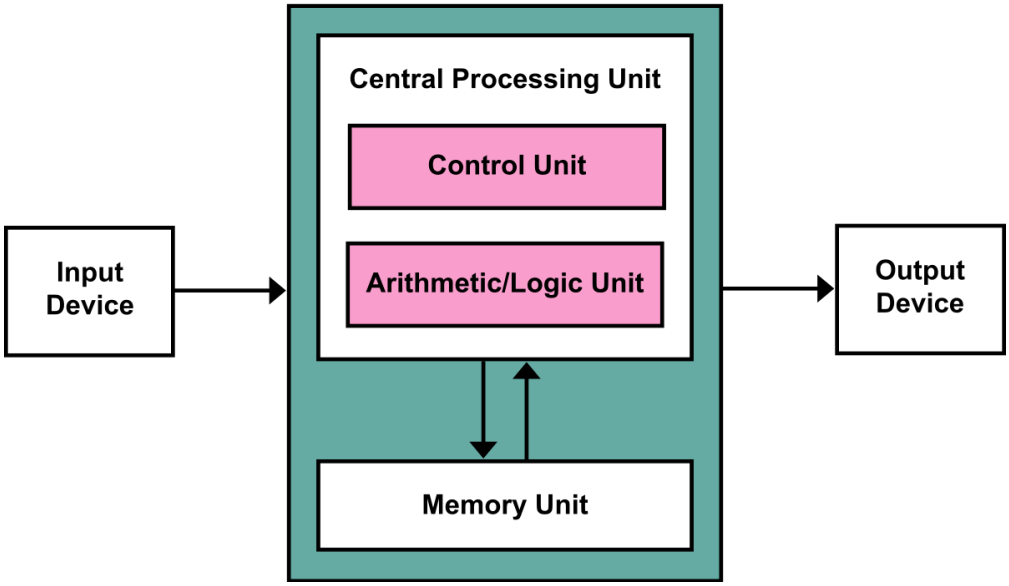
Basics

Concurrency: één processor/core meerdere programma's tegelijk door ze op te splitsen
parallellisme: Worden nog steeds in kleine stukjes verdeeld. Verschilllende stukken kunnen nu op het zelfde moment worden uitegevoerd

Nomanclatuur

- **Node:** een computer
  - Eén of meerdere processoren/cores
  - Geheugen dat gedeeld is over deze cores (**shared memory**)
- **Processor:** Fysieke chip met mogelijk meerdere rekenkernen (cores)
- **Core:** stukje hardare dat onafhankelijk van andere cores instructies en data kan opvragen en uitvoeren
- **GPU computing:**
  - Bedoeld voor snel renderen van 3D scenes
  - Kan gebruikt worden voor meer algemene berekeningen
- **Parallellisme:** Bepaalde berekeningen/instructies kunnen op hetzelfde moment uitgevoerd worden.
- **Distributed computing:** wanneer verschillende nodes samenwerken om berekeningen uit te voeren
  - Vorm van parallellisme
  - Gegevens uitwisselen noodzakelijk: verdelen/samenbrengen data, coördineren van werk
  - Netwerk nodig: bvb Ethernet met TCP/IP, infiniband
- **Grid en cluster**: Rekenkracht en opslag van vele nodes bundelen
- **Cluster:**
  - Nodes fysiek dicht in elkaars buurt
  - Typish zelfde soort nodes: homogeen
  - Hoge kwaliteit verbindingen
- **Grid**
  - Verschillende delen zijn fysiek verspreid
  - veelal heterogeen, vele verschillende types nodes die amenwerken

- Von Neumann architectuur



- **Memory wall:** CPU moet telkens wachten op geheugen access
  - Beter caching nodig (L1-3), beperk RAM access, prefetching
- **UMA**
  - Uniform memory access: alle processoren rechstreeks toegang tot zelfde fysieke geheugen
- **NUMA**
  - Non-uniform memory access: Processor heeft eigen deel van RAM geheugen
  - Kan nog steeds aan rest van geheugen, mits extra communicatiekost

Parallelle problemen

- **Deadlock:** elk proces/thread wacht totdat een ander een resource vrijfeeft
  - Threads zijn echt geblokkeerd
- **Livelock:** Threads zitten in een oneidige lus
- **Starvation:** thread wordt gedwongen te achten omdat andere resources niet vrijgeven
- **Data race** meerder threads updaten zelfde geheugen (zonder bvb mutex, atomic)
- **Race condition:** Gaat over volgorde van operaties waar bij multithreading iets mis kan lopen (zonder locking)
  - Term wordt ogal verward met data race
  - Komen vaak samen voor (maar moet niet)
- **Load imbalance:** werk over threads/nodes niet gelijkmatig verdeeld
  - Er zijn threads die relatief veel wachten, die geen nuttig werk verrichten

Schaalbaarheid

- Vaak is men geïnteresseerd in relaties tussen o.a. uitvoortijd en aantal processoren/nodes/probleemgrootte etc
  = schalingsgedra, schaalbaarheid

- **Strong scaling:**
  - Probleemgrootte blijft zelfde
  - Compute power wordt verhoogd
  - Doel is uitvoertijd te verkorten
  - ijdele hoop: Nkeer meer cores, N keer minder uitvoertijd
- **Weak scaling:**
  - Probleemgrootte wordt verhoogd
  - Compute power wordt verhoogd met zelfde factor
  - doel is in zelfde tijdeen groter probleem te behandelen
  - ijdele hoop: N keer meer cores en N keer groter probleem met zelfde uitvoertijd

Speedup

$$Tserieel/Tparallel$$

- Best verwachte speedup?
  - superlineaire speedup mogelijk

Amddahl's law

Wordt gebruikt om de theoretische speedup te voorspellen
Stel:

- S = tijd voor echt seriele code, niet paralleliseerbaar
- P = Tijd van deel dat geparallelliseerd kan worden (gemeten in serieel programma)
- N = # processoren/cores/threads

$$Tserieel = S + P$$

$$T(N) = S + P/N$$

$$speedup = \frac{Tserieel}{T(N)} = \frac{S+P}{S+\frac{P}{N}}$$

stel $s = \frac{s}{S+P}$ en $p = \frac{P}{S+P}$
Dan geldt

$$s + p = 1$$

En wordt speedup:

$$Speedup = \frac{1}{s+\frac{p}{N}} = \frac{1}{s+\frac{1-s}{N}}$$

Limieten van Speedup

- Als s → 0, speedup → N
  - Is wat we verwachten als code perfect paralleliseerbaar is
- Als N → ∞, speedup → $\frac{1}{s}$ ← wordt typisch als Amdahl's law gezien

Gustafson's law

Adreseerd de tekortkomingen in Amdahl's law die gebasseert is op de assumptie van een vaste probleem grote. Gustafson constanteerde dat programmers de probleem grote zullen vergroten om de volledige computer kracht te benutten.

Strategieën en patronen

Serieel naar parallel

- Zoek delen van algoritme ('taken') die parallel uitgevoerd kunnen worden
- Map deze taken op parallele uitvoereenheden
- Programeer

Foster's methodology

1. Partitioning = taken identificeren
2. Communiation = communicatiepatronen tussen taken
3. Agglomeratie = groeperen van geïdentificeerde taken
4. Mapping = toewijzen aan cores/nodes/...

Work vs span

- **Work:** hoeveel berekeningen/taken/operaties zijn er in totaal nodig
  - Bvb: bij elk van tien getallen iets optellen, work == 10
  - Hoeveel tijd heeft algoritme nodig als serieel uitgevoerd
- **Span:** hoeveel 'stappen' zijn er nog nodig als je parallel kan werken
  - Ook wel critical path, step complexity
  - Elke stap kan dan verschillende delen van het work mogelijk tegelijk uitvoeren
  - Bvb: bij elk van tien getallen iets optellen, span == 1 voor parallel algoritme, span == 10 voor serieel algoritme
  - Hoeveel tijd heeft het algorime nodig als volledig parallel uitgevoerd

Parallel algortihm strategy patterns

Task parallelism

- Zoek dingen die parallel kunnnen gebeuren
  - Ga na welke data nodig is voor welke taak

    - Hou rekening met load balancing bij uitvoer/scheduling
    - Misschien pre-processing nodig om onafhankelijke taken te krijgen, gevolgd door post-processing voor volledige resultaat
- Task is geen strikt afgelijnd begrip:
    - Sommigen gebruiken het als er echt verschillende zaken uitgevoerd moeten worden
    - In handboek wordt het veel minder restrictief gebruikt
- Liefst minstens evenveel taken dan cores
    - Let op granulariteit: load balancing vs overhead
- Seperable dependencies
    - Afhankelijkheid tussen taken kan weggewerkt worden
    - Wordt vaak gewerkt met data replication & reduction

Data parallelism

- Essentie: zelfde operatie toegepast op elementen van grote dataset
    - Vector operaties, matrix operaties
    - GPU computing: shaders die zeggen wat er per pixel moet gebeuren
        - Zelfs binnen shader nog data-parallellisme

Divide and conquer

- Verdeel en heers: opslitsen van probleem in steeds kleinere deelproblemen definieert uit te voerren taken
- Deelproblemen kunnen nog op verschillende manieren over cores/processoren/nodes verdeeld worden
- Voorbeelden:
    - speelboom doorzoeken
    - Problemen met bomen

Pipeline

- Probleem in verschillende stadia verdelen
- State van element dat door de pipeline reist kan worden aangepast
- Aanpak kan worden gebruikt als er een duidelijke equentie van taken is die uitgevoerd moeten worden
- Tijdens opstart en einde niet volledig gevuld
    - Moeten genoeg elementen zijn
    - Uitvoertijden stadia moeten goed gebalanceerd zijn
    - Daarom mogelijk efficënter om volledige pipeline als één taak te beschouwen en deze paralel uit te voeren

Geometric decomposition

- Deeltaken hebben meestal niet alle data nodig: enkel bepaald gebied **+ randen**
  - Als er een iteratieve berekening nodig is moeten die **randen** bij elke stap weer uitgewiseld worden

Implementation stragy patterns

SIMD

- Single Instruction Multiple Data
- Exact dezelfde instructie (low-level!) wordt uitgevoerd op verschillende data-elementen tegelijk
  - Typisch heel basic data types: float, int, bytes
- Uitvoring op verschillende data-elementen gebeurt in 'lockstep'
- Wordt veelal door hardware voorzien

SPMD

- Single Program Multiple Data
  - Zelfde prorammacode
  - Voor parallelle uitvoering:
    - Meerdere threads
    - Meerdere instanties van progamma worden gestart, mogelijk op meerdere nodes
  - Uitgevoerde stappen van code kunnen verschillen voor threads/processen
    - Gebruikte data verschilt ook typisch
  - Op basis van ID bepalen wat er gedaan moet worden door thread/core/node
  - Zeer algemeen, typisch programma met parallellisme

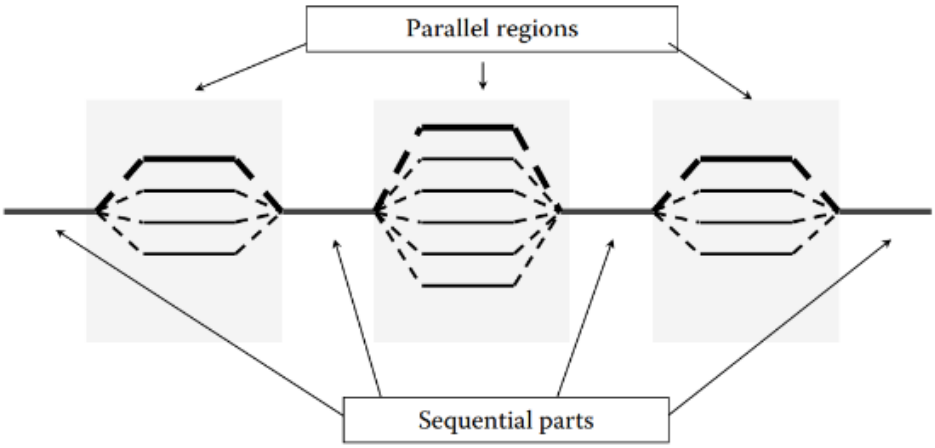
Loop-level parallelism

- Loop in code verdelen, bvb over threads
- Loop moet intensief genoeg zijn, paralleliseren moet voordeel geven vor programma
  - Veel iteraties, of veel werk per iteratie
- Rekening houden met dependencies
- Zeer makkelijk te gebruiken in OpenMP (als dependencies opgelost zijn)

