Ett parallelliserat verktyg för simulering av artificiella neurala nätverk.

(HS-IDA-EA-97-104)

Alexander Foborg (dv3alefo@ida.his.se)

Institutionen för datavetenskap
Högskolan Skövde, Box 408
S-54128 Skövde, SWEDEN

Examensarbete på det datavetenskapliga programmet under vårterminen 1997.

Handledare: Mikael Bodén

Key words: Artificial neural networks, Parallel computing, Implementation, Source code

Abstract

Högskolan i Skövde has recently purchased a new number crunching machine, dedicated for simulation of Artificial Neural Networks (ANNs). There is a need for a new tool for simulation of these ANNs, that uses the new machine's parallel architecture. This Final Year Paper examines if such a tool would be sufficiently faster than the now available serialized simulator and in which conditions the parallelisation is most efficient.

A number of ways to parallelize a simulator for ANNs is presented and an epoch-based method is chosen for implementation. This implementation is then tested with a large number of ANNs, showing a 3-5 times speed-up compared to the serialized simulator.

1 Introduktion....6
1.1 Parallellisering....6
1.1.1 Hårdvarustöd för parallellisering....6
1.1.2 Synkroniseringsprimitiver....7
1.1.3 Prestandaökning i parallelliserade system....8
1.1.4 Trådstöd i Solaris 2.5....9
1.2 Neurala nätverk....10
1.2.1 Backpropagation....11
1.2.2 Steepest descent....11
1.2.3 Motiveringar för epokbaserad inlärning....11
1.2.4 Parallell implementation av neuralt nätverk....12
2 Problembeskrivning....14
2.1 Mål....14
2.1.1 Typ av nätverk....14
2.1.2 Inlärningsalgoritmer....14
2.2 Målsystem....14
2.3 Kriterier för bedömning av lösning....14
2.3.1 Performance (Effektivitet)....14
2.4 Andra ambitioner....14
2.4.1 Usability (Användbarhet)....15
2.4.2 Flexibility (Flexibilitet)....15
3 Implementation....16
3.1 Parallelliserad implementation....16
3.2 Serialiserad implementation....18
3.3 Filformat för beskrivning av ANN....18
3.3.1 Nätverksbeskrivning....19
3.3.2 Exempel och testdata....19
3.4 Generering av körbar fil....20
4 Korrekthetstester....21
4.1 Introduktion....21
4.2 XOR....21
4.3 Nencode 8....21
5 Prestandatester....22
5.1 Genomförande av prestandatester....22
5.1.1 Antal epoker....23
5.1.2 Storleken på det neurala nätverket....23
5.1.3 Antal exempel....25
5.1.4 Antal trådar....26
6 Slutsatser....30
6.1 Utvärdering av mål och ambitioner....30
6.1.1 Performance (Effektivitet)....30
6.1.2 Usability (Användbarhet)....31
6.1.3 Flexibility (Flexibilitet)....31
6.2 Riktlinjer för användande av simulatorn....31
7 Fortsatt arbete....32
Appendix A: Källkod för parallell implementation

Appendix B: Källkod för seriell implementation

Appendix C: Skript för generering av körbart ANN

Appendix D: Simulering av XOR

Appendix E: Simulering av Nencode 8

Appendix F: Tider för testkörningar

Sammanfattning

Den forskningsgrupp som vid Högskolan i Skövde bedriver forskning om neurala nätverk, har nyligen köpt in en ny beräkningsmaskin, dedikerad för simuleringar av dessa neurala nätverk. Maskinen är en Sun Enterprise 4000, en MIMD-maskin med 6 st UltraSparc CPUer.

Det finns behov av nya verktyg, som underlättar utveckling av ANN modeller till denna maskin, och som dessutom utnyttjar maskinens parallellitet. Detta verktyg ska uppvisa så god prestanda som möjligt. I denna rapport studeras förutsättningarna för ett sådant verktyg.

Av de tänkbara metoder som kan användas för att parallellisera en nätverksimulator, är troligtvis epokbaserad parallellisering den som ger högst prestandaökning på en MIMD-maskin. Detta beror på att den inte kräver lika täta synkroniseringar som exempelvis en neuronbaserad parallellisering.

En implementation av den epokbaserade metoden har skett, både med och utan parallelliserad summering av viktförändringar. Den parallelliserade summeringen av viktförändringar behöver teoretiskt endast $\log_2(n)$ så lång tid som den serialiserade summeringen kräver (n = antal trådar). Summeringen av viktförändringar är dock en ganska liten del av varje epok, och effektivisering av denna bit har inte så stor praktiskt betydelse.

Ett mindre antal tester har gjorts för att kontrollera om simulatorn är korrekt implementerad. De tester som gjordes gav lyckat resultat och talar för att simulatorn är korrekt.

Ur prestandatesterna som genomförts kan följande slutsatser dras:

- Ökning av antal exempel ökar prestandaökningen
- Ökning av antal vikter minskar prestandaökningen
- Antal epoker påverkar inte prestandaökningen

På det målsystem med 6 CPUer som testerna genomförts på, gäller följande:

- 4-5 trådar ger oftast högst prestandaökning
- 3-5 gångers prestandaökning är normalt

Eftersom prestandan på många av nätverken ligger uppåt 75% (4.5 gånger prestandaökning med 6 st CPUer) av vad målmaskinen klarar av, anser författaren av denna rapport att parallelliseringen är lyckad.

1 Introduktion

Den forskningsgrupp som vid Högskolan i Skövde bedriver forskning om neurala nätverk, har nyligen köpt in en ny beräkningsmaskin, dedikerad för simuleringar av dessa neurala nätverk. De verktyg som nu finns tillgängliga för att simulera neurala nätverk utnyttjar inte den nya maskinens parallella arkitektur till fullo. Därför finns det behov av nya verktyg, som underlättar utveckling av ANN modeller till denna maskin, och som dessutom utnyttjar maskinens parallellitet.

I denna rapport studeras förutsättningarna för ett sådant verktyg.

1.1 Parallellisering

De första datorerna som utvecklades klarade bara att utföra en uppgift i taget. Efter ett tag insåg man dock värdet i att kunna köra flera olika program samtidigt, så man införde stöd för detta i operativsystemen. Detta kallas bl a multitrådning (multithreading), eller multitasking.

Denna möjlighet att köra flera olika uppgifter samtidigt, införde dock vissa problem. De olika uppgifterna som kördes på samma dator kunde på olika sätt förstöra för varandra, ofrivilligt eller avsiktligt. Med hjälp av diverse mekanismer såg man till att operativsystemet skyddade de olika programmen från varandra. Dessa olika enheter, som körs skyddade från varandra på samma hårdvara, benämns i stället för processer. Process-stöd ansåg länge överlägset trådstöd, och de moderna operativsystemen valde i allmänhet att stödja dessa, i stället för trådar.

Nackdelen med processer, jämfört med trådar, är dock att de mekanismer som används för att skydda processerna från varandra kräver en hel del datorkapacitet. Denna datorkapacitet, som bara är administrativ och inte hjälper det egentliga arbetet framåt, skulle i de fall när processkydd inte är så viktigt bättre utnyttjas om det användes till processens egentliga uppgift. Ett programs interna parallella delar har i allmänhet inte något behov av att bli skyddade från varandra, och att tillgodose dem med detta skydd skulle vara rent slöseri med processorkraft.

Utvecklarna av operativsystem löste detta genom att stödja *både* trådar och processer. Detta sker i allmänhet som så, att en tungrodd process kan innehålla flera effektiva trådar. Detta är en mycket lyckad lösning, som i de flesta fall ger tillgång till det bästa från båda koncepten. Program som har höga effektivitetskrav kan använda trådar för sin interna parallellitet, samtidigt som de är skyddade från andra program genom att programmet i sig består av en enda process. Ett exempel på operativsystem som stöder detta är Solaris 2.

Mer information om stöd för parallellisering i operativsystem finns i [SG94].

1.1.1 Hårdvarustöd för parallellisering

De flesta av dagens persondatorer har bara en enda central processor (CPU). När det gäller lite större system, har det dock under en tid varit populärt att ha flera processorer i ett system. En av fördelarna med ett flerprocessorsystem är att det är billigare att använda sig av flera simpla CPUer än att använda en mer avancerad och dyr CPU. Det är också möjligt att

bygga ett system med flera av den just då bästa CPUn för att göra en dator med högre prestanda än annars skulle vara möjligt vid denna tid.

Flynn har identifierat följande typer av parallellitet i datorsystem:

SISD: Single instruction, single data. Detta är den vanliga typen av datorsystem, med endast en CPU.

SIMD: Single instruction, multiple data. Denna typ av system används ofta för simuleringar där stora mängder av likformig data hanteras. SIMD-system är exempelvis mycket effektiva på att göra matrismultiplikationer. Ett exempel på när detta används är vädersimuleringar, som ofta körs på SIMD-system.

MISD: Multiple instruction, single data.

MIMD: Multiple instruction, multiple data. Dessa system innehåller flera fullvärdiga CPUer, som klarar av att arbeta oberoende av varandra. Detta kan t ex vara flera enskilda datorer som samarbetar för att slutföra en gemensam uppgift, eller en enskild dator som innehåller flera processorer.

MIMD är den arkitektur som är relevant för denna rapport, eftersom målsystemet (beskrivet i ”2.2 Målsystem”) är en MIMD-maskin. Det finns två olika sätt att realisera en MIMD-arkitektur:

Tightly coupled: I denna arkitektur har alla CPUer ett gemensamt adressutrymme. Eftersom alla processer har tillgång till samma data, behövs ingen explicit kommunikationsmetod.

Loosely coupled: I dessa system har varje CPU ett eget, oberoende adressutrymme, och processer som behöver utbyta data med varandra får göra det med någon form av explicit kommunikation.

Målsystemet för detta projekt är tightly coupled, se ”2.2 Målsystem”.

1.1.2 Synkroniseringsprimitiver

Det finns ett par olika metoder som kan användas för att låta programmets trådar utbyta information med varandra på ett kontrollerat sätt:

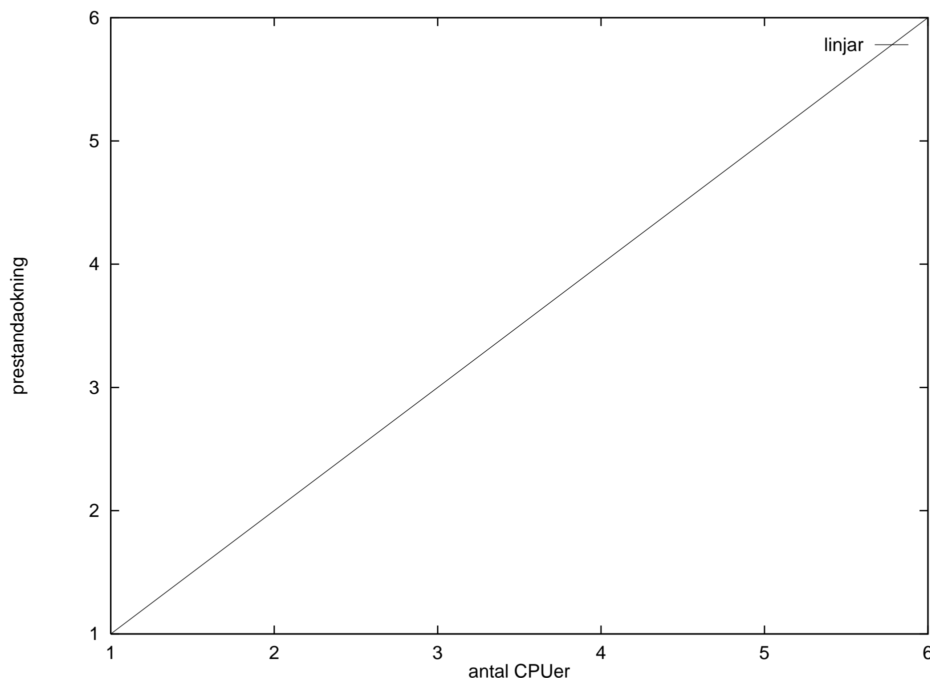
delat minne: Primitiver som semaforer och monitorer används för att synkronisera och hantera åtkomst till gemensam data. Denna metod går bara att använda på arkitekturer som är tightly coupled.

message-passing: Primitiver för att uttryckligen begära kommunikation och synkronisering med hjälp av meddelanden används. Denna typ av interprocesskommunikation går att implementera både på system som är loosely coupled och tightly coupled.

