Een parallelle LU-decompositie

Lars Hemel Arthur van Dam

8 december 1999

Inhoudsopgave

Hoofdstuk 1

Inleiding

Het probleem dat bekeken wordt, zal veel te maken hebben met het volgende systeem van lineaire vergelijkingen,

$$Ax = b \tag{1.1}$$

Hierin is A een $n \times n$ matrix, b een bekende vector ter lengte n en x de vector ter lengte n die berekend moet worden.

Vooraf moet gezegd worden dat dit niet zomaar een vergelijking is. In de praktijk blijkt dat zeer veel problemen te schrijven zijn als een stelsel van vergelijkingen $Ax = b$. Hoe meer onbekenden er zijn des te groter en nauwkeuriger wordt het probleem. Uit tijd en dus geld-overwegingen is het zeer belangrijk dat dit stelsel snel opgelost kan worden.

Hoe dit probleem opgelost gaat worden, is als volgt: Omdat er met heel grote matrices gewerkt wordt (n is vaak van orde 10^3 of hoger), kan het probleem niet opgelost worden door de inverse van A uit te rekenen. Hoe het wel kan is met behulp van de zogenaamde LU-decompositie. De matrix A moet als een vermenigvuldiging van twee andere matrices L en U geschreven worden. Omdat iedereen inzag dat met het oplossen van dit probleem geld te verdienen was, zijn er talrijke algoritmes bedacht. De een lost het probleem nog sneller en efficiënter op dan de ander. Het parallelle algoritme dat hier gebruikt wordt, is het algoritme *BSP-LU1* van Rob Bisseling, zoals beschreven in hoofdstuk 2 van [2]. Door allerlei kleine veranderingen zal zijn programma sneller en sneller gemaakt worden. De oorspronkelijke *BSP-LU1* is in ANSI C geïmplementeerd, met het pakket BSPlib. Dit is een software pakket dat de gebruikers ervan parallelle progamma's laat schijven volgens het BSP model.
BSP (Bulk Synchronous Parallel) betekent in het kort het volgende: Een bepaalde hoeveelheid gegevens (bulk hoeveelheid) wordt parallel bewerkt. Iedere processor doet een bepaalde hoeveelheid werk. Als dit gedaan is, wisselen de processors belangrijke data met elkaar uit die ze nodig hebben voor hun volgende stap. Aan het eind van zo'n stap wordt er gesynchroniseerd. Nu heeft elke processor de gegevens die het nodig heeft en wordt met een nieuwe stap begonnen. Dit gaat net zo lang door totdat het probleem opgelost is.

Hoofdstuk 2

Wiskundige achtergrond

Voordat ook maar iets verteld kan worden over hoe een parallelle LU-decompositie berekend wordt, moet eerst verteld worden hoe een sequentiële decompositie werkt.

2.1 Sequentiële LU-decompositie

Zoals al eerder vermeld, wordt A omgeschreven in de matrixvermenigvuldiging LU. Hierbij is L een $n \times n$ eenheids beneden driehoeksmatrix en U een $n \times n$ boven driehoeksmatrix. Zodra deze matrices L en U bekend zijn, kan (1.1) zeer eenvoudig opgelost worden: los eerst $Ly = b$ op en vervolgens $Ux = y$. Omdat tijdens het algoritme rijen verwisseld zullen worden, wordt nog een permutatievector π bijgehouden, die aangeeft waar rijen in de oorspronkelijke matrix A stonden.

Het algoritme gaat als volgt: Er wordt begonnen met de eerste kolom en eerste rij (fase 0). Bekijk welk element in absolute waarde in de eerste kolom het grootste is en dit wordt het pivot-element genoemd.

Indien het pivotelement a_{i0} niet op de diagonaal staat ($i > 0$), wordt rij i met rij 0 verwisseld. In π wordt opgeslagen waarheen deze rijen verwisseld zijn. Ook als rijen niet verwisselen, kan dat uit de vector π afgelezen worden. Door het verwisselen wordt ervoor gezorgd dat er geen 0 op de diagonaal komt te staan en dat er dus niet door 0 gedeeld wordt. Dit zorgt ervoor dat het algoritme goed gaat voor niet-singuliere matrices. Wanneer een pivot wordt gevonden, die wel gelijk is aan 0, dan is meteen duidelijk dat de matrix A singulier is.

Nu het pivotelement bekend is, worden alle andere elementen in de eerste kolom gedeeld door de pivot. Dan wordt van alle elementen a_{ij} met $(0 < i < n)$ en $(0 < j < n)$ het produkt $a_{i0}a_{0j}$ afgetrokken. Fase 0 is nu klaar en er wordt verder gegaan met fase 1. De tweede kolom en de tweede rij worden bekeken en er wordt precies hetzelfde gedaan als wat in fase 0 gedaan werd. Op deze manier wordt de hele matrix doorlopen en op het eind kunnen in de matrix A, de lower- en uppertriangular matrices L en U teruggevonden worden. Het voorgaande staat uitgewerkt in Algoritme 2.1. De kosten van een sequentiële LU-decompositie bedragen $T_{seq} = 2/3n^3 - n^2/2 - n/6$.

2.2 Parallelle LU-decompositie

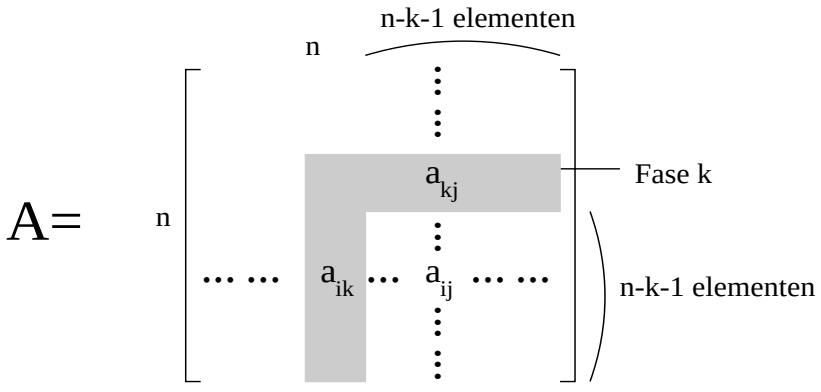
Hoe kan nu een parallelle LU-decompositie gemaakt worden? Zo gauw er meerdere processors in het spel zijn, zal er over het algemeen behoorlijk wat communicatie plaats moeten vinden.

```
input:    De n × n matrix A
output:   De n × n matrix A = L − I − U, met
          n × n eenheid-beneden driehoeksmatrix,
          U = nxn boven driehoeksmatrix,
          π = permutatievector met lengte n, waarin
          opgeslagen staat waarheen welke rij verhuisd.
    for i := 0 to n − 1 do π_i := i;
    for k := 0 to n − 1 do
          r := argmax(|a_ik| : k ≤ i < n);
          swap(π_k, π_r);
          for j := 0 to n-1 do swap(a_kj, a_rj);
          for i := k + 1 to n − 1 do a_ik := a_ik/a_kk;
          for i := k + 1 to n − 1
                for j := k + 1 to n − 1
                      a_ij := a_ij − a_ik a_kj;
```

Algoritme 2.1: Sequentieel LU-algoritme

Om deze communicatiekosten tot een minimum te beperken, moet eerst eens gekeken worden naar wat er precies gecommuniceerd moet worden.

In fase k zullen bij het updaten van $a_{ij} := a_{ij} - a_{ik}a_{kj}$ over het algemeen de elementen a_{ij}, a_{ik} en a_{kj} niet op dezelfde processor liggen en moet er dus gecommuniceerd worden. Zoals in



Figuur 2.2: De matrix A met fase k

figuur (2.2) te zien is, moeten er $(n-k-1)^2$ elementen a_{ij} geupdate worden, waarbij alleen maar gebruik gemaakt wordt van een deel van de elementen uit kolom k van A en van een deel van de elementen van rij k van A. Dus om geen grote hoeveelheden data te versturen, is het handig dat de update van a_{ij} gedaan wordt door de processor die a_{ij} bevat. In fase k hoeft dan alleen kolom en rij k van A verstuurd te worden. Omdat de elementen a_{ij} met $(k < j < n)$ alleen element a_{ik} van kolom k nodig hebben, kan de communicatie nog meer naar beneden gehaald worden door alle matrixrijen te verdelen over een aantal processors M. De communicatie van element a_{ik} komt dan neer op een broadcast naar M processors. Bij de

kolommen geld hetzelfde. Elk element a_{ij} met $(k < i < n)$ heeft uit rij k van A alleen element a_{kj} nodig. De kolommen van A worden dus over N processors verdeeld. In het vervolg wordt er gesproken over processorrijen en processorkolommen.

