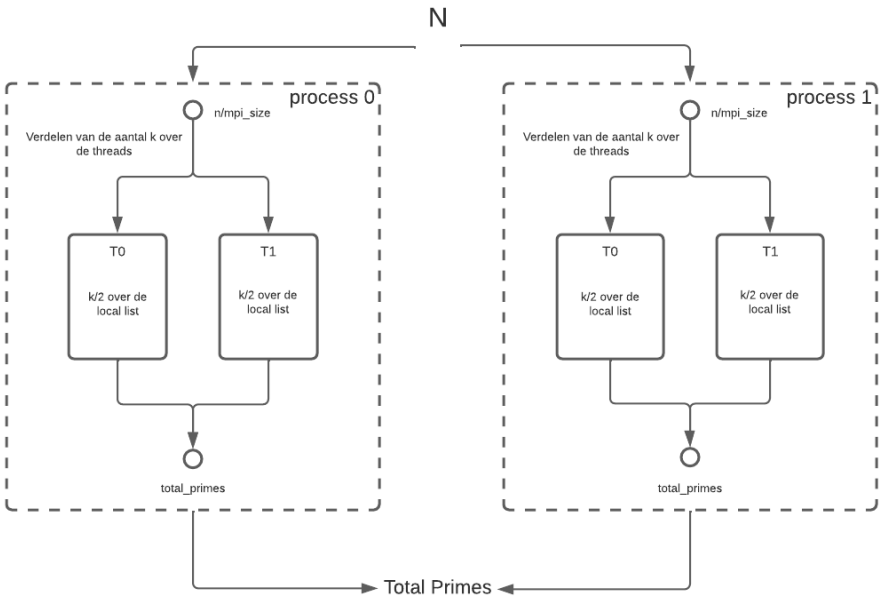
Inleiding

In deze opdracht moesten we de zeef van eratosthenes realiseren. Het doel van dit opdracht was om gebruik te maken van MPI en openMP om dit opdracht goed te kunnen realiseren. Verder ging de opdracht over hoe je priemgetallen kon vinden door verschillende processen en threads te gebruiken. Dit zou zorgen dat de snelheid van de programma sneller zou gaan.

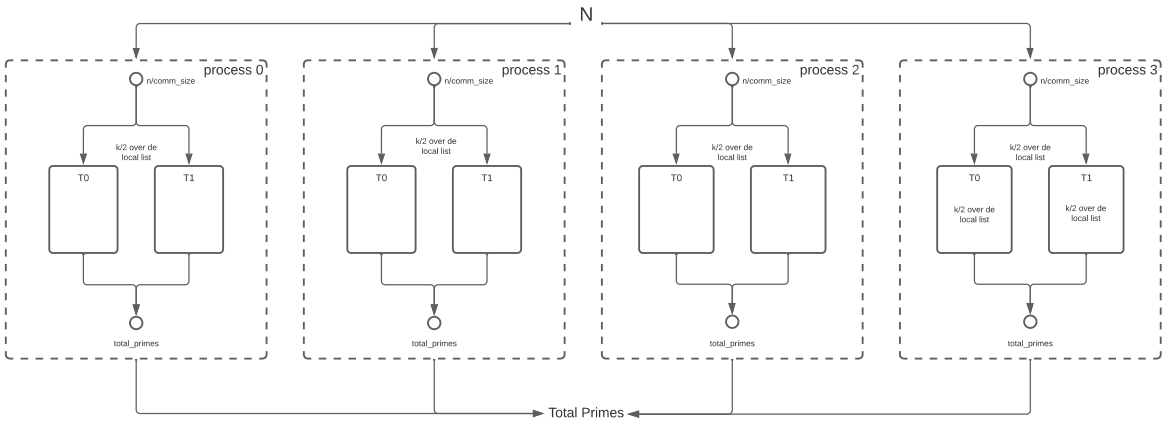
Ontwerp

In de onderstaande kopje (MPI Met OpenMP) kunnen we het ontwerp zien van hoe je dus MPI en openMP samen kan gebruiken om de applicatie sneller te laten verlopen.

MPI met OpenMP (2 processen) (4 threads)



MPI met OpenMP (4 processen) (8 threads)



Realisatie

Hieronder in de twee kopjes ga ik uitleggen hoe ik dit opdracht heb kunnen realiseren en hoe ik tot een oplossing ben gekomen.