Vilken av dessa två metoder som är bäst beror på omständigheterna, men messagepassing anses av många minska risken för fel i programmet, medan metoden med delat minne möjliggör effektivare implementationer.

1.1.3 Prestandaökning i parallelliserade system

När nya CPUer sätts i en maskin skulle denna maskin kunna förväntas bli exakt så mycket snabbare som de nya CPUerna klarar av att beräkna (se Figur 1). En maskin med 3 st CPUer skulle alltså bli tre gånger så effektiv som ett system med bara en CPU av samma typ. Så är dock sällan fallet. Eftersom de uppgifter som datorn ska utföra sällan är möjliga att helt parallellisera så måste datorns olika CPUer synkronisera med varandra. Dessa synkroniseringsaktiviteter använder dels CPU-kraft för själva synkroniseringen, dessutom gör de så att alla CPUer inte kan arbeta i full fart hela tiden, då de CPUer som blir klara först måste vänta på de som ännu inte är det.



Figur 1: Linjär prestanda i ett multi-CPU system.

För att få ett program så effektivt parallelliserat som möjligt, måste därför antalet synkroniseringar hållas nere till ett minimum. Programmet måste designas så att dess olika processer/trådar jobbar så lite som möjligt med gemensam data, och så mycket som möjligt med egen data som de kan nå utan att behöva synkronisera med andra processer.

I [AC95] presenteras benchmark-tester för Suns olika multi-CPU maskiner. De flesta system visar linjär prestandaökning upp till 4 CPUer, men det finns system (exempelvis SparcCenter 2000) som i benchmarktesterna visar linjär prestandaökning ända upp till 12 CPUer. Linjär prestandaökning upp till 4 CPUer anses populärt vara vad man kan räkna med i ett normalt system. Processorer utöver detta tillför sällan särskilt mycket CPU-kraft i praktiken. I vissa fall blir systemet till och med mindre effektivt när en extra CPU sätts i.

En intressant effekt som kan fås i ett parallelliserat program, är att programmet efter parallellisering blir mer än linjärt effektivare [AC95]. Ett program som trådas upp i två trådar kan alltså bli mer än dubbelt så effektivt. Detta sker i de fall där programmets inre loopar,

efter parallellisering, helt får plats sin CPUs cache. Eftersom CPUn i detta fall inte behöver vänta på minnesaccesser lika länge som i den serialiserade versionen av programmet, kan den arbeta så mycket effektivare att ovanstående effekt kan uppstå. Detta tillhör dock inte normalfallet. Normalt sett får parallelliserade program mindre än linjär prestandaökning.

Ett program bör inte delas upp i mer processer än det finns CPUer i maskinen som programmets ska köras på. Så länge antalet körande processer är mindre än antalet lediga processorer i maskinen, slipper operativsystemet avbryta processer och lägga dem i viloläge för att en annan process ska kunna köra (sk contextswitch). Contextswitchar är kostar mycket i CPU-kraft för processer och även en del för trådar. Om contextswitchar kan undvikas genom att minska antalet trådar i ett program, så kan den insparade CPU-kraften i stället användas för att låta programmet utföra sin uppgift.

Prestandatester

Det är svårt att jämföra olika algoritmers effektivitet mot varandra. Det mest uppenbara sättet att kunna jämföra dem, är att provköra de olika algoritmerna med samma testdata och mäta hur lång tid de behöver för att slutföra uppgiften. I detta fall måste dock hänsyn tas till de externa omständigheter som påverkar tidsåtgången, men som ligger utanför algoritmens kontroll. Exempel på detta skulle kunna vara operativsystemets egna aktiviteter, och andra konkurerande processers CPU-utnyttjande. Dessa händelser är slumpartade, och går inte att förutsäga i förväg. Med tanke på detta, är det lämpligaste sättet att mäta en algoritms reella tidsförbrukning troligtvis att provköra den på en helt ledig maskin, flera gånger, och sedan använda den genomsnittsliga tiden för att jämföra med andra algoritmer. Dock är det inte ens på detta sätt möjligt att få ett exakt jämförsvärde.

1.1.4 Trådstöd i Solaris 2.5

Solaris 2.5 levereras med två olika trådbibliotek; ett för Solaris egna API (Application Programming Interface) och ett för POSIX API. Dessa två APIer erbjuder i stort sett samma funktionalitet, med ganska små skillnader. De är fullständigt kompatibla med varandra, och det går att blanda anrop till de båda APIerna i ett program.

Tyvärr är inte POSIX API helt färdigimplementerat för Solaris 2.5.1 ännu. Bl a saknas fungerande POSIX-semaforer. Detta löses dock enkelt genom att använda Solaris semaforer i stället. På grund av ovan nämnda kompatibilitet mellan de två funktionsbiblioteken är detta fullt möjligt.

En stor fördel för POSIX APIt, är att det är ett standardiserat API som finns tillgängligt på de flesta Unix-plattformar. Detta möjliggör portning till dessa plattformar utan att förändring av programmets källkod behöver ske.

För båda trådbiblioteken gäller att endast en pekare kan skickas till en tråd som startas. Vill man skicka många variabler till tråden går det bra att skicka en pekare till en sammansatt variabel (i c kallad struct), som initierats med lämplig data.

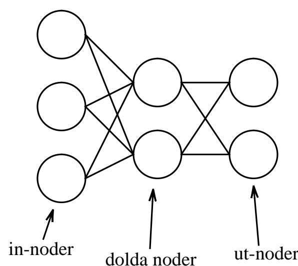
1.2 Neurala nätverk

Artificiella Neurala Nätverk (ANN) är en metod, framtagen med inspiration från den mänskliga hjärnans uppbyggnad, för att approximera matematiska funktioner.

Genom att träna ett neuralt nätverk med hjälp av exempeldata, kan det lära sig att känna igen trender och mönster i exempeldatan. För varje exempel det får, beräknar nätverket sitt eget svar och kontrollerar hur mycket fel det hade, jämfört med facit. Det försöker sedan korrigera sitt beteende för att få sin beräkning att komma närmare det riktiga svaret. Den vanligaste inlärningsmetoden kallas backpropagation och beskrivs närmare nedan.

Eftersom det neurala nätverket inte lär sig exempeldatan utantill, klarar det i allmänhet av att ge vettiga svar på indata som liknar den data det har tränats på, men som inte ingår i exempelmängden. Neurala nätverk har därför blivit populära att användas till problem där det gäller att förutsäga trender och känna igen olika typer av mönster, så som att tolka text och förutsäga börskursförändringar.

Den vanligaste typen av nätverk kallas multi-layer feedforward perceptron neural network och brukar representeras enligt Figur 2.



Figur 2: Förenklad representation av neuralt nätverk

Indata presenteras vid in-noderna, och flödar via länkarna till de dolda noderna. Varje länk har en vikt som påverkar datan som passerar dem. De dolda noderna summerar den förändrade datan som kommer på länkarna och skickar den vidare via länkarna (också dessa med vikter som förändrar datan) till ut-noderna. Ut-noderna summerar datan från varje länk, och presenterar dem som nätverkets resultat.

Nätverkets olika noder och länkar arbetar oberoende och parallellt med varandra. Varje nod och länk kan ses som en egen process som arbetar med sin egen del av nätverket.

För att det neurala nätverket ska ge rätt svar måste dess vikter vara inställda så att de påverkar datan så att rätt svar presenteras av ut-noderna. Det är alltså vikterna som innehåller nätverkets kunskap.

Detta är en mycket förenklad beskrivning på hur ett neuralt nätverk används, och ger ingen teoretisk förklaring till varför det fungerar bra för approximation av funktioner. För denna typ av information rekommenderas [MS93].

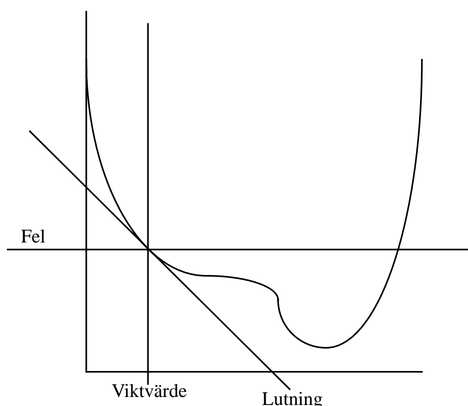
1.2.1 Backpropagation

Att träna ett neuralt nätverk består i att korrigera nätverkets vikter så att nätverket beräknar önskad utdata från den indata som den får som inlärningdata. Den populäraste metoden för att träna ett nätverk kallas backpropagation [RH86].

För varje exempel som presenteras för nätverket, beräknas den utdata som nätverket anser hör ihop med presenterad indata. Detta är det "framåtgående passet". Den beräknade utdatan jämförs sedan med den önskade utdatan. Felet som beräknats för varje utnod, propageras sedan tillbaka via länkarna till de dolda noderna. Felet för varje dold nod blir därför också känd. Vikten för en länk som går till en nod kan nu förändras, med utgångspunkt från nodens felvärde, för att minimera felvärdet. Detta sker med hjälp av en inlärningsalgoritm. Den vanligaste algoritmen för ANN är Steepest Descent, som beskrivs nedan.

1.2.2 Steepest descent

Efter att felet på den nod som en länk är kopplad till är känt, kan länkens vikt förändras så att felet minskas. Steepest Descent-metoden försöker göra detta genom att räkna ut felkurvans derivat (lutning) för nuvarande viktvärde och sedan förändra viktens värde beroende på åt vilket håll kurvan lutar åt.



Figur 3: Felkurva för en vikt

Hur mycket viktens värde ska förändras avgörs inte bara av derivatans värde, utan också av inlärningsfaktorn, som multipliceras med derivatan. Inlärningsfaktorn väljs manuellt. Ju högre inlärningsfaktorn är, desto större förändringar av vikten kommer att ske.

1.2.3 Motiveringar för epokbaserad inlärning

Vid användning av Backpropagation och Steepest Descent, beräknas normalt felen för hela exempelmängden, innan viktkorrigeringen sker (s k epokbaserad inlärning). En alternativ metod skulle kunna vara att vikterna i stället korrigeras efter varje exempel. Denna, exempelbaserade, inlärningsmetod har i allmänhet effekten att felkurvan lutar brantare i de första epokerna, jämfört med epokbaserad inlärning.

I [MS93] ges dock tre motiveringar till varför epokbaserad inlärning bör användas framför exempelbaserad:

- Det är inte säkert att exempelbaserad inlärning verkligen minskar antalet epoker som behöver köras för att nätverket skall hitta de optimala vikterna. I de flesta fallen konvergerar de två metodernas felkurvor innan inlärningen avslutas. Trots att den exempelbaserade inlärningens felkurva lutar brantare till att börja med, planar den i allmänhet ut med tiden, så att den epokbaserade inlärningens felkurva kommer i kapp innan de optimala vikterna har hittats.
- Trots att exempelbaserad inlärning kan kräva mindre antal epoker för att tränas, är det inte säkert att det krävs mindre reell tid. Exempelbaserad inlärning kräver nämligen mer beräkningar, eftersom vikterna måste beräknas efter varje exempel, till skillnad mot den epokbaserade som endast beräknar vikterna efter varje epok.

  Epokbaserad inlärning är också effektivare att parallellisera, eftersom exempel enkelt kan portioneras ut till flera beräknande enheter. Sammanfattning av de olika enheternas arbete (och därmed synkronisering av enheterna) behöver endast ske i slutet på varje epok, till skillnad mot efter varje exempel, som fallet skulle vara med exempelbaserad inlärning.
- De exempel som körs sist, får större betydelse i exempelbaserad inlärning, eftersom de får sista chansen att påverka vikterna. Därför har ordningen på exemplen betydelse i denna typ av inlärning. Exemplen får alltså inte presenteras för nätverket sorterade efter typ, eller liknande. [MS93] anser till och med att exemplens ordning bör vara olika för varje epok, om exempelbaserad inlärning används. Epokbaserad inlärning undviker dessa problem, eftersom den tar hänsyn till alla exempel när de nya vikterna beräknas.

Dessa problem går att lösa på olika sätt, men detta ökar komplexiteten på lösningen, och [MS93] avslutar därför med att rekommendera att epokbaserad inlärning används.