Het verdelen van de elementen uit A over de processorkolommen en processorrijen wordt gedaan met behulp van een zogenaamde matrix-distributie. Dit is een afbeelding ϕ die aan elk element $A(i,j)$ een processorrij (s) en processorkolom (t) toekent. Dit is als volgt gedefinieerd:

$$\phi : \{(i,j) : 0 \leq i,j < n\} \rightarrow \{(s,t) : (0 \leq s < M) \ \wedge \ (0 \leq t < N)\} \tag{2.1}$$

De functie ϕ heeft twee coördinaten. nl:

$$\phi(i,j) = (\phi_0(i,j), \phi_1(i,j)), \ met \ 0 \leq i,j < n$$

ϕ_0 geeft de verdeling van de processors over de rijen weer en ϕ_1 die over de kolommen. Twee vaak voorkomende distributies zijn een grid distributie en een blok distributie, die ook te zien zijn in figuur (2.3).

s= 0	00	01	02	00	01	02
1	10	11	12	10	11	12
0	00	01	02	00	01	02
M=2 1	10	11	12	10	11	12
0	00	01	02	00	01	02
1	10	11	12	10	11	12
t=	0	1	2	0	1	2

N=3

0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
2	2	2	3	3	3
2	2	2	3	3	3
2	2	2	3	3	3

Figuur 2.3: Links: Grid distributie. Rechts: Blok distributie

De grid-distributie is een speciaal geval van een Cartesische distributie. Bij een Cartesische distributie hangt de verdeling van processors over de kolommen van A niet af van de verdeling van de processors over de rijen van A; $\phi_0(i,j)$ hangt niet af van $\phi_1(i,j)$. Bij de grid-distributie geldt: $0 \leq \phi_0(i,j) < M$ en $0 \leq \phi_1(i,j) < N$. De waarden van $\phi_0(i,j)$ voor opeenvolgende i worden cyclisch verdeeld over $0 \ldots M-1$. Ditzelfde geldt voor ϕ_1.

Het is eenvoudig in te zien dat een grid distributie het beste te gebruiken is voor het parallel oplossen van een LU-decompositie. Als een bepaalde fase k bekeken wordt is in figuur (2.3) is in te zien dat bij een grid distributie veel processoren tegelijkertijd bezig zijn (in dezelfde 'haak' k zitten) en weinig bij een blok distributie. Bij deze laatste is de inefficiëntie vooral te zien als de laatste fase bekeken wordt. Dan is namelijk alleen processor 3 nog bezig.

Hoofdstuk 3

Algoritme en implementatie

Zoals gezegd, is de implementatie van het LU-algoritme in ANSI C, met het software-pakket BSPlib[1]. Er is gebruik gemaakt van een bestaand algoritme *BSP-LU1*, onderdeel van `bsppack` [1] door Rob Bisseling.

Het oorspronkelijke programma `bsplu` is een directe implementatie van het parallelle LU-algoritme *BSP-LU1*, zoals beschreven in [2]. In het kort komt het op het volgende neer:
De $n \times n$-matrix A wordt in een lus doorlopen. Voor iedere fase k wordt de pivot r van de huidige kolom bepaald en rij k met rij r verwisseld. Wanneer k en r op de zelfde processor liggen, kunnen hierbij overschrijvingen plaats vinden en daarom worden de `put`'s eerst locaal gebufferd.

Zoals in figuur 2.2 te zien is, moet een element a_{ij} bijgewerkt worden met alle elementen links van hem, vermenigvuldigd met alle elementen boven hem. Anders gezegd: een element a_{ik} (met $k < i < n$) moet bekend worden gemaakt aan alle elementen in de zelfde rij, rechts van de k-de kolom. Evenzo geldt voor elementen boven de diagonaal, dat zij bekend moeten worden bij elementen in de zelfde kolom (onder de k-de rij). Om een redelijk gebalanceerde h-relatie ($h \approx h_s \approx h_r$) te verkijgen voor deze grote communicatiestap, wordt een *two-phase randomised broadcast* gebruikt. Deze doet eerst een kleine ongebalanceerde stap, door van ieder matrix-element één kopie naar een (telkens andere) processor te sturen. Vervolgens gaan die processoren tegelijkertijd dat elementje naar alle andere processoren (in de zelfde rij c.q. kolom) sturen.

Zodra de elementen locaal bekend zijn, kan op de submatrix $A(k_1 : n-1, k_1 : n-1)$ locaal de rank-1 update worden uitgevoerd.

In [2] staat uitgewerkt, dat de kosten van *BSP-LU1* neerkomen op:

$$T_{\text{BSP-LU1},\sqrt{p}\times\sqrt{p}\text{grid}} \approx \frac{2n^3}{3p} + \frac{n^2}{\sqrt{p}} + \frac{3n^2 g}{\sqrt{p}} + 8nl. \tag{3.1}$$

Hierin is p het aantal processoren, n de matrixdimensie, g de communicatiekosten en l de synchronisatiekosten, allen in flops.

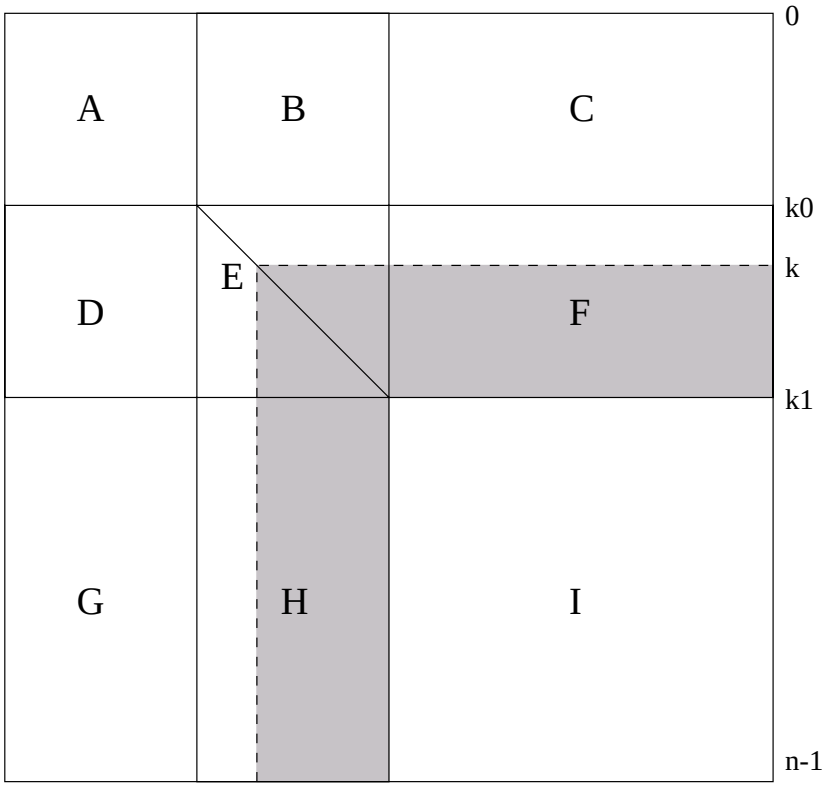
Afgezien van verbeterde communicatie, staan in *BSP-LU1* geen verdere optimalisaties. Er kan echter nog van alles geoptimaliseerd worden.

[1] Voor meer informatie over BSPlib: `http://www.bsp-worldwide.org`

3.1 Optimalisaties in *BSP-LU1*

Eén van de punten waar veel tijdswinst op kan worden gemaakt is het uitstellen van rekenstappen. In dit specifieke geval is het bijvoorbeeld mogelijk om de rank-1 updates uit te stellen en na een aantal iteraties ineens uit te voeren met een matrix-matrix-vermenigvuldiging. Dit laatste kan zeer efficiënt gedaan worden met een voor de machine geoptimaliseerde `dgemm`. `dgemm` is een BLAS-routine die zeer snel $C := \alpha\hat{A}\hat{B} + \beta C$ berekent.
De nu volgende aanpak staat schematisch in figuur 3.1.

In het algoritme worden de rank-1 updates van opeenvolgende fases $k_0 \leq k < k_1$ gecombi-



Figuur 3.1: Opdeling van matrix A in verschillende blokken

neerd tot een enkele matrix-update van de submatrix $A(k_1 : n-1, k_1 : n-1)$, (I in de figuur) aan het eind van fase $k_1 - 1$. Natuurlijk worden binnen de haak van fase k_0 to $k_1 - 1$ (E, H) wel de rank-1 updates uitgevoerd. Ook F valt binnen deze haak, maar de updates hiervan worden ook uitgesteld. E en H moeten wel geupdate worden, omdat op basis van de waarden in de kolommen in E en H gepivoteerd wordt.

Voor het achteraf bijwerken van de submatrix I moet bekend zijn welke updates er zijn uitgesteld. Dit wordt gedaan door de achtereenvolgende rank-1 updates te schrijven als het produkt van een $(n - k_1) \times (k_1 - k_0)$ matrix met een $(k_1 - k_0) \times (n - k_1)$ matrix, afgetrokken van de nog onaangepaste I. Deze twee matrices worden in het vervolg respectievelijk lb en ub genoemd. De update in fase k is te schrijven als $a_{ij} \leftarrow a_{ij} - a_{ik} \cdot a_{kj} \; \forall \; i, j \in \{k_1 \ldots n-1\}$.

Alle opeenvolgende updates tussen k_0 en k_1 zijn dus te schrijven als:

$$a_{ij} \leftarrow a_{ij} - \sum_{k=k_0}^{k_1-1} a_{ik} \cdot a_{kj} \ \forall \ i,j \in \{k_1 \ldots n-1\}$$

De volledige matrixupdate is dus te schrijven als:

$$\begin{array}{rcl} I & \leftarrow & I - \begin{pmatrix} a_{k_1k_0} & a_{k_1k_0+1} & \cdots & a_{k_1k_1-1} \\ a_{k_1+1k_0} & a_{k_1+1k_0+1} & \cdots & a_{k_1+1k_1-1} \\ \vdots & \vdots & \ldots & \vdots \\ a_{n-1k_0} & a_{n-1k_0+1} & \cdots & a_{n-1k_1-1} \end{pmatrix} \cdot \begin{pmatrix} a_{k_0k_1} & a_{k_0k_1+1} & \cdots & a_{k_0n-1} \\ a_{k_0+1k_1} & a_{k_0+1k_1+1} & \cdots & a_{k_0+1n-1} \\ \vdots & \vdots & \ldots & \vdots \\ a_{k_1-1k_1} & a_{k_1-1k_1+1} & \cdots & a_{k_1-1n-1} \end{pmatrix} \\ I & \leftarrow & I - lb \cdot ub \qquad (I = A(k_1 : n-1, k_1 : n-1)) \end{array} \tag{3.2}$$

De twee matrices uit (3.2) zijn in figuur 3.1 in repectievelijk H en F te vinden. In `bsplu` zullen deze matrices gedurende de iteratie tussen k_0 en k_1 opgebouwd worden.

