Faculty of Engineering
Master of Science in Computer Science

Project
Hoog-performant Rekenen

Decaestecker Dieter
Naessens Tom

1 Probleemstelling

Ons doel is om het volgend stelsel op te lossen op een computationeel snelle manier:

$$\overbrace{\begin{bmatrix} X^TX & X^TZ \\ Z^TX & Z^TZ + \frac{\sigma_e^2}{\sigma_s^2}I \end{bmatrix}}^{A,\text{ gegeven}} \underbrace{\begin{bmatrix} b \\ u \end{bmatrix}}_{x,\text{ gevraagd}} = \overbrace{\begin{bmatrix} X^Ty \\ Z^Ty \end{bmatrix}}^{B,\text{ gegeven}}$$

De aangeduide benamingen zullen we ook gebruiken doorheen de rest van dit verslag. Zoals aangegeven is de data om de matrix A en de vector y op te stellen gegeven. Concreet zijn de volgende problemen op te lossen:

1. Het opstellen van de 2 matrixes op basis van de gegeven data.
2. Het oplossen van de vergelijking a.d.h.v deze 2 matrixes.
3. Het berekenen van $\hat{y}$ en de determinatiecoëfficiënt.

Willen we dit probleem oplossen voor de gegeven inputdata hebben we een aanzienlijk hoeveelheid geheugen nodig om de matrixes in het geheugen te houden. Daarnaast zijn er ook een aanzienlijk hoge hoeveelheid operaties nodig om de matrix-matrix vermenigvuldigingen op te lossen (voornamelijk dankzij de bewerkingen waar de Z matrix deel van uitmaakt). Een HPR-oplossing is voor dit probleem dan ook uiterst geschikt: matrix-matrix vermenigvuldigingen zijn sterk parallelliseerbaar en voor dit soort operaties worden ook standaard implementaties aangeboden door HPR softwarebibliotheken. In het bijzonder door de bibliotheek die wij zullen gebruiken; ScaLAPACK.

2 High-level overzicht van de oplossing

We hebben voornamelijk gebruik gemaakt van de voorgestelde parallelle oplossing (de gebruikte ScaLAPACK functies zijn telkens tussen haakjes vermeld):

1. Initialisatie van MPI en van het procesraster (dit laatste aan de hand van de functies `sl_init_ en blacs_gridinfo_`).
2. De data wordt ingelezen en verspreid vanaf het rootproces:
   (a) De bestanden X, Y en Z worden beschreven aan de hand van `descinit_`.
   (b) Het rootproces leest de data in X en Y en verspreid deze blokcylisch onder de processen (aan de hand van `dgesd2d_` en `dgerv2d_`).
   (c) Het rootproces leest enkele lijnen van de het Z bestand in, berekent de matrix-lijn van Z en verspreid deze lijn blokcylisch naar de verschillende processen. Dit gaat door tot Z volledig ingelezen en verspreid is.
3. A wordt opgesteld en berekend:

   (a) De matrix A wordt beschreven aan de hand van `descinit_`.

   (b) A wordt gevuld (aan de hand van `pdlaset_`) met de foutcoëfficiënt op de diagonaal.

   (c) De deelmatrices van A: X^TX, X^TZ, Z^TX en Z^TZ worden berekend (via `pdgemm_`) en worden op hun correcte plaats in A geplaatst door een offset mee te geven aan de `pdgemm_` oproep.

4. B wordt opgesteld en berekend:

   (a) De vector B wordt beschreven (aan de hand van `descinit_`).

   (b) De deelvectoren van B: X^TY en Z^TY worden berekend via `pdgemv_` en worden op hun plaats in B geplaatst door een offset mee te geven aan de `pdgemv_` oproep.

5. Het stelsel wordt opgelost (aan de hand van `pdgesv_`). Hierbij wordt A overschreven door zijn LU-factorisatie en B door de oplossing van het stelsel.

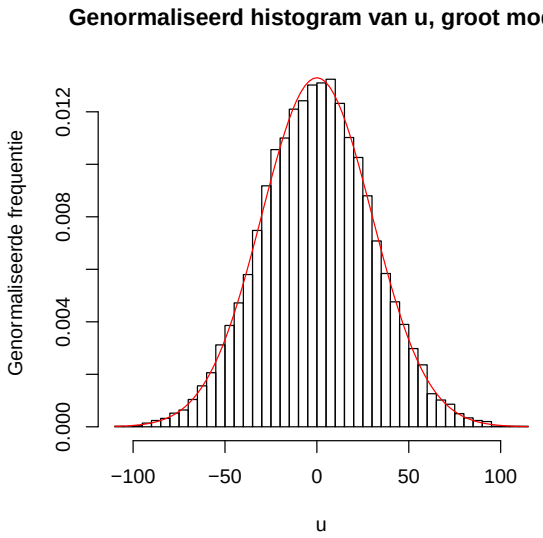
6. De evaluatieresultaten worden berekend en verzameld op het rootproces:

   (a) De oplossing van het stelsel terug verzameld (aan de hand van `dgesd2d_` en `dgerv2d_`).