Fork-join

- "Fork" aantal threads die nodige berekeningen doen

- Na berekeningen doen ze "join" met hoofdthread



- Complexer want threads kunnen ook zelf weer andere threads forken!
  - recursieve algoritmes, divide and conquer
- Focus bij fork-join is op shared memory systemen: threads starten/joinen relatief goedkoop

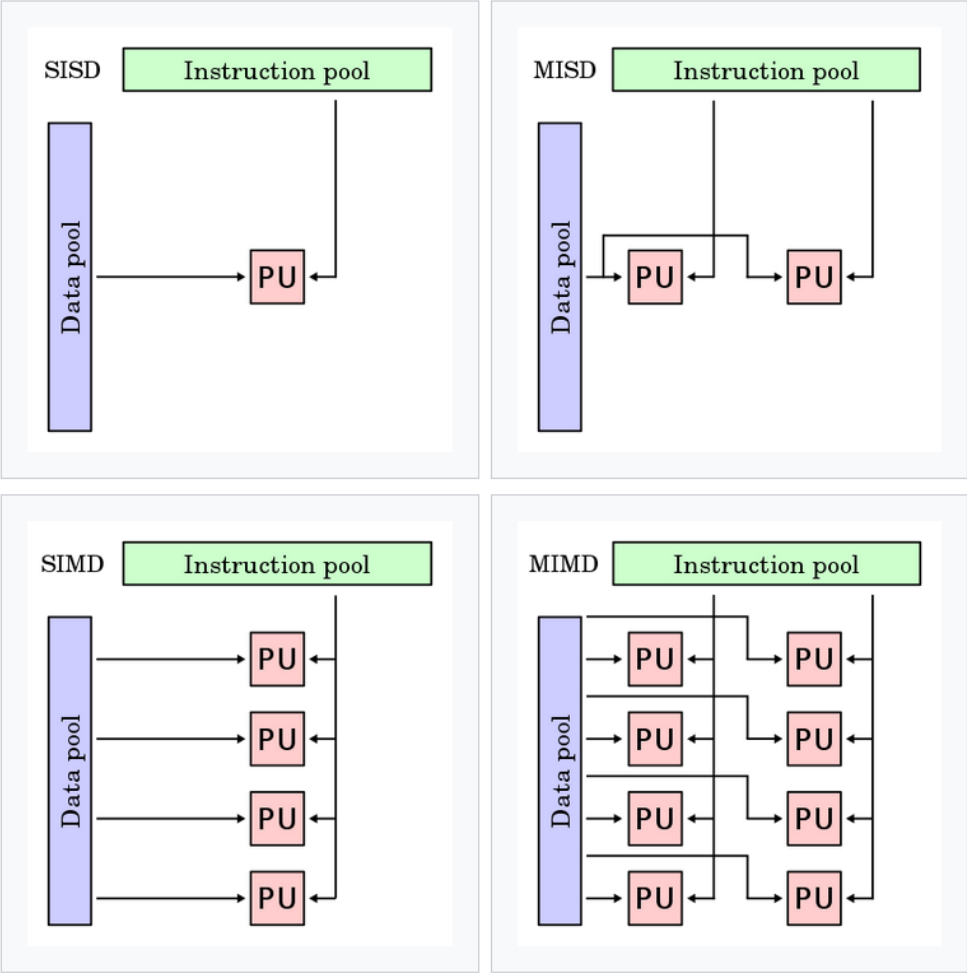
Master-worker/task queue

- Voorbeeld: verdelen van render-werk
  - 64 workers vragen telkens aan master voor welke lijn uit grid berekeningen gedaan moeten worden
- Master(s) reiken werk/taken aan:
  - Worker vraagt neiuw werk als vorige berekeningen gedaan zijn
  - Op deze manier: automatisch load-balancing
    - Mate van load balacing en overhead uiteraard afhankelijk van granulariteit

Flynn's taxonoy: meer xIxD afkortingen

- Gaat over "instruction streams" en "data streams"
  - Single instruction single data (vroeger)
  - Single instruction mutliple data
  - multiple instruction multiple data (NU!)

- SPMD maakt van dit model gebruik



Map/reduction/scan

- **Map:** pas functie/operatie toe op verschillende elementen
  - data parallellisme
  - Geen dependencies
- **Reduction:**
  - Combineer verschillende elementen via (associatieve) operator
    - bvb som van getallen in array
  - Kan bvb via divide & conquer aanpak voorzien worden
- **Scan:** Verang getal in array door som tot dan toe
  - Kan in principe ook andere operatie zijn
  - Inclusive/exclusive
  - Hoe paralleiseren?
    - Bestaan algoritme:
      - Hilis-steele algoritme,
      - Blelloch Algoritme
    - Meer 'work' voor kortere 'span'

OpenMP

Open standaar voor shared memory multi-processing

OpenMP onderdelen

- Compiler directives: praga's
- Extra library functies:
  - `#include <omp.h>` nodig om functienamen te kennen
  - Met juiste OpenMP compiler flag wordt automatisch tegen de juiste library gelinkt
- Environment variabelen
  - Defalt waarden beïnvloeden

Parallel

- Initieel heeft programma één thread
- "parallel" pragma start aantal OS-level threads
  - Thread die tema van threads startte is master van dat team
  - Thread kan zelf weer andere threads starten
- Na parallel werk joinen met master
- In pragma `num_threads(...)`
- op voorhand `omp_set_num_threads(...)` om het aantal threads vast te leggen
- Environment variabele
  - `OMP_NUM_THREADS`
- Je krijgt niet noodzakelijk het aantal gevraagde threads
  - Als nodig: controleer met `omp_get_num_threads()`
- Controleer dat nesting aanstaat met `omp_get_nested()`
- **Wat doet compiler met "parallel"**
  - Maakt van parallel gebied een functie
  - Start aantal threads die deze functie uitvoeren

Private & shared

- Default gedrag:
  - Compiler zal binnen de threads naar deze variabelen verwijzen met pointers
    - Opletten met data races!
  - Binen parallel gebied: private
  - `private(...)` Duid een variable aan als private
  - `firstprivate(...)` initialiseerd een private variabel
  - **Declareer variable pas binnen parallel gebied indien mogelijk**

Synchronisatie

- OpenMP: veel data is shared tussen threads
  - Opassen met data races en race conditions
  - 'atomc' en 'critical' voorkomen dat meerdere threads tegelijk iets uitvoeren/aanpassen
  - Bestaan ook low-level lock (mutex) objecten
  - 'atomic' gaat over geheug, wordt dan atomair uitgevoerd
    - eenvoudige reads/writes
  - 'critical' algemeer, maar één thread tegelijk voort dit uit

Worksharing constructs

- Binnen parallel gebied: werk verdelen over beschikbare threads met worksharing constructs:
- Geen extra synchronisatie aan start van zo'n gebied
- Wel extra synchronisatie aan einde ervan
  - voorkomen met 'nowait'

single

- Maar één thread van het huidige team voert deze code uit
- Er wordt gewacht tot 'single' code gedaan is
  - Als niet nodig: 'nowait' opgeven

Sections

- Eerst een 'sections' gebied aanduiden, en daarbinnen een aantal 'section' region's
- De 'section' gebieden worden verdeeld over de beschikbare threads in het team
- task-parallelism
- Minder secties dan threads:
  - threads idel
- Meer secties dan threads
  - Sommige threads doen meer

For

- Verdeel de iteraties van een loop over het team van threads
  - Grenzen moeten gekend zijn
  - Iteratieveriabele altijd private
  - Kan ook verkort worden
    - `#pragma omp parallel for`
  - Met `collapse` geneste loops samen voegen.
    - Mogelijk werk beter verdelen

        - Moeten eenvoudige, perfect geneste loops zijn

Schedulers

- Static: Vaste opsplitsing van loop in chunks, vaste verdeling
    - Default chunk grootte verdeelt aantal iteraties gelijkmatig over threads
    - Erg goed als iteraties even lang duren
- Dynapic: vaste opsplitsing, maar geen vaste verdeling
    - Vrije thread neemt telkens volgende chunk; pas op: default chunksize is 1
    - Goed als iteraties verschillen in uitvoertijd
- Guided: zoals dyamic, maar chunk groootte wordt aangepast
    - aanpassing is van hoog naar laag, chunksize geeft minium
- Auto: compiler beslist, hoeft geen van voorgaande schedulers te zijn
- runtime: gebruik verdeling op basis van env var OMP_SCHEDULE

Master

- Zorgen dat code alleen door master van team uitgevoerd kan worden
    - Andere threads slaan de code gewoon over
- Geen automatische sync aan het einde
    - in tegenstelling tot single
    - +/- zelfde gedrag als single met 'nowait'

Reduction

- Duidelijker en efficienter
- Hoe werkt het?
    - Elke thread maakt private variabele
    - Initalisatie van variable volgens operator
    - Aan het eind van reductie worden private versies gecombineerd volgens operator
    - Merk op: wat je tijdens berekening met de variabele doet bepaal je zelf
        - oeft niet in overeenstemming te zijn met wat je zou verwachten
- Voor for, sections en parallel

Task

- Bij recursie/devide en conquer: gebruik van geneste sections zou tot enorm veel threads leiden
    - Mogelijk te veel voor OS
- Als we wel conceptueel dit paradigma willen gebruiken, maar geen 'echte threads starten: 'task'

    - Moet binnen parallel gebied gespecifieerd worden
    - Duidt blok coda aan die door één van de threads in het team moet uitgevoerd worden
- Thread die task tegenkomt mag die ook zelf dadelijk uitvoeren
- Tasks kunnen vanaf meerdere threads gestart worden, hoeft niet vanuit master te zijn
- Kan helpen met load balancing

Simd

Zegt tegen de compiler dat volgende loop SIMD operaties bevat. De loop zal gesplits woren in stukken die compatieble zijn met beschikbaar SIMD registers/instructies
Dit hoeft niet noodzakelijk in 'paallel' gebied te staan.