E en H werden tijdens het itereren al bijgewerkt, I is bijgewerkt met bovengenoemde matrixupdate, blijft over het updaten van F. De bovenste (nulde) rij van F (k_0-de rij van A) hoeft niet bijgewerkt te worden, aangezien de updates uit fases 0 tot $k_0 - 1$ al zijn afgerond. Van de eerste rij van F moet $a_{k_0+1k_0}$ maal de nulde rij afgetrokken worden. Algemeen:

$$a_{ij} \leftarrow a_{ij} - \sum_{k=k_0}^{i-1} a_{ik} \cdot a_{kj} \qquad \forall \ i \in \{k_0 \ldots k1-1\}, j \in \{k_1 \ldots n-1\} \tag{3.3}$$

Iedere rij kan dus uit de voorgaande rijen bepaald worden.

Een ander punt van tijdswinst is het communiceren in bulk-hoeveelheden. In dit geval kan dat bijvoorbeeld door rijwisselingen uit te stellen. Althans, delen daarvan; het is namelijk niet nodig om een rijwisseling ook direct in de matrixgedeelten D, G, F en I uit te voeren. Pas wanneer F en I bijgewerkt moeten worden, aan het eind van de $k_1 - 1$-e fase, worden de wisselingen daar uitgevoerd. Ook D en G worden dan bijgewerkt. Voordeel hier is dat alleen de nodige wisselingen worden uitgevoerd. Wanneer bijvoorbeeld tijdens de iteratie rij a met rij b en later rij b met rij c gewisseld is, is het goedkoper om na afloop in één keer rij a met rij c te wisselen. Er wordt dus bijgehouden welke wisselingen uitgevoerd moeten worden, en aan het eind van fase $k_1 - 1$ worden de 'netto-wisselingen' bepaald en uitgevoerd.

Ook in de matrix lb kunnen de wisselingen uitgesteld worden, omdat deze pas op het laatst nodig is bij de matrix-update. Hier ligt het echter een stuk gecomliceerder. In kolom k van A zijn namelijk de eerste $k - k0$ wisselingen al uitgevoerd en die kolom wordt vervolgens in lb gezet. De daarop volgende wisselingen moeten *wel* achteraf uitgevoerd worden. Het aantal uit te voeren wisselingen in lb verschilt dus per kolom. In plaats van te wisselen, worden achteraf in lb de gewisselde rijen volledig opnieuw ingelezen vanuit A.
Zoals reeds eerder gezegd, wordt de matrix ub achteraf bepaald. Dit kan zonder problemen, omdat de wisselingen geen bijwerkingen vereisen in ub. Van een rij in ub moeten namelijk een aantal andere rijen afgetrokken worden, en de voorfactoren van die rijen zijn gewoon meegewisseld.

3.2 Algoritme

Nu zal worden beschreven hoe de aanpassingen op *BSP-LU1*, zoals beschreven in sectie 3.1 in een algoritme gestopt worden.

Het verbeterde algoritme *BSP-LU2* ontvangt de zelfde parameters als *BSP-LU1* en ook nog de blokgrootte

Algoritme *BSP-LU2*
input: $A : n \times n$ matrix, $A = A_0$, distr$(A) = (\phi_0, \phi_1)$,
met $\phi_0(i) = i \bmod M$ en $\phi_1(j) = j \bmod N$.
output: $A : n \times n$ matrix, distr$(A) = (\phi_0, \phi_1)$, $A = L - I + U$, with
$L : n \times n$ eenheid-benedendriehoeksmatrix,
$U : n \times n$ bovendriehoeksmatrix,
π : permutatie vector ter lengte n, distr$(\pi) = (\phi_0, 0)$,
zodat $A_0(\pi_i, j) = (LU)(i, j)$, voor $0 \leq i, j < n$.
tswaps : matrix met 2 kolommen, waarin uitgestelde swaps worden geregistreerd.
blocksize : blokgrootte $k_1 - k_0$

Zoals gezegd, wordt de matrix in blokken bijgewerkt, dus in het algoritme wordt een lus over de blokken gelopen. Het aantal stappen in deze lus is gelijk aan $\lceil n/blocksize \rceil$. De matrixgrootte zal lang niet altijd een veelvoud zijn van de blokgrootte. Alle blokken worden grootte *blocksize* genomen, en het eerste blok wordt passend gemaakt aan wat overblijft binnen de matrix.
Binnen een blok wordt over de rijen/kolommen gelopen, dit worden de fasen k genoemd. Nog buiten alle lussen, wordt de permutatievector π geïnitialiseerd door processorkolom 0.

if $t = 0$ **then for all** $i : 0 \leq i < n \ \wedge \ \phi_0(i) = s$ **do** $\pi_i := i$;

Vervolgens wordt de lus over de blokken gestart en daarin de lus binnen een blok. Als eerste wordt in de k-de kolom de pivot op iedere processor lokaal bepaald (0) en naar iedereen verstuurd (1). Daarna bepalen alle processoren in kolom k redundant de globale pivot r (2) en geven dit door aan de processoren in dezelfde rij (3). De processoren in de k-de en r-de rij doen een put van hun rij-elementen (tussen kolom k_0 en k_1) in de buffer $\hat{\pi}_r$, resp. $\hat{\pi}_k$ van de andere processor (4). Deze stap wordt afgesloten met een synchronisatie, waarna de betrokken processoren de rijen k en r vanuit de buffers definitief in de matrix A zetten (5). De wisseling wordt door alle processoren geregistreerd in *tswaps*, zodat achteraf de andere matrixgedeelten ook gewisseld kunnen worden. Na de wisseling, worden de elementen in de k-de kolom door *pivot* gedeeld wegens de toegepaste *partial row pivoting*.

for $b := 0$ **to** $n - 1$ **step** *blocksize* **do**
  for $k := b$ **to** $b +$ *blocksize* **do**
(0) **if** $k \bmod N = t$ **then** $r_s := \text{argmax}(|a_{ik}| : k \leq i < n \ \wedge \ i \bmod M = s)$;

(1) **if** $k \bmod N = t$ **then** put r_s en $a_{r_s,k}$ in $P(*, t)$;

(2) **if** $k \bmod N = t$ **then**
    $s_{\max} := \text{argmax}(|a_{r_q,k}| : 0 \leq q < M)$;
    $r := r_{s_{\max}}$; $pivot := a_{rk}$;

(3) **if** $k \bmod N = t$ **then** put r in $P(s, *)$;

    $tswaps(k - k_0, 0) := k$; $tswaps(k - k_0, 1) := r$;
(4) **if** $k \bmod M = s$ **then**

if $t = 0$ **then** put π_k als $\hat{\pi}_k$ in $P(r \bmod M, 0)$;
for all $j : k_0 \leq j < k_1 \ \wedge \ j \bmod N = t$ **do**
put a_{kj} als $\hat{a}_{kj}$ in $P(r \bmod M, t)$;
if $r \bmod M = s$ **then**
if $t = 0$ **then** put π_r als $\hat{\pi}_r$ in $P(k \bmod M, 0)$;
for all $j : k_0 \leq j < k_1 \ \wedge \ j \bmod N = t$ **do**
put a_{rj} als $\hat{a}_{rj}$ in $P(k \bmod M, t)$;

(5)
if $k \bmod M = s$ **then**
if $t = 0$ **then** $\pi_k := \hat{\pi}_r$;
for all $j : k_0 \leq j < k_1 \ \wedge \ j \bmod N = t$ **do** $a_{kj} := \hat{a}_{rj}$;
if $r \bmod M = s$ **then**
if $t = 0$ **then** $\pi_r := \hat{\pi}_k$;
for all $j : k_0 \leq j < k_1 \ \wedge \ j \bmod N = t$ **do** $a_{rj} := \hat{a}_{kj}$;
if $k \bmod N = t$ **then**
for all $i : k \leq i < n \ \wedge \ i \bmod M = s$ **do** $a_{ik} := a_{ik}/pivot$;

Nu haak k van de matrix A is bijgewerkt, kunnen deze elementen aan lb en ub worden toegevoegd. Voor de uitgestelde rank-1 update moet lb de grootte van matrixgedeelte H hebben (zie fig.3.1). Om echter aan het eind van een blok matrixgedeelte F te kunnen bijwerken, is de onderdriehoek van E nodig. Hiervoor wordt lb gebruikt en lb heeft dus nog steeds breedte $k_1 - k_0$, maar hoogte $n - k_0$ (i.p.v. $n - k_1$). Ditzelfde geldt voor ub, deze heeft nu breedte $n - k_0$. De werkelijke grootten van lb en ub zullen fors kleiner zijn, aangezien deze lokaal worden opgeslagen en iedere processor dus een eigen kleine matrix heeft. Zoals reeds eerder opgemerkt, wordt het bijwerken van matrixgedeelte F uitgesteld, dus de broadcast naar ub beslaat alleen gebied E en loopt dus van k tot k_1. De broadcast naar lb loopt van k tot n.

Bij het toevoegen van elementen uit A aan lb en ub, wordt een *two-phase randomised broadcast* gebruikt ((6) en (7)). Details hierover staan in [2].

