KOMMUNIKATION I MULTI-GPU KLUSTER MED MPI OCH CUDA

David Eränen

Kandidatavhandling i datateknik
Handledare: Jan Westerholm
Institutionen för informationsteknologi
Åbo Akademi
4 april 2011

Referat

Mitt abstrakt

Nyckelord: mina nyckelord

Innehåll

1 Inledning **1**

2 Kommunikationsbibliotek mellan parallella processer: MPI **2**

2.1 Översikt 2

2.2 Operationer 2

2.3 Användningsexempel 2

3 General purpose GPU computing (GPGPU computing) **4**

3.1 Översikt 4

3.2 CUDA (Computer Unified Device Architecture) 4

3.2.1 Tesla GPU:n 5

4 Datorkluster **6**

4.1 Översikt 6

4.1.1 GPU kluster 7

5 Användande av MPI och CUDA vid kommunikation i GPU kluster **9**

5.1 Problem 1: Mappande av MPI processer till grafikprocessorer . . 9

5.1.1 Lösning 9

5.2 Problem 2: Procedur för överföring av data mellan grafikprocessorer 10

5.2.1 Rättfram implementation 11

5.3 Problem 3: Delning av gränsdata mellan grafikprocessorer 12

5.3.1 En möjlig lösning 12

5.3.2 En första lösning 13

5.3.3 Överlappning av send/receive med kopiering av data . . 13

5.3.4 Överlappning av beräkningar med MPI kommunikation . 14

5.3.5 Slutlig lösning 16

6 Slutsatser **18**

7 Litteraturförteckning **19**

1 Inledning

Här kommer min inledning

2 Kommunikationsbibliotek mellan parallella processer: MPI

2.1 Översikt

Message Passing Interface, även använt i dess förkortade form MPI, är ett kommunikations API som används för att skicka meddelanden mellan parallella processer. MPI är avsett att tillhandahålla en virtuell topologi, synkronisering och förmågan att utbyta data mellan processer på ett språkoberoende sätt. MPI siktar också på att ge hög prestanda, skalbarhet och portabilitet. En MPI process kan mappas till exempelvis en nod i ett kluster, eller en individuell kärna på en CPU. [1] Detta gör MPI mycket användbart då man skapar parallella program för användning i högprestationskluster och andra parallella system.

2.2 Operationer

MPI stöder ett flertal grundläggande typer av biblioteksfunktioner, såsom elementära send/receive operationer, sammanställning av ofullständiga resultat med gather/reduce operationer och synkronisering med barrier operationen. Det finns även stöd för erhållning av information om det underliggande nätverket som MPI programmet körs på, till exempel antalet processer i beräkningssessionen, processor identitet, angränsande processer med flera. Punkt-till-punkt operationer såsom send/receive funktioner kan anges som synkrona, asynkrona, buffrade och ”redo”, vilka låter programmeraren utnyttja parallellism så effektivt som möjligt. [1]

2.3 Användningsexempel

Varje process i ett MPI program exekverar samma kod. Om vi vill definiera processen med id ’0’ som en så kallad ”huvudprocess” som koordinerar de andra processerna (eller ”slav processer”), så blir man tvungen att avgrena till exempel via ett if-uttryck. Detta demonstreras i Pseudokod 1 på sid. 3. Processen med id ’0’ fungerar som huvudprocess, medan alla andra processer kommer att stiga in i else-blocket.
Som ett exempel, anta att vi vill att huvudprocessen skall distribuera numrena

$1, 2, 3, ..., n-1$ (där n är antalet processer) till slavprocesserna, sedan kvadrera numrena, och slutligen skicka tillbaka resultaten till huvudprocessen för uppvisning på skärmen. Pseudokod 2 och 3 visar hur huvudprocessen respektive slavprocesserna åstadkommer detta.
Som framgår av pseudokodexemplena är användandet av MPI för utveckling av parallella program mycket okomplicerat. Detta, tillsammans med dess prestanda och skalbarhet, gör MPI ett lönsamt val för körandet av parallella program på datorkluster och superdatorer.

Pseudokod 1 Avgrening till huvudprocess och slavprocess

```
if id of current process is 0 then
   master process instructions here
else
   slave process instructions here
end if
```

Pseudokod 2 Huvudprocess distribuerar inputdata i form av nummer till slavprocesser. Huvudprocessen väntar sedan på att slavprocesserna skickar tillbaka resultat som sedan visas på skärmen.

```
while there is more data to send to slaves do
   send data to slave using MPI_Send
end while
while all results have not been received do
   receive result from slave using MPI_Recv
   store result in array
end while
print contents of array
```

Pseudokod 3 Slavprocesserna tar emot data från huvudprocessen, kvadrerar datan för att sedan skicka tillbaka resultatet till huvudprocessen.

```
ta emot data från huvudprocessen med MPI_Recv
kvadrera datan
ta emot data från huvudprocessen med MPI_Send
```