GPU computing

GPU

- Gespecialiseerde hardware voor grafische operaties
    - Doel: snel kunnen weergeven van realistische 3D scenes
    - Vaak embarrassingly parallel
- Beschikbare APIs: OpenGL/Vulkan, Direct3D, WebGL/WebGPU
- Berekeningen hoeven niet voor 3D scene te zijn
    - Zelfde principes kunnen gebruikt worden voor algemeen rekenwerk
      ⇒ General purpose GPU
- Bestaan ook speciale APIs voor GPU computing
    - CUDA, OpenCL, DirectCompute

Renderen van 3D scenes

object space → world space → eye space → clip space → device space → screen space
Vak noemt nie computer graphics

Traditionel GPGPU

- Door programmeerbaarheid van shaders worden algemene berekenigen mogelijk
- Typische aanpak
    - Vul framebuffer met twee driehoeken
        - Hoeft niet op scherm getoond te worden, kan ook in geheugen
    - Vertex shader: voort meestal geen speciale berekeningen uit
    - Berekeningen gebeuren per pixel/fragment in fragment shader

- Input voor berekeningen wordt aangeleverd als texture, output wordt opgeslagen in andere texture
  - Als weerdere iteraties nodig: wissel input/output om

Hardware

CPU vs GPU

CPU: multi-core, cores kunnen heel verschillend ingezet worden, heavyweight threads
GPU: many-core, cores voeren heel gelijkaardige instructies uit, lightweight threads

NVIDIA vs AMD

NVIDIA:

- Aantal streaming multiprocessors (SM)
- Elk aantal CUDA cores (scalar processors, SPs)
  AMD:
- Aantal compute units (CUs)
- Elk aantal stream processors (SPs)

Memory wall

- Ook bij GPUs is communicatie met (eigen) DRAM traag tov rekensnelheid
- We willen veel rekenwerk per geheugenaccess
- Shared memory kan DRAM communicatie beperken
- Latency hiding: wanneer groep threads/warp moet wachten op data, verder gaan met ander warp
- Een byte opvragen kost evenveel tijd als 128 opeenvolgende
- Per SM is er snel shared memory waarvan threads gebruik kunnen maken
- Typische aanpak:
  - *Coalescing*: opeenvolgende threads lezen opeeenvolgende adressen
  - Sla op in shared memory zodat andere threads er ook gebruik van kunnen maken
  - Synchronisati nodig: aangeven wannneer shared memory volleidg ingelzen is: `_syncthreads`

Shared memory

- Threads in één block
  - Kunnen gebruik maken van zeer snel lokaal geheugen
  - Kunnen gesynchroniseerd worden via barrier
- Op deze manier

    - Threads halen samen data naar shared memeory op geordende manier (memory coalescing, opletten met bank conflicts)
    - Synchroniseren zodat shared memory zeker klaar is voor gebruik
    - Doen berekeningen op basis van data in shared memory

Branch divergence

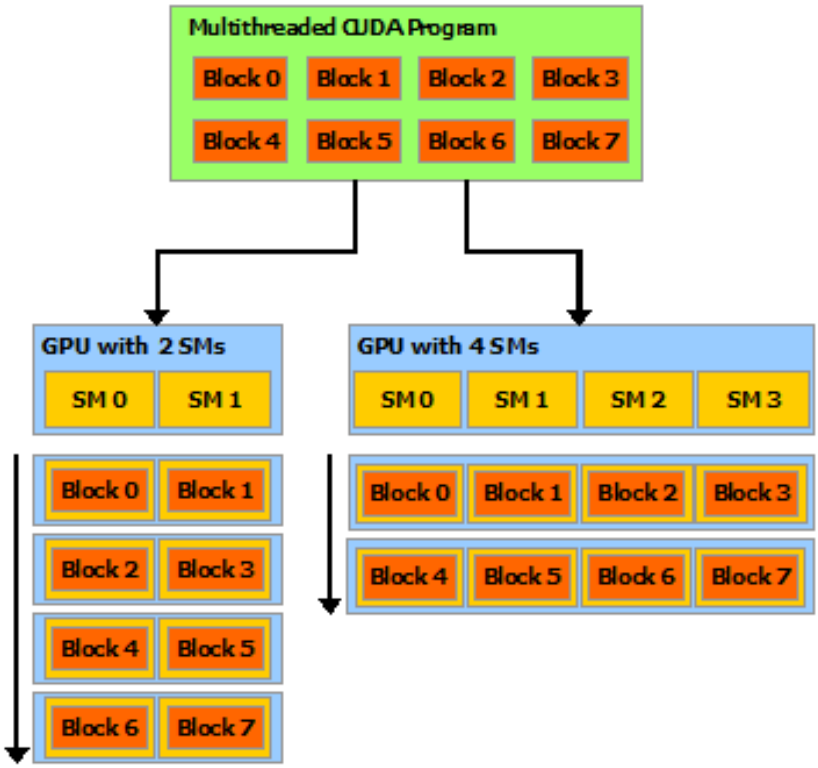
- Kernel is flexibel, kan op basis van if-tests andere code uitvoeren
- GPU threads zijn lightweight, liefst zoveel mogelijk zelfde berekeningen
- if-test kan leiden tot geserialiseerde code
    - De 'if' en 'else' code wordt niet parallel luitgevoerd, maar na elkaar
    - Daarna gaan threads pas synchroon verder
- bij cuda gebeurt dit op warp niveau

Cuda

Van 'old school' GPGPU naar CUDA

- Klassiek GPGPU
    - Pak probleem aan door het onder te verdelen in kleine stukjes, (typisch in 2D layout) die samen voor oplossing zorgen
    - Maak fragment shader die de code voor zo'n stukje bevat
- CUDA
    - Zelfde principe, maar meer controle over onderverdeling van grid
        - Grid bestaat uit blocks, block bestaat uit threads
        - Grid en blocks kunnen 1D/2D/3D zijn
    - Maak kernel die de code bevat en door een CUDA thread zal worden uitgevoerd
    - Threads worden in groepjes ('warps') van 32 uitgevoerd
- In biede gevallen: upload/download data naar/van GPU enkel wanneer nodig

Blocks



- Threads worden georganisseerd in blocks: helpt met schaalbaarheid
- Je kent de volgorde van blocks niet
  - Sommige parallel, sommige sequentieel
  - Mogelijk verschillende SMs en GPUs
- Je kent de volgorde van threads niet
  - Worden wel steeds in warp uitgevoerd

Unified memory / Managed memory

- Ipv cudaMalloc: cudaMallocManaged
- Alloceert zowel op CPU als GPU stuk geheugen
- Zorgt dat zelfde pointer geldig is op CPU en GPU
- Datatransfer gebeurt impliciet
- Mogelijke sync nodig met: `cudaDeviceSynchronize()`
- Soms wat trager dan expliiet geheugenbeheer

Architectuur

- NVIDIA GPU: aantal SMs
  - Echte parallele processoren
- Elk block zal op één SM uitgevoerd worden
- Aantal 'warp schedulers' per SM
  - 2 voor P100, 4 voor 'Volta'
- Zijn erg goed in latency hiding
  - Van zodra warp even moet wachten wordt er verder gegaan met andere

- Moeten genoeg warps zijn!

Memory coalescing

- Alignment van data met thread speelt rol
- Volgorde van access kan rol spelen afhankelijk van compute capability

Bank conflicts

- Shared memory: georganiseerd in aantal verschillende geheugenbanken
  - Opeenvolgende 32-bit waarden in andere bank
- As verschillende threads zelfde bank gebruiken:
  - "Bank conflict"
  - Performanteiverlies

Occupancy

- SM ondersteunt bepaald maximaal aantal actieve warps
- Occupancy = verhouding behaalde aantal tot dit maximum
- Vele zaken kunnen zorgen voor lagere occupancy:
  - Shared memory: als block relatief veel nodig heeft, kunnen minder blocks tegelijk actief zijn op SM
  - Aantal registers nodig per thread (SM heeft maar beperkt aantal registers)
- Optimaal gebruik maken van GPU is niet tiviaal

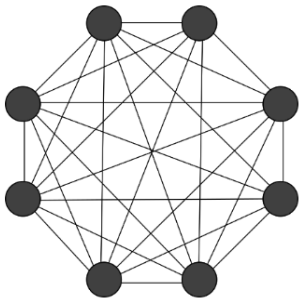
Distributed memory prallellisme met MPI

- Geen shared memory → expliciet zenden/ontvangen van berichten = 'message passing'
- Voordelen van messag passing
  - Niet beperkt tot één node ⇒ distributed computing
  - Algemeen bruikbaar paradigma
  - Minder problemen met bugs als data races
- Nadelen:
  - Latenc door expliciet uitwisselen/kopiëren
  - Mogelijk meer geheugen nodig (als anders geheugen gedeeld zou kunnen worden)
- Meer dan één node: cluster & grid computing
  - MPI: in HPC de aanapk om werk te coördineren tussen nodes
- **MPI**: de aanpak om werk te coördineren tussen nodes
  - Meer rekenkracht
  - Meer geheugen

- Netwerk nodig voor communicatie

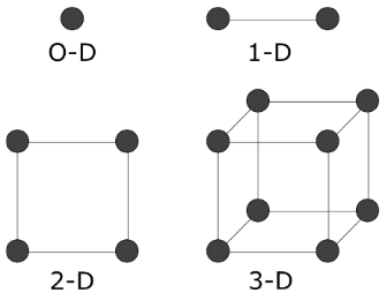
Topologieën

Fully connected mesh



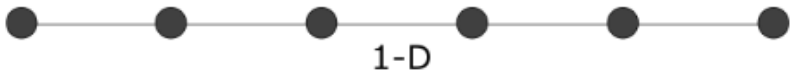
- N nodes elke node rechstreekse verbinding alle andere nodes
- N(N-1)/2 links, minste omweg om node te bereiken

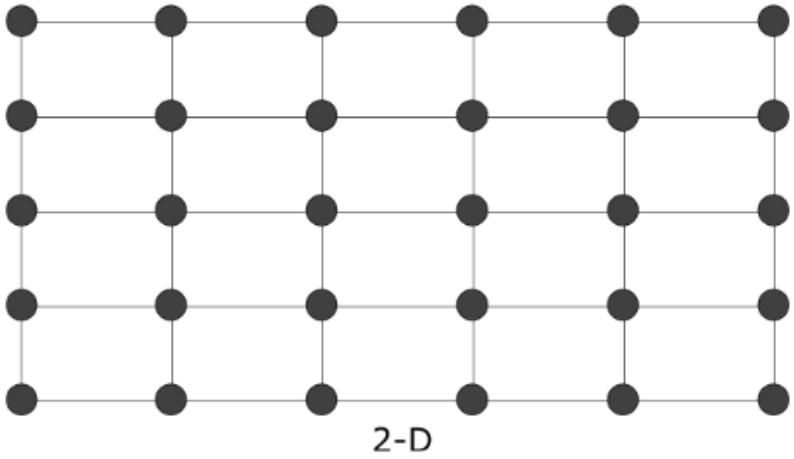
N-D hypercube



- Procedure: verdubbel situatie, maak verbindingen tussen verdubbelde nodes
- 2^N nodes
- $Links(N) = 2 * Links(N-1) + 2^{(N-1)}$
  $= (2^N * N)/2$
- Maximale afstand: N verbindingen

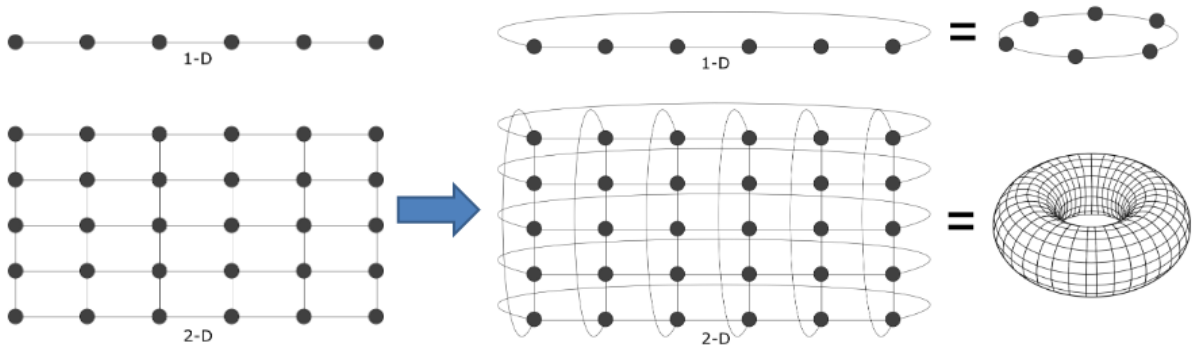
N-D grid





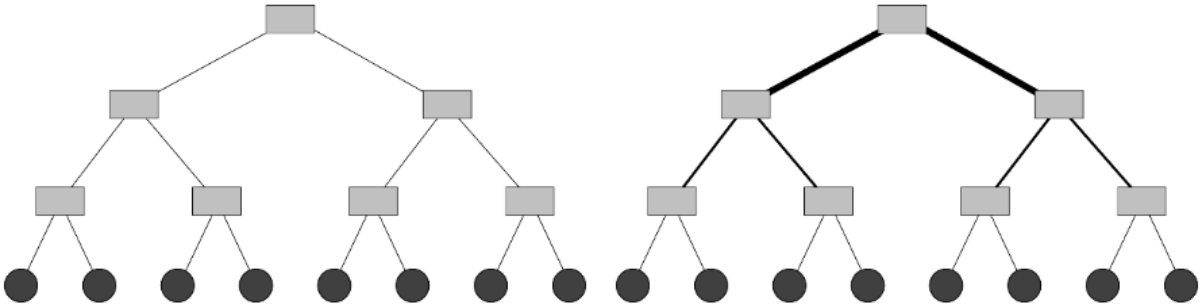
- Elk punt heft N 'uitgaande' verbindingen, behalve rand
- zoals hypercube, maar meer tussenliggende punten
- Niet noodzakelijk evenveel herhalingen in elke rechting

N-D torus



- Start van N-D mesh, slut randen
- Alle nodes in grid layout hebben N uitgaande verbindingen

Tree, fat tree



Meer bandbreete voorzien dicht bij de root.