(6)
if $k \bmod N = t$ **then for all** $i : k < i < n \ \wedge \ i \bmod M = s$ **do**
put a_{ik} in $P(s, (i \text{ div } M) \bmod N)$ als $lb_{i-k_0,k-k_0}$;
if $k \bmod M = s$ **then for all** $j : k < j < k_1 \ \wedge \ j \bmod N = t$ **do**
put a_{kj} in $P((j \text{ div } N) \bmod M, t)$ als $ub_{k-k_0,j-k_0}$;

(7)
for all $i : k < i < n \ \wedge \ i \bmod M = s \ \wedge \ (i \text{ div } M) \bmod N = t$ **do**
put $u_{i-k_0,k-k_0}$ in $P(s, *)$;
for all $j : k < j < k_1 \ \wedge \ j \bmod N = t \ \wedge \ (j \text{ div } N) \bmod M = s$ **do**
put $u_{k-k_0,j-k_0}$ in $P(*, t)$;

Het laatste wat nu nog moet plaatsvinden is de rank-1 update van de submatrix onder de k-de haak. Het bijwerken van F en I wordt uitgesteld, dus alleen E en H blijven over:

(0')
for all $i : k < i < n \ \wedge \ i \bmod M = s$ **do**
for all $j : k < j < k_1 \ \wedge \ j \bmod N = t$ **do** $a_{ij} := a_{ij} - a_{ik}a_{kj}$;

Afgezien van wat kleine grens-aanpassingen, was het voorgaande weinig verschillend van het oorspronkelijke algoritme. Wanneer nu echter de lus over k binnen een blok wordt beëindigd, moeten de uitgestelde stappen uitgevoerd worden.

Als eerste worden de netto-wisselingen bepaald uit de gegevens in *tswaps*. Dit wordt op iedere processor redundant gedaan, zodat hierover achteraf geen communicatie meer nodig is. Vervolgens worden alsnog de wisselingen uitgevoerd in de matrixgedeelten D, G, F en I (8).

(8)
for all overall rowswaps $u \to v$ **do**
  if $u \bmod M = s$ **then**
    for all $j : 0 \le j < k_0 \ \wedge \ j \bmod N = t$ **do**
      put a_{uj} in $P(v \bmod M, t)$ als a_{vj}
    for all $j : k_1 \le j < n \ \wedge \ j \bmod N = t$ **do**
      put a_{uj} in $P(v \bmod M, t)$ als a_{vj}

In het voorgaande is er vanuit gaan, dat de put-operatie automatisch gebufferd wordt bij het direct 'putten' in a. Dit is inderdaad het geval in BSPlib.

Het eerste gedeelte van het uitgestelde werk is verricht, nu blijven de rank-1 updates over. Hiervoor moeten eerst de juiste lb en ub bekend zijn. In de lus over k is lb al gedeeltelijk opgebouwd; alleen de gewisselde rijen moeten gecorrigeerd worden. Dit gebeurt door een eenvoudige resend vanuit A naar lb (9). De elementen van lb moeten globaal (in de hele rij) bekend zijn, dus volgt een broadcast van lb (10):

(9)
for all overall rowswaps $u \leftrightarrow v$ **do**
  if $u \bmod M = s$ **then**
    for all $j : k_0 \le j < k_1 \ \wedge \ j \bmod N = t$ **do** $lb_{u,j-k_0} = a_{uj}$;

(10)
for all overall rowswaps $u \leftrightarrow v$ **do**
  if $u \bmod M = s$ **then**
    for all $j : k_0 \le j < k_1 \ \wedge \ j \bmod N = t$ **do**
      put $lb_{u,j-k_0}$ in $P(s, *)$ als $lb_{u,j-k_0}$

Het gedeelte van ub dat nodig is, is nog leeg en moet nog helemaal bepaald worden met rank-1 updates. Van een rij moeten rijen boven hem afgetrokken worden, met voorfactoren die in de onderdriehoek van E staan (opgeslagen in lb). Zodra een element van ub is bepaald, wordt deze waarde ook direct lokaal in A geschreven. (11). De waarden van ub worden lokaal bepaald en worden vervolgens met een *two phase broadcast* aan de rest van de kolom bekend gemaakt ((12) en (13)).

for all $i : k_0 \le i < k_1 \ \wedge \ i \bmod M = s$ **do**

(11)
  for all $j : k_1 \le j < n \ \wedge \ j \bmod N = t$ **do**
    $ub_{i-k_0,j-k_0} := a_{ij}$;
    for $h := k_0$ **to** i **do**
      $ub_{i-k_0,j-k_0} := ub_{i-k_0,j-k_0} - lb_{i-k_0,h-k_0} \cdot ub_{h-k_0,j}$
    $a_{ij} := ub_{i-k_0,j-k_0}$

(12)
    put $ub_{i-k_0,j-k_0}$ in $P((j \text{ div } N) \bmod M, t)$;

(13)
  for all $j : k1 \le j < n \ \wedge \ j \bmod N = t \ \wedge \ (j \text{ div } N) \bmod M = s$ **do**
    put $u_{i-k_0,j-k_0}$ in $P(*, t)$;

Met uitzondering van gedeelte I, is de hele matrix A nu bijgewerkt. lb en ub zijn correct ingevuld, dus de `dgemm` kan aangeroepen worden (14).

(14) **for all** $i : k1 \leq j < n \ \wedge \ i \bmod M = s$ **do**
for all $j : k1 \leq j < n \ \wedge \ j \bmod N = t$ **do**
for $h := 0$ **to** *blocksize* **do**
$a_{ij} := a_{ij} - a_{ih}a_{hj}$;

3.2.1 Kostenanalyse

Het zojuist beschreven algoritme is opgebouwd uit 14 superstappen. Hiervan zijn (0), (2), (5), (0′), (9), (11) en (14) reken superstappen. De kosten worden onder andere geformuleerd in:

$$R_{k,l} = \max_{0 \leq s < M} \{i : k \leq i < l \ \wedge \ \phi_0(i) = s\}, \tag{3.4}$$

oftewel, $R_{k,l}$ is het maximum aantal locale matrixrijen met globale index tussen k en l, en

$$C_{k,l} = \max_{0 \leq t < N} \{j : k \leq j < l \ \wedge \ \phi_1(j) = t\}, \tag{3.5}$$

oftewel, $C_{k,l}$ is het maximum aantal locale matrixkolommen met globale index tussen k en l.

De kosten van stappen (0), (2) en (9) zijn verwaarloosbaar, omdat hierin geen floating-point berekeningen uitgevoerd worden. Superstap (5) kost $T_{(5)} = R_{k+1,n}$, superstap (0') kost $T_{(0')} \leq 2R_{k+1,n}C_{k_0,k_1}$ (C_{k_0,k_1} is een bovengrens voor aantal kolommen tussen k en k_1). In rekenstap (11) kan de lus over j afgeschat worden met een factor $C_{k+1,n}$, de lussen over i en h hangen samen en kunnen dus niet als losse factoren geschreven worden. De lus over i kan worden beschreven als een lus over $0 \leq i < R_{k_0,k_1}$, aangezien alleen voor de lokale rijen werk wordt verricht. De lus over h loopt dan echter tussen 0 en $M \cdot i$, aangezien h wel alle globale elementen in het blok langs loopt. De kosten voor stap (11) komen zo uit op:

$$T_{(11)} = C_{k_1,n} \sum_{i=0}^{R_{k_0,k_1}} \sum_{h=0}^{i \cdot M} 2 \leq C_{k_1,n}(M \cdot R_{k_0,k_1}^2 + (M+2)R_{k_0,k_1} + 2).$$

De laatste rekenstap (14) kost $T_{(14)} = 2R_{k_1,n}C_{k_1,n}blocksize$.

Vervolgens de communicatiekosten; $T_{(1)} = 2(M-1)g$, $T_{(3)} = (N-1)g$, $T_{(4)} = (C_{k_0,k_1}+1)g$, $T_{(6)} = (R_{k+1,n} + C_{k+1,k_1})g$, $T_{(7)} = (R_{k+1,n} + N - 1 + C_{k_0,k_1} + M - 1)g$.
De kosten bij superstap (6) en (7) vereisen enige uitleg. Bij het versturen van een kolom naar *lb* worden door één processor $R_{k+1,n}$ elementen verstuurd. Omdat er niet meer elementen ontvangen kunnen worden dan er verzonden zijn, geldt $h_r \leq h_s = R_{k+1,n}$. Ditzelfde geldt woor de rij broadcast naar *ub*, alleen is de lengte van de verstuurde rij slechts C_{k_0,k_1}. De kosten voor superstap (6) komen dus uit op $T_{(6)} = (R_{k+1,n} + C_{k+1,k_1})g$.
Bij de tweede fase van de broadcast worden maximaal $h_r = R_{k+1,n}$ elementen ontvangen. Door de verspreiding van de elementen in de eerste fase, moeten door één processor nu hoogstens $\lceil R_{k+1,n}/N \rceil$ verschillende elementen verstuurd worden. Van ieder element worden $N-1$ kopieën verstuurd binnen de processorrij, dus $h_s = \lceil R_{k+1,n}/N \rceil (N-1) \leq (R_{k+1,n}/N + 1)(N-1) \leq R_{k+1,n} + N - 1$. Hetzelfde geldt voor de rij-broadcast (ook nu weer met breedte C_{k_0,k_1}), dus $T_{(7)} = (R_{k+1,n} + N - 1 + C_{k_0,k_1} + M - 1)g$.