3 General purpose GPU computing (GPGPU computing)

3.1 Översikt

Förklara termen GPU. Before DirectX8, which is a collection of application programming interfaces for handling, among other things, game programming and graphics, was released in 2000, GPUs were used almost exclusively for computer graphics. However, with the introduction of Shader Model 1.1 alongside DirectX8 (and all the subsequent versions of Shader Model and DirectX), the programmability of the graphics cards has increased dramatically, and has therefore become more suitable for doing general computation.
Grafikprocessorer är konstruerade för att effektivt lösa problem som är mycket parallelliserbara, såsom manipulation av pixlar i en bild eller vertex:ar i en tredimensionell polygon mesh. Det finns flera icke-grafikrelaterade områden som kan dra nytta av mycket parallella beräkningar, som till exempel video och ljud processering, fysik-baserad simulering, computational finance, medicinsk bildbehandling med flera. Eftersom användningsområdet för grafikprocessorer i huvudsak är grafikprocessering, så är sättet de kan bli programmerade mycket begränsande. Det finns inget stöd för rekursiva funktioner och aritmentik med dubbelprecision antingen stöds ej eller är mycket långsam. [2]

3.2 CUDA (Computer Unified Device Architecture)

CUDA, utvecklat av NVIDIA, är en parallell beräkningsarkitektur som accelererar beräkningar på NVIDIA grafikprocessorer från och med G8X serien och framåt, vilken inkluderar GeForce, Quadro och Tesla varuslagen. C med NVIDIA utvidgningar används för programmering, och kompilering hanteras av en PathScale Open64 C kompilerare. CUDA gör möjligt GPGPU beräkning vid sidan av grafikrendering för att öka till exempel fysikmotor prestanda, men användning i icke-grafiska applikationer är även möjligt. [2] From the official NVIDIA website we can read that “CUDA has been enthusiastically received in the area of scientific research.” It is also mentioned that there is “more than 700 GPU clusters installed around the world”.

3.2.1 Tesla GPU:n

NVIDIAs Tesla GPU är den första dedikerade GPU:n avsedd för högpresterande kluster. För att betona detta, så har kortet inte ens en videoutgång. Tesla GPU:na har ECC skydd för kompromisslös datatillförlitlighet, stöd för C++ och flyttals prestanda med dubbelprecision. Vid jämförelse med den nyaste fyrkärniga CPU:n så förbrukar Tesla 20-serie-GPU:n, enligt den officiella NVIDIA hemsidan, 1/20:e-del av strömmen av CPU:n medan den leverar motsvarande prestanda. [3, 4]

Figur 3.1. Tesla GPGPU

4 Datorkluster

4.1 Översikt

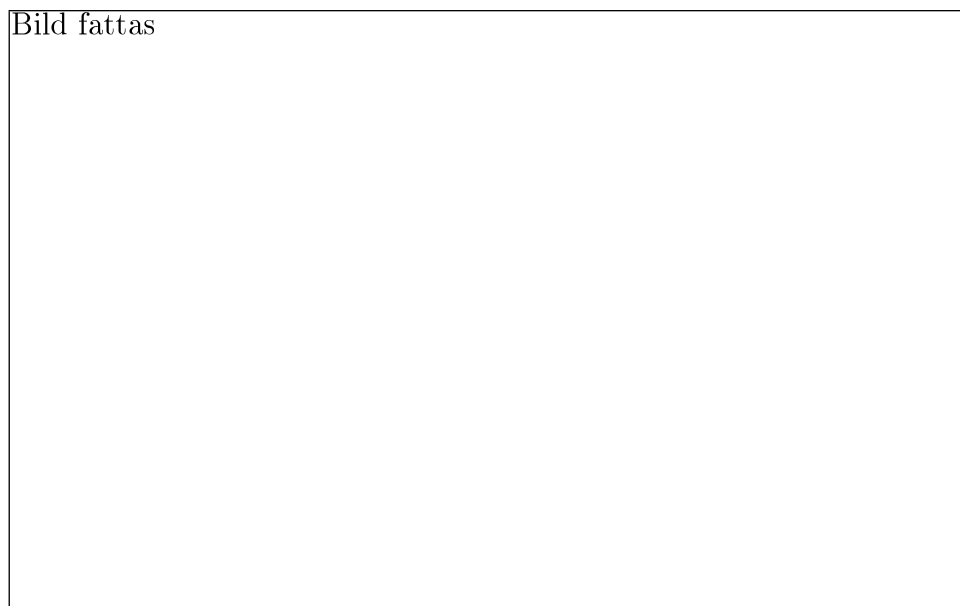
Ett datorkluster är en samling av två eller mera datorer som är kopplade till varandra via ett nätverk. Denna samling av ihopkopplade datorer kan bland annat användas till processorkrävande beräkningar genom att dela upp uppgifterna i mindre delar och köra dem parallellt på varje dator, kallad nod. Andra användningssyften kan vara till så kallade ”högtillgänglighetskluster”, där redundanta noder kan ta över om andra noder går sönder. ”Lastbalanseringskluster” ruttar all trafik genom lastbalanseringsnoder som används för att distribuera arbetsbördan så effektivt som möjligt till andra lediga noder. ”Högprestanda kluster” är kluster som används för parallell beräkning, som nämndes ovan. Datorkluster har många fördelar, och bland dessa är:

- **Kostnad versus prestanda och processeringskraft.** Priset på en bordsdator är billigt jämfört med special-purpose hårdvara, och deras prestanda ökar konstant på grund av konsumentarnas efterfrågan. Deras kollektiva processeringskraft utnyttjas bäst i högprestanda kluster, eftersom varje nod kan (potentiellt) utnyttjas till fullo.

- **Snabbare och förmånligare nätverksteknologi.** Precis som med bordsdatorer så ökar nätverkshårdvara i både hastighet och kapacitet samtidigt som utvecklingskostnaden minskar.

- **Skalbarhet och tillgänglighet.** Processeringskapaciteten för ett datorkluster kan enkelt expanderas genom att lägga till mera noder till klustret. Om en nod går sönder kan man utan ansträngning ersätta den med en ny nod, men program som körs kommer att stängas och blir tvugna att köras igen.

Datorkluster använder ett message-passing gränssnitt, till exempel MPI, vid kommunikation melland noder, och lyder under distribuerat minne, eftersom varje individuell nod är en komplett dator med dess eget RAM minne. Då en nod behöver dela på data med en annan nod så måste datan i fråga kopieras och överföras över nätverket. Detta inför en signifikant overhead ifall det görs ofta, och därför försöker man undvika att högprestanda kluster delar på data. Nedan finns ett diagram av ett simpelt datorkluster som illustrerar hur noder är anslutna genom nätverket, och hur klientdatorer kommunicerar med klustret.



Figur 4.1. Ett simpelt GPU kluster

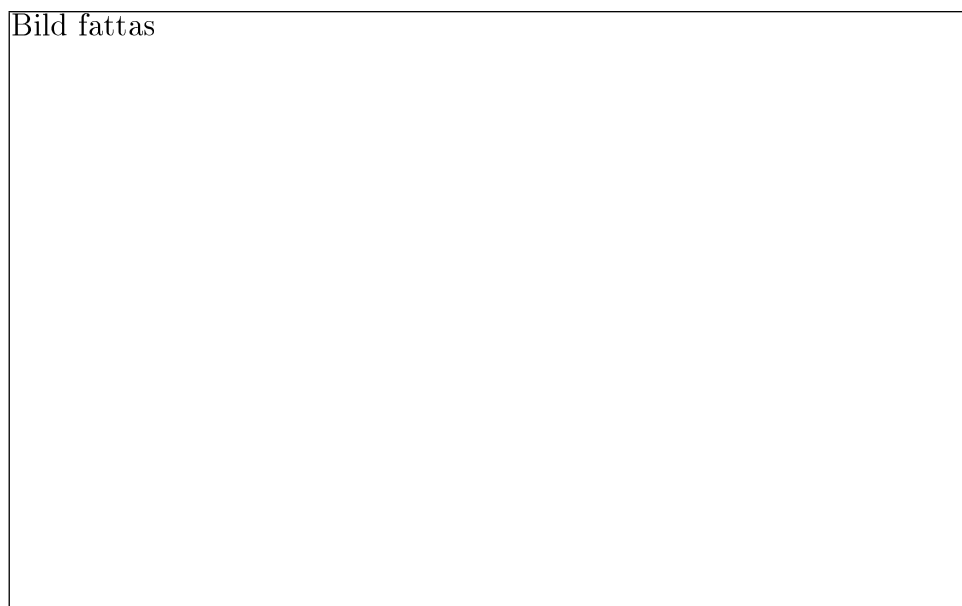
4.1.1 GPU kluster

Allt eftersom grafikprocessorer har blivit bättre anpassade till allmän processering, så är det ett naturligt steg framåt att börja använda GP-grafikprocessorer i högprestanda kluster. Varje nod i klustret rymmer en eller flera GP-grafikprocessorer som tar hand om den beräkningsintensiva delen av en uträkning. Detta har tidigare gjorts med CPU:ar. CPU:ns nya ansvar är hantering av kommunikation mellan grafikprocessorer, både mellan grafikprocessorer belägna på samma nod (i ett multi-GPU kluster) och grafikprocessorer på separata noder. Användande av GP-grafikprocessorer i kluster ökar inte bara den totala processeringskapaciteten, utan de har ävan potential att vara mera kostnadseffektiva, inte mycket utrymmeskrävande, samt strömsnåla. GPU kluster inför även flera utmaningar, till exempel hur man skall skedulera jobb mellan grafikprocessorer samt hantera resurser mellan ROM på noderna, och globalt/lokalt minne på GPU:na själva. [5]