Sequentiële

Voor de realisatie van de sequentiële versie ben ik eerst analyse gaan doen over hoe de zeef van eratosthenes werkt. Door op Wikipedia te gaan kwam ik al snel algoritmes tegen die eventueel zouden kunnen werken. Na een analyse ben ik begonnen met het werk environment opzetten.

Na dat het environment opgezet was ben ik begonnen met het maken van de find_prime_numbers functie.

```
std::vector<int> find_prime_numbers(int n) {
```

Ik creëerde eerst een vector om alle nummers erin te hebben. Na dat ik de vector had gevuld ging ik door met de daadwerkelijke algoritme

```
std::vector<int> zeef;

for(int i = 2; i < n; i++) {
  zeef.push_back(i);
}
```

Ik creëerde een variabelen genaamd k. ik zette de waarde van k op twee aangezien 0 en 1 geen primes kunnen zijn

```
int k = 2;
```

Na dat ik de k variabelen had aangemaakt ging ik door met het schrijven van een while loop. Hier had ik een expression in gezet van “k * k <= n” dit wilt zeggen dat die de while loop doet tot dat k * k kleiner is dan n. na dat creëer ik een iterator om door de zeef te gaan itereren. Ik pak voornamelijk de index waar het volgende veelvoud is. Ik loop met een for loop waar i op dit veelvoud index zit. Ik kijk of die index binnen de zeef niet gelijkwaardig is aan k wat dus betekent dat de waarde op de index binnen de zeef geen prime is.

Na de for loop maak ik nog een itterator aan om alle 0’s te verwijderen uit mijn lijst zodat ik alleen maar de primes over heb. Dit doe ik voornamelijk aangezien ik later de count ga pakken van het lijst wat dus mij de totale prime nummers gaat geven.

Tot slot zet ik k met index waarde binnen mijn eigen zeef

```cpp
std::vector<int> find_prime_numbers(int n) {

  std::vector<int> zeef;

  for(int i = 2; i < n; i++) {
    zeef.push_back(i);
  }

  int k = 2;

  while(k * k <= n) {
    std::vector<int>::iterator it = std::find(zeef.begin(), zeef.end(), k * k);
    int index = std::distance(zeef.begin(), it);

    for(int i = index; i < n; i++) {
      if(zeef[i] % k == 0) {
        zeef[i] = 0;
      }
    }

    zeef.erase(std::remove(zeef.begin(), zeef.end(), 0), zeef.end());

    std::vector<int>::iterator it1 = std::find(zeef.begin(), zeef.end(), k);
    int k_index = std::distance(zeef.begin(), it1);

    k = zeef[k_index + 1];

  }
```

Na dat de find_prime_numbers functie klaar was ging ik door met het maken van mijn main functie. Ik heb roep voornamelijk de find_prime_numbers functie op en kijk wat de tijd is en wat de totale gevonden prime nummers zijn. Ook zet ik deze waardes in een excel sheet om dit te kunnen tonen binnen dit opdracht

```cpp
int main() {

  int n = 1000;

  auto start = std::chrono::system_clock::now();
  std::vector<int> outcome = find_prime_numbers(n);
  auto stop = std::chrono::system_clock::now();

  auto duration = stop - start;
  typedef std::chrono::duration<float> float_seconds;
  auto secs = std::chrono::duration_cast<float_seconds>(duration);

  std::ofstream myFile;
  myFile.open("spreadsheet/sequentiele.csv", std::ios::app);
  myFile << secs.count() << "; " << outcome.size() << "; " << n << std::endl;
  myFile.close();

  std::cout << "total amount of numbers: " << n << std::endl;
  std::cout << "found prime numbers: " << outcome.size() << std::endl;
  std::cout << "Total time taken: " << secs.count() << std::endl;


  return 0;
}
```

MPI

Om de mpi versie te maken ging ik zoals eerder een analyse/ ontwerp maken hoe je dat precies zou kunnen maken. Snel kwam ik achter dat je over de processen de k's moest gaan verdelen zodat ieder proces weet bij welke stukje van het data zit om calculaties te maken. Ook kwam ik snel achter dat je vast een first element en last element moest hebben om dus niet out of bounds te gaan. Na dat ik dit analyse/ ontwerp/ voor gedachtes had ontwikkeld ging ik door met de daadwerkelijk realisatie van de applicatie.