Tot slot zijn er nog communicatiestappen, die na afloop van een blok worden uitgevoerd: $T_{(8)} = ((C_{0,n} - C_{k_0,k_1}) \cdot blocksize)g$, $T_{(10)} = (blocksize \cdot C_{k_0,k_1} \cdot (N-1))g$, $T_{(12)} = (R_{k_0,k_1} \cdot C_{k_1,n})g$ en $T_{(13)} = R_{k_0,k_1}(C_{k_1,n} + M - 1)g$.
Bij het wisselen van de rijen in superstap (8) hoeft het middenstuk $k_0 \leq k < k_1$ niet meegenomen te worden, vandaar breedte $C_{0,n} - C_{k_0,k_1}$. Aangezien een processor maximaal *blocksize*

rijen heeft verwisseld met andere processoren, geldt $T_{(8)} = ((C_{0,n} - C_{k_0,k_1}) \cdot blocksize)g$. Vervolgens de resend naar lb; voor maximaal $blocksize$ wisselingen moeten C_{k_0,k_1} elementen aan $N-1$ processoren worden gestuurd, dus $T_{(10)} = (blocksize \cdot C_{k_0,k_1} \cdot (N-1))g$. In de gecombineerde superstap (11), (12) en (13) kost de eerste put $T_{(12)} = (R_{k_0,k_1} \cdot C_{k_1,n})g$. De tweede put is wat gecompliceerder, maar lijkt weer erg veel op de vorige tweede fase van een broadcast (7), maar nu voor R_{k_0,k_1} rijen: $h_r = ((blocksize - R_{k_0,k_1})C_{k_1,n})g$ en $h_s = (R_{k_0,k_1}(C_{k_1,n}/M + 1)(M-1))g$, zodat $T_{(13)} = (R_{k_0,k_1}(C_{k_1,n}/M + 1)(M-1))g \approx R_{k_0,k_1}(C_{k_1,n} + M - 1)g$.

Voor het verkrijgen van een bovengrens voor de kosten, is de volgende afschatting te maken:

$$R_{k,l} \leq \left\lceil \frac{l-k}{M} \right\rceil \leq \frac{l-k}{M} + 1$$

Natuurlijk is dit ook voor de kolommen te doen:

$$C_{k,l} \leq \left\lceil \frac{l-k}{N} \right\rceil \leq \frac{l-k}{N} + 1$$

De totale kosten van *BSP-LU2* worden verkregen door de kosten van alle superstappen op te tellen. In (3.6) geldt: $B = blocksize$.

$$\begin{aligned}
T_{\text{BSP-LU2}} &= \sum_{k=0}^{n-1} T_{(1)} + T_{(3)} + T_{(4)} + T_{(5)} + T_{(6)} + T_{(7)} + T_{(0')} \\
&\quad + \sum_{b=0}^{n/b+1} T_{(8)} + T_{(10)} + T_{(11)} + T_{(12)} + T_{(13)} + T_{(14)} \\
&= \sum_{k=0}^{n-1} 2\frac{n-k-1}{M}g + \left(\frac{n}{B} + 1\right) \sum_{k=1}^{B-1} \frac{k}{N} g \\
&\qquad + \sum_{k=0}^{n-1} \left(\left(\frac{B}{N} + 1\right) \cdot 2 + 1\right)\left(\frac{n-k-1}{M} + 1\right) + \left(3M + 2N + 2\frac{B}{N} + 1\right) g \\
&\quad + \left(\frac{B^2}{M} + 3B + 2M + 2\frac{B}{M} + 4\right) \sum_{b=1}^{\frac{n}{B}+1} \left(\frac{n-bB}{N} + 1\right) \\
&\qquad + 2B \sum_{b=1}^{\frac{n}{B}+1} \left(\frac{n-bB}{N} + 1\right)\left(\frac{n-bB}{M} + 1\right) \\
&\qquad + (n+B)\left(\frac{n-2B}{N} + N + B - 1\right) g + \left(\frac{B}{M} + 1\right) \sum_{b=1}^{\frac{n}{B}+1} \left(2\frac{n-bB}{N} + 1 + M\right) g \\
&\quad + \left(8n + 6\left(\frac{n}{B} + 1\right)\right) l
\end{aligned} \tag{3.6}$$

Een optimale bovengrens voor (3.6) ontstaat wanneer $R_{i,j} = C_{i,j}$ en $N = M = \sqrt{P}$. Weg-

strepen van de termen lager dan $\mathcal{O}(n)$, levert voor de kosten:

$$\begin{aligned} T_{\text{BSP-LU2}} \approx \frac{n^2}{p}\left(1+\frac{1}{2}B+\frac{2}{3}n\right) + \frac{n^2}{B}\left(1+\frac{12}{\sqrt{p}}+\frac{5B}{\sqrt{p}}\right) \\ + \left(\frac{n^2}{B\sqrt{p}}+\frac{n^2}{p}+\frac{2n^2}{\sqrt{p}}\right)g \\ + \left(8n+6\left(\frac{n}{B}+1\right)\right)l \end{aligned} \tag{3.7}$$

In vergelijking met (3.1) zijn de termen $\frac{2}{3}\frac{n^3}{p}$ en $\frac{n^2}{\sqrt{p}}g$ gelijk gebleven, de term $\frac{n^2}{\sqrt{p}}$ is hier nog door B gedeeld, en de synchronisatiekosten liggen nu hoger. Verder zijn er nog wat extra lagere orde termen. In het programma liggen de synchronisatiekosten lager dan in (3.7) staat, aangezien er superstappen samengenomen worden.

3.3 Implementatie

Hier zullen in het kort wat details van de programmacode van `bsplu` worden toegelicht. De volledige listing staat in appendix B.

Het programma is een directe uitwerking van het algoritme, met wat kleine verschillen. Omdat BSPlib in één superstap zowel berekeningen als communicatie toestaat, kunnen superstappen gecombineerd worden. Zo worden superstappen (0) en (1) uit het algoritme gecombineerd tot `Superstep 0` in het programma. Superstap (2) en (3) worden gecombineerd tot `Superstep 1`, superstap (5) en (6) worden gecombineerd tot `Superstep 3`, superstap (9) en (10) worden gecombineerd tot `Superstep 6` en superstap (11) en (12) worden gecombineerd tot `Superstep 7`. Superstap (0') staat in `Superstep 0'`, maar kost geen extra synchronisatie, omdat dit pas in de volgende lus, in `Superstep 0` nodig is. De rest van de superstappen (4), (7), (8), (13) en (14) zijn in het programma ook losse `Superstep`'s, resp. `2`, `4`, `5`, `8` en `9`.

Bovenin de methode `bsplu` worden allerlei hulp-variabelen aangemaakt; `lk` en `uk` komen overeen met respectievelijk *lb* en *ub* uit het algoritme. `lbw` (local blockwidth) en `lbh` (local blockheight) zijn respectievelijk C_{k_0,k_1} en R_{k_0,k_1}. De rest van de variabelen worden uit het commentaar duidelijk.

De matrix wordt in blokstappen doorlopen, maar omdat *blocksize* geen deler van n hoeft te zijn, heeft het eerste blok grootte (n mod *blocksize*). Vervolgens wordt ieder blok doorlopen, over `Superstep 0/0'` t/m `7`. Bij het uitvoeren van een rijwisseling, wordt in `ts` de globale waarde van `k` en `r` geschreven. Op de processoren die verwisselde elementen hebben, wordt daarnaast in `raffect` opgeslagen dat de betreffende rij veranderd (affected) is. Aan het eind van het blok wordt nu door iedere processor `determineSwaps` aangeroepen. Deze methode bepaalt voor de betreffende processor, waar de verwisselde rijen uiteindelijk terecht gekomen zijn, en levert dit op in een matrix `swaps`. Aangezien alleen de rijen uit `raffect` zijn nagelopen, is de eerste index van deze matrix (`swaps[i][0]`) alleen maar van rijen die op de huidige processor liggen. Hierdoor kan iedere processor zijn eigen `swaps` doorlopen en de betreffende elementen naar rij `swaps[i][1]` putten. Er wordt niet extra gebufferd, omdat `bsp_put` dat al doet (`Superstep 5`).
Dezelfde matrix `swaps` kan nogmaals gebruikt worden, om de veranderde rijen in `lk` bij te werken. Dit gebeurt met een eenvoudige broadcast; *two phase* is hier niet nodig, omdat alle kolommen tussen `k0` en `k1` van verschillende rijen verstuurd moeten worden en de meeste processoren dus toch al tegelijkertijd bezig zijn (`Superstep 6`).

Vervolgens de bepaling van `uk`; het bij te werken blok wordt van rij `k0` tot `k1` doorlopen, de eerste gegevens worden uit `a` gehaald en vervolgens worder de voorgaande rijen er vanaf getrokken (`Superstep 7`). De uiteindelijke waarde van `uk` wordt in `a` geschreven, en met een *two phase broadcast* naar alle processorrijen verstuurd (`Superstep 7` en `8`). Deze broadcast gebeurt binnen de lus, omdat de huidige rij van `uk` van andere rijen afgetrokken moet worden. Tot slot wordt met een `SGEMM` het laatste gedeelte van `a` bijgewerkt (`Superstep 9`).

Hoofdstuk 4

Experimenten

Om de effectiviteit van de nieuwe *BSP-LU2* te onderzoeken, zijn verschillende experimenten uitgevoerd. Vooraf zijn echter al een aantal voorspellingen te doen over de looptijd.