1.2.4 Parallell implementation av neuralt nätverk

Det finns minst två huvudsakliga metoder för att göra en parallell implementation av ett neuralt nätverk. Dels kan man utnyttja den parallellitet som finns i själva det neurala nätverket, dvs att alla neuroner jobbar parallellt med varandra. Detta är en parallellisering på neuronnivå. Den andra möjligheten är att utnyttja den parallellitet som finns i inlärningsalgoritmerna som används för att träna nätverket, dvs att inlärningen parallelliseras.

Dessa två metoder utesluter inte varandra, men det finns en rad olika skäl för att parallelliteten i inlärningsalgoritmerna går att utnyttja bättre än det neurala nätverkets interna parallellitet:

- Det oberoende arbete som en neuron kan genomföra innan den måste synkroniseras med de andra neuronerna är ganska kort, i förhållande till hur mycket synkroniseringsarbete som måste genomföras. Om implementationen parallelliseras på neuronnivå kommer därför tillgänglig processorkraft att utnyttjas ineffektivt, då mycket kraft går till administrativt arbete.

- Om man parallelliserar inlärningen kan däremot en mycket stor mängd beräkningar ske innan synkronisering måste ske. Teoretiskt skulle varje exempel kunna köras på en egen processor, men i praktiken är det troligen lämpligare att köra ett antal exempel på varje CPU för att minska andelen administrativt arbete jämfört med produktivt arbete. Om fler processorer blandas in behövs det också mer synkroniseringsarbete.

  Synkroniseringen sker först på slutet av en epok, efter att alla exempel har körts. Detta innebär att det mesta av processorns kraft går till produktivt arbete och att den CPU-kraft som slösas bort på synkronisering är liten.

- Inlärningsfasen är den fas som är mest CPU-intensiv. När nätet väl är färdigtränat och det neurala nätverket endast körs i framåtgående läge, är det ofta inte längre fråga om samma mängd av körningar som sker. Nätverket hinner då med att utföra sin uppgift, trots att det inte är parallelliserat.

2 Problembeskrivning

2.1 Mål

Forskningsgruppen önskar ett paket som underlättar utvecklande av ANN modeller i C++. Detta paket bör innehålla funktioner för att läsa in inlärningsdata från fil och funktioner som underlättar analysering av hur nätverket beter sig. Paketet bör också innehålla en eller flera inlärningsalgoritmer som justerar vikterna i nätverket.

2.1.1 Typ av nätverk

Paketet ska hantera Multiple Layer Feedforward Networks (se ”1.2 Neurala nätverk”). Denna typ av nätverk har i de flesta fall endast ett dolt lager, och detta fall kommer att prioriteras vid implementation. Det är dock önskvärt att paketet byggs så att det i framtiden går att förändra så att det klarar godtyckligt antal dolda lager, eller att det redan i första versionen hanterar det.

2.1.2 Inlärningsalgoritmer

Backpropagation är den mest använda inlärningsalgoritmen för ANN. Denna algoritm kommer därför att prioriteras i detta projekt. Beskrivningar av parallelliserade implementationer av backpropagation finns tillgängliga för allmänheten, och en av dessa skulle kunna användas.

2.2 Målsystem

Målsystemet för paketet är en Sun Enterprise 4000. Den är ett tightly coupled MIMD-system som just nu har 6 st UltraSparc CPUer. Maskinen är för tillfället utrustad med 384 Mb minne, 4.2 Gb hårddiskutrymme och kör Solaris 2.5.1.

2.3 Kriterier för bedömning av lösning

Följande kriterier kommer att tas hänsyn till vid utveckling av systemet, samt användas för att bedömma kvaliten på det implementerade systemet.

2.3.1 Performance (Effektivitet)

Beställarna önskar en lösning som utnyttjar målsystemet så effektivt som möjligt. Att få ett snabbt system är det primära önskemålet.

Den implementerade parallella inlärningsalgoritmen kommer att jämföras med den snabbaste serialiserade algoritmen som finns tillgänglig på målsystemet. Dessa tester kommer att ske enligt beskrivning i ”1.1.3 Prestandaökning i parallelliserade system”.

2.4 Andra ambitioner

Dessa ambitioner kommer att tas hänsyn till vid utveckling av systemet, men kommer inte att prioriteras. Att få en hög effektivitet är viktigare än nedanstående ambitioner.

2.4.1 Usability (Användbarhet)

Det ska med det utvecklade paketet gå snabbt och enkelt att göra en prototypimplementation av ett standard ANN. Användaren ska i stort sett bara behöva ange hur nätverket ska se ut (antal in, ut och dolda noder), samt vilken av de tillgängliga inlärningsalgoritmerna som ska användas, för att ett körbart ANN ska kunna generas.

2.4.2 Flexibility (Flexibilitet)

Lösningen ska i möjligaste mån vara anpassningsbar till framtida krav. Det bör framför allt vara enkelt att lägga till nya inlärningsalgoritmer till systemet.

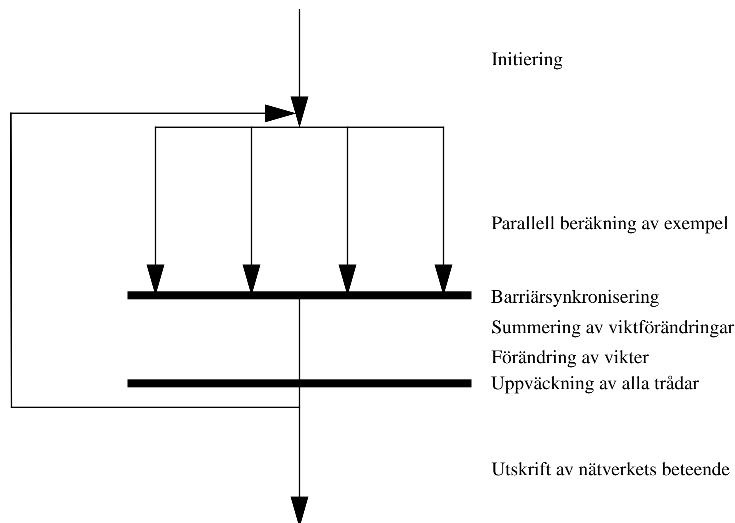
Detta kriterie är det minst viktiga kriteriet. Eftersom effektivitet prioriteras kommer paketet att byggas så att det är så effektivt som möjligt med de nuvarande kraven implementerade, även om det minskar möjligheten för framtida förändringar av paketet.

3 Implementation

3.1 Parallelliserad implementation

Med stöd av motiveringen som ges i ”1.2.4 Parallell implementation av neuralt nätverk” har en epokbaserad parallellisering av simulatorn gjorts. Denna beskrivs närmare nedan.

Eftersom POSIX-trådar och Solaris-trådar är så gott som likvärdiga (se ”1.1.4 Trådstöd i Solaris 2.5”) har POSIX-trådar använts för att underlätta eventuell framtida portning av simulatorn. Solaris semaforer har dock använts, eftersom Solaris 2.5.1 inte stöder POSIX-semaforer.



Figur 4: Epokbaserad parallelliserad nätverksimulator

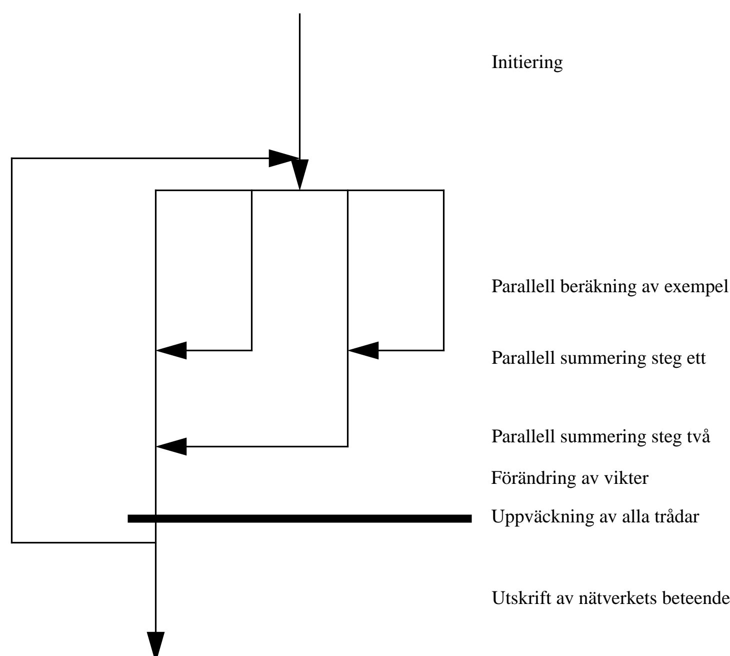
I Figur 4 beskrivs hur den implementerade simulatorn arbetar. I initieringen av nätverket slumpas värden fram för alla vikter och uppdelning av inlärningsexemplen till programmets trådar sker. Efter att slav-trådarna startats upp och skickats till inlärningsfunktionen, anropar mastertråden själv inlärningsfunktionen där programmets yttre loop finns. Denna loop avslutas först när alla epoker genomförts.

För varje epok körs först den parallelliserade beräkningen av alla exempel. Varje tråd har egna exempel att beräkna. Denna beräkning kan ske utan inblandning från de andra trådarna. Efter exempelberäkningen sker en barriärsynkronisering, dvs alla trådar väntar på att alla andra trådar blir klara med att beräkna sina exempel. Efter denna barriärsynkronisering lägger sig alla slavtrådar i viloläge och väntar på att mastertråden ska summera alla viktförändringar som beräknats och uppdatera vikterna enligt dessa förändringsvärden. När mastertråden är klar med detta, signalerar den slavtrådarna att de kan starta på nästa epok. Mastertråden påbörjar sedan själv nästa epok.

Efter att alla epoker är genomförda avslutas slavtrådarna. Mastertråden lämnar inlärningsfunktionen och går över till testläget där nätverkets beteende testas och utskrift av resultat sker.

Det är alltså endast exempelberäkningen som är parallelliserad. Summering av viktförändringar och uppdatering av vikter är svårare att parallellisera, eftersom under dessa steg påverkas variabler som är globala för alla trådar. En metod för att parallellisera summeringen av viktförändringarna har dock implementerats, inspirerad av [LC96]. Denna metod beskrivs närmare nedan. Se även Figur 5.

Den parallelliserade summeringen sker stegvis. Eftersom trådarna summerar ihop sin data i par, kommer det krävas $\log_2(n)$ (avrundat uppåt) steg för att summera n trådars data. I optimala fall (när antal trådar är 2, 4, 8, osv) uppnås alltså teoretiskt $\log_2(n)$-prestanda vid summering av trådarna, jämfört med seriell summering. Metoden har därför döpts till logfetch i denna rapport.



Figur 5: Parallelliserad nätverksimulator, parallelliserad summering av viktförändringar

Logfetch-summeringen ersätter den barriär-synkronisering som sker efter exempelberäkningen i den första versionen av simulatorn. Trådarna arbetar stegvis i par (se Figur 5 för exempel med 4 trådar), där en tråd blir mastertråd och den andra slavtråd. Den tråd med högst nummer i paret blir slav.

I steg nummer s, ska tråd nummer n prata med en tråd på 2^{s-1} stegs avstånd från tråden. Om n modulo 2^s är lika med 0 blir tråden master och ska prata med tråd $n+2^{s-1}$ (om denna tråd finns), i annat fall blir den slav och ska prata med $n-2^{s-1}$. När tråden har varit slav en gång, övergår den till att vänta på signal från tråd nummer 0 (som blir sist kvar) att epoken är klar och att nästa epok kan påbörjas.

Tråd nummer 0 blir aldrig slav. Den avslutas först när 2^{s-1} är större än antalet trådar. Då övergår tråden till att uppdatera alla vikter med de nu summerade viktförändringarna.

Förkortad källkod för logfetch följer nedan. Se ”Appendix A: Källkod för parallell implementation” för komplett källkod.

```
for(delta = 1; thread_no%delta == 0 &&
               delta < NOOFTHREADS; delta *= 2) {

  if(thread_no % (delta * 2) == 0) {
    if(thread_no + delta < NOOFTHREADS) {
      pratamed = thread_no + delta;
      /* barrier med pratamed */

      /* summera första halvan av datan med dennas array

      /* barrier med pratamed */
    }
  } else {
    pratamed = thread_no - delta;
    /* barrier med pratamed */

    /* summera andra halvan av datan med pratameds array

    /* barrier med pratamed */
  }
}
```

Figur 6: Förkortad källkod för logfetch

I källkoden är delta = 2^{s-1} och thread_no = n.