Ethernet vs InfiniBand

- **Ethernet**:
  - 1Gb ethernet ingeburgerd
  - Ook 10Bb, 40Gb
  - Gebruik van TCP/IP zorgt voor overhead door tussenkomst OS en extra kopieën van data
- **InfiniBand**:
  - Low latency
    - Programma's communiceren rechstreeks met IB adapter
    - Ondersteunt ook Remote DMA
  - High bandwidth

MPI

- MPI is een specifictie van een API
- Veel implementaties
  - Intel MPI, OpenMPI, MPICH, MVAPICH, MSMPI
- Is een C API, bruikbaar vanuit C++
- Is geen compiler o fcompiler-extensie
  - Kent bvb de grootte/offsets van willekeurige struct niet
- Werking:
  - Start veel processen (evt op verschillende nodes) die met elkaar kunnen communiceren
    - Zo data uit wisselen, algoritme coördineren
    - Er zijn geen automatisch gedeelde variabelen, geheugen van elk proces is private

Basisstructuur MPI programma

- `MPI_Init`: initialiseer MPI library
  - Pas daarna zijn andere MPI calls toegelaten

- `MPI_Finalize`: cleanup van MPI library
  - Daarna geen MPI calls meer toegelaten
- Levert één enkele executable
  - Spawnt zelf geen andere processen
  - meerdere processen starten via `mpirun`
- Call naar MPI_Init:
  - Zorgt dat processen met elkaar verbonden worden
  - Zorgt dat command line argumenten in elk van de processen beschikbaar zijn
  - Parameters mogen NULL zijn als dit niet nodig is
- **Communicator**: identificatie voor groep processen
  - Heeft bepaalde grootte: MPI_Comm_size
  - Proces heeft bepaalde ID binnen communicator: MPI_Comm_rank
- Alle processen maken automatisch deel uit van communicator MPI_COMM_WORLD

Point-to-point communicatie: send & receive

- Deze MPI_Send & MPI_Recv zijn blocking calls
  - MPI_Send: gaat verdre wanneer de buffer aangepast mag worden
  - MPI_Recv: gaat verder wanneer velledig bericht ontvangen werd
- Sends/receives moeten gebalanceerd zijn: berichten die verstuurd worden moeten opgevangen worden
- Wordt gefilterd op: communicator, tag, source rank
  - Wildcards in receive: MPI_ANY_SOURCE, MPI_ANY_TAG
  - 'status' bevat dan source & tag
    - Mag MPI_STATUS_IGNORE zijn als niet belangrijk
- **Non-blocking** send and receive
  - Uitvoor programma gaat onmiddelijk verder
    - MPI_Isend: buffer mag nog niet dadelijk aangepast worden
    - MPI_Irecv: buffer bevat nog niet alle ontvangen data
  - MPI_Request argument: informatie over de non-blocking call
    - Status nagaan met MPI_Test, MPI_Testall
    - Wachten tot operaties klaar zijn: MPI_Wait, MPI_Waitall
  - Operatie moet afgesloten worden via een van de MPI_Wait functies om resources weer vrij te geven
  - Buffer moet blijven bestaan tot einde data transfer
  - Kan handig zijn om deadlocks te vermjijden
  - Non-blocking en bocking calls mogen gmeixt worden
  - Maakt ook overlap communication/computatio mogelijk

- Aanroep van MPI_Wait/MPI_Test functies noodzakelijk om voortgang te maken met non-blocking communicatie

Datatypes

- Voorgedefinieerd: MPI_INT, MPI_CHAR, MPI_DOUBLE, ...

User-defined

- Stel dat N^2 waarden van NxN matrix voorstellen
  - Rijen: opeenvolgende waarden, kunnen makkelijk met send/recv gecommuniceerd worden
  - Kolommen?
    - kolom 4×4 matrix met MPI_Type_vector
      - Meer algemeen submatrix
      - Vergeet MPI_Type_commit niet
- Structs moeten we ook in MPI definieren `MPI_Type_create_struct()`

Stel: willen (array van) deze struct uitwisselen

```
struct MyStruct
{
    char a;
    int b;
    double xy[2];
};
```

MPI tegenhanger maken:

```
int blocklen[] = { 1, 1, 2 };
MPI_Aint displacements[] = { offsetof(MyStruct, a), offsetof(MyStruct, b),
                             offsetof(MyStruct, xy) };
MPI_Datatype dataTypes[] = { MPI_CHAR, MPI_INT, MPI_DOUBLE };

MPI_Datatype mpiMyStruct;
MPI_Type_create_struct(3, blocklen, displacements, dataTypes, &mpiMyStruct);
MPI_Type_commit(&mpiMyStruct);
```

(vergeet commit niet!)

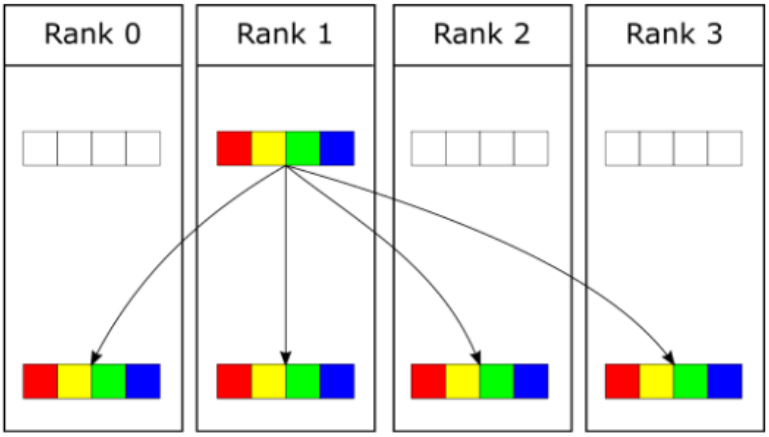
```
...

MPI_Type_free(&mpiMyStruct);
```

Collectives

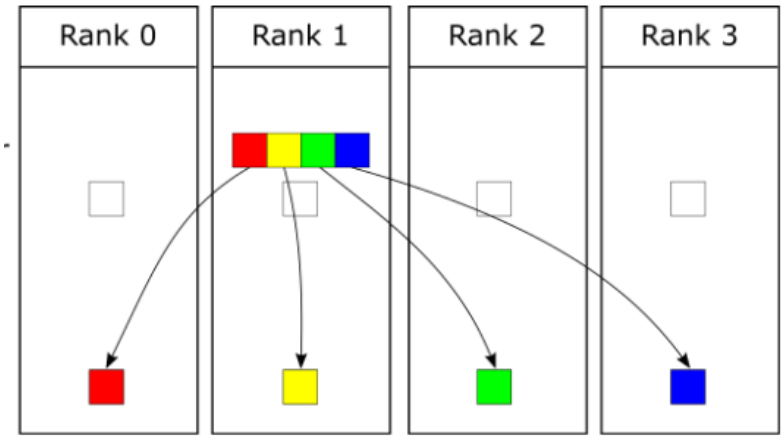
MPI_Bcast

Broadcast data van root naar andere processen



MPI_Scatter

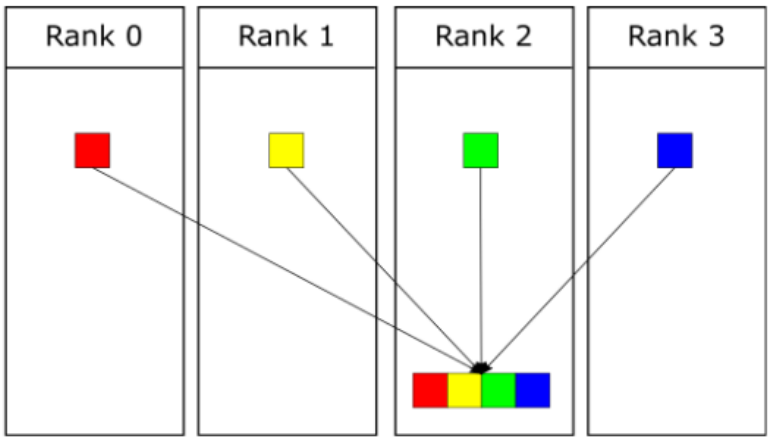
Verdeel data over processen



Voor meer controle: MPI_Scatterv

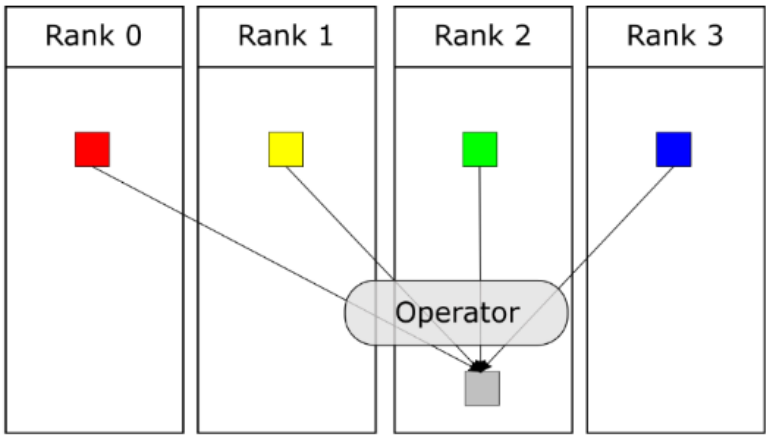
MPI_Gather

Verzamel data van processen in communicator



MPI_Reduce

Voor samen operatie uit op verdeelde data



- Voor gedefinieerde operatoren: MPI_SUM, MPI_PROD, ...
- Eigen reductieoperator definiëren is mogelijk
  - Maak functie als:
  - `void MyUserDefineReduction(void *invec, void* inoutvec, int *len, MPI_Datatype* datatype)
  - Creëer oeprator via MPI_Op_create
  - Gebruik die perator in MPI_Reduce
- Operator moet associatief zijn

Blocking & non-blocking

- Vorige collective calls waren allemaal blocking
- Bestaan ook non-blocking versies
  - Bvb MPI_Ireduce, MPI_Ibcast, ...
- Blocking & non-blocking collectives mogen niet gemengd worden
  GEBRUIK ALTIJD COLLECTIVE IPV SEND/RECEIVE ALS DAT MOGELIJK IS