Na dat ik mijn environment op had gezet ging ik door met het programmeren van de applicatie. Ik ging eerst alle variabels maken die nodig waren bijvoorbeeld (comm_size, comm_rank, n, first element, last element etc.). Dit waren allemaal noodzakelijk om voor ieder proces de juiste data bij te houden

```
int comm_size;
int comm_rank;
unsigned long n = 1000;
unsigned long f_elem;
unsigned long l_elem;
unsigned long size;
unsigned long i;
unsigned long k;
unsigned long first_multiple_index;
unsigned long next_k;
unsigned int count = 0;
unsigned int global_count = 0;
```

Ik ging toen door met het schrijven van de initialisatie van mpi en het verdelen van de data over de aantal processen uiteraard keek dit naar welke proces het op het moment was.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
MPI_Comm_rank(MPI_COMM_WORLD, &comm_rank);

double start = MPI_Wtime();

// we splitten de lijst in blocks voor elke process. ook zorgen we dat voor elke process de first en last elem en de size word opgeslagen.
f_elem = floor(comm_rank * (n - 2) / comm_size) + 2;
l_elem = floor((comm_rank + 1) * (n - 2) / comm_size) - 1 + 2;
size = l_elem - f_elem + 1;
```

Verder ging ik door met het allocating van het lijst waar n elementen in gaan zitten.

```
// we initialiseren een lijst met daarin elementen tot size die true zijn
bool *array = new bool[size];
for(i = 0; i < size; i++){
    array[i] = true;
}
```

We zetten alle waardes daar naar true wat wilt zeggen dat elke element daar binnen een prime nummer is. Dit heb ik voornamelijk gedaan aangezien het makkelijk voor mij is om verwerkingen te maken.

Na de allocatie van de lijst gingen we door met het standaard algoritme die we ook hebben geïmplementeerd in de sequentieel versie. Deze versie van het algoritme was wel wat anders dan de voorgaande aangezien we nu met meerdere processen bezig zijn.

wat hier voornamelijk gebeurd is dat we weer een while loop schrijven waarin de expressie (k * k <= n) zit. We bepalen de index van de eerste veelvoud van k in het eerste process. Na dit kijken we of de eerste element een veelvoud is of niet. Zo niet zetten we de veelvoud index op 0 zo wel zetten we de index veelvoud op k – first element van de veelvoud k.
na dit kijken we zoals gewoonlijk of de eerste veelvoud index binnen de lijst met elements in zitten. Zo ja zetten we de element die.1 is naar een 0 wat wilt zeggen dat dit element geen prime is.

Uiteindelijk zetten we weer zoals gewoonlijk de volgende waarde van k naar de kleinste getal en kijken we of dat nummer groter is dat de huidige k. en uiteindelijk geven we aan dat de eerste process dit waarde moet gaan vinden voor de volgende process.

Wanneer de process de volgende waarde k heeft gevonden gebruiken we de mpi broadcast functie om dit waarde door te geven naar de volgende process.

```
k = 2;

while(k * k <= n){
    // we bepalen de index van het eerste veelvoud van k in de huidige thread
    first_multiple_index;
    if(f_elem % k == 0) {
                                    first_multiple_index = 0;
                            } else {
                                    first_multiple_index = k - f_elem % k;
                            }


    // van deze eerste veelvoud marken we als een non prime voor elke n element.
    for(i = first_multiple_index; i < size; i += k){
        array[i] = false;
    }
    if(comm_rank == 0) {
                                    array[k - 2] = true;
                            }


    // we zetten de volgende waarde van k naar de kleinste nummer en we kijken of dat
    // we geven aan dat thread 0 dit volgende waade moet gaan vinden. Dit waarde is v
    if(comm_rank == 0){
        next_k = k + 1;

        while(!array[next_k - 2]) {
                                                    next_k = next_k + 1;
                                            }
        k = next_k;
    }