3.2 Serialiserad implementation

För att kunna kontrollera hur effektiv den parallelliserade nätverksimulatorn är, har även en serialiserad nätverksimulator implementerats till detta projekt (se ”Appendix B: Källkod för seriell implementation”). Denna är tänkt att användas som jämförelseimplementation. Merparten av den serialiserade simulatorns kod är identisk med den parallelliserade simulatorns implementation. Dock har de parallelliserade delarna skrivits om till seriellt beteende. Detta har gjort att vissa variabler har kunnat göras globala och detta har skett av effektivitetsskäl.

Den serialiserade nätverkssimulatorn använder samma indatafiler för beskrivning av nätverkslayout som den parallelliserade versionen.

3.3 Filformat för beskrivning av ANN

Kompilatorn klarar av att optimera ett program bättre om så många som möjligt av programmets variabler är kända redan vid kompileringen. Därför har header-filer med #define-satser använts som filformat för att specificera utseendet på nätverken med hjälp av konstanter. Detta för att kompilatorn ska känna till nätverkets utseende och antal epoker redan när kompiliering sker.

Beskrivningsfilerna för nätverken består av två avdelningar; första delen är en beskrivning av nätverket. Därefter följer exempeldatan som nätverket ska tränas på. Denna information läggs i en c-headerfil enligt Figur 7.

Denna information skulle kunna ha lagts i två olika filer, men under arbetet på detta projekt uppdagades det att det är bekvämare att ha all data om ett nätverk i en och samma fil.

```
#ifdef NETHEAD
/* Beskrivning av nätverk */
#else
/* Exempeldata och testdata */
#endif
```

Figur 7: Grundläggande struktur för net-fil

För kompletta exempel på net-filer, se ”Appendix D: Simulering av XOR” och ”Appendix E: Simulering av Nencode 8”.

3.3.1 Nätverksbeskrivning

Nätverksdelen av net-filen beskriver för simulatorn hur nätverkets layout ser ut och diverse annan information. Denna är:

- antalet trådar som ska användas under inträningsfasen (minst 2)

  #define NOOFTHREADS 2
- om parallelliserad summering (logfetch) ska användas. I så fall definieras LOGFETCH. Om LOGFETCH är odefinierad används normal summering.

  #define LOGFETCH
- hur många epoker nätverket ska tränas

  #define NOOFEPOKS 10000
- inlärningsfaktorn (större än 0)

  #define ETA 0.2
- antalet in-noder

  #define INPUT 2
- antalet dolda noder

  #define HIDDEN 2
- antalet ut-noder

  #define OUTPUT 1

3.3.2 Exempel och testdata

Förutom exempeldatan, dvs den data som nätverket ska tränas på under inlärningsfasen, kan i denna avdelning också två olika typer av testdata anges.

Inträningsdata anges enligt Figur 8.

```
#define NOOFEXAMPLES 4
Example examples[NOOFEXAMPLES] = {
  { { 0.0, 0.0 }, { 0.0 } },
  { { 0.0, 1.0 }, { 1.0 } },
  { { 1.0, 0.0 }, { 1.0 } },
  { { 1.0, 1.0 }, { 0.0 } }
};
```

Figur 8: Inträningsdata för nätverk

Den första typen av testdata är den data där svaret redan är känt och skillnaden mellan nätverkets uträkning och rätt svar ska presenteras. Kontroll mot denna data och presentation sker efter att träningsfasen slutförts.

```
#define NOOFCHECKS 1
Example checks[NOOFCHECKS] = {
  { { 0.0, 0.0, 0.0, 1.0 }, { 0.0, 0.0, 0.0, 1.0 } }
};
```

Figur 9: Testdata som svaret är känt för

För den testdata som rätt svar inte är känt för, kan inte felet beräknas. För denna testdata presenteras därför nätverkets resultat i stället. Kontroll mot denna data och presentation sker efter att träningsfasen slutförts.

```
#define NOOFTESTS 1
Example tests[NOOFTESTS] = {
   { 0.0, 1.0, 0.0, 0.0 }
};
```

Figur 10: Testdata utan känt svar

3.4 Generering av körbar fil

Generering av körbar fil från källkod kan ske med valfri ANSI-kompatibel C eller C++ kompilator. Länkning sker med matematik-bibliotek och bibliotek med POSIX-funktioner. Förslagsvis används också maximal optimering för bästa prestanda.

Vid kompilering måste NETFILE definieras till filnamnet på den net-fil som beskriver det nätverk som körbar fil ska genereras för. På de flesta kompilatorer används optionen

```
-DNETFILE=”filnamn”
```

för att göra denna definiering.

För ett sh-script som underlättar kompileringsprocessen, se ”Appendix C: Skript för generering av körbart ANN”.

Under detta projekt har lyckad kompilering skett med Suns cc och CC, samt Gnus gcc och g++.

4 Korrekthetstester

4.1 Introduktion

Det är inte helt elementärt att kontrollera om en implementation för simulering av neurala nätverk är korrekt. Detta beror på att simuleringar av samma exempelmängd flera gånger inte ger samma resultat. I vissa fall kan nätverket misslyckats att lära sig mönstret i exemplen på det antal epoker som är normalt. Detta beror på att vikterna initierats till olämpliga värden (dessa slumpas fram i initieringen av nätverket).

Med hänsyn tagen till ovanstående potentiella problem, har simulering skett av två standardproblem, Nencode16 och XOR. Dessa tester beskrivs närmare nedan.

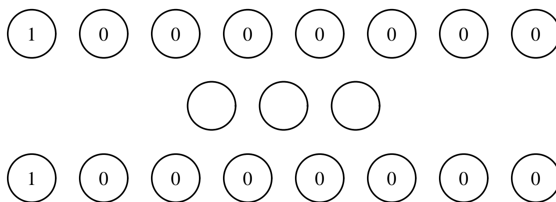
4.2 XOR

XOR är en mycket populär funktion att använda för introduktion till neurala nätverk. Detta beror på att den är en av de minsta funktionerna som kräver att det neurala nätverket har dolda noder för lyckad inträning. En genomgående beskrivning av hur neurala nätverk hanterar XOR finns i [MS93].

Ett 2-2-1 nätverk tränades i 10000 epoker på XOR (se ”Appendix D: Simulering av XOR” för inträningsdata och resultat). Detta nätverk klarade av att lära sig XOR med normal precision.

4.3 Nencode 8

Ett annat problem som brukar användas för test av simulatorers korrekthet är Nencode [AG85]. Nencode består av n antal dolda noder och 2^n st in- och ut-noder. För detta nätverk genereras sedan 2^n st exempel där endast en in- och en ut-nod är satt till aktiv (dvs satt till 1) för varje exempel. De övriga in- och ut-noderna sätts till noll (se Figur 11 där första in- och ut-noden är aktiv). I Nencode 8, som använts i detta fall, generas alltså 8 st exempel, där var och en av in-noderna är aktiv för varsitt exempel (se ”Appendix E: Simulering av Nencode 8”). En fungerande simulator ska lyckas träna ett nätverk på Nencode.



Figur 11: Exempel på träningsdata till Nencode 8

Exempel för Nencode 8 generades och nätverket tränades på denna exempeldata med lyckat resultat (se ”Appendix E: Simulering av Nencode 8” för inträningsdata och resultat).

5 Prestandatester

Som nämnts i ”1.1.3 Prestandaökning i parallelliserade system”, finns det många faktorer som kan påverka en parallell implementations effektivitet. En bra tumregel är, att ju mindre de olika trådarna måste synkronisera sitt arbete med andra trådar, desto effektivare kan de jobba. Detta beror naturligtvis på att tråden slipper vänta på de andra och kan i stället jobba på att slutföra sin uppgift.

Effektiviteten för den implementation för simulering av neurala nätverk som presenterats tidigare i denna rapport, påverkas främst av fyra olika faktorer:

- Antal epoker
- Storleken på det neurala nätverket (antal vikter)
- Antal exempel
- Antal trådar

Dessa faktorer kommer att presenteras i efterföljande avsnitt. En analys om hur dessa faktorer påverkar implementationens effektivitet kommer också att ges.

5.1 Genomförande av prestandatester

För att testa nätverksimulatorns prestanda har den provkörts med en mängd olika testfall. Dessa testfall har utformats så att de olika faktorer som nämns ovan, varierats i olika kombinationer.

Den parallelliserade simulatorn har provkörts på testfallen med 2-7 st trådar, på testmaskinen som beskrivs i ”2.2 Målsystem”. Varje testfall har provkörts en gång och tidtagning har skett med zsh-kommandot “time”. Tiderna för dessa tester har sedan jämförts med den serialiserade simulatorns tider. Prestandaökningen för den parallelliserade simulatorn jämfört med de serialiserade simulatorn har sedan beräknats enligt:

prestandaökning = serialiserad tid/parallelliserad tid

För att hålla nere simuleringstiden till rimliga tider, har antalet epoker minskats när simulering av de större nätverken skett. Avdelning ”5.1.1 Antal epoker” visar dock att antalet epoker inte har någon större betydelse för simulatorns prestanda. Detta bör därför inte påverka testerna i någon större utsträckning.

I graferna anges nätverkens konfiguration enligt IN-DOLDA-UT. Ett nätverk med 32 innoder, 100 dolda noder och 30 ut-noder skrivs då 32-100-30. Om nätverket körts med parallelliserad summering (Logfetch) läggs ett F till på slutet. För nätverk som körts med normal summering läggs i stället ett N till.

I vissa grafer presenteras endast resultatet för testkörningar med 5 st trådar. Simulatorn visar oftast på högst prestandaökning vid just 5 trådar och dessa grafer bedömdes därför vara mest intressanta.

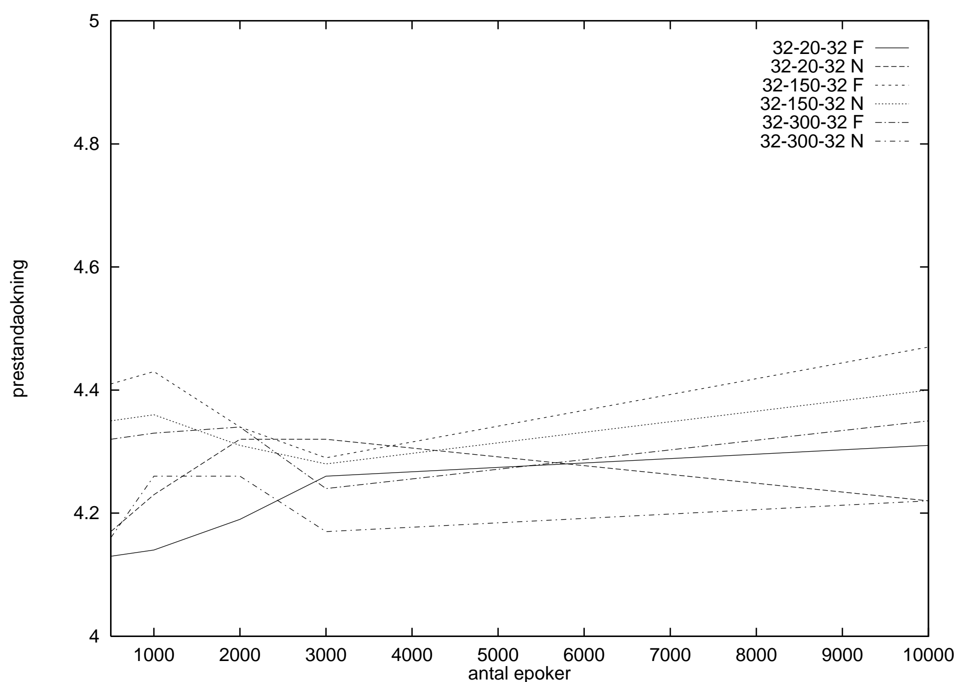
Samtliga körbara filer är genererade från källkodsfilerna presenterade i ”3.1 Parallelliserad implementation” och ”3.2 Serialiserad implementation”. De är genererade med kompilatorn CC (se ”3.4 Generering av körbar fil”) som ingår i Suns SunWSPro-paket.