One-sided communication, RMA

- Send/Receive: moet gebalanceerd zijn, samenwerking processen nodig
- Bij one-sided communication/Remoe Memory Acces
  - Processen duiden geheugen aan dat remote toegankelijk is
  - Andere processen kunen dan schrijven of lezen uit dit geheugen
  - Kan hardware ondersteuning van RDMA gebruiken

combinatie MPI/Threads

- Voor combinatie met std::thread, OpenMP, ...
  - Idee: binnen node threads, commuicatie tussen nodes met MPI

Combinatie MPI/GPU

- Perfect mogelijk
    - Berekeningen op GPU
    - Data-uitwisseling via MPI

Gedistribueerde Systemen

- Collectie van computers/systemen/onderdelen ("nodes")
- Werk samenaan een taak, coöperatie
- Coördineren samenwering via uitwisseling berichten ("messages")
    - Een soort netwerk nodig
        - Ethernet/infiniBand
        - CAN-bus
        - Trucks
- Kunnen/zullen "partial failures" optreden
    - Zowel op vlak van nodes als van netwerk
        - Power failure, segfault
- HPC vs DS
    - HPC is zeker een DS (gezien bij MPI)
    - Alles stopt als er iets misgaat
- Een distributed syste is erg complex
    - Kan veel misgaan
- Kan noodzakelijkzijn
    - Fysieke redenen (smartphones die communiceren, client/server, ...)
    - Performantie (rekenkracht combineren, data dichter bij gebruiker)
    - Betrouwbaarheid, "fault tolerance"
    - Te veel data voor één systeem

Systeemmodel

- Twee onderdelen: nodes & messages
    - Andere benamingen: prcess, host
    - Vaak voorgesteld in "lamport diagram"
    - Richting van tijd verschilt nogal
- Duidelijk aangeven wat vereiste zijn, wat mis kan gaan:
    - Netwerk: hoe betrouwbaar
        - Kan netwerkpartitie optreden?
    - Nodes: crashes, betrouwbaarheid
    - Timing

Node failures

- Fail-stop, crash-stop
  - Het enige dat mis kan gaan is dat een node (of relevant proces) er helemaal mee stopt
  - Stroomonderbreking, kernel panic, segmentation fault
- Fail-recovery, crash-recovery
  - analog, maar node herstart
  - Kan wel inhoud van geheugen verloren zijn
- Byzantine errors, Byzantine faults
  - Arbitreire fouten
  - Afwijking van algoritme, mogelijk met opzet door aanvaler

Timing

- Synchroon systeem
  - Grenzen op latency, clock drift
  - Algoritme wordt aan gekende snelheid uitgevoerd
  - Moeilijk, maar bestaat (stabiele circuits nodig)
- Asynchroon systeem
  - Er zijn helemaal geen timing garanties
  - Nodes kunnen vertragen, berichten kunnen lang onderweg zijn
- Partieel synchroon systeem
  - Echt asynchroon systeem is misschien te pessimistisch
  - Vaak is het systeem eenhele tijd synchroon, met af en toe meer asynchroon gedrag

Algoritmes

- Bij een gedistribueerd algoritme:
  - Belangrijk wat veronderstellingen zijn
  - Met wat voor fouten kan algortitme om?
  - Wat voor timing is er nodig?
- **Saftey property:**
  - "Something bad will never happen"
  - Bvb bij "leader election" zal er maar één gekozen worden
- **Liveness property:**
  - "Something good will eventually happen"
  - bvb bij "leader election" kunnen er tijdelijk conflicten zijn (meerdere nodes willen leader zijn), maar uiteindelijk wrdt er één gekozen
- **Remote Prcedure Calls (RPC)**
  - Vaak gebruikt paradigma
  - Lokale implementatie van functie is zgn "stub"

    - Parameters worden geencodeerd, verzonden, gedecodeerd
        - Marshalling/unmarshalling
    - Interface definition Lanuage (IDL) wordt vaak gebruikt
        - Op basis hiervan bvb automatisch stub/marshalling code genereren
    - Tegenwoordig vaak web services
        - Via represenational state transfer
        - JSON voor marshalling

Tijd

Fysische tijd

is erg complex:

- Seconden is gedefinieerd op basis van quantumechanische eigenschappen van Cesiu ⇒ atoomklok
- Meer betaalbare klokken (quartz kristallen) zijn minder nauwkeurig
- Verloop van tijd is afhankelijk van beweging en zwaartekrachtveld
- Onze ervaring met tijd is baseerd op aardrotatie
- Bij DS fysiek verspreide systemen
    - Vaak geïnterseerd in 'wat gebeurde er eerst?'
    - Maar klokken 'synchronisern' heeft altijd eidige resolutie
    - Deze 'Time of day' tijd si niet altijd bruikbaar in DS
- "Time of day" wordt beïnvloed door vele zaken
    - Synchronisatie protocol (NTP), zomertijd, schikkelseconde, ...
    - Tijdsduurmeting zelf wordt gedaan via "monotone klok"
        - Helemaal niet te vergelijken tussen systemen
        - Nog steeds clock drift

Logische tijd

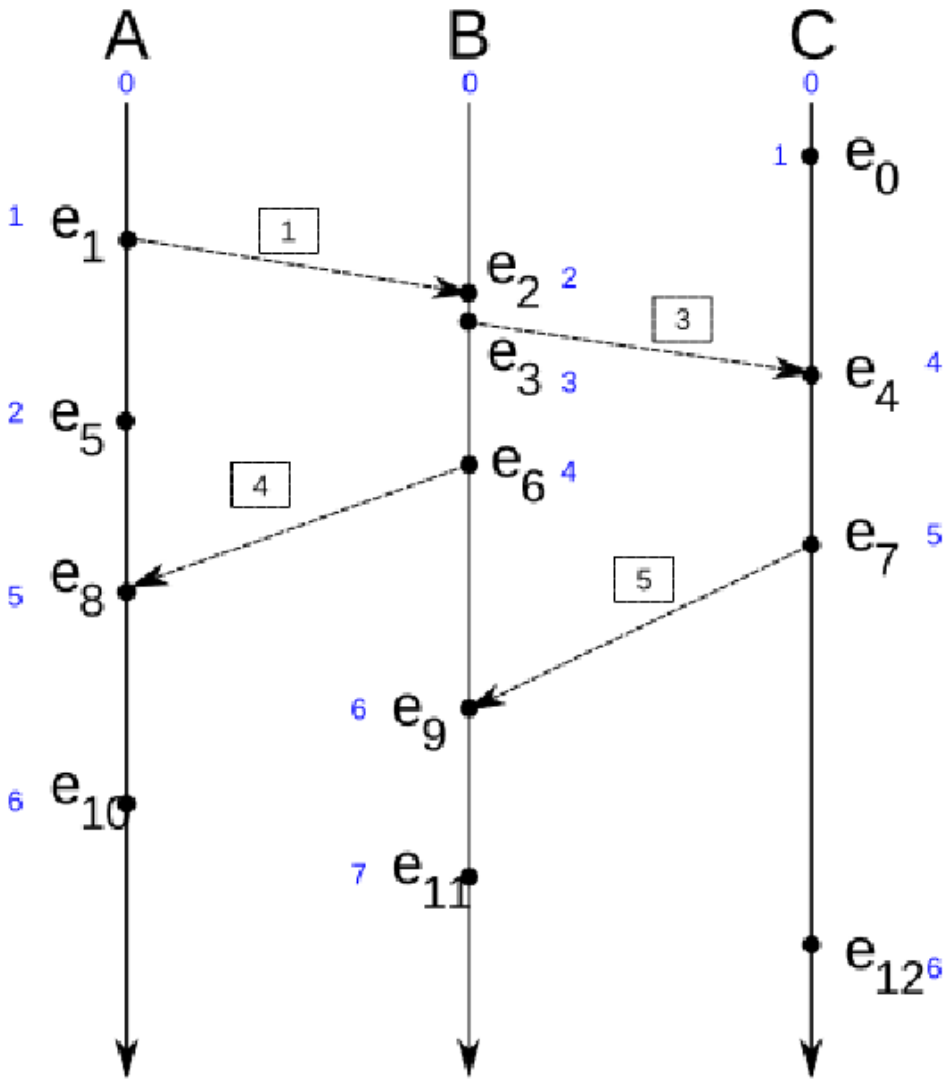
- Misschien 'echte' tijd niet nodig, maar wel welk event voor een ander gebeurt
- "Happens before" relatie X → Y
- Concurrent: X || Y
    - Betekend dat we niet kunnen zeggen wat de volgorde is

Orde

- "happens before" is een causale orde
    - Defineert maar een partiële orde op een set van events A en B is het niet altijd A → B of B→ A (A || B is ook mogelijk)

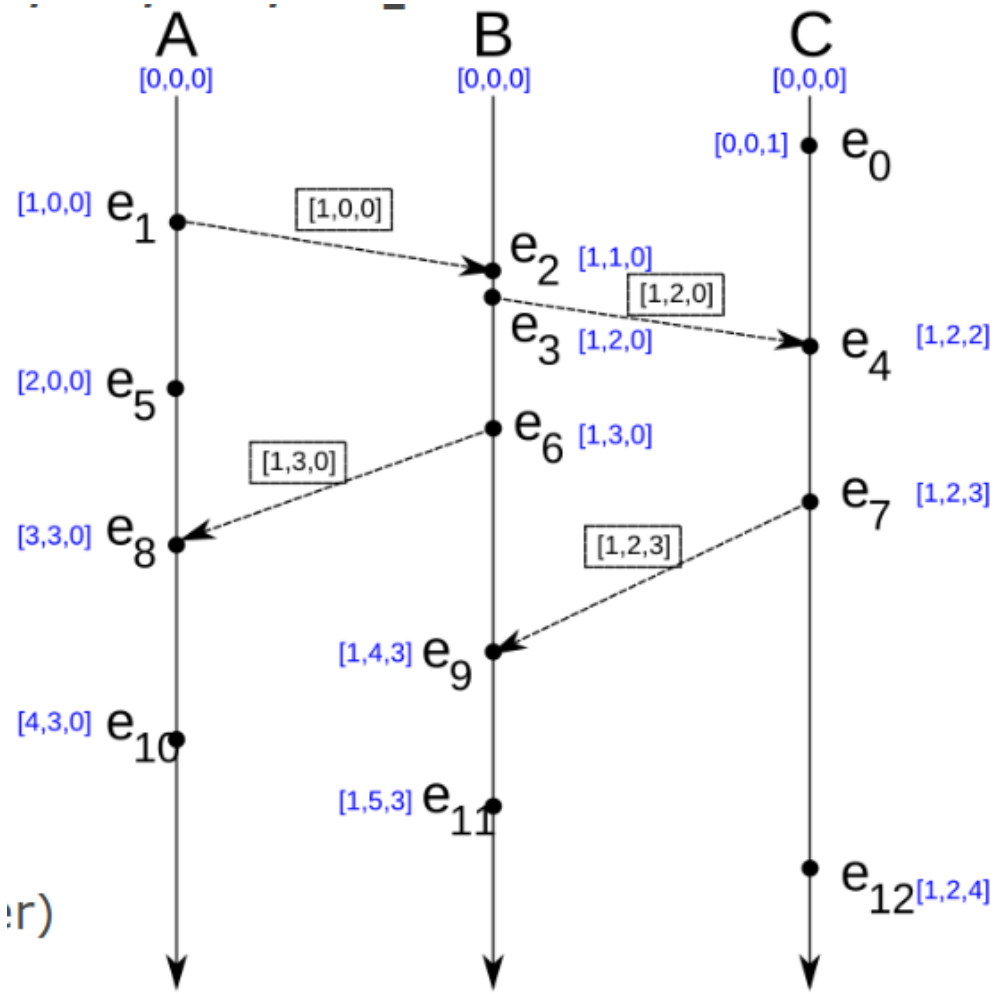
- Dit in tegenstelling tot een totale orde
  - Voor verschillende events A en B zou altijd ofwel A → B of B → A
- Beide types ordes voldoen ook aan transitiviteit, in dit geval als A → en B→ C, Dan A → C

Lamport clock



- Elke node heeft eigen LC, geïnitialiseerd op 0
- Bij gewoon event:
  - Verhoog LC met 1
- Bij versturen van bericht:
  - Verhoog LC met 1
  - Stuur LC mee met bericht
- Bij ontvangen berict
  - Zet LC op max(LC, LC in bericht)
  - Verhoog LC met 1
- Verschilende events kunnen zelfde LC hebben
- Door contructie:

    - Als X → Y, dan is LC(X) < LC(Y)
- Omgekeerd geldt niet
    - Als LC(X) < LC(Y), kan X → Y of X || Y
- Wel weet je:
  - Als LC(X) >= LC(Y), dan x ↛ Y
  - Wat met e3 en e10, e0 en e6?