    // we broadcasten de volgende k naar de volgende processen.
    MPI_Bcast (&k,  1, MPI_INT, 0, MPI_COMM_WORLD);
}
```

Uiteindelijk kijken we naar de totale aantal primes die binnen de lijst nog zitten. Dit sturen we door naar een lokaal variabelen wat we uiteindelijk naar globaal count gaan reducen. Na dat dit proces klaar is reducen we van de lokaal variabelen naar de globaal count. We checken op het laats of de proces id 0 is aangezien dit als de root proces gezien wordt. Hierin printen we de totalen prime nummers de hoeveelheid tijd. Ook schrijven wij deze waardes net als de sequentiële versie naar een excel sheet om later grafieken te kunnen tonen

```
//we counten de totale nummers van prime en summen dat naar een variabelen.
for(i = 0; i < size; i++){
    if (array[i]) {
      count++;
    }
}

MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

double duration = MPI_Wtime() - start;

if(comm_rank == 0){

    std::ofstream myFile;
    myFile.open("spreadsheet/mpionly.csv", std::ios::app);
    myFile << comm_size << "; " << duration << "; " << global_count << "; " << n << std::endl;
    myFile.close();

    std::cout << "amount of numbers: " << n << std::endl;
    std::cout << "amount of threads used: " << comm_size << std::endl;
    std::cout << size+1 << " elements per thread" << std::endl;
    std::cout << "found total prime numbers: " << global_count << std::endl;
    std::cout << "total time taken: " << duration << std::endl;
}
// we geven de memory weer vrij
delete(array);