5.1.1 Antal epoker

Innan den parallelliserade simulatorn kan starta inlärningen, sker initialisering av nätverket och uppstart av trådarna som behövs för den parallelliserade delen av programmet. När nätverket har tränats klart, sker testning av nätverkets beteende och utskrift av resultatet från detta test. Dessa delar av programmet är inte parallelliserade i nuvarande implementation och påverkar den parallelliserade simulatorns prestanda negativt.

Ju längre tid som går åt till exempelberäkning, desto mindre del av programmets totala tid åtgår till initiering av programmet och desto mindre påverkar denna del simulatorns totala prestanda. När antalet epoker ökas, ökas exempelberäkningens tid, och detta borde påverka simulatorns effektivitet positivt.

Hur många epoker nätverket tränas bör dock inte påverka simulatorns prestandaökning i någon större utsträckning. Att detta också är fallet går att se i nedanstående figur (Figur 12). Skillnaden på nätverket prestandaförbättring på 500 epoker och 10000 epoker, rör sig bara om någon tiondels skillnad uppåt eller neråt.



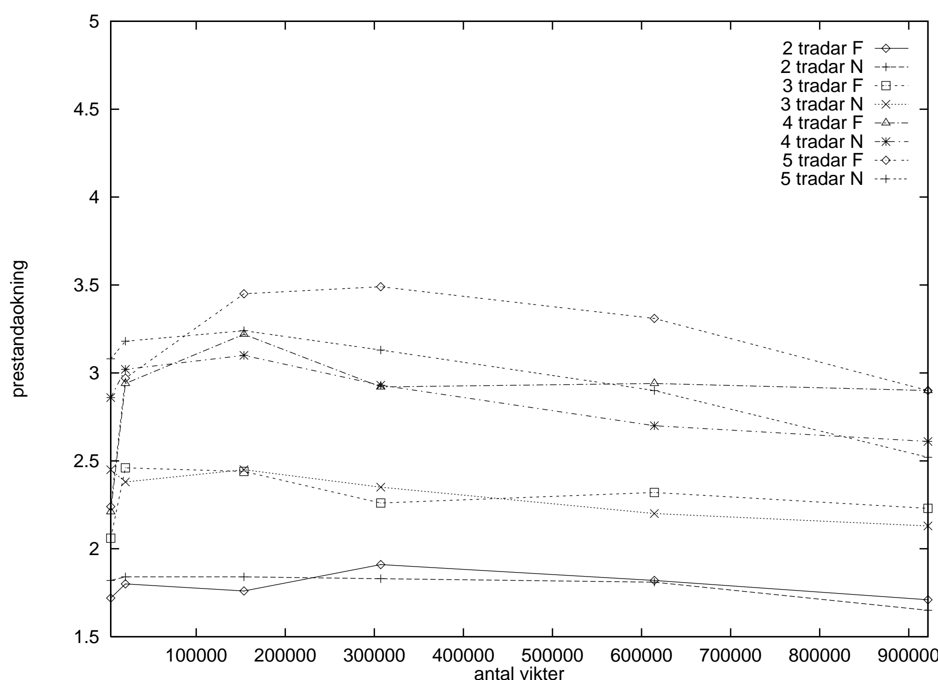
Figur 12: Antal epokers påverkan på prestandaökning (5 trådar)

5.1.2 Storleken på det neurala nätverket

I början och slutet på varje epok, måste summering av vikförändringarna och uppdatering av nätverkets vikter ske. Denna fas i epoken kan inte parallelliseras i samma utsträckning som beräkningen av exemplen kan parallelliseras. Ju större del av en epok som denna summering tar upp, jämfört med exempelberäkningen, desto mindre effektiv blir den parallelliserade simulatorn jämfört med den serialiserade simulatorn.

Detta betyder att storleken på nätverket (antalet vikter) har betydelse för den parallelliserade simulatorns effektivitet. När storleken på nätverket växer, ökar dels antalet beräkningar som behöver ske för varje exempel, men också antalet vikter som måste summeras i slutet på varje epok. När tiden för genomförandet av denna icke parallelliserade del ökar, minskar simulatorns totala effektivitet.

För att kontrollera detta provkördes simulatorn på olika nätverk i storlekar mellan 32-4-32 och 32-900-32 med 32 exempel. Endast antalet dolda noder varierades, mellan 4 och 900 noder. Resultaten presenteras i Figur 13 nedan.

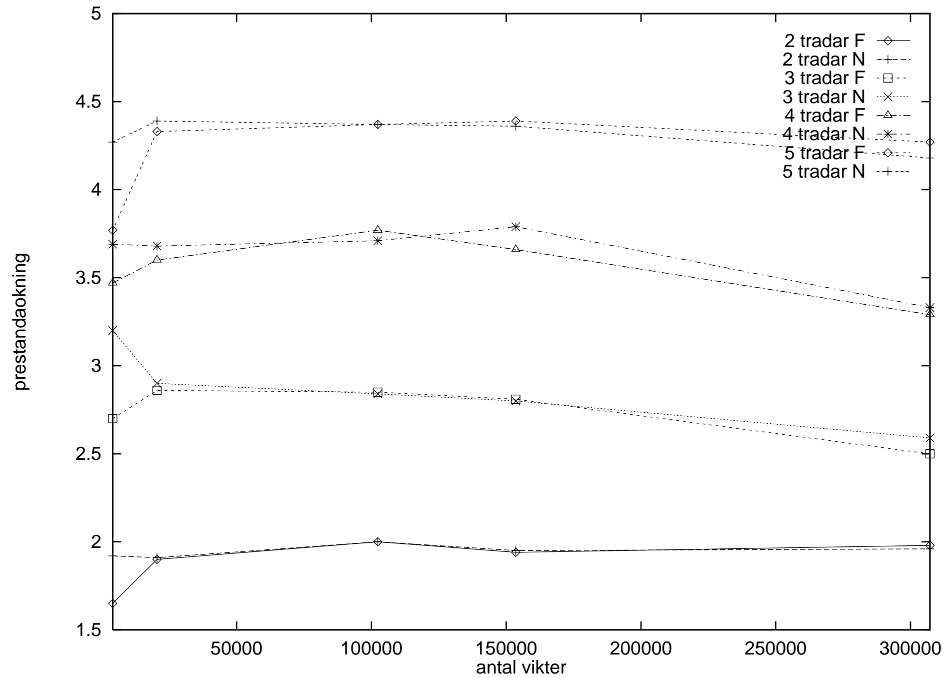


Figur 13: Antal vikters påverkan på prestandaökning (32 exempel)

Som synes minskar prestandaökningen när antalet vikter ökar. Det visar att större nätverk med många vikter ger mindre prestandaökning än mindre nätverk.

Den parallelliserade summeringen av viktförändringar som beskrivs i ”3.1 Parallelliserad implementation” infördes för att minska effektivitetsförlusten vid stora nätverk. I Figur 13 syns att de simuleringar som utnyttjat den parallelliserade summeringen (märkta med F i figuren) är effektivare än de som utnyttjat den normala summeringen (märkta med N i figuren) på stora nätverk, men i de flesta fallen mindre effektiva än den normala summeringen på mindre nätverk.

Liknande beteende går att se i Figur 14, där ovan presenterade simulering skett, men med 128 exempel i stället för 32.

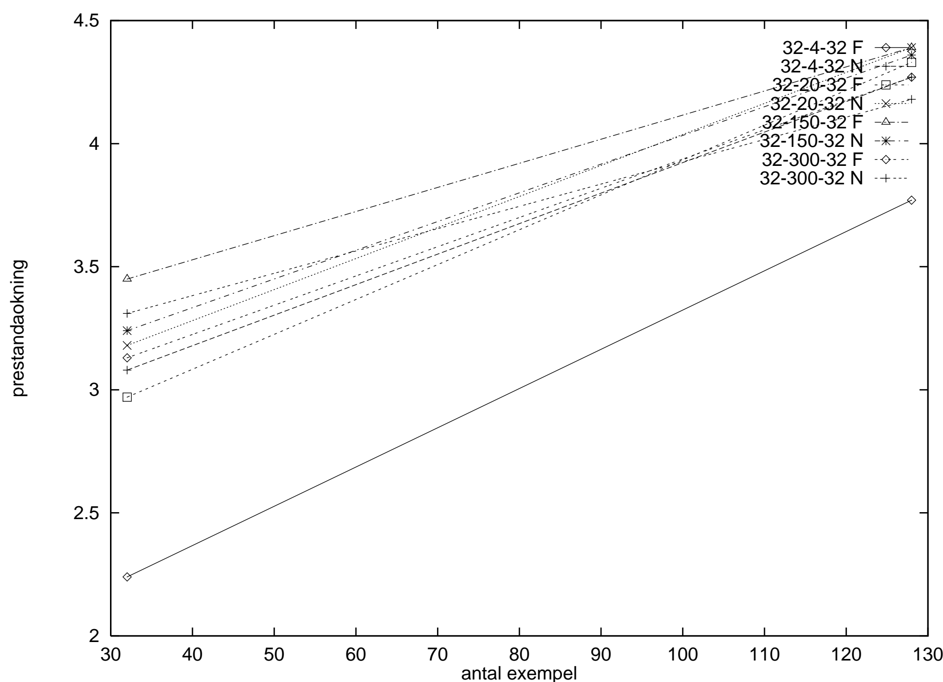


Figur 14: Antal vikters påverkan på prestandaökning (128 exempel)

5.1.3 Antal exempel

När antalet exempel ökas, ökar den parallelliserade delen av simulationen, utan att påverka de andra delarna av programmet. Detta borde därför påverka simulatorns effektivitet positivt. För att kontrollera detta, testkördes simulatorn på nätverk mellan 32-4-32 och 32-300-32 med 32 exempel och 128 exempel. Resultaten, som redovisas i Figur 15, visar att ju fler exempel simulatorn tränas på desto högre prestandaökning uppvisar den parallelliserade simulatorn, jämfört med den serialiserade. Samtliga nätverk uppvisar en högre prestandaökning när antalet exempel 128 jämfört med när antalet exempel är 32.

En ökning av antalet exempel ger en högre prestandaökning på i stort sett alla typer av nätverk, oavsett hur många noder det är i dem.



Figur 15: Antal exempels (32 och 128) påverkan på prestandaökning (5 trådar)

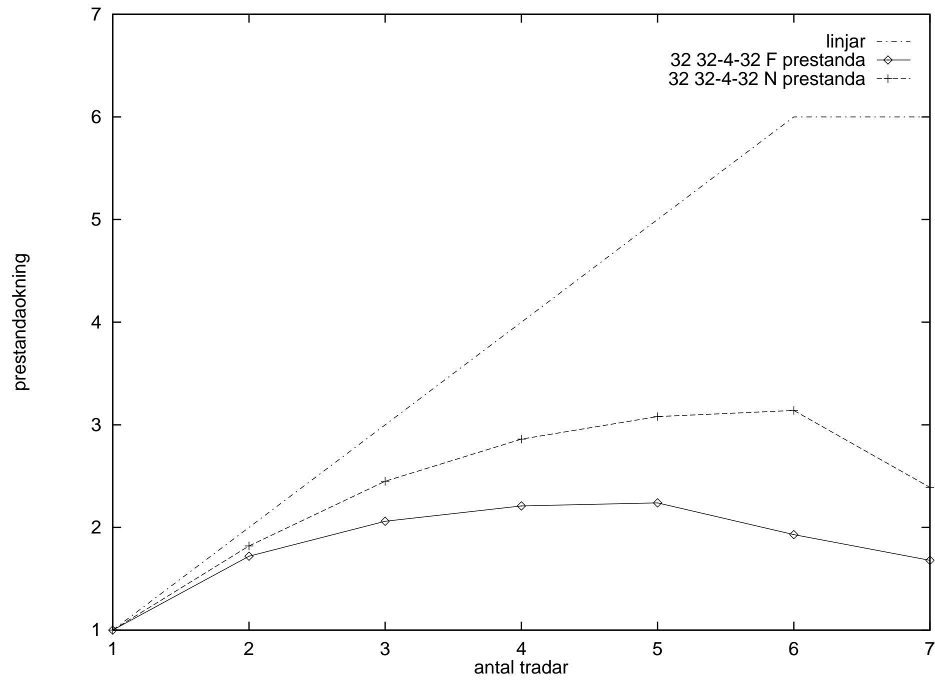
5.1.4 Antal trådar

Som motiverats i ”1.1.3 Prestandaökning i parallelliserade system”, så kan man inte förvänta sig linjär prestandaökning i ett parallelliserat program. Detta gäller även för den parallelliserade nätverkssimulator som presenteras i denna rapport. För att mäta prestandaökningen i den parallelliserade simulatorn har den provkörts på en mängd olika nätverk, med 2-7 trådar. Typiska kurvor över prestandaökningen som uppnåddes presenteras i Figur 16, Figur 17, Figur 18, Figur 19 och Figur 20.