4.1 Invloed van blokgrootte

De cruciale verandering die in *BSP-LU2* is in gebouwd, is het doorlopen in blokken. Een interessante vraag is nu, hoe groot deze blokken moeten zijn. Voor een gegeven matrixgrootte n en aantal processoren p is te bepalen wat de verwachte kosten T zijn, uitgedrukt in *blocksize*, g en l. De laatste twee kunnen ingevuld worden, wanneer de machine-parameters bekend zijn. De afgeleide naar *blocksize* van deze functie T op 0 stellen, levert een minimum van de kosten. Voor kosten-funtie (3.7) met $n = 500$, $p = 32$, $l = 1250$ en $g = 1.9$ levert dit een minimum op bij $blocksize \approx 34$. In het gebruikte programma zijn de synchronisatiekosten echter iets minder, vanwege de gecombineerde superstappen: $T_{\text{sync}} = (4n + 5(n/B + 1))l$. Hierbij ligt het minimum op $blocksize \approx 15$.

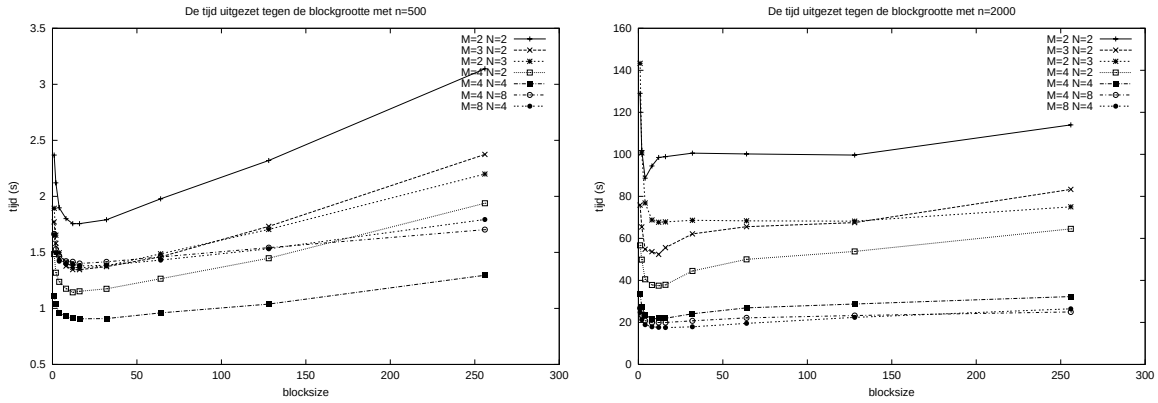
Om de invloed van de *blocksize* experimenteel te onderzoeken, is het programma `bsplu` op de Cray T3E van de TU Delft verscheidene malen uitgevoerd, voor verschillende parameters n, p, m, N en *blocksize*. De volledige output is opgenomen in appendix A.

In figuur 4.1 is te zien dat de optimale blokgrootte rond 15 ligt, zowel voor $n = 500$ als $n = 2000$. Dit komt dus inderdaad goed overeen met de hiervoor voorspelde blokgrootte.
Wat opvalt, is dat experimenten met 8×4 en 4×8 processoren slechter zijn dan met minder processoren. Misschien dat de extra communicatiekosten bij deze configuratie zeer sterk terugkomen in de totale kosten, maar het verschil is dusdanig groot dat er mogelijk ook externe factoren bij betrokken zijn. Misschien was de computer op moment van uitvoeren druk bezet.

4.2 Invloed van SGEMM

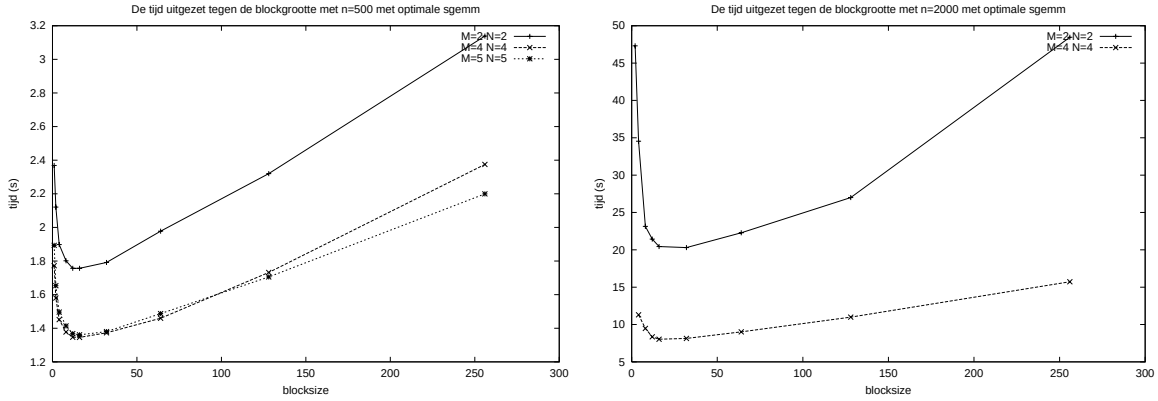
Om de invloed van de geoptimaliseerde `SGEMM` op de tijd te onderzoeken, zijn weer experimenten uitgevoerd, maar nu met `SGEMM` in superstap (14).

Als naar de tijd in figuur (4.2) gekeken wordt en dit wordt vergeleken met de tijd in figuur (4.1) wordt duidelijk dat er flink tijdswinst is gemaakt. Het verloop van de curven is nog steeds het zelfde; het optimum ligt nog steeds rond blokgrootte 15.
De afname van de kosten bedraagt soms zelfs een factor 5. Bij de meeste configuraties was dit



Figuur 4.1: Invloed van blokgrootte op tijd

een factor 2 à 3. In kostenformule (3.7) is de term $\frac{2}{3}\frac{n^3}{p}$ het grootst en die is afkomstig van de matrixvermenigvuldiging in de laatste superstap (14). De gebruikte matrixvermenigvuldiging wordt door `SGEMM` versneld, en dit is de belangrijkste term in de kostenformule, zodat de kosten drastisch afnemen.
`SGEMM` heeft dezelde kosten $\mathcal{O}(\frac{n^3}{p})$, maar de voorterm is een flinke factor kleiner.
Dit verklaart de enorme tijdswinst bij gebruik van `SGEMM`.



Figuur 4.2: Invloed van blokgrootte op tijd met SGEMM

Hoofdstuk 5

Conclusie

Het blijkt dat uitstellen van werk – zowel rekenoperaties als communicaties – kostenbesparing oplevert. Door het uitstellen van rekenoperaties, konden opeenvolgende rank-1 updates met een matrixvermenigvuldiging uitgevoerd worden, hetgeen erg snel is met de BLAS-operatie `SGEMM`. Voor verdere optimalisaties zouden eventueel nog andere berekeningen door BLAS-operaties vervangen kunnen worden.

Verder heeft het uitstellen van communicaties duidelijk nut; alleen de nodige communicatie wordt gedaan ('netto' rijwisselingen) en deze kan in grote (bulk) hoeveelheden gedaan worden. Voor verdere optimalisaties kunnen een aantal `bsp_put`'s door `bsp_hpput`'s vervangen worden. Deze `put`'s worden niet automatisch gebufferd, maar worden direct uitgevoerd. Dit kan bij alle `bsp_put`'s, behalve in superstap 8, want daar is uitgegaan van het bufferen door BSP.

Tot slot de keuze van de blokgrootte. Bij grotere blokgrootte worden meer operaties uitgesteld, maar broadcasts en resends naar lb en ub worden groter, dus stijgen de communicatiekosten. Het optimum vindt een goede balans tussen deze twee kostensoorten.

Bibliografie

[1] Rob. H. Bisseling, `BSPPACK_EDU04`, http://www.math.uu.nl/people/bisselin/software.html

[2] Rob. H. Bisseling, Parallel Scientific Computing with BSP, draft manuscript, 1999

Bijlage A

Output van experimenten

In dit hoofdstuk zijn de data te vinden die verkregen zijn door het geoptimaliseerde programma *BSP-LU2* te testen op een parallelle supercomputer, de Cray T3E van de TU Delft. Deze computer
De tijd is gemeten voor verschillende samenstellingen van processorrijen M en processorkolommen N. Ook is bekeken wat invloed van de *blocksize* en de matrixgrootte n op de tijd is. Er zijn zowel voor de eenvoudige als voor de geoptimaliseerde dgemm experimenten uitgevoerd.