MPI_Finalize();
```

Uitkomsten

In de onderstaande twee kopjes (sequentiële, MPI) kunnen we de uitkomsten zien die uiteindelijk zijn gekomen van mijn realisatie. We zien bijvoorbeeld hoe de vergelijking is tussen de aantal processen tegen tijd.

Sequentiële

sequentiele

NumberOfprocesses	Time	Prime Numbers	Amount of numbers	
1	0.776781	78498	1000000	
1	19.8041	664579	10000000	
1	508.149	5761455	100000000	

We zien bijvoorbeeld dat voor de sequentiële versie van de realisatie best wel lang heeft geduurd voor grootte getallen. Dit versie is ook alleen maar tot 100 m gegaan aangezien het voor mij veel te lang duurde. Toch zien we dat de uitkomsten correct zijn maar wel dus langzaam.

MPI

We zien bijvoorbeeld bij de voorbeelden hieronder dat de run time vermindert des de meer processen we op het applicatie zetten. Boven in het spreadsheets kunnen we goed de tijden vergelijken tussen sequentieel versie en mpi only versie.

Spreadsheet

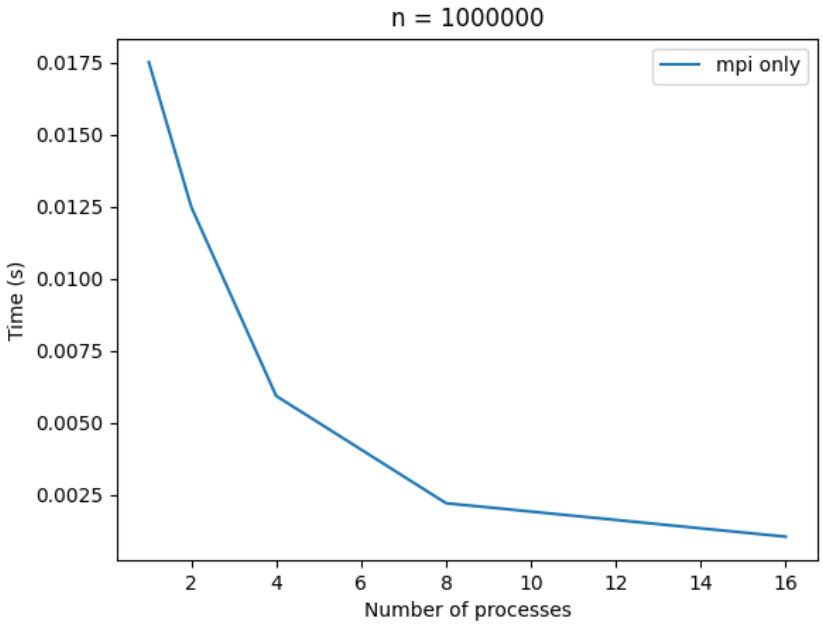
We zien in de onderstaande afbeelding de spreadsheet van de mpi only versie. We zien tegenover de vergelijking van de sequentiële versie dat er grote verbetering is in de run time.

Dit heeft uiteraard te maken met de hoeveelheid processen we inzetten en hoe we de data verdelen.

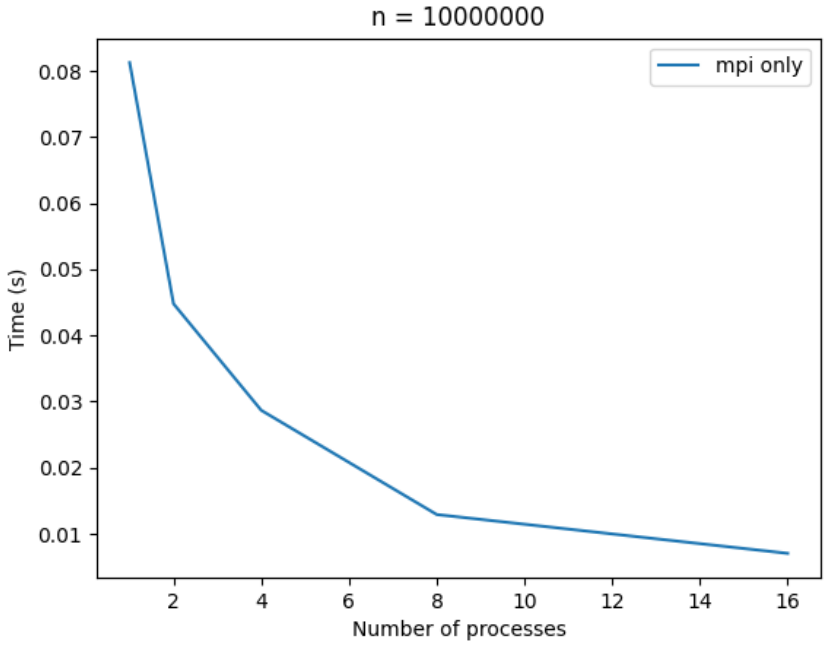
mpionly

Number Of processes	Time	Prime Numbers	Amount of numbers
1	0.0175081	78498	1000000
2	0.0124805	78498	1000000
4	0.00592736	78498	1000000
8	0.00220715	78498	1000000
16	0.00104564	78498	1000000
1	0.0812708	664579	10000000
2	0.0448028	664579	10000000
4	0.0286856	664579	10000000
8	0.01292	664579	10000000
16	0.00704845	664579	10000000
1	1.20898	5761455	100000000
2	0.858926	5761455	100000000
4	0.627794	5761455	100000000
8	0.539096	5761455	100000000
16	0.500368	5761455	100000000
1	14.5265	50847534	1000000000
2	10.4411	50847534	1000000000
4	9.39897	50847534	1000000000
8	7.94653	50847534	1000000000