Arkitektur

Arkitekturen hos en grundläggande GPU kluster nod som använder en Tesla S1070 [6] är visualiserad i Figur 4.2. En nod är ansluten till resten av klustret med en InfiniBand [7] (IB i figuren) kommunikationslänk med hög genomströmmning och låg latens. Prestandan hos InfiniBand länken är mycket viktig, eftersom den behöver försöka matcha bandbredden hos kommunikationen mellan noden och GPU:n, som är mycket hög tack vare PCI Express x16 [8]

bussarna belägna på nodens moderkort. För att utnyttja dataöverföringsprestandan till fullo borde man överväga att matcha värdminnesstorleken med grafikprocessorernas minnesstorlek. Tesla S1070 i maskinen i Figur 4.2 rymmer fyra grafikprocessorer med 4 GB minne var, totalt 16 GB. [5]



Figur 4.2. Ett exempel på en GPU kluster nod

5 Användande av MPI och CUDA vid kommunikation i GPU kluster

5.1 Problem 1: Mappande av MPI processer till grafikprocessorer

Vi behöver ett sätt att mappa MPI processer till grafikprocessorer så att varje MPI process enkelt kan överlämna ett beräkningsjobb den har blivit tilldelad till en specifik GPU. CUDA stöder endast en värdtråd för varje GPU enhet, och samma process kan inte ge två grafikprocessorer instruktioner samtidigt. [2]

5.1.1 Lösning

Detta problem har en enkel lösning om det finns lika många grafikprocessorer per nod som det finns CPU kärnor (MPI startar en process per kärna), eftersom varje process skulle mappas till en GPU. Om det finns till exempel fyra grafikprocessorer på noder med 6-kärniga CPU:ar så skulle två kärnor bli utan grafikprocessorer. Detta kan fixas med att ha varje slavprocess att rapportera tillbaka (via MPI) till huvudprocessen om dom har en GPU mappad till sig eller inte. Huvudprocessen kan då skicka tillbaka information till varje slavprocess som informerar vilka GPU-mappade processer är lediga mot norr och söder (eller vänster och höger). Detta gör så att endast GPU-mappade kärnor används för kommunikation och beräkningar, medan de andra lämnas oanvända. Vi kommer att använda id:n av en process och antalet processer/kärnor per nod för att räkna ut vilken GPU vi skall mappa till vilken process. Nedan finns ett kodexempel som illustrerar hur man skall initiera slavprocesserna genom att mappa grafikprocessorer till dem och hur man informerar dem vilka deras närmaste GPU-mappade processer är.

Funktionen `cudaGetDeviceCount` returnerar antalet CUDA-stödda grafikprocessorer som finns på noden. Vi sätter `device` variabeln att innehålla GPU:n som en process vill mappa till. Vi har följande exempel: en slavprocess vars id är 7 och finns belägen på ett kluster som har 6 kärnor per nod skulle vilja mappa till GPU nummer 1 på nod 1 (räck-index startar från 0; se Figur 5.1). Det finns en brist i designen som uppenbarar sig då man ser på Figur