   (b) $\hat{y}$ wordt beschreven (aan de hand van `descinit_`) en berekend door (aan de hand van `pdgemv_`) $X * b$ en $Z * u$ op te tellen bij deze lege vector.

   (c) $\hat{y}$ wordt terug verzameld op het rootproces.

Een single-core/node implementatie is quasi onmogelijk vanwege de redenen beschreven in deel 1, in deel 3.3 gaan we hier dieper op in.
De grootste bottleneck is sowieso het inlezen van de matrices uit de opgegeven bestanden. Dit is een IO-gebonden proces dat niet of zeer moeilijk te paralleliseren is. Momenteel leest het rootproces alle bestanden in en verdeelt deze onder de overige processen. Mocht dit geperalleliseerd worden zou dit veel versneld kunnen worden. Voor bijvoorbeeld 96 processen neemt het uitvoeren van de vermenigvuldigingen zelf 208 seconden in beslag. Het inlezen echter 4305 seconden. Dit is dus een gigantische bottleneck. Meer over de resultaten bespreken we in deel 3.2 op pagina 3.
Een ander probleem waar we tegen aan zijn gelopen is dat niet enkel het berekenen van Z^TZ te zwaar is voor één proces, maar zelfs het inlezen van de data voor en omzetten naar Z is te data-intensief. Daarom hebben we het inlezen en het verdelen van de data gecombineerd zodat de data iteratief in kleine stukken wordt ingelezen en verspreid. Omdat X en Y een stuk kleiner zijn was het wel mogelijk om deze data eerst volledig in te lezen en dan te verspreiden. Voor grotere datasets zal een soortgelijke oplossing zoals Z ook nodig zijn voor X en Y.



Figuur 1: *Genormaliseerde distributie van u tegenover de normaaldistributie (in het rood weergegeven)*

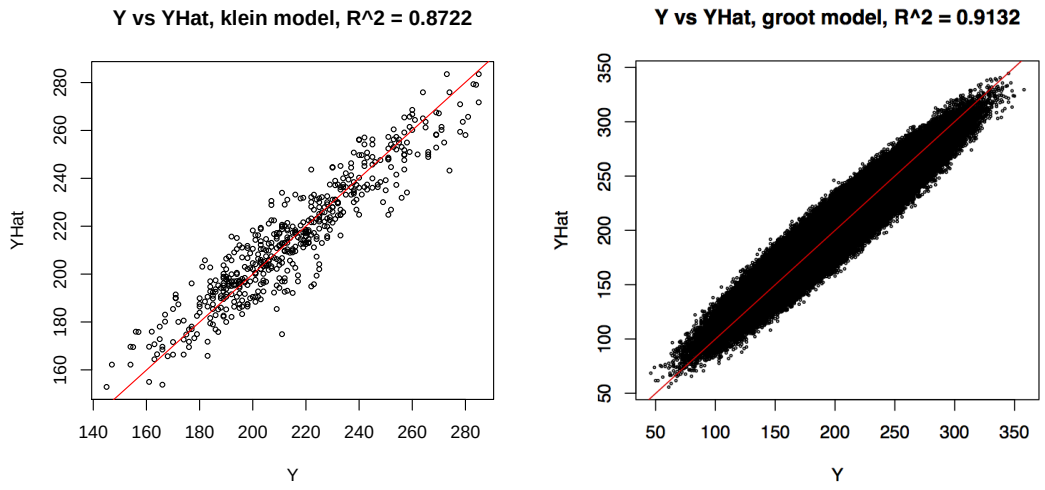
3 Kwaliteitsmetrieken

3.1 Aantonen juistheid

We kunnen de juistheid aantonen aan de hand van drie metrieken:
Ten eerste zouden de waarden van de b vector, zoals vermeld in de opgave, in de buurt moeten liggen van [100, 1]. De bekomen waarden voor b van het kleine model zijn [110.869, 1.04896]. Voor het grote model bekomen we [99.4539, 1.00051].
Daarnaast zou de correlatie van Y en $\hat{Y}$ overeen moeten komen. Dit kunnen we berekenen aan de hand van de determinatiecoëfficiënt R^2. Voor het kleine model is R^2 gelijk aan 0.8722 en voor het grote model 0.9132.
Als laatste zou de bekomen u vector een normaaldistributie moeten volgen. In figuur 1 op pagina 3 staat een genormaliseerd histogram van de waarden in u met de normaaldistributie in het rood.
Alle metrieken liggen zeer dicht bij de verwachte en benodigde waarden. We kunnen er dus van uit gaan dat onze oplossing correct is voor zowel het kleine als het grote model.