16	7.5305	50847534	1000000000

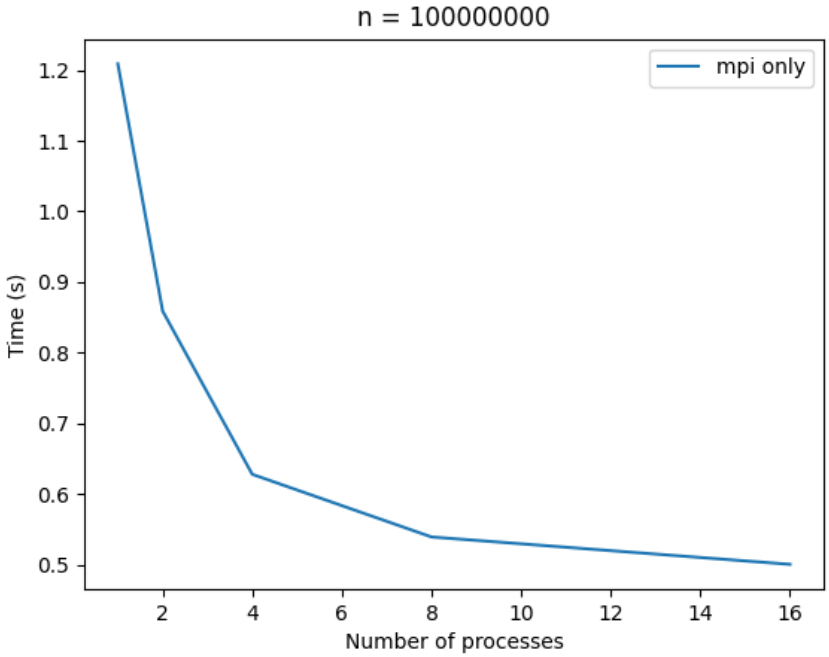
MPI (1 tot 16 processen) (n = 1m)



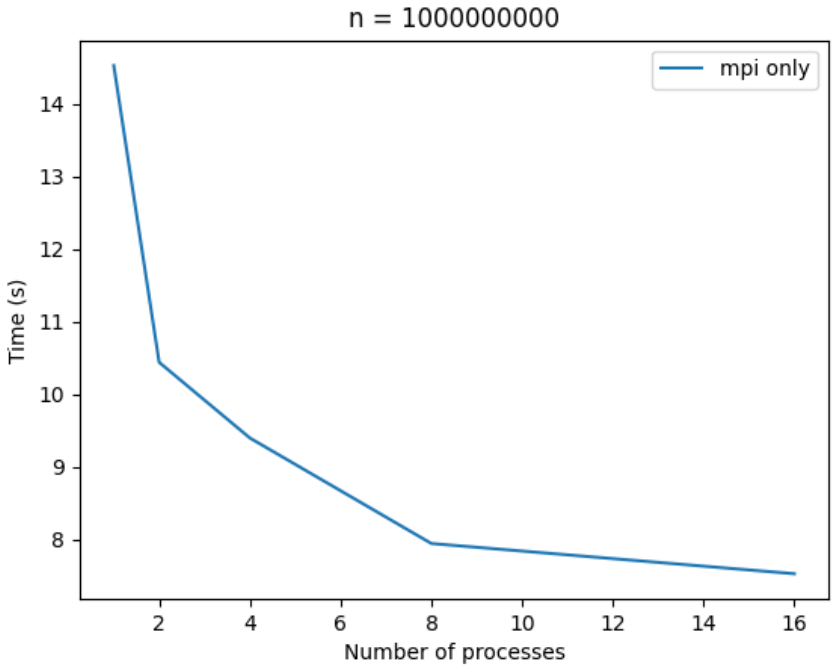
MPI (1 tot 16 processen) (n = 10m)



MPI (1 tot 16 processen) (n = 100m)



MPI (1 tot 16 processen) (n = 1b)



Zelfreflectie

Wat ging goed?

Wat voornamelijk goed ging in deze opdracht was het doorzetting ’s vermogen die ik heb getoond om dit opdracht op tijd af te kunnen krijgen. Ook wat best goed ging was het maken van een pseudo code voordat ik bezig ging met het programmeren. Dit hielp voornamelijk met het denkproces voor dit opdracht.

Wat vond ik lastig?

De mogelijk realisatie van deze dracht in mpi en openmp vond ik het lastigste. Het was voornamelijk moeilijk om de juiste k binnen de threads te kunnen verdelen. Om dit opdracht toch tot een succes te brengen heb ik alleen mpi gebruikt.

Wat kon beter?

In het begin begon ik gelijk met het programmeren van het probleem maar later kwam ik achter dat beginnen met een pseudo code veel beter voor mij uitkwam. Zoals al eerder vertelt hielp dit erg met het denkproces dat nodig was voor dit opdracht.

Wat heb ik geleerd?

Wat ik voornamelijk heb geleerd tijdens deze opdracht is dat ik nu weet hoe je mpi kan gebruiken om priemgetallen te vinden. Ook heb ik geleerd hoe je het veelvoud kan verdelen over verschillende processen. Verder heb ik geleerd hoe je schematisch ontwerpen kan bouwen voor complexe opdrachten zoals dit.

Verder heb ik ook geleerd hoe je omp ook kan gebruiken bij het maken van threads. Door uiteraard eerdere opdrachten in omp te maken heeft dit me erg geholpen in het begrijpen van de techniek.

Conclusie

Ik kan concluderen dat dit opdracht tevens de drukke dagen en drukke schema’s toch tot een succes is gekomen aangezien we wel de data over de hoeveelheid processen hebben kunnen verdelen. Ook zien we bijvoorbeeld dat er groot verschil zit tussen de run time van de twee versies.