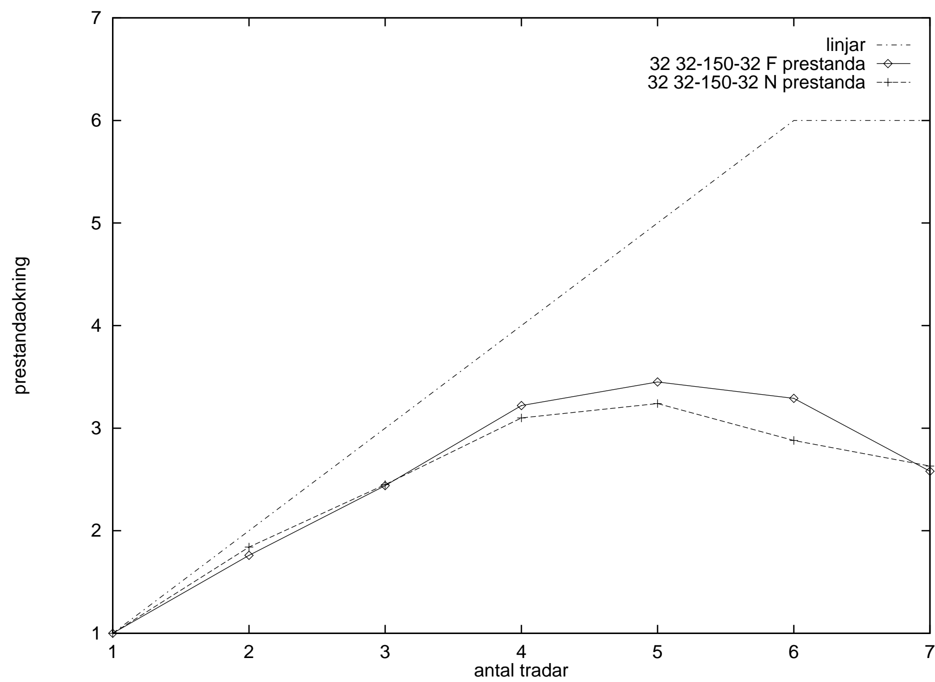
Eftersom testmaskinen (se ”2.2 Målsystem”) har 6 st CPUer uppvisas oftast högst prestandaökning när 4 eller 5 trådar används. Detta syns extra tydligt i Figur 19. Att simulering med 5 trådar i bland uppvisar högre prestandaökning än med 6 trådar, beror troligtvis på synkroniserings-overhead och att operativsystemet behöver CPU-tid, som kan köras på den lediga processorn.

I Figur 16 presenteras simulatorns beteende för ett ganska litet neuralt nätverk. Linjär prestandaökning uppnås inte ens när simulatorn körs med endast 2 trådar och prestandaökningen ökar inte särskilt mycket när simulatorn körs med fler trådar. Den normala summeringen (märkt med N i diagrammet) är avsevärt mycket effektivare än den parallelliserade summeringen (märkt med F i diagrammet). Detta beror på att nätverket är så litet, att den parallelliserade summeringsalgoritmen tillbringar mer tid med att vänta på att de

andra trådarna ska bli klara än att arbeta med själva summeringen. För så här små nätverk är det därför lämpligast att köra med 2-3 trådar, och normal summering.

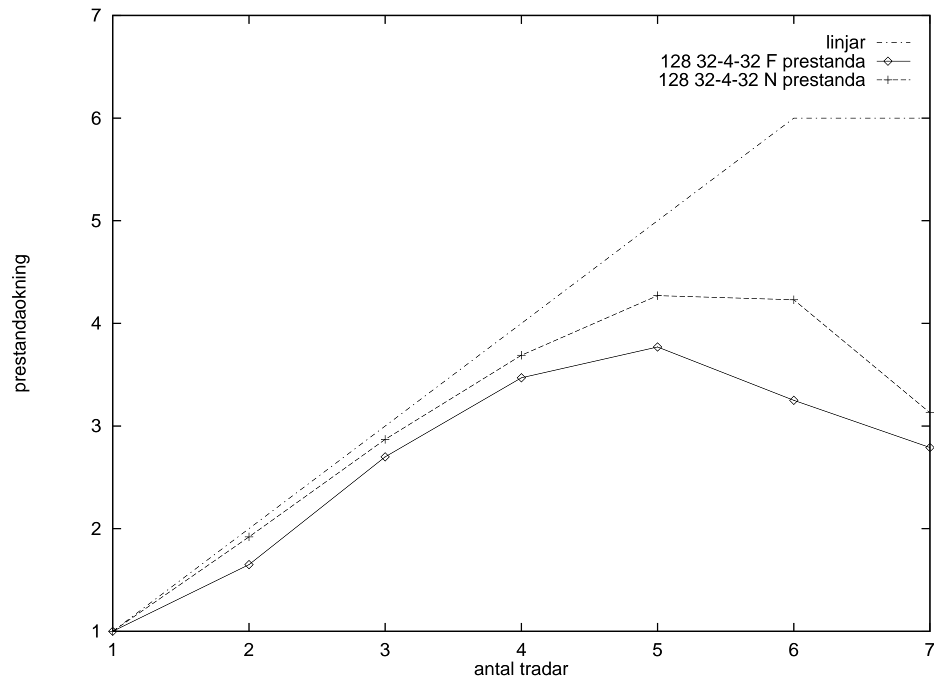


Figur 16: Prestandaökning för 32-4-32 nätverk med 32 exempel



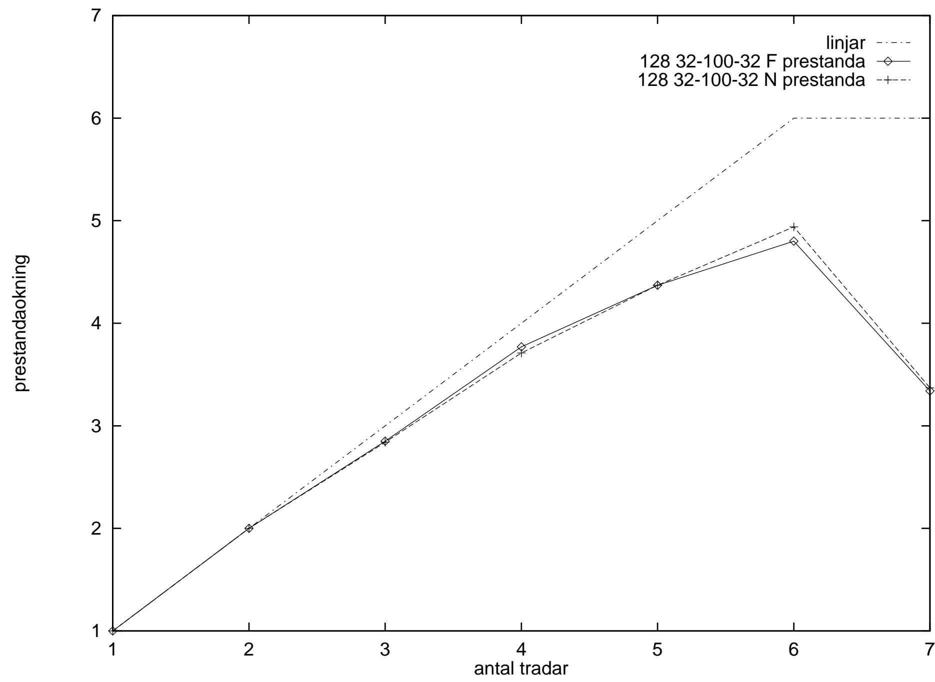
Figur 17: Prestandaökning för 32-150-32 nätverk med 32 exempel

Figur 17, visar simulatorns beteende för ett lite större nätverk. Här ökar prestandaökningen ganska jämnt ända upp till 4 trådar. För den här storleken på nätverk är normal summering och parallell summering ganska jämbördiga, men den parallelliserade summeringen är aningen effektivare på 4 trådar. Simulatorn är effektivast när den använder 5 trådar, men skillnaden mot när den använder 4 trådar är så liten att 4 trådar rekommenderas för detta nätverk.

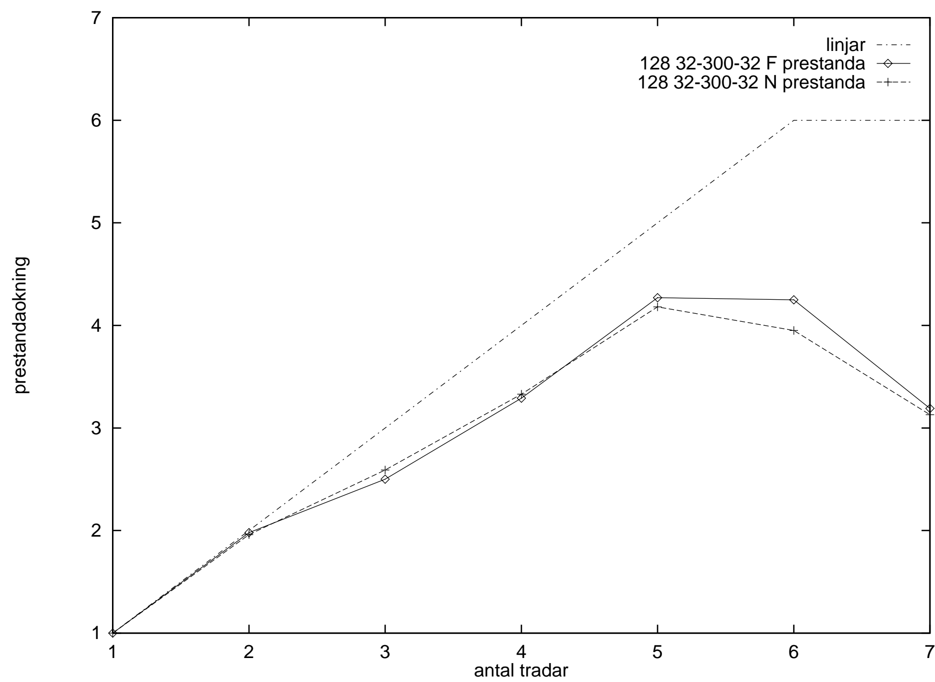


Figur 18: Prestandaökning för 32-4-32 nätverk med 128 exempel

Ovan ses Figur 18. Trots att detta nätverk är exakt likadant som det som presenteras i Figur 16 fås här en ganska jämn prestandaökning ända upp till 5 trådar. Detta beror naturligtvis på att nätverket körts med 4 gånger så många exempel. I ”5.1.3 Antal exempel” visades också att en ökning av antalet exempel påverkar prestandaökningen positivt.



Figur 19: Prestandaökning för 32-100-32 nätverk med 128 exempel



Figur 20: Prestandaökning för 32-300-32 nätverk med 128 exempel

6 Slutsatser

6.1 Utvärdering av mål och ambitioner

I ”2.3 Kriterier för bedömning av lösning” och ”2.4 Andra ambitioner” presenterades de mål och ambitioner som fanns vid starten av detta projekt. Nedan redovisas dessa, och ett resonemang sker om hur väl dessa mål och ambitioner är uppfyllda.

6.1.1 Performance (Effektivitet)

Det viktigaste målet för detta arbete var att utvärdera om det går att öka prestandan på en nätverkssimulator genom att parallellisera den. I avdelning ”4 Korrekthetstester” och avdelning ”5 Prestandatester” visades att det går att göra en parallelliserad nätverkssimulator som beter sig korrekt, och som dessutom uppvisar en prestandaökning jämfört med en serialiserad simulator.