Pseudokod 4 Mappande av MPI processer till grafikprocessorer

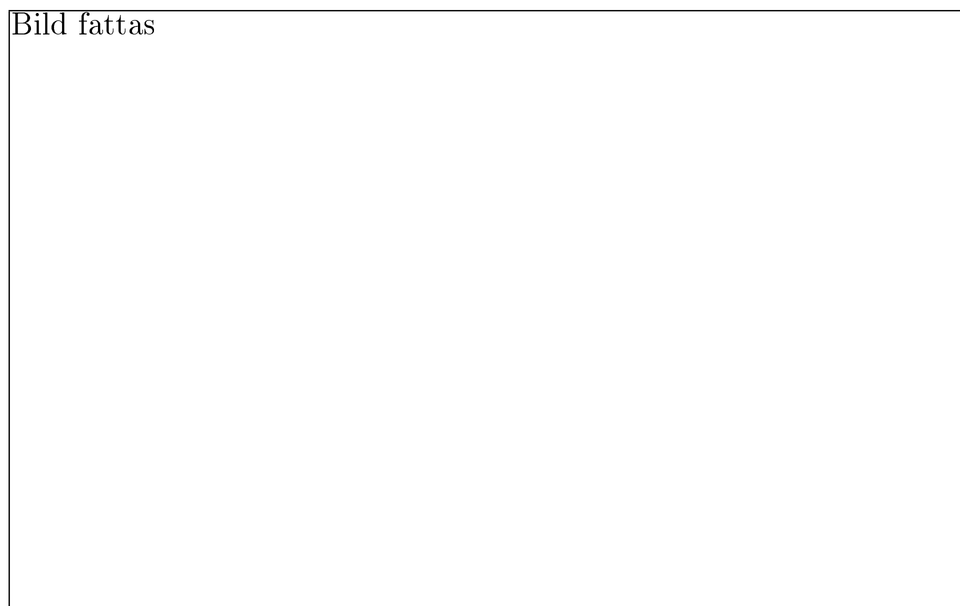
```
if huvudprocess then
  while alla slavprocesser har inte rapporterat do
    MPI_Recv(hasDevice från vilken som helst slavprocess)
  end while

  for all slavprocesser do
    MPI_Send(närmaste nord/syd GPU-mappade processer)
  end for
  /* Här kommer resten av huvudprocess koden */
else if slavprocess then
  deviceCount ← cudaGetDeviceCount()
  device ← (myId%processesPerNode)
  if device < deviceCount then
    cudaSetDevice()
    hasGPU ← true
  else
    hasGPU ← false
  end if
  MPI_Send(hasGPU till huvudprocessen)
  MPI_Recv(närmaste nord/syd GPU-mappade processer)
  /* Här kommer den egentliga beräkningen */
end if
```

5.1. En GPU lämnas oanvänd på nod 0 eftersom huvudprocessen befinner sig där. Detta är lätt att korrigera, men gör pseudokoden onödigt komplicerad för denna studie.

5.2 Problem 2: Procedur för överföring av data mellan grafikprocessorer

När grafikprocessorer introduceras till ett kluster så behöver man ett sätt att kommunicera mellan dem. Eftersom CUDA SDK:n (Software Developmen Kit) inte stödjer detta blir man tvungen att hitta på en egen lösning. Vi kommer här att introducera en metod för kommunikation mellan grafikprocessorer i form av funktionerna `MPI_GPUSend` och `MPI_GPURecv`. Det är klart att vi hamnar att använda både MPI och CUDA funktionsanrop i båda funktionerna. `MPI_GPUSend` skall kopiera en angedd mängd data från GPU:n som är mappad till dess MPI process, samt sända datan till valfi GPU som befinner sig i klustret. `MPI_GPURecv` igen förväntas ta emot data från valfi källa (`MPI_Source`) och överföra datan till GPU:n som är mappad till dess egen MPI process. Vi kommer först att presentera en enkel implementation av dessa funktioner,



Figur 5.1. Noder, kärnor, grafikprocessorer och MPI processer

identifiera vad som kan förbättras och sedan göra en bättre version av båda.

5.2.1 Rättfram implementation

Pseudokod 5 visar en implementation av `MPI_GPUSend` uttryckt i pseudokod, medan Pseudokod 6 visar `MPI_GPURecv`.

Pseudokod 5 MPI_GPUSend

```
MPI_GPUSend(integer node, integer gpu, pointer GPUData,
                                                  integer size)
{
    pointer hostData = malloc(size)
    cudaMemcpy(hostData, GPUData, size, cudaMemcpyDeviceToHost)
    integer GPURank = node*GPUsPerNode + gpu
    MPI_Send(GPURank, hostData, size)
    free(hostData)
}
```

I de mest allmänna av fallen, där en GPU kan när som helst ta emot data från en annan GPU, så borde source argumentet specificeras som `MPI_ANY_SOURCE`. Det finns ett problem med den nuvarande implementationen. Hur vet ”receive” proceduren i förväg hur stor mängd data som kommer att sändas? Om applikationen vet exakt vad som sänds och när samt i vilken ordning datan sänds; då är detta inget problem. Men vad händer om en slavprocess inte vet detta i förväg? I sådana fall krävs en så kallad “handskakning” före den egentliga data överföringen tar plats. Denna handskakning kommer att berätta för

Pseudokod 6 MPI_GPURecv

```
MPI_GPURecv(integer source, pointer hostData, integer size,
                                               pointer GPUData)
{
    hostData = malloc(size)
    MPI_Recv(hostData, size, source)
    cudaMalloc(GPUData, size)
    cudaMemcpy(hostData, GPUdata, size, cudaMemcpyHostToDevice)
    free(hostData)
}
```