- Door één counter te gebruiken verliezen we informatie
    - Oplossing vector van counters bijhouden, één per proces
- Iedere node initialiseert eigen VC op [0, 0, 0]
- Bij gewoon event:
    - Verhoog VC met 1 op pos van node
- Bij versturen bericht:
    - Verhoog VC met 1 op pos van node (zender)
    - Verstuur hele vector klok mee met bericht
- Bij ontvangen bericht:
    - Stel VC = max(eigen VC, ontvangen VC)
    - Verhoog VC met 1 op pos van node
- Verschillende event hebben verschillende VC
- We stellen VC(X) < VC(Y), als
    - voor ek component kleiner of gelijk aan

    - **En** voor minstens één component strict kleiner
- Dit is een partiële orde
- Weer door constructie:
    - Als X → Y, dan is VC(X) < VC(Y)
- Nu geldt omgekeerde ook:
    - Als VC(X) < VC(Y), dan X → Y
- De vector clocks van event weerspiegelen dus "happens before"!
- Wet met e_3 en e_{10}, e_0 en e_6
    - $e_3 \rightarrow e_{10}$
    - $e_0 \mid\mid e_6$

Replication

= bijhouden van verschillende data op verschillende nodes

- Kan om verschillende redenen, o.a.
    - Fault tolerance: zorgen dat data blijft bestaan, of toegankelijk blijf
    - Performantie:
        - Load van data access verspreiden over nodes
        - Data dichter bij gebruiker
- Twee grote klassen om zelfde state te verkrijgen
    - State machine replication
    - state transfer

State machine replication (SMR)

- Telkens bijhouden wat de actie was, hoe de 'state' veranderde
- Actie kan vanales zijn (wijziging key/value entry, muisbeweging, ...)
- Idee is dat 'afspelen' van die acties steeds dezelfde state veroorzaakt
- Verschillende nodes bouwen zo kope van state op
- Moet deterministisch zijn
- De acties/state veranderingen worden typisch in log/commit log bijgehouden

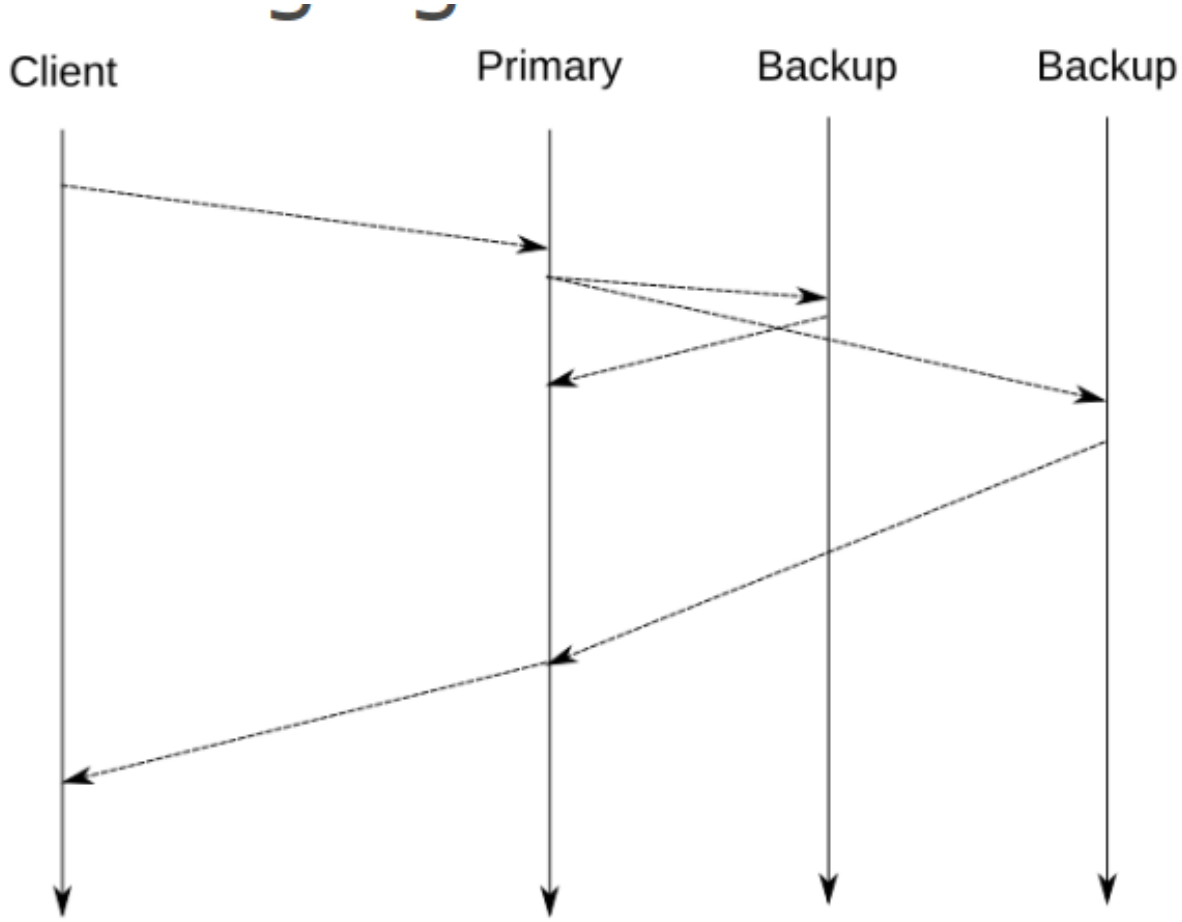
State transfer: volledige state wordt doorgestuurd

- Veeleisend voor netwerk, trager
- Vaak cominatie met SMR

Enkele systemen

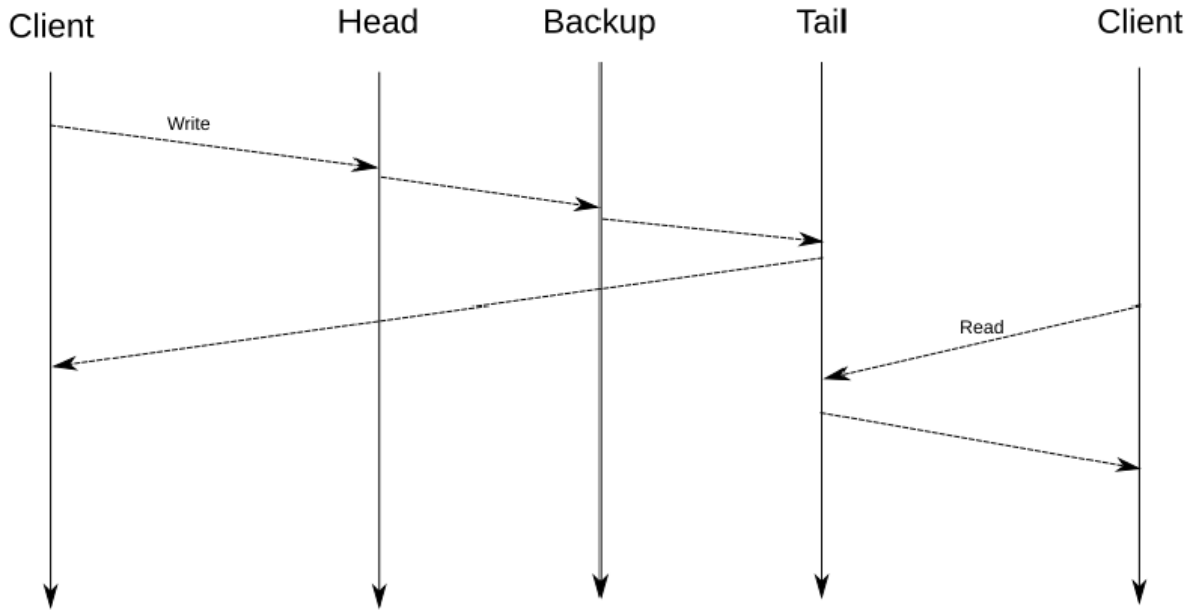
Deze hebben allemaal één node waar writes gebeuren, bestaan vele andere systemen met ander consistency modellen

Primary/backup



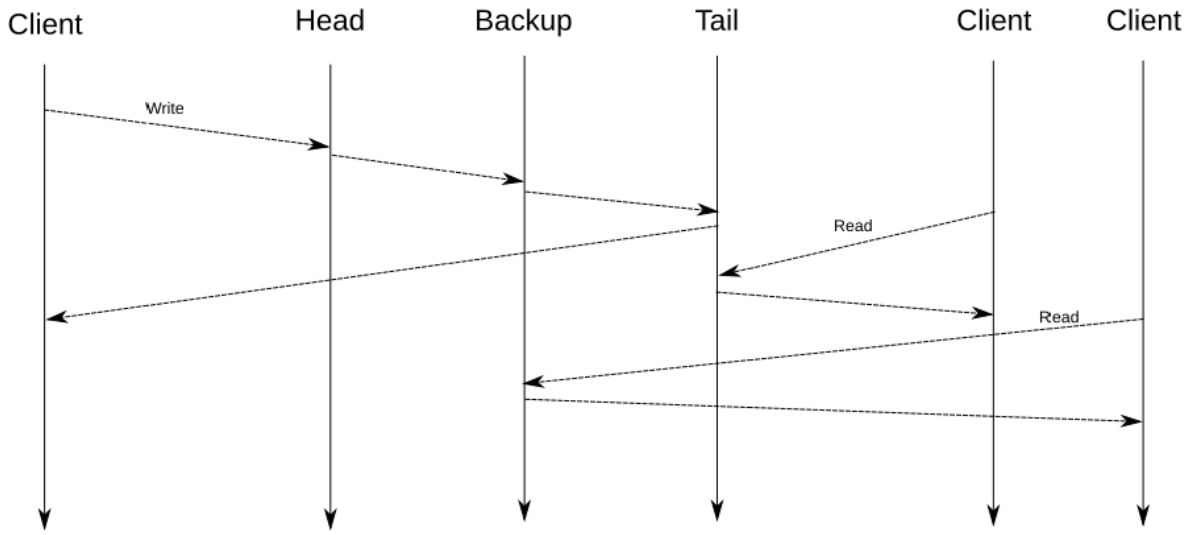
- Eén van de replica's dient als primary, de rest is backup
- Alles request gaan via primary
- Bij write: primary bereidt aanpassingen voor een vraagt backups hetzelfde te doen
- Als alle backups bevestigd hebben, bevestiging aan client
- In databases vaaak "two phase commit" (2PC)
  - Eerst een "prepare phase"
  - Als alle backups bevestigd hebben: "commit phase"

Chain replication



- Keten van replica's: head, aantal gewone backups en tail
- Write request proageert door chain
  - Elke replica houdt zo aanpassingen bij
  - Commit als bij tail aankomt
- Read requests gebeuren enkel via tail

CRAQ



= Chain replication with apportioned queries

- Extensie van chain replication
- Write gebeurt nog steeds op dezelfde manier
- Read mag bij eender welke replica gebeuren
- Wat als zelfde client van verschillende replica's leest?