A.1 Output bij eenvoudige DGEMM

matrixgrootte 500

#N	M	b	n	tijd
2	2	1	500	2.368704
2	2	2	500	2.120843
2	2	4	500	1.898192
2	2	8	500	1.801276
2	2	12	500	1.756456
2	2	16	500	1.756533
2	2	32	500	1.791478
2	2	64	500	1.977850
2	2	128	500	2.320164
2	2	256	500	3.138626

N	M	b	n	tijd
3	2	1	500	1.772262
3	2	2	500	1.579920
3	2	4	500	1.451558
3	2	8	500	1.377712
3	2	12	500	1.346700
3	2	16	500	1.345787
3	2	32	500	1.372680
3	2	64	500	1.459721
3	2	128	500	1.731982
3	2	256	500	2.374543

N	M	b	n	tijd
2	3	1	500	1.893521
2	3	2	500	1.653476
2	3	4	500	1.496113
2	3	8	500	1.413652
2	3	12	500	1.368167
2	3	16	500	1.360925
2	3	32	500	1.380002
2	3	64	500	1.487234
2	3	128	500	1.705109
2	3	256	500	2.199608

4	2	1	500	1.484942
4	2	2	500	1.318867
4	2	4	500	1.237282
4	2	8	500	1.175442
4	2	12	500	1.143660
4	2	16	500	1.153179
4	2	32	500	1.174771
4	2	64	500	1.265133
4	2	128	500	1.447882
4	2	256	500	1.938296

4	4	1	500	1.115013
4	4	2	500	1.036866
4	4	4	500	0.961687
4	4	8	500	0.929478
4	4	12	500	0.910848
4	4	16	500	0.907006
4	4	32	500	0.909223
4	4	64	500	0.959945
4	4	128	500	1.038341
4	4	256	500	1.295381

```
 4 8 1   500 1.664791      8 4 1   500 1.654687
 4 8 2   500 1.528550      8 4 2   500 1.496921
 4 8 4   500 1.465128      8 4 4   500 1.421763
 4 8 8   500 1.421406      8 4 8   500 1.394281
 4 8 12  500 1.416388      8 4 12  500 1.394516
 4 8 16  500 1.400820      8 4 16  500 1.379088
 4 8 32  500 1.415753      8 4 32  500 1.385197
 4 8 64  500 1.459502      8 4 64  500 1.431395
 4 8 128 500 1.542468      8 4 128 500 1.531947
 4 8 256 500 1.703189      8 4 256 500 1.793731
```

matrixgrootte 2000

```
#N M  b   n    tijd           N M  b   n    tijd          N M  b   n    tijd
 2 2 1   2000 129.009762      3 2 1   2000 75.839755     2 3 1   2000 143.297054
 2 2 2   2000 101.783808      3 2 2   2000 65.458482     2 3 2   2000 100.674603
 2 2 4   2000 88.645520       3 2 4   2000 54.822649     2 3 4   2000 76.833152
 2 2 8   2000 94.475817       3 2 8   2000 53.677356     2 3 8   2000 68.770145
 2 2 12  2000 98.530908       3 2 12  2000 52.369288     2 3 12  2000 67.688672
 2 2 16  2000 98.889151       3 2 16  2000 55.574940     2 3 16  2000 67.842358
 2 2 32  2000 100.576748      3 2 32  2000 62.150635     2 3 32  2000 68.605736
 2 2 64  2000 100.185375      3 2 64  2000 65.565507     2 3 64  2000 68.341857
 2 2 128 2000 99.651921       3 2 128 2000 67.501658     2 3 128 2000 68.176312
 2 2 256 2000 114.024690      3 2 256 2000 83.330044     2 3 256 2000 75.010166
```

```
 4 2 1   2000 56.673900       4 4 1 2000 33.684447
 4 2 2   2000 49.882888       4 4 2 2000 27.511828
 4 2 4   2000 40.538411       4 4 4 2000 23.634033
 4 2 8   2000 37.815462       4 4 8 2000 21.625753
 4 2 12  2000 37.484557       4 4 12 2000 21.918540
 4 2 16  2000 37.917900       4 4 16 2000 21.992885
 4 2 32  2000 44.506840       4 4 32 2000 24.113928
 4 2 64  2000 50.035417       4 4 64 2000 26.914493
 4 2 128 2000 53.787352       4 4 128 2000 28.750207
 4 2 256 2000 64.545980       4 4 256 2000 32.319538
```

```
 4 8 1   2000 26.790684       8 4 1 2000 24.786767
 4 8 2   2000 23.290669       8 4 2 2000 21.316969
 4 8 4   2000 20.810319       8 4 4 2000 18.891798
 4 8 8   2000 19.879664       8 4 8 2000 17.944915
 4 8 12  2000 19.952845       8 4 12 2000 17.640950
 4 8 16  2000 19.963029       8 4 16 2000 17.541922
 4 8 32  2000 20.768417       8 4 32 2000 17.908213
 4 8 64  2000 22.173920       8 4 64 2000 19.553056
 4 8 128 2000 23.271705       8 4 128 2000 22.400600
 4 8 256 2000 24.996838       8 4 256 2000 26.475575
```

A.2 Output bij geoptimaliseerde SGEMM

Matrixgrootte 500

```
#N M  b   n    tijd        N M  b   n    tijd        N M  b   n    tijd
 2 2 2   500 1.381677
 2 2 4   500 1.195831       4 4 4   500 0.795326
 2 2 8   500 1.074398       4 4 8   500 0.755660
 2 2 12  500 1.062860       4 4 12  500 0.743019
 2 2 16  500 1.035095       4 4 16  500 0.744165
 2 2 32  500 1.076707       4 4 32  500 0.756650       5 5 32  500 1.195435
 2 2 64  500 1.219461       4 4 64  500 0.795494       5 5 64  500 1.226950
 2 2 128 500 1.543975       4 4 128 500 0.900339       5 5 128 500 1.337933
 2 2 256 500 2.520067       4 4 256 500 1.149659       5 5 256 500 1.552943
```

Matrixgrootte 2000

```
 2 2 2   2000 47.311200
 2 2 4   2000 34.535173     4 4 4   2000 11.295492
 2 2 8   2000 23.135696     4 4 8   2000 9.483359
 2 2 12  2000 21.438220     4 4 12  2000 8.351119
 2 2 16  2000 20.440932     4 4 16  2000 8.029337
 2 2 32  2000 20.304247     4 4 32  2000 8.134558
 2 2 64  2000 22.285107     4 4 64  2000 9.014943
 2 2 128 2000 26.995756     4 4 128 2000 10.986467     5 5 128 2000 14.513082
 2 2 256 2000 48.453424     4 4 256 2000 15.727450     5 5 256 2000 18.089754
```