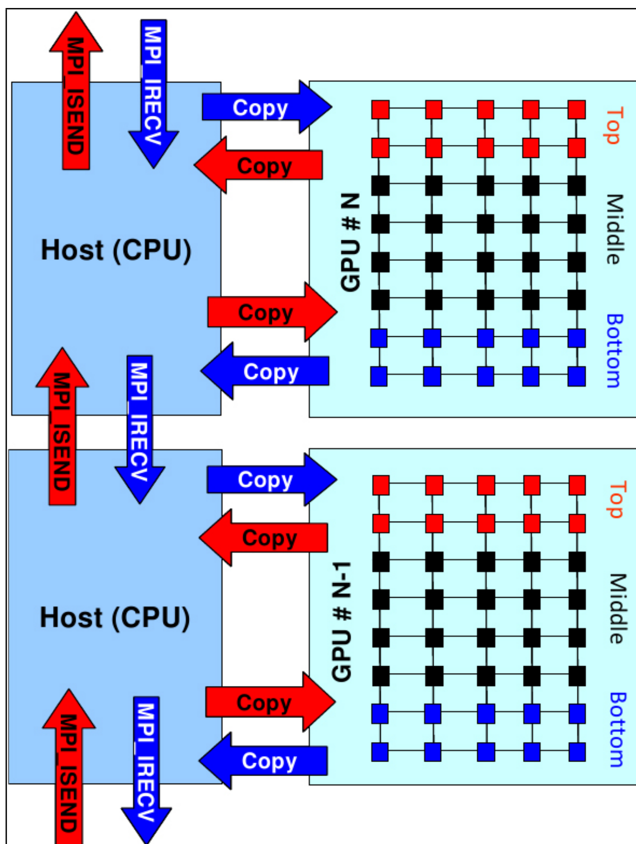
mottagaren vilken slags data kommer att skickas, samt vilken storlek datan har.

5.3 Problem 3: Delning av gränsdata mellan grafikprocessorer

Figur 5.2 illustrerar en allmän parallell beräkningsuppgift där data måste delas mellan processer för att kunna lösa ett särskilt problem. De övre och undre gränserna i ett data gitter måste överföras till de angränsande GPU:na. Det är viktigt att påpeka att det specifika sättet som denna procedur hanteras är avgörande för klusters prestanda och skalbarhet. [9] Det är mycket lätt att introducera oönskade flaskhalsar i designen, och avlägsnandet av dessa flaskhalsar kan ta upp en stor del av utvecklingstiden för programmet.

5.3.1 En möjlig lösning

MPI hanterar kommunikation mellan processer, som kan bli mappade till fysiska kärnor i CPU:ar. Kommunikation mellan grafikprocessorer kan då åstadkommas genom att vidare mappa CPU kärnor till grafikprocessorer som befinner sig på samma nod. Figur 5.3 visar kommunikationen som behövs mellan grafikprocessorer på samma nod och CPU kärnorna (intra-nod kommunikaiton) och mellan grafikprocessorer på olika noder (inter-nod kommunikation). Om en GPU behöver dela med något av resultaten den beräknat mellan en annan GPU, så måste datan kopieras från GPU:ns minne till värddatorns RAM minne. Datan måste sen antingen kopieras till en annan GPU som befinner sig på samma nod, eller överföras via nätverket till en annan nods RAM minne, och sedan kopieras till en GPU på den noden. All intra- och internod kommunikationsöverföringar mellan kärnor hanteras enkelt med MPI. Överföring av data mellan värddatorer och GPU enheter görs med CUDA. Att ha en



Figur 5.2. Visar kommunikationen som krävs för att dela data med angränsande grafikprocessorer

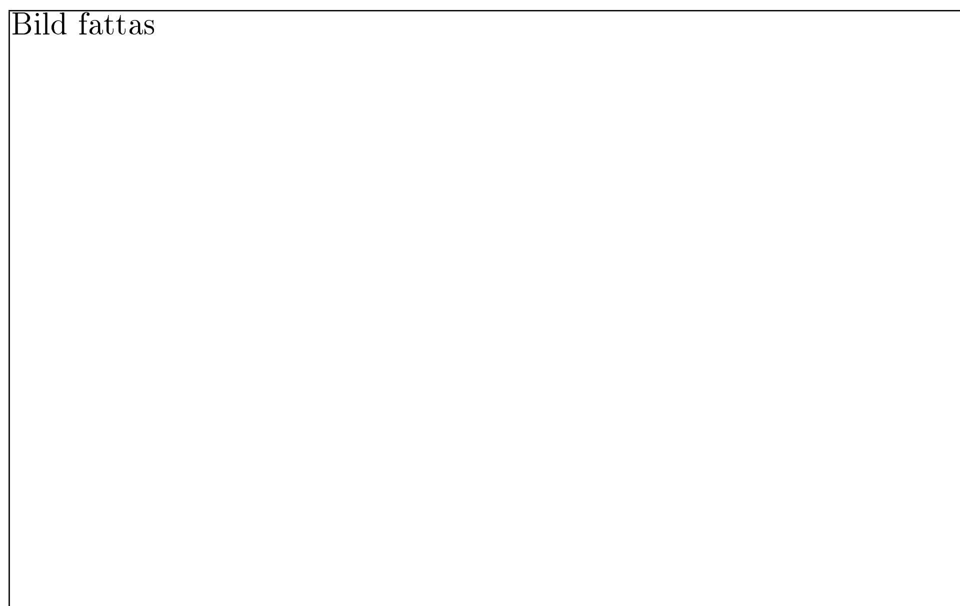
enda kärna att hantera mer än en GPU introducerar många utmanande problem som till exempel skedulering och belastningsbalansering, så detta vill vi naturligtvis undvika.