Coördinator

- Hele idee achter replica's is "fault tolerance"

    - Systeem kan blijven werken als node wegvalt
    - Mischien moet er een nieuwe primary komen, nieuwe head, tail,....
- Er is typisch een aparte coördinator die beslit welke node welke rol krijgt
- Mag op zich geen 'single point of failure' worden!
- 'Coördinator' is vaak zelf systeem van meerdere nodes
    - Beslissen dan samen via 'consenus algoritme'

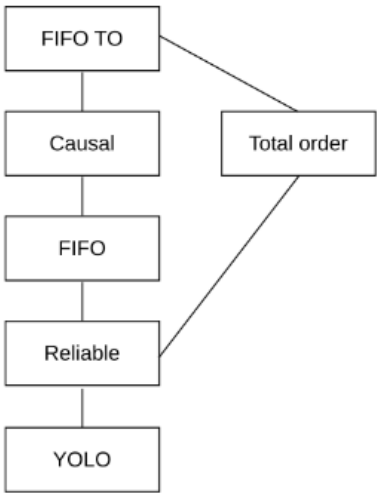
Sharding

- Vaak wordt niet alle data op alle replica's bijgehouden
- Verschillende delen van de data op verschillende replica's
    - = Data partitioning
    - = sharding

Broadcast

- Types broadcast:
    - Best effort: berichten al dan niet afgeleverd ('YOLO')
    - Reliable: berichten zeker afgeleverd; maar in wellekeurige volgorde
    - FIFO: berichten van zelfde zender worden afgeleverd in volgorde van verzenden
    - Causal: aflevering resprectert de "happens before" relatie
    - Total order (TO)/atomic broadcast op elke node worden berichten in zelfde volgorde afgeleverd
        - Hiervoor moeten beichten ook naar zichzelf gaan, mogelijk pas later afgeleverd
    - FIFO total rder: combinatie van FIFO en TO
- Hoe bericht verspreiden
    - Eén keer naar iedereen sturen misschien niet fault tolerant genoeg
    - Eager reliable broadcast: iedereen stuurt bericht nog verder $O(n^2)$
    - Gosip/epidemic protocols: iedereen stuurt naar aantal ander nodes (random)
        - Blockchian

Hiërarchie



- FIFO TO impliceert zwel TO als causal broadcast
- Causal impliceert FIFO broadcast

Implementatie

- FIFO broadcast: kan via sequence numbers
- Causal broadcast: Kan met systeem dat lijkt op vector clocks
- TO: kan bvb met 'single leader' aanpak
  - Eén node beslist over volgorde
  - Is analoog aan consensus probleem: daar moet groep van nodes ook overeenkomen in welke volgorde beslissingen genomen worden
  - Men kan bewijzen dat TO broadcast en consensus equivalente problemen zijn

Consensus

- Consensus probleem:
  - Groep nodes
  - Beslissing nemen, waarde overeenkomen
  - Typisch herhaalderlijk
  - Fault telerant tegen:
    - Netwerkpartitie: 'split brain' vermijden
    - Wegvallen aantal nodes
- Hangt vaak samen met leader election

FLP

- Per beslissingsronde wil je dat consensus algoritme voldoet aan deze criteria:
  - Termination: er wordt uiteindelijk een beslissing genomen
  - Agreement: alle processen komen tot dezelfde beslissing
  - Validity/integrity/non-triviality: De overeengekomen waarde/beslissing moet een van de voorstellen zijn.

- FLP resultaat:
  - Zegt dat beslissing voortdurend uitgested kan worden door ongusntige timing in het algoritme
- Gaat over echt asynchrone systemen, in praktijk eerder partieel synchroon

Bekende algoritmes

- **Paxos**
  - Paxos: één beslissingsronde
  - Multi-paxos: steeds nieuwe beslissingen overeenkomen
  - Wordt beschouwd als een complex systeem
- **Viewstamped replication** (VR)
  - Bevat consensus mechanisme als onderdeel voor systeem rond replicated state machine
- **Raft**
  - Zorgt voor overeenkomst van log (van beslisingen)
  - Een 'leader' mag toevoegingen aan log voorstellen
    - Moet dan nog door meerderheid bevestigd worden
  - Als er nog een leader is, moet die eerst verkozen worden
  - Mogelijkheid om aantal nodes in systeem te wijzigen

Quorums

- Quorum: hoeveelheid nodes die succesvol moeten reageren op bepaald request
- Komt ook terug bij replication/consistency
  - Soms apart voor reads/writes: read quorum, wirte quorum
  - Ook hier belangrijk dat ze overlappen

Generalen: Two generals problem, Byzantine generals problem

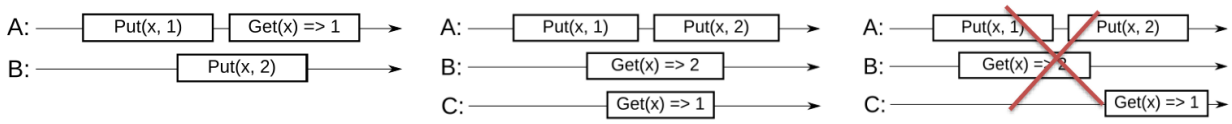
- Two generals problem
  - Twee legers, overeenkomen of ze stad aanvallen of niet
  - Generaals zijn betrouwbaar, eerlijk, kunnen berichten sturen
  - Met communicatie kan iets mis gaan
  - Probleem met ack: is het oorspronkelijk bericht niet aangekomen of de acknowledgment?
  - Je kan nooit helemaal zeker zijn dat je consensus bereikt hebt!
- The Byzantine Generals Problem:
  - Aantal generaals moeten overeenkomen of ze aanvalen
    - De eerlijke generaals moeten tot consensus komen

- Sommige generaals kunnen verraders zijn
  - Iets anders doen dat ze beloofd hadden, tegenstrijige berichten sturen
- Berichten kunnen nu probleemloos verzonden worden
- Belangrijk resultaat 3f+1 generaals nodig om f valse te doorstaan
  - Minder nodig via cryptografie (digital signatures)

Consistency

Linearizability

- Strong consistency/linearizability
  - Lezen/schrijven kan naa rmogelijk meerdere systemen
  - Gaat over verschillende processen die concurent operaties uitvoeren
  - Een operatie (bvb get/put) heeft begin en einde
  - Ergens daartussen is er een ogenblik waarin atomair uitvoer gebeurt
  - Er is conceptueel een globale tijd om deze ogeblikken te vergelijken
  - Een hystory van oeraties (met begin/einde) is linearizable als zulke ogenblikken geïdentificeerd kunnen worden



- In ysteem met zowel read als write naar meerder replica's:
  - Linearizability verkijgen kan internsief zijn
  - Moet consensus bereikt worden over volgorde
- Afhankelijk van toepassing: geen linearizabiltiy nodig?
- Typisch wil je minstens
  - Read-your-writes consistency/read-afer-write consistency
  - Als client net geschreven data leest, moet dit deze waarde, of nieuwere, opleveren

Eventual consistency

- Queries bij verschillende nodes kunnen tijdelijk inconsistente resultaten opleveren
- Uiteindelijk, als er geen updates meer gebeuren wordt consistente toestand gebruikt
- Vrij zwak model: wat als updates nooit stoppen?

Strong eventual consistency

- replica's die op zeker moment zelfde updates gezien hebben, hebben compatibele toestand
- Geen eenduidige aanpak, toepassingsafhankeljk
- Vaak conflict resolution nodig
- CRDTs: conflict-free replicated data types

CAP

- **= Consistency, Availability, Partition tolerance**
  - Consistency: gaat over strong eonsistency
  - Partition: gaat over netwerk partitie, dat nodes elkaar
- Zegt dat je niet alledrie kan garanderen, dat er trade-offs nodig zullen zijn
  - Bvb bij Paxos/Raft consensu
    - Voorziet dat het niet misloopt bij partitie
    - Maar mogelijk duurt het langer om majority quorum te krijgen, ten koste van availability
  - bvb bij eventual consistency:
    - Kan overweg met partities, zonder dat ysteem minder available wordt
    - Geen strong consistency

Blockchain

Gedistribueerde database die je kan vetrouwen

Hashing

- In stukjes door elkaar halen
- Een operatie die input van willekeurige lengte omzet in iets van specifieke lengte
  - Deterministisch
- Voor cryptografische toepassingen:
  - Erg gevoelig aan input: één andere bit ⇒ volledig andere hash
  - Zit er random uit op basis van uitzicht input is hash onvoorspelbaar
  - enkel via brute force inverteerbaar
  - Hash collisions in praktijk niet mogelijk
    - In theorie uiteraard perfect mogelijk
- Wanneer zo'n hash collision zich toch voordoet: hash functie is niet alnger cryptografisch veilig
- MD5 in 2005 aangetoond dat collisions kunnen
- SHA-1: begin 2017 aangetoond

Gebruikte hash functies

- Bitcoin
  - SHA-256
- Ethereum
  - Keccak-256
  - Variant van SHA-3
- Voor blockchain op zich is enkel hash functie nodig
- Voor inhoud van blocks: wat cryptografie nodig

Public key cryptografie

- Basisidee:
  - Genereer een 'key pair': private key & public key
  - Public key mag iedereen weten, en kan gebruikt worden om berichten te encrypteren
  - Jij alleen kent private key en kan bericht weer decrypteren

RSA

- Private en public key eigenlijk gelijkwaardig
  - Met public key encrypteren ⇒ met private key decrypteren
  - Met private key encrypteren ⇒ met public key decrypteren
- Kan ook handtekening onder een bericht zetten:
  - Gabasseerd op dat specifieke bericht
  - Op basis van private key
  - Iemand met public key kan nagan
    - Dat jij overeenkomstige private key hebt
    - Dat signature idd voor vermelde bericht dient
    - Kortom dat 'eigenaar' van public key een concreet bericht gestuurd heeft

ECC

- Veel recentere aanpak: elliptic curve cryptography
- Nog steeds private & public keys, maar zijn van fundamenteel verschillende aard
- Met veel minder bits een gelijkaardige beveiliging als bij RSA

ECDSA

- Voor blockchain: Elliptic Curve Digital Signature Algorithm
- Is een 'signature algorithm', dient niet voor encryptie en decryptie
- Bewijzen dat een specifiek bericht geschreven werd door de eigenaar van een zekere public key

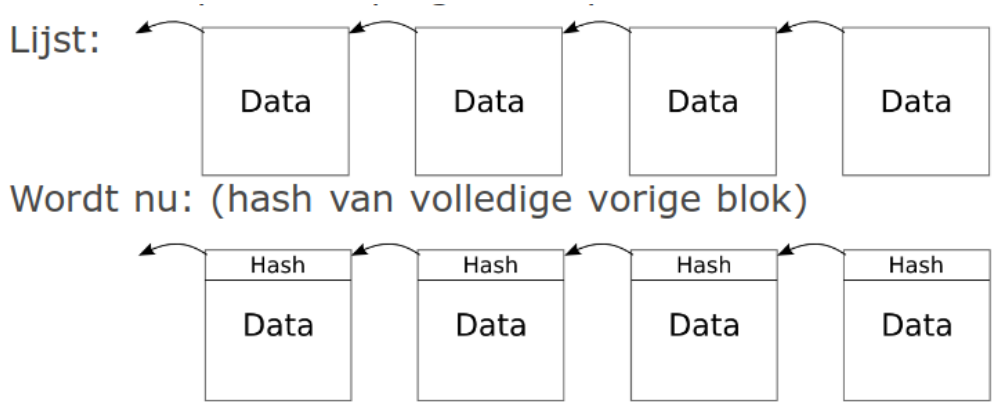
- Zowel bitcoin als ethereum gebruiken ECDSA:
  - Secp256k1
  - 256 bits voor private key, 257 bits voor public key