Ur prestandatesterna som redovisas i ”5 Prestandatester” går det att dra en del slutsatser om hur effektiv den parallelliserade implementation som gjorts är på olika typer av nätverk:

- Ökning av antal exempel ökar prestandaökningen
- Ökning av antal vikter minskar prestandaökningen
- Antal epoker påverkar inte prestandaökningen

På det målsystem med 6 CPUer (se ”2.2 Målsystem”) som använts gäller följande:

- 4-5 trådar ger oftast högst prestandaökning
- 3-5 gångers prestandaökning är normalt

Om det går att få högre prestandaökning på ett system med fler CPUer är tyvärr omöjligt att säga, utan att genomföra tester på ett sådant system, men som nämnts i avdelning ”1.1.3 Prestandaökning i parallelliserade system” så ökar ytterligare CPUer inte alltid prestandan så mycket som skulle kunna vara förväntat. Trots detta är, för de nätverk som uppvisat en så gott som linjär prestandaökning ända upp till 6 trådar (för ex, se Figur 19), chansen stor att prestandaökningen skulle bli ännu större med fler trådar om ytterligare CPUer fanns tillgängliga.

Eftersom prestandan på många av nätverken ligger uppåt 75% (4.5 gånger prestandaökning med 6 st CPUer) av vad målmaskinen teoretiskt klarar av, anser författaren av denna rapport att parallelliseringen är lyckad och att målet för arbetet är uppfyllt.

Parallelliserad av summering av viktförändringar

Den lilla skillnaden på effektivitet mellan den vanliga summeringen (som inte är parallelliserad) och Logfetch (parallelliserad med tät synkronisering av trådarna) visar att tätsynkroniserad parallellisering inte fungerar så bra. Detta tyder på att en parallellisering på neuron- och länk-nivå inte skulle ge lika stora prestandaökningar som den epokbaserade parallelliseringen gett.

6.1.2 Usability (Användbarhet)

Ett av önskemålen var att det med hjälp av framtagen simulator ska vara lätt att ta fram en prototypimplementation av ett standard ANN. Simulatorn som implementerats uppfyller denna ambition. För att simulera den typ av nätverk som simulatorn hanterar behöver ingen programmering ske. Det räcker för användaren att specificera nätverkets utseende och antal trådar som ska användas för simulering, samt förse nätverket med de exempel som ska användas för inlärning, för att en simulering ska kunna ske.

6.1.3 Flexibility (Flexibilitet)

Den sista ambitionen som presenterades var att simulatorn ska vara flexibel och anpassningsbar för framtida krav. Eftersom kravet på effektivitet prioriterats högre än denna ambition, är den simulator som implementerats inte särskilt flexibel. Det är möjligt att lägga till nya inlärningsalgoritmer i simulatorn, men andra önskemål, såsom stöd för flera dolda lager, kan vara svårt att bygga in i simulatorn i dess nuvarande form.

6.2 Riktlinjer för användande av simulatorn

Med utgångspunkt enbart från de prestandatester som presenterats, kan det vara svårt att veta hur simulatorn ska ställas in för att få kortast möjliga simuleringstid för det nätverk som ska simuleras. Därför presenteras här ett par råd om hur simulatorn bör ställas in om målet är att få så hög prestanda som möjligt:

- Antalet trådar bör vara en mindre än antalet lediga CPUer som finns i den maskin som simulatorn ska köras på. En klar majoritet av de tester som gjorts har uppvisat högst prestandaökning när 5 trådar har använts till simulatorn och en av målmaskinens 6 CPUer har lämnats ledig. Detta gäller både för små nätverk med få vikter och för nätverk med ett stort antal vikter.
- Logfetch kan användas om antalet vikter är fler än ca 300000. Figur 13 och Figur 14 tyder på att gränsen för när Logfetch-summeringen blir effektiv för de testade nätverken går vid ca 300000 vikter. Detta gäller troligtvis även för andra nätverk.

7 Fortsatt arbete

På flera ställen i denna rapport har den epokbaserade metod för parallellisering som använts, jämförts med en möjlig neuron- och länk-baserad metod för parallellisering. Trots att det finns mycket som talar för att en neuron- och länk-baserad metod inte skulle bli särskild effektiv i praktiken, så vore det intressant att jämföra den med den epokbaserade metod som har implementerats under arbetet med denna rapport. Att implementera och testa den neuron- och länk-baserade metoden är ett arbete jämförbart i storlek med det arbete som skett för denna rapport, och skulle kanske passa som ett examensarbete.

Det finns också arbete kvar att göra på den implementation som gjordes under denna rapport. Saker som skulle vara intressant att implementera för att få ett komplett funktionsbibliotek för utveckling av neurala nätverk är:

- Nya inlärningsalgoritmer

  En inlärningsmetod som är särskilt intressant att implementera är en Steepest Descent med Adaptive Learning Rates [MS94]. Enligt [MS94] behöver denna metod i de flesta fall endast tränas en tiondel så många epoker som vanlig Stepeest Descent för att få lyckad inträning.

  Skillnaden mellan den Steepest Descent som implementerats och en implementation av Steepest Descent med Adaptive Learning Rates är inte så stora. Anledningen att den inte implementerats till denna rapport är att det inte fanns tillräckligt med tid för att göra tillräckliga prestandatester av båda metoderna.

- Hantera mer än ett dolt lager
- Hantera nätverk med feedback

  Det finns vissa problem när det gäller feedbacknätverk, som gör dem svåra att parallellisera. En utredning av detta vore intressant.

- Hjälpfunktioner

  Funktioner som kan vara aktuella att implementera är möjlighet att spara och ladda in nätverkets vikter, räkna ut genomsnittsligt fel på nätverkets ut-noder och liknande saker. Detta är i allmänhet ganska enkla funktioner, men de behövs för att få ett komplett funktionsbibliotek.

Referenser

[AC95] Adrian Cockcroft (1995), Sun Performance and Tuning, SunSoft Press

[AG85] David H. Ackley och Geoffrey E. Hinton ochTerrence J. Sejnowski (1985), A learning algorithm for Boltzmann machines, Cognitive Science nr 9/85 sid 147-169

[LC96] Louis Coetzee (1996), Parallel approaches to training feedforward neural nets, Faculty of Engineering, University of Pretoria

[MS93] Murray Smith (1993), Neural Networks for Statistical Modeling, Van Nostrand Reinhold, New York

[RH86] Rumelhart, et al (1986)

[SG94] Silberschatz, A och Galvin, P (1994), Operating System Concepts Fourth Edition, Addison-Wesley Publishing Company