5.3.2 En första lösning

En enkel men ineffektiv lösning visas i Pseudokod 7. Koden visar instruktioner för huvud och slavprocesserna för att beräkna ett partiellt resultat.

5.3.3 Överlappning av send/receive med kopiering av data

Den första versionen av lösningen gör sitt jobb, om än dåligt. Om vi inte var tvugna att vänta på att `MPI_receive` funktionerna körs färdigt, så kunde vi under tiden till exempel kopiera data mellan GPU:n och värden. De ickeblockerande MPI funktionerna `MPI_Isend` och `MPI_Irecv` låter oss göra detta genom att genast returnera och låta påföljande kod exekvera. Det är dock inte tillräckligt att helt enkelt byta ut de blockerande MPI funktionerna mot icke-



Figur 5.3. En översikt över kommunikation mellan grafikprocessorer i ett multi-GPU kluster

blockerande, eftersom man hamnar att noga överväga exekveringsordningsföljden hos funktionerna. Pseudokod 8 visar en förbättrad lösning som överlappar send/receive funktioner med datakopiering.

`MPI_Wait` funktionen är en synkroniseringsfunktion som används för att blockera exekveringen av programmet tills båda rand data gränserna har tagits emot. När de har tagits emot återupptas exekveringen igen och båda delarna av datan kopieras till GPU:n för att användas i beräkningen.

5.3.4 Överlappning av beräkningar med MPI kommunikation

Det finns ännu rum för förbättring. I denna version väntar programmet tills hela resultatet är uträknat, och därför skickas rand datan och tas emot onödigt sent. Vi kunde dela upp kerneln i tre delar, varav två delar beräknar norr respektive söder gränsdata, och den tredje delen beräknar resten av datagittret, vilket är det mest beräkningsintensiva delen (se Figur 5.2). Vi kunde då starta loopen med att beräkna gränsdata, sedan skicka och ta emot gränsdatan och slutligen beräkna den mittersta delen av datagittret. Detta skulle betyda att vi överlappar beräkning med MPI kommunikation. Pseudokod 9 visar en ytterligare förbättrad version som gör detta.

Den viktigaste kodraden här är det sista `MPI_Wait` anropet som följer beräkningen av gittrets mittersta del. Detta gör så att vi beräknar den mittersta

Pseudokod 7 En första version för delande av data mellan grafikprocessorer

```
/* Kopiera den nyligen uträknade norra gränsdatan och skicka den till den
norra GPU:n */
cudaMemcpy(norr gränsdata från GPU till värd)
MPI_send(norr gränsdata till norra GPU:n)
/* Samma som ovan men med söder gränsdata */
cudaMemcpy(söder gränsdata från GPU till värd)
MPI_send(söder gränsdata till södra GPU:n)

/* Ta emot rand data från norr och kopiera det till GPU:n */
MPI_recv(rand data från norra GPU:n)
cudaMemcpy(norr gränsdata från värd till GPU)
/* Samma som ovan men med söder data */
MPI_recv(rand data från södra GPU:n)
cudaMemcpy(söder gränsdata från värd till GPU)

/* Beräkna resultat och repetera processen */
compute_result()
```

Pseudokod 8 En förbättrad version som överlappar send/receive funktionsanrop med kopiering av data för att åstadkomma delande av data mellan grafikprocessorer

```
MPI_Irecv(rand data från norra GPU:n)
cudaMemcpy(söder gränsdata från GPU till värd)
MPI_Isend(söder gränsdata till södra GPU:n)

MPI_Irecv(rand data från södra GPU:n)
cudaMemcpy(norr gränsdata från GPU till värd)
MPI_Isend(norr gränsdata till norra GPU:n)

MPI_Wait(tills alla rand data är mottagna)
cudaMemcpy(norr gränsdata från värd till GPU)
cudaMemcpy(söder gränsdata från värd till GPU)

/* Beräkna resultat och repetera processen */
compute_result()
```

Pseudokod 9 En ytterligare förbättrad version som överlappar beräkningar med MPI kommunikation vid delande av data mellan grafikprocessorer

```
compute_result(norr gräns)
compute_result(söder gräns)

MPI_Irecv(rand data från norra GPU:n)
cudaMemcpy(söder gränsdata från GPU till värd)
MPI_Isend(söder gränsdata till södra GPU:n)

MPI_Irecv(rand data från södra GPU:n)
cudaMemcpy(norr gränsdata från GPU till värd)
MPI_Isend(norr gränsdata till norra GPU:n)

compute_result(mittendelen)

MPI_Wait(tills alla rand data är mottagna)
cudaMemcpy(norr gränsdata från värd till GPU)
cudaMemcpy(söder gränsdata från värd till GPU)
```

delen av gittret medan vi väntar på att ta emot rand data.