Hash pointers

- Combinatie van:
  - Pointer naar stuk geheugen
  - Hash van dat stuk geheugen
- Hash functie is heel gevoelig aan precieze inhoud
- Zo nagaan ofgeheugen gewijzigd is
- Conceptuele voorstellig
  - Gaat over verwijzing naar data (pointer) en de hash van die data
  - Hoeft niet over ram geheugen te gaan, kan evengoed op HD etc

Lijst

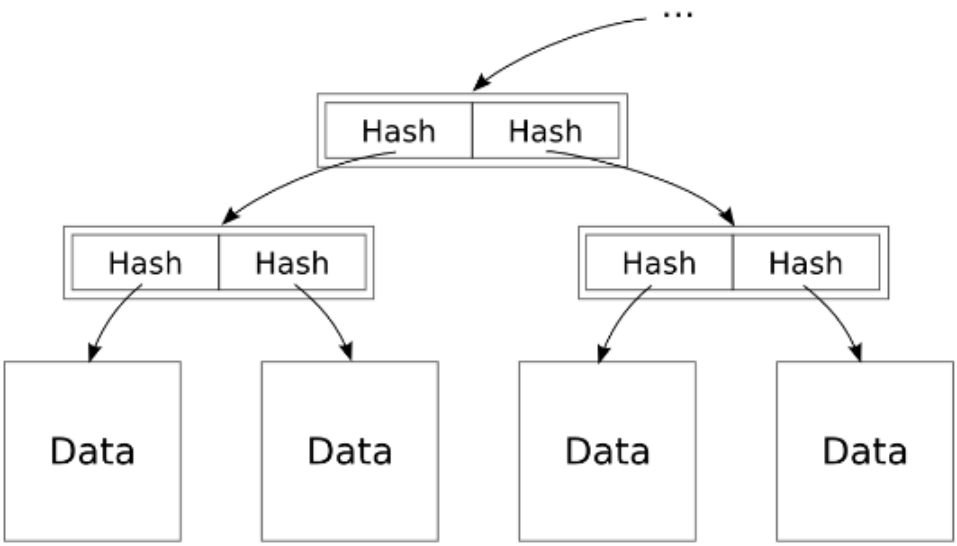
- Gekende datastructuren kunnen veiliger gemaakt worden met hash pointers ipv gewone pointers



  = Blockchain
- Hash in het block beschrijft data+hash vorige block
- Voor blockchain structuur zelf:
  - enkel hash functie nodig
  - voor de inhoude zullen we cryptografie nodig hebben

Boom

Een boomstructuur met hash pointers wordt een Merkle tree genoemd



Blockchain

- Achterliggende datastructuur: lijst met hash pointers
- Gedistribueerd
  - Vele nodes hebben kopie van hele blockchian, controleren allemaal geldigheid van blocks
- Nodes zijn los gekoppeld: relatief weinig verbindingen

Inhoud van blocks

- Bitcoin is een cryptocurrency
  - In block zitten aantal transacties van ene eigenaar naar andere
- Eigenar wordt geïdentificeerd adhv public key: 'adres'
  - Zoals een rekening nummer
  - Een gebruiker kan meerdere adressen hebben
- Schets van overschrijven van bitcoin in transactie
  - Transactie vermeldt input adres en output adres
  - Eigenaar van input adres tekent transactie om te laten zien dat hij idd dat adres (en dus de bitcoin) bezit
- Flexibiliteit:
  - Meerdere inputs en outputs mogelijk
  - Eigenlijk worden niet ewoon adressen gebruikt in transactie, maar worde een eenvoudige programmeertaal gebruikt: 'Script'
  - Dit is een stack gebasseerde taal zonder loops ze is nie turing compleet en kan niet vastlopen.
  - Ethereum zijn transacties en de programmeertaal uitgebreidere mogelijkheden hebben

Hoeveel crypto bezit een adres?

- Voor de hand liggende aanpak
  - Hou soort 'state' bij per adres, de hoeveelheid van de currency
  - Ontvangen van currency verhoogt het opgeslagen getal
  - Betalen verlaagt dat getal
- Ethereum volgt deze aanpak, bitcoin niet
- Bij bitcoin moet (*) alle bitcoin uit input opgebruikt worden
  - Om kleiner bedrag over te schrijven: schrijf de rest nog over naar jezelf
  - Laatste overschrijvingen naar jouw adres bepalen hoeveel bitcoin je bezit = Unspent Transaction outputs (UTXO)

Mining

- Transacties die uitgevoerd moeten worden
  - Stuur door naar geconnecteerde nodes
  - Stuen die op hun beurt weer verder
- Nodes hebben zo naat de blockchain ook een hele reeks transacties die in de blockchain opgenomen willen worden
- Sommige nodes zulen proberen een block te vomen met transacties die aan blockchain toegevoegd kan worden
- Niet elk block is zomaar geldig
- De miners moeten een cryptografische puzzel oplossen
  - Doel is de hash van het block te laten beginnen met aantal 0-bits
  - Da hash van block zal in eerste instantie iets vrij willekeurig zijn
  - Kan echter beinvloed worden dor enkele vrij in te vullen zake nin het block
  - Zo kan op brute force manier gezocht worden naar block waarvan hash met bepaalde aantal 0-bits begint
  - Als een miner dit vindt:
    - Voegt toe aan eigen blockchain
    - Stuurt block door naar verbonden nodes, die het weer verder sturen
    - Krijgen hiervoor beloning: voegen zelf transactie toe zonder input
    - Krijgen ook transactiekosten: de overschotten in transacties
- Duurt tijdje voor block zich over hele netwek verspreidt
- Is dus zeker mogelijk dat andere miner een ander, even geldig block toevoegt
- Mines beslissen zelf op welke tak verder te werken
- Afspraak is dat de langste keten de geldige is
- Doordag geldig block zoeken niet makkelijk is zal de blockchain zich vrij snel weer stabiliseren

Confirmations & double spending

- Als je transactie net in een block zit, ben je dus niet zeker dat die echt in de geldige blockchain tak zit
- Hoe meer blocks er bovenop dat block komen, hoe zekerder je bent dat de transactie echt in blockchain zit
  - Wordt confirmations genoemd
- Wacht op 'voldoende' confirmations om zogenaamde double spending attack te voorkomen

Double spending

- A sturt transactie met aantal bitcoin naar B, voor aankoop product
- Stel dat B al onmiddellijk (0 confirmations) product opstuurt
- A maakt ook transatie waarbij zelfde bedrag naar ander adres C gestuurd wordt (of naar zichzelf)
  - Begint te minen zodat die transactie in gelijkwaardig (of eerder) block terechtkomt
  - Zorgt door verder minen dat die transactie in langste chain komt
- Resultaat: B geen geld ontvangen, maar product kwijt

51% attack

- Iemand krijgt volledige controle over blockchain
  - Heeft meer rekenkracht nodig dan de rest van het netwerk
- Wordt daarom ook 51% aanval genoemd

Bitcoin mining regels

- Moeilijkheidsgraad van mining puzzel
  - Aantal 0-bits waarmee hash moet beginnen
  - Wordt aangepast (bij bitcoin om de twee weken)
  - Op deze manier: gemiddeld een nieuw block elke 10min
- Belonging voor block reward neemt af: halveert elke 210000 (+- elke 4 jaar):
  - Limiteerd hoeveelheid bitcoin die in omloop kan komen
- Hash wordt enkel van een header berekend
  - Bevat op zich Merkel root hash van transacties
  - Helpt toevoegen kleine of zelfs lege blocken te voorkomen

Ethereum

- 15s gemiddelde blocktime
  - Moeilijkheidsgraad wordt continu aangepast
- Geen UTXO systeem, maar balans per adres:

    - Aan elk adres wordt een state, bepaalde data geassocieerd, bevat balans van cryptocurrency
    - Kan ook veel algemenere data bevatten, zelfs code: smart contract

Smart contracts

- Niet elke balans aan adres koppelen, maar ook code en algemene data
- Met transactie naar contract adres: code aanroepen die contract data kan manipuleren

Solidity

- Solidity: high-level programmeertaal voor smart contract
    - Lijkt wat op Java
    - Ipv 'class' maak je 'contract'
- Inheritance mogelijk
- Kan verwijzen naar ander contract dat al bestaat op blockchain
- Events kunnen gegenereerd worden, externe applicaties kunnen daarop reageren
- Dapp = distributed application
- Aanroep van smart contract moet via transactie gebeuren
- Kan niet zomaar op bepaald moment zelf iets uitvoeren

Gas

- Blockchain is verspreid over vele nodes
- Moeten allemaal alle transacties verwerken
    - Dus ook alle EVM code uitvoeren in alle transacties
    - Taal laat lops toe, is turing complete
- Hoe voorkomen dat nodes te lang bezig zij met transactie of zelfs vastlopen?
    - Gas
- Elke operatie (+, x, functie-aanroep, ...) komt overeen met een bepaalde operationele kost, een hoeveelheid 'gas'
    - Voor specifieke smart contract aanroep zal er zoeen welbepaalde hoeveelheid gas nodig zijn (al weet jepas hoeveel na uitvoering!)
- Je betaald dus voor de code die je uitvoert.
- Aparte eenheid omdat de waarde van ether hard fluctueert

Proof of work

- Block toevoegen aan blockchain: cryptografische puzzel oplossen
    - Wordt 'Proof of Work' genoemd
- Ecologisch niet zo goed, enorm energie verbruik

- Minig evolutie bij bitcoin: CPU, GPU, Field Programmable Gate Arrays, Application specific integrated circuits (ASICs)
- ASICs domineren bitcoin mining
- ethash was ASIC resistant
- Miners organiseren zich in pools
    - Block reward wordt dan verdeeld onder deelnemers
    - Zorgt voor centralisatie. Gaat tegen het principe in

Proof of Stake

- Bewijs van aandeel
- Idee is nutteloze berekeningen bij PoW achterwege te laten
- Beslissingen gebeuren op basis van aandeel in de cyrptocurrency
    - Als je 10% van de crypto bezit mag je in 10% van de gevallen beslissen welk block het volgende wordt
    - Maakt 51% aanvaal moeilijker
    - Ecologisch veel meer verantwoord
- Maar:
    - hoe eerlijk is dit ("the rich get richer")
    - Hoe vertakking voorkomen
    - Wordt er enkel cryptocurrency herverdeeld, of komt er toch nog bij?

Forks

- Code fork: start van bvb bitcoin code, met wat andere opties
- Soft fork (van blockchain!):
    - Nieuwe regels vor blocks zijn subset van oude, nieuwe blocks worden ook aanvaard door oude nodes
    - Om tak te krijgen met nieuwe versie: meerderheid moet die versie gebruiken
- Hard fork
    - Maar drastische anpassing, typisch gestart vanaf bepaald block
    - Nieuwe blocks niet aanvaard door oude nodes
    - Soms blijvende split in blockchian als niet iedereen wil upgraden
        - BTC vs bitcoin cash, ETH vs ETC