3.2 Bespreking resultaten

We hebben als visuele voorstelling van de resultaten voor beide modellen de Y vs $\hat{Y}$ verhouding geplot, zie hiervoor figuur 2 op pagina 4. Zoals in voorgaand puntje hebben we ook de determinatiecoëfficiënt berekend. De lineaire regressielijn is in het rood als overlay toegevoegd aan de plots.
We zien duidelijk dat voor beide modellen de waarden zich langs de diagonaal bevinden. Dit duidt op een goeie fitting van het model.



(a) *Klein model* **(b)** *Groot model*

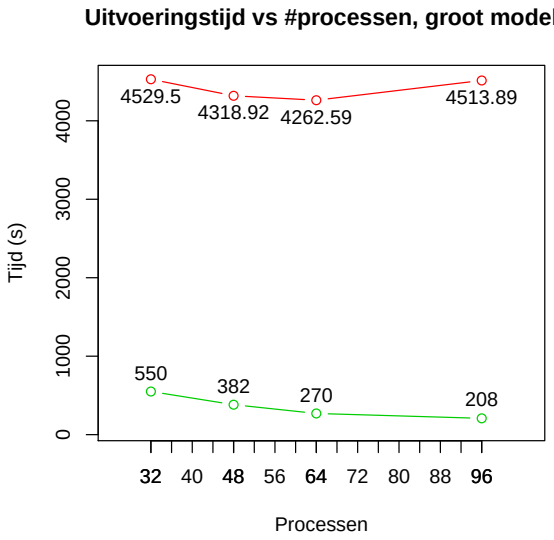
Figuur 2: *Y versus* $\hat{Y}$

Tabel 1: *Speedup voor de verschillende uitvoeringstijden per proces, met als basis de uitvoeringstijd op 32 processen.*

#Processen	Uitvoeringstijd (s)	Speedup
32	550	100%
48	382	143%
64	270	204%
96	208	264%

3.3 Bespreking HPC-aspect

In figuur 3 op pagina 5 is een grafiek te vinden van het aantal processen ten opzichte van de uitvoeringstijd. We brengen twee opzichten in rekening: enerzijds de volledige uitvoeringstijd van begin tot einde (inclusief het inlezen van de data) en anderzijds de tijd nodig om het stelsel zelf op te lossen (exclusief het inlezen van data). We zien duidelijk dat het inlezen van de data de grootste overhead vormt. Bij 96 processen bedraagt dit $\pm 95\%$ van de volledige tijd. Vandaar ook dat er tussen de verschillende volledige uitvoeringstijden weinig verschil te zien is.
Als we naar de groene curve kijken, die enkel de berekeningen voorstelt, zien we wel een duidelijke daling van uitvoeringstijd bij het toenemen van het aantal processen. In tabel 1 of pagina 5 is een overzicht te vinden van de speedup tegenover het miminum aantal benodigde processen, namelijk 32. Met een lager aantal processen dan 32 kon het probleem niet worden opgelost op de Delcatty cluster wegens geheugenlimieten.
We zien duidelijk dat de uitvoeringstijd van de bewerkingen halveert als het aantal processen verdubbeld. Hieruit kunnen we besluiten dat dit probleem goed schaalt ten opzichte van het aantal gebruikte processen en ook zeer goed te paralleliseren is.
Een single-core/node oplossing is niet mogelijk voor dit probleem op de huidige Delcatty



Figuur 3: *Uitvoeringstijden tegenover gebruikt aantal processen*

infrastructuur. De matrix die we dienen te vermenigvuldigen bestaat uit 1M rijen van 10.000 doubles. Dit resulteert in een geheugengebruik van $(1.000.000 * 10.000 * 8B =)$ 80GB. Een node op de Delcatty cluster heeft een fysiek geheugen van 62.9GB, deze matrix past dus niet in het geheugen. Mocht de dataset kleiner zijn en wel binnen het geheugen passen van 1 node, is dit wel oplosbaar, ook al zal dit dan zeer lang duren. Een single-core/node oplossing zou enkel mogelijk zijn moest de infrastructuur voldoende geheugen hebben en als de gebruiker veel geduld heeft.

3.4 Conclusie

We kunnen kort concluderen dat de berekeningen nodig om dit probleem op te lossen zeer goed te paralleliseren zijn. Het inlezen van de data is de grootste bottleneck. Mocht dit ook geparalleliseerd worden zou de volledige uitvoeringstijd sterk afnemen.