Bijlage B

Listing bsplu

```
#include "bsppack.h"

int f(int k, int u, int R){
    /* Compute min(m: m*R+u>=k) for k>=0 and 0<=u<R */
    return ( u<k%R ?  k/R+1 : k/R );
} /* end f */

void bsplu(int M, int N, int s, int t, int n, int *pi, double **a,
           int blocksize){
    /* Compute LU decomposition of A with partial pivoting
       Program text for P(s,t) = processor s+t*M, with 0<=s<M and 0<=t<N
       A is distributed according to the M by N grid distribution.
       Author: Rob Bisseling.  Date: 1-6-1996. Last modified: 13-11-96.
       Copyright reserved.
       Modified by: Gerben Wolterink, Joost Rommes,
                    Lars Hemel and Arthur van Dam. Date: 6-12-1999
    */

    void determineSwaps(int n, int **ts,int naffect, int *raffect, int **swap);
    int f(int k, int u, int R);

    double *ak, *ar, *uk, *lk, *gmax, absmax, max, pivot, *pa ;
    int nr, nc, k, i, r, pik, pir, *gimax, lbw, lbh, k0r, k0c, k1r,
        k1c, b, k0, k1, kr, kr1, kc, kc1, j, imax, s1, t1, g, h;
    int **ts,         /* Used to store swapped rowindices redundantly */
        **swap,       /* Used to store global swap indices without
                         intermediate swaps */
        *raffect;     /* Used to store rowindices of local swapped rows */
    int naffect,      /* Used to count number of swapped rows */
        tmpBlocksize; /* Temporary storage of original blocksize */
    int wA, hA, block, lda, ldb, ldc;
    double alpha, beta;

    bsp_push_reg(&r,SZINT);
    bsp_push_reg(&pik,SZINT);
    bsp_push_reg(&pir,SZINT);
    nr=  f(n,s,M);  /* number of local rows */
    nc=  f(n,t,N);  /* number of local columns */
    lbh = (int)ceil((double)blocksize/(double)M);  /* number of local
                                                      rows within block*/
    lbw = (int)ceil((double)blocksize/(double)N);  /* number of local
```

```
                                                columns within block*/
ts = matalloci(blocksize,2);
swap = matalloci(blocksize + lbh,2);
raffect = vecalloci( blocksize + lbh ) ;

ak= vecallocd(lbw);  bsp_push_reg(ak,lbw*SZDBL);
ar= vecallocd(lbw);  bsp_push_reg(ar,lbw*SZDBL);
uk= vecallocd(nc*blocksize);  bsp_push_reg(uk,nc*blocksize*SZDBL);
lk= vecallocd(nr*blocksize);  bsp_push_reg(lk,nr*blocksize*SZDBL);
gmax= vecallocd(M);  bsp_push_reg(gmax,M*SZDBL);
gimax= vecalloci(M);  bsp_push_reg(gimax,M*SZINT);
pa = a[0];  bsp_push_reg(pa,nc*nr*SZDBL);

/* Initialise permutation vector pi */
if (t==0){
    for(i=0; i<nr; i++)  pi[i]= i*M+s; /* global row index */
}
bsp_sync();

tmpBlocksize = blocksize ;
if( n%blocksize != 0 ) blocksize = n%blocksize ;

for (b = 0; b < n; b += blocksize){
    if( b > 0 ) blocksize = tmpBlocksize;
    k0 = b; k1 = b+blocksize;
    k0r = f(k0,s,M);  k0c = f(k0,t,N);
    k1r = f(k1,s,M);  k1c = f(k1,t,N);
    lbw = k1c - k0c;  lbh = k1r-k0r;
    naffect = 0;

    for (k=k0; k < k1; k++){
        /****** Superstep 0 ******/
        kr= f(k,s,M); /* first local row with global index >= k */
        kr1= f(k+1,s,M);
        kc= f(k,t,N);
        kc1= f(k+1,t,N);

        if (k%N==t){   /* k=kc*N+t */
        /* Search for local absolute maximum in column k of A */
            absmax= 0.0;
            for (i=kr; i<nr; i++){
                if (fabs(a[i][kc])>absmax){
                    absmax= fabs(a[i][kc]);
                    imax= i;
                }
            }
            if (absmax > EPS)  max= a[imax][kc];
            else {  max= 0.0; imax= -1;  }

            /* Broadcast value and local index of maximum to P(*,t) */
            for(s1=0; s1<M; s1++){
                bsp_put(s1+t*M,&max,gmax,s*SZDBL,SZDBL);
                bsp_put(s1+t*M,&imax,gimax,s*SZINT,SZINT);
            }
        }
        bsp_sync();
```

```
/****** Superstep 1 ******/
if (k%N==t){
/* Determine global absolute maximum (redundantly) */
    absmax= 0.0; r=-1;
    for(s1=0; s1<M; s1++){
        if (fabs(gmax[s1])>absmax){
            absmax= fabs(gmax[s1]);
            r= gimax[s1]*M+s1; /* global index of pivot row */
            pivot= gmax[s1];
        }
    }
    if (r==-1 && s==0){ /* no pivot found */
        bsp_abort("bsplu at stage %d: matrix is singular\n",k);
    }

    /* Broadcast index of pivot row to P(*,*) */
    for(t1=0; t1<N; t1++){
        bsp_put(s+t1*M,&r,&r,0,SZINT);
    }
}
bsp_sync();

/****** Superstep 2 ******/
ts[k-k0][0] = k; /* Store rowindices of swapped rows redundantly */
ts[k-k0][1] = r;

if (k%M==s){
    raffect[naffect] = k; /* Store locally affected row */
    naffect ++;
    /* Store pi(k) in pi(r) on P(r%M,0) */
    if (t==0)  bsp_put(r%M,&pi[k/M],&pik,0,SZINT);
    /* Store row k in ak on P(r%M,t) */
    bsp_put(r%M+t*M,&a[k/M][k0c],ak,0,lbw*SZDBL);
}
if (r%M==s){
    raffect[naffect] = r;
    naffect ++;
    if (t==0)  bsp_put(k%M,&pi[r/M],&pir,0,SZINT);
    bsp_put(k%M+t*M,&a[r/M][k0c],ar,0,lbw*SZDBL);
}
bsp_sync();

/****** Superstep 3 ******/
if (k%M==s){
    if (t==0)  pi[k/M]=pir;
    /* Assign row r to A(k,*), but only from column k0 to k1 */
    for(j=0; j<lbw; j++)  a[k/M][j+k0c]= ar[j];
}
if (r%M==s){
    if (t==0)  pi[r/M]=pik;
    for(j=0; j<lbw; j++)  a[r/M][j+k0c]= ak[j];
}

/* First phase of two-phase randomised broadcast */
if (k%M==s){
    /* Store new row k in uk - first phase */
    for(j=kc1; j<k1c; j++){
```

```
        bsp_put(j%M+t*M,&a[kr][j],uk, ((j-k0c)*blocksize+k-k0)*SZDBL,SZDBL);
        }
    }
    if (k%N==t){   /* k=kc*N+t */
        /* Compute column k and store it in lk - first phase */
        for(i=kr1; i<nr; i++){
        a[i][kc] /= pivot;
        bsp_put(s+(i%N)*M,&a[i][kc],lk, ((k-k0)*nr+i-k0r)*SZDBL,SZDBL);
        }
    }
    bsp_sync();

    /****** Superstep 4 ******/
    /* Second phase of two-phase randomised broadcast */
    /* Store row k in uk - second phase */
    for(j=f(kc1,s,M)*M+s; j<k1c; j +=M){
        for(s1=0; s1<M; s1++){
            bsp_put(s1+t*M,&uk[(j-k0c)*blocksize+k-k0],uk,
                  ((j-k0c)*blocksize+k-k0)*SZDBL,SZDBL);
        }
    }
    /* Store column k in lk - second phase */
    for(i=f(kr1,t,N)*N+t; i<nr; i +=N){
        for(t1=0; t1<N; t1++){
            bsp_put(s+t1*M,&lk[(k-k0)*nr+i-k0r],lk,
                  ((k-k0)*nr+i-k0r)*SZDBL,SZDBL);
        }
    }
    bsp_sync();

    /****** Superstep 0' ******/
    /* Rank-1 update of A, only from column k to k1, global rows k to n */
    for(i=kr1; i<nr; i++){
       for(j=kc1; j<k1c; j++){
           a[i][j] -= lk[(k-k0)*nr+i-k0r]*uk[(j-k0c)*blocksize+k-k0];
       }
    }
}/* end of loop over one block */

/****** Superstep 5 ******/
determineSwaps(blocksize, ts,naffect,raffect,swap);
/* swap now contains the entire swaps (without intermediate swaps) */

/* execute postponed row-swaps */
for(i = 0; i < naffect; i++){
    if(swap[i][0] != -1){ /* update postponed swaps */
        bsp_put(swap[i][1]%M+t*M,a[swap[i][0]/M], pa,
               (swap[i][1]/M*nc)*SZDBL,k0c*SZDBL);
        bsp_put(swap[i][1]%M+t*M,&a[swap[i][0]/M][k1c], pa,
               (swap[i][1]/M*nc + k1c)*SZDBL,(nc-k1c)*SZDBL);
    }
}
bsp_sync();

/****** Superstep 6 ******/
for(i = 0; i < naffect; i++){ /* resend data to L */
    if(swap[i][0] != -1){
```

```
        for(j = k0c; j < k1c; j++){
            lk[(j*N+t-k0)*nr+swap[i][0]/M-k0r] = a[swap[i][0]/M][j];
           /* kopieer eigen data uit A naar lk */
        }
    }
}

/* Broadcast of affected rows of L to all procs in this row */
for(i = 0 ; i < naffect ; i++){
    if(swap[i][0] != -1){
        for(h = k0c ; h < k1c ; h++){
            for(j = 0 ; j < N ; j++){
                bsp_put( s+j*M,&lk[(h*N+t-k0)*nr+swap[i][0]/M-k0r],
                        lk, ((h*N+t-k0)*nr+swap[i][0]/M-k0r)*SZDBL, SZDBL );
            }
        }
    }
}

/* Construction of u */
for(i = k0; i < k1; i++){
    /****** Superstep 7 ******/
    if(i%M == s){
        for(j = k1c; j < nc; j++){
            uk[(j-k0c)*blocksize+i-k0] = a[i/M][j];
            for(h = k0; h < i; h++){
                uk[(j-k0c)*blocksize+i-k0] -= lk[(h-k0)*nr+i/M-k0r]*
                   uk[(j-k0c)*blocksize+h-k0];
            }
            /* update a */
            a[i/M][j] = uk[(j-k0c)*blocksize+i-k0];

            /* Two fase broadcast of u , Store u in uk - first fase  */
            bsp_put(j%M+t*M,&uk[(j-k0c)*blocksize+i-k0],uk,
                 ((j-k0c)*blocksize+i-k0)*SZDBL,SZDBL);
        }
    }
    bsp_sync() ;
    /****** Superstep 8 ******/
    /* Store u in uk - second fase */
    for(j=f(k1c,s,M)*M+s; j<nc; j +=M){
        for(s1=0; s1<M; s1++){
            bsp_put(s1+t*M,&uk[(j-k0c)*blocksize+i-k0],uk,
               ((j-k0c)*blocksize+i-k0)*SZDBL,SZDBL);
        }
    }
    bsp_sync();
}/* End of construction of U */

/****** Superstep 9 ******/
/* straightforward DGEMM */
/*
for(j = k1c; j < nc; j++){
    for(i = k1r; i < nr; i++){
        for(h = 0; h < blocksize; h++){
            a[i][j] -= uk[(j-k0c)*blocksize + h] * lk[h*nr+i-k0r];
        }
```

```
            }
        }
        */
        /* efficient SGEMM */
        wA=nr-k1r; hA=nc-k1c; block=blocksize; alpha= -1; beta = 1;
        lda=tmpBlocksize; ldb=nr; ldc=nr;

        SGEMM( modet, modet, &wA, &hA, &block, &alpha,&uk[(k1c-k0c)*blocksize],
               &lda, &lk[k1r-k0r],&ldb,&beta, &a[k1r][k1c], &ldc );
        bsp_sync();
    }
    bsp_pop_reg(gimax);  free(gimax);
    bsp_pop_reg(gmax);  free(gmax);
    bsp_pop_reg(lk);  free(lk);
    bsp_pop_reg(uk);  free(uk);
    bsp_pop_reg(ar);  free(ar);
    bsp_pop_reg(ak);  free(ak);
    bsp_pop_reg(&pir);  bsp_pop_reg(&pik);
    bsp_pop_reg(&r);  bsp_pop_reg(pa) ;
    matfreei(ts);
    matfreei(swap);

    bsp_sync();
} /* end bsplu */

/* determineSwaps determines the necessary swaps */
void determineSwaps(int n, int **ts,int naffect, int *raffect, int **swap){
    int i, j, x;
    void unique(int n, int *x);
    unique(naffect, raffect);
    for(j = 0; j < naffect; j++){
        x = raffect[j];
        swap[j][0] = x;
        if(x != -1){ /* if it is a real swap */
            for(i = 0; i < n; i++){
                if(ts[i][0] == x){ /* if another swap follows */
                    swap[j][1] = ts[i][1]; /* then make that the target row */
                    x = ts[i][1];
                }
                else if(ts[i][1] == x){ /* swapped with a row between k0 and k1 */
                    swap[j][1] = ts[i][0];
                    break; /* no further swaps will follow up this one */
                }
            }
        }
    }
}

/* unique marks reoccurences of a rownumber */
void unique(int n, int *x){
    int i, j;
    for(i = 0; i < n-1; i++){
        for(j = i+1; j < n; j++){
            if(x[i] == x[j] && x[i] != -1)  x[j] = -1;
        }
    }
}
```