5.3.5 Slutlig lösning

Om vi på något sätt kunde överlappa kernelberäkning med CUDA minnesöverföringar så kunde vi förbättra vårt program ytterligare. Lyckligtvis så stöds detta med så kallade CUDA strömmar. De möjliggör asynkron kopiering med `cudaMemcpyAsync`, vilket betyder att det är möjligt att göra beräkningar medan man överför minne till och från GPU:n. Kom ihåg att denna funktion endast fungerar med grafikprocessorer som stödjer "compute capability 1.1öch högre. Pseudokod 10 visar en slutlig version av applikationen som använder CUDA strömmar.

Vi kan se från Pseudokod 10 att funktionen `compute_result` använder sig av ström 0, 1 och 2 vid beräkning av nord-, syd- respektive mittendelen av datagittret. Funktionen `cudaMemcpyAsync` specificerar även vilken ström som skall användas vid kopiering av minne från och till GPU:n. Som kan ses så används olika strömmar för båda funktionsanropen för minneskopiering samt för beräkningen av mittendelen i datagittret. Detta betyder att de kan alla exekvera asynkront, vilket möjliggör överlappning av beräkning och minnesöverföring. Funktionen `cudaThreadSynchronize` ser till så att beräkningen av nord- och sydgränserna har avslutats före det kopieras från GPU:n till värdminnet. `cudaStreamSynchronize` igen returnerar då all exekvering hos en given ström har avslutats.

Pseudokod 10 En slutlig version som med hjälp av "CUDA strömmar"överlappar beräkningar med CUDA minnesöverföringar vid delande av data mellan grafikprocessorer

```
/* Asynkron beräkning */
compute_result(norr gräns, ström 0)
compute_result(söder gräns, ström 1)
cudaThreadSynchronize()

cudaMemcpyAsync(söder gränsdata från GPU till värd, ström 0)
cudaMemcpyAsync(norr gränsdata från GPU till värd, ström 1)

/* Asynkron beräkning */
compute_result(mittendelen, ström 2)

MPI_Irecv(rand data från norra GPU:n)
cudaStreamSynchronize(ström 0)
MPI_Isend(söder gränsdata till södra GPU:n)

MPI_Irecv(rand data från södra GPU:n)
cudaStreamSynchronize(ström 1)
MPI_Isend(norr gränsdata till norra GPU:n)

MPI_Wait(tills södra rand data är mottagen)
cudaMemcpyAsync(söder rand data från värd till GPU, ström 0)
MPI_Wait(tills norra rand data är mottagen)
cudaMemcpyAsync(norr rand data från värd till GPU, ström 1)

cudaThreadSynchronize()
```

6 Slutsatser

Här kommer mina slutsatser

7 Litteraturförteckning

[1] Jack Dongarra, David Walker, Ewing Lusk, Bob Knighten, Marc Snir, William Gropp, Al Geist, Marc Snir, Steve Otto, Rolf Hempel, Ewing Lusk, James Cownie, Tony Skjellum, Lyndon Clarke, Richard Littlefield, Mark Sears, Steven Huss-Lederman. MPI: A Message-Passing Interface Standard Version 2.2. `http://www.mpi-forum.org/docs/mpi22-report/mpi22-report.htm`, April 2011.

[2] NVIDIA Corp. Nvidia c cuda programming guide. `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf`, November 2010.

[3] NVIDIA Corp. High Performance Computing - Supercomputing with Tesla GPUs. `http://www.nvidia.com/object/tesla_computing_solutions.html`, April 2011.

[4] NVIDIA Corp. Nvidia tesla - gpu computing technical brief. `http://www.nvidia.com/docs/IO/43395/tesla_technical_brief.pdf`, May 2007.

[5] Zhe Fan, Feng Qiu, Arie Kaufman, Suzanne Yoakum-Stover. GPU Cluster for High Performance Computing. *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, sc:pp.47, 2004.

[6] NVIDIA Corp. Tesla Workstation Solutions. `http://www.nvidia.com/object/personal-supercomputing.html`, April 2011.

[7] InfiniBand® Trade Association. IBTA. `http://www.infinibandta.org/`, April 2011.

[8] NVIDIA Corp. PCI Express. `http://www.nvidia.com/page/pci_express.html`, April 2011.

[9] Dana A. Jacobsen, Julien C. Thibault, Inanc Senocak. An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters. *??*, ??:??, 2010.