Appendix A: Källkod för parallell implementation

epokbp.c:

```
/*
 * Parallellized backprop simulator (by Alexander Foborg)
 * Based on:
 *   A very simple backprop simulator (by M. Boden).
 */

#ifndef __cplusplus    /* Optimize for C++ if available */
#define inline         /* Standard C don't know about inline */
#endif

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/types.h>
#include <time.h>
#include <pthread.h>
#include <thread.h>

#define NETHEAD
#include NETFILE
#undef NETHEAD

typedef struct {
  double input[INPUT];
  double target[OUTPUT];
} Example ;

#include NETFILE

typedef struct {
  int thread_no;
  double (*cin2hid)[INPUT+1][HIDDEN];
  double (*chid2out)[HIDDEN+1][OUTPUT];
  double (*hidden)[HIDDEN];
  double (*output)[OUTPUT];
  Example *examples;
  int examplesToDo;
  sema_t *sema_done;
  sema_t *sema_go;
} learnStruct;

/* weight values between units; fully connected network */
double in2hid[INPUT+1][HIDDEN], hid2out[HIDDEN+1][OUTPUT];

sema_t sema_master_done[NOOFTHREADS], sema_master_go[NOOFTHREADS];

void init()
{
  int i,h,o;

  srand48(time(NULL));
  for (i=0; i<INPUT+1; i++)
    for (h=0; h<HIDDEN; h++)
      in2hid[i][h]=drand48()-0.5;
  for (h=0; h<HIDDEN+1; h++)
    for (o=0; o<OUTPUT; o++)
      hid2out[h][o]=drand48()-0.5;
}

/* the output function determines the level of output activity of a
   unit */
inline double sigmoidActivation(double x)
{
  return (1.0/(1.0+exp(-(x))));
}
```

```
/* the derivative of the output function is used to determine errors
   on units */
inline double sigmoidDerivative(double x)
{
  return (x*(1.0-x));
}

/* the activity of the network is determined by calculating the summed
   input of each receptive unit and then applying the output function
   on that sum. Function must be reentrant */

inline void feedforward(double hidden[HIDDEN],
                 double output[OUTPUT],
                 double input[INPUT])
{
  int i,h,o;
  double sum;

  for (h=0; h < HIDDEN; h++) {
    sum = input[0]*in2hid[0][h];
    for (i = 1; i < INPUT; i++)
      sum += input[i] * in2hid[i][h];
    hidden[h] = sigmoidActivation(sum+in2hid[INPUT][h]);
  }

  for (o = 0; o < OUTPUT; o++) {
    sum = hidden[0] * hid2out[0][o];
    for (h = 1; h < HIDDEN; h++)
      sum += hidden[h] * hid2out[h][o];
    output[o] = sigmoidActivation(sum+hid2out[HIDDEN][o]);
  }
}

/* reset errors, function must be reentrant */

inline void resetError(double cin2hid[INPUT+1][HIDDEN],
                double chid2out[HIDDEN+1][OUTPUT])
{
  int i,h,o;

  for (o=0; o<OUTPUT; o++) {
    for (h=0; h<HIDDEN; h++)
      chid2out[h][o] = 0.0;
    chid2out[HIDDEN][o] = 0.0;
  }
  for (h=0; h<HIDDEN; h++) {
    for (i=0; i<INPUT; i++)
      cin2hid[i][h] = 0.0;
    cin2hid[INPUT][h] = 0.0;
  }
}

/* calculate the error for each unit, function must be reentrant  */
void calcError(double hidden[HIDDEN],
               double output[OUTPUT],
               double cin2hid[INPUT+1][HIDDEN],
               double chid2out[HIDDEN+1][OUTPUT],
               double input[INPUT],
               double target[OUTPUT])
{
  /* error values for units */
  double errout[OUTPUT], errhid[HIDDEN];

  int i,h,o;

  for (o = 0; o < OUTPUT; o++)
  {
    errout[o] = (target[o] - output[o]) * sigmoidDerivative(output[o]);
  }

  for (h = 0; h < HIDDEN; h++) {
    errhid[h] = 0.0;
    for (o = 0; o < OUTPUT; o++)
```

```
      errhid[h] += errout[o] * hid2out[h][o];
    errhid[h] *= sigmoidDerivative(hidden[h]);
  }

  for (o = 0; o < OUTPUT; o++) {
    for (h = 0; h < HIDDEN; h++)
      chid2out[h][o] += errout[o]*hidden[h];
    chid2out[HIDDEN][o] += errout[o];
  }

  for (h = 0; h < HIDDEN; h++) {
    for (i = 0; i < INPUT; i++)
      cin2hid[i][h] += errhid[h] * input[i];
    cin2hid[INPUT][h] += errhid[h];
  }
}
/* adjust weights */

inline void adjustWeights(double cin2hid[INPUT+1][HIDDEN],
                   double chid2out[HIDDEN+1][OUTPUT])
{
  int i,h,o;

  for (o = 0; o < OUTPUT; o++) {
    for (h = 0; h < HIDDEN; h++)
      hid2out[h][o] += ETA * chid2out[h][o];
    hid2out[HIDDEN][o] += ETA * chid2out[HIDDEN][o];
  }
  for (h = 0; h < HIDDEN; h++) {
    for (i = 0; i < INPUT; i++)
      in2hid[i][h] += ETA * cin2hid[i][h];
    in2hid[INPUT][h] += ETA * cin2hid[INPUT][h];
  }
}

inline void startThreads(sema_t *sema_go) {
  int threadCount;

  for (threadCount = 1; threadCount < NOOFTHREADS; threadCount++) {
    sema_post(&sema_go[threadCount]);
  }
}

inline void waitThreads(sema_t *sema_done) {
  int threadCount;

  for (threadCount = 1; threadCount < NOOFTHREADS; threadCount++) {
    sema_wait(&sema_done[threadCount]);
  }
}

void waitAndAdd(sema_t *sema_done,
                double cin2hid[NOOFTHREADS][INPUT+1][HIDDEN],
                double chid2out[NOOFTHREADS][HIDDEN+1][OUTPUT]) {

  static int i,h,o, count;

  for (count = 1; count < NOOFTHREADS; count++) {
    sema_wait(&sema_done[count]);
    for (o=0; o<OUTPUT; o++) {
      for (h=0; h<HIDDEN; h++)
        chid2out[0][h][o]+=chid2out[count][h][o];
      chid2out[0][HIDDEN][o]+=chid2out[count][HIDDEN][o];
    }
    for (h=0; h<HIDDEN; h++) {
      for (i=0; i<INPUT; i++)
        cin2hid[0][i][h]+=cin2hid[count][i][h];
      cin2hid[0][INPUT][h]+=cin2hid[count][INPUT][h];
    }
  }
  adjustWeights(cin2hid[0], chid2out[0]);
}
```

```
void *learnfunction(void *learnInfoPtr) {
    int examplecount, delta, pratamed, o, h, i, epokCount;

    learnStruct *learnInfo = (learnStruct *)learnInfoPtr;
    int thread_no = learnInfo->thread_no;

    for (epokCount = 0; epokCount < NOOFEPOKS; epokCount++) {
      resetError(learnInfo->cin2hid[thread_no],
                 learnInfo->chid2out[thread_no]);

      for (examplecount = 0; examplecount < learnInfo->examplesToDo;
           examplecount++) {
        feedforward(*(learnInfo->hidden),
                    *(learnInfo->output),
                    (learnInfo->examples+examplecount)->input);

        calcError(*(learnInfo->hidden),
                  *(learnInfo->output),
                  learnInfo->cin2hid[thread_no],
                  learnInfo->chid2out[thread_no],
                  (learnInfo->examples+examplecount)->input,
                  (learnInfo->examples+examplecount)->target);
      }

      /* Parallell summation of weightchanges */

      #ifdef LOGFETCH
        for(delta = 1; thread_no%delta == 0 &&
                       delta < NOOFTHREADS; delta *= 2) {

          if(thread_no % (delta * 2) == 0) {
            if(thread_no + delta < NOOFTHREADS) {
              pratamed = thread_no + delta;
              /* barrier med pratamed */
              sema_post(&(learnInfo->sema_done[pratamed]));
              sema_wait(&(learnInfo->sema_go[pratamed]));

              /* summera första halvan av datan med dennas array */
              for (o = 0; o < OUTPUT; o++) {
                for (h = 0; h < HIDDEN; h++)
                  learnInfo->chid2out[thread_no][h][o]  +=
                                    learnInfo->chid2out[pratamed][h][o];
                learnInfo->chid2out[thread_no][HIDDEN][o]  +=
                                    learnInfo->chid2out[pratamed][HIDDEN][o];
              }

              /* barrier med pratamed */
              sema_post(&(learnInfo->sema_done[pratamed]));
              sema_wait(&(learnInfo->sema_go[pratamed]));
            }
          } else {
            pratamed = thread_no - delta;
            /* barrier med pratamed */
            sema_post(&(learnInfo->sema_go[thread_no]));
            sema_wait(&(learnInfo->sema_done[thread_no]));

            /* summera andra halvan av datan med pratameds array */

            for (h = 0; h < HIDDEN; h++) {
              for (i = 0; i < INPUT; i++)
                learnInfo->cin2hid[pratamed][i][h]  +=
                                    learnInfo->cin2hid[thread_no][i][h];
              learnInfo->cin2hid[pratamed][INPUT][h]  +=
                                    learnInfo->cin2hid[thread_no][INPUT][h];
            }

            /* barrier med pratamed */
            sema_post(&(learnInfo->sema_go[thread_no]));
            sema_wait(&(learnInfo->sema_done[thread_no]));
          }
        }
      #endif
```

```
      if (thread_no == 0) {
        #ifdef LOGFETCH
          waitThreads(sema_master_done);
          adjustWeights(*learnInfo->cin2hid, *learnInfo->chid2out);
        #else
          waitAndAdd(sema_master_done, learnInfo->cin2hid, learnInfo->chid2out);
        #endif
        startThreads(sema_master_go);
      } else {
        /* Send DONE! */
        sema_post(&(sema_master_done[thread_no]));
        /* Wait for GO! */
        sema_wait(&(sema_master_go[thread_no]));
      }
    }

    return 0;
}

int main()
{
  thread_t thread_id[NOOFTHREADS];      /* array of thread IDs */
  pthread_attr_t attr;
  int epokCount, exampleCount, threadCount, testCount, lineCount, checkCount;
  learnStruct learnInfo[NOOFTHREADS];

  /* weight changes to be done at end of epoch */
  double cin2hid[NOOFTHREADS][INPUT+1][HIDDEN],
         chid2out[NOOFTHREADS][HIDDEN+1][OUTPUT];

  /* network unit activities */
  double hidden[NOOFTHREADS][HIDDEN], output[NOOFTHREADS][OUTPUT];

  sema_t sema_done[NOOFTHREADS], sema_go[NOOFTHREADS];

  #ifdef LOGFETCH
    fprintf(stderr,
    "Nät: %d-%d-%d Epoker: %d Antal exempel: %d Antal Trådar: %d LOGFETCH\n",
    INPUT, HIDDEN, OUTPUT, NOOFEPOKS, NOOFEXAMPLES, NOOFTHREADS);
  #else
    fprintf(stderr,
    "Nät: %d-%d-%d Epoker: %d Antal exempel: %d Antal Trådar: %d ej LOGFETCH\n",
    INPUT, HIDDEN, OUTPUT, NOOFEPOKS, NOOFEXAMPLES, NOOFTHREADS);
  #endif

  init();   /* Initialise weights */

  for (threadCount = 0; threadCount < NOOFTHREADS; threadCount++) {
    learnInfo[threadCount].thread_no = threadCount;
    learnInfo[threadCount].cin2hid = cin2hid;
    learnInfo[threadCount].chid2out = chid2out;
    learnInfo[threadCount].hidden = &hidden[threadCount];
    learnInfo[threadCount].output = &output[threadCount];
    learnInfo[threadCount].examples =
                              &examples[(NOOFEXAMPLES/NOOFTHREADS)*threadCount];

    learnInfo[threadCount].examplesToDo = NOOFEXAMPLES/NOOFTHREADS;
    if (threadCount < NOOFEXAMPLES%NOOFTHREADS) {
      learnInfo[threadCount].examplesToDo += 1;
      learnInfo[threadCount].examples += threadCount;
    } else {
      learnInfo[threadCount].examples += NOOFEXAMPLES%NOOFTHREADS;
    }

    learnInfo[threadCount].sema_done = sema_done;
    sema_init(&sema_done[threadCount], 0, 0, NULL);

    learnInfo[threadCount].sema_go = sema_go;
    sema_init(&sema_go[threadCount], 0, 0, NULL);

    sema_init(&sema_master_done[threadCount], 0, 0, NULL);
    sema_init(&sema_master_go[threadCount], 0, 0, NULL);
  }
```

```
  pthread_attr_init(&attr); /* initialize attr with default attributes */
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM); /* OS-scheduling
                                                        * for multi-CPUs */
  for (threadCount = 1; threadCount < NOOFTHREADS; threadCount++) {
    pthread_create(&thread_id[threadCount], &attr,
                   learnfunction, (void *)(learnInfo+threadCount));
  }

  learnfunction((void *)(learnInfo+0));  /* Start Master Thread (no 0) */

  /* the test phase consists of setting input values, calculating the
     actual output and printing errors on the screen, for each example. */

  for (exampleCount = 0; exampleCount < NOOFEXAMPLES; exampleCount++) {
    feedforward(*(learnInfo[0].hidden),
                *(learnInfo[0].output),
                (examples+exampleCount)->input);
    for (lineCount = 0; lineCount < OUTPUT; lineCount++) {
      printf("(% 5.3f-% 5.3f = % 5.3f) ", learnInfo[0].output[0][lineCount],
             examples[exampleCount].target[lineCount],
             learnInfo[0].output[0][lineCount]-
                                 examples[exampleCount].target[lineCount]);
    }
    printf("\n");
  }

  #ifdef NOOFCHECKS
    printf("\n");

    for (checkCount = 0; checkCount < NOOFCHECKS; checkCount++) {
      feedforward(*(learnInfo[0].hidden),
                  *(learnInfo[0].output),
                  (checks+checkCount)->input);
      for (lineCount = 0; lineCount < OUTPUT; lineCount++) {
        printf("(% 5.3f/% 5.3f) ", checks[checkCount].target[lineCount],
               learnInfo[0].output[0][lineCount]-
                                     checks[checkCount].target[lineCount]);
      }
      printf("\n");
    }
  #endif

  #ifdef NOOFTESTS
    printf("\n");

    for (testCount = 0; testCount < NOOFTESTS; testCount++) {
      feedforward(*(learnInfo[0].hidden),
                  *(learnInfo[0].output),
                  (tests+testCount)->input);
      for (lineCount = 0; lineCount < OUTPUT; lineCount++) {
        printf("% 5.3f ",
               learnInfo[0].output[0][lineCount]);
      }
      printf("\n");
    }
  #endif

  return 0;
}
```

Appendix B: Källkod för seriell implementation

epokbp_serial.c:

```
/*
 * Serialized backprop simulator (by Alexander Foborg)
 * Based on:
 *   A very simple backprop simulator (by M. Boden).
 */

#ifndef __cplusplus    /* Optimize for C++ if available */
#define inline         /* Standard C don't know about inline */
#endif

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/types.h>
#include <time.h>

#define NETHEAD
#include NETFILE
#undef NETHEAD

typedef struct {
  double input[INPUT];
  double target[OUTPUT];
} Example;

#include NETFILE

/* network unit activities */
double hidden[HIDDEN], output[OUTPUT];

/* weight values between units; fully connected network */
double in2hid[INPUT+1][HIDDEN], hid2out[HIDDEN+1][OUTPUT];

/* weight changes to be done at end of epoch */
double cin2hid[INPUT+1][HIDDEN], chid2out[HIDDEN+1][OUTPUT];

void init()
{
  int i,h,o;

  srand48(time(NULL));
  for (i=0; i<INPUT+1; i++)
    for (h=0; h<HIDDEN; h++)
      in2hid[i][h]=drand48()-0.5;
  for (h=0; h<HIDDEN+1; h++)
    for (o=0; o<OUTPUT; o++)
      hid2out[h][o]=drand48()-0.5;
}

/* the output function determines the level of output activity of a
   unit */
double sigmoidActivation(double x)
{
  return (1.0/(1.0+exp(-(x))));
}

/* the derivative of the output function is used to determine errors
   on units */
double sigmoidDerivative(double x)
{
  return (x*(1.0-x));
}

/* the activity of the network is determined by calculating the summed
   input of each receptive unit and then applying the output function
```

```
   on that sum. */
void feedforward(double input[INPUT])
{
  int i,h,o;
  double sum;

  for (h=0; h<HIDDEN; h++) {
    sum=0.0;
    for (i=0; i<INPUT; i++)
      sum+=input[i]*in2hid[i][h];
    hidden[h]=sigmoidActivation(sum+in2hid[INPUT][h]);
  }
  for (o=0; o<OUTPUT; o++) {
    sum=0.0;
    for (h=0; h<HIDDEN; h++)
      sum+=hidden[h]*hid2out[h][o];
    output[o]=sigmoidActivation(sum+hid2out[HIDDEN][o]);
  }
}

/* reset errors */

void resetError(/* double cin2hid[INPUT+1][HIDDEN],
                double chid2out[HIDDEN+1][OUTPUT] */)
{
  int i,h,o;

  for (o=0; o<OUTPUT; o++) {
    for (h=0; h<HIDDEN; h++)
      chid2out[h][o] = 0.0;
    chid2out[HIDDEN][o] = 0.0;
  }
  for (h=0; h<HIDDEN; h++) {
    for (i=0; i<INPUT; i++)
      cin2hid[i][h] = 0.0;
    cin2hid[INPUT][h] = 0.0;
  }
}

/* calculate the error for each unit */
void calcError(/* double cin2hid[INPUT+1][HIDDEN],
               double chid2out[HIDDEN+1][OUTPUT],
                */Example example)
{
  /* error values for units */
  static double errout[OUTPUT], errhid[HIDDEN];

  int i,h,o;

  for (o=0; o<OUTPUT; o++)
    errout[o]=(example.target[o]-output[o])*sigmoidDerivative(output[o]);
  for (h=0; h<HIDDEN; h++) {
    errhid[h]=0.0;
    for (o=0; o<OUTPUT; o++)
      errhid[h] += errout[o]*hid2out[h][o];
    errhid[h] *= sigmoidDerivative(hidden[h]);
  }

  for (o=0; o<OUTPUT; o++) {
    for (h=0; h<HIDDEN; h++)
      chid2out[h][o]+=errout[o]*hidden[h];
    chid2out[HIDDEN][o]+=errout[o];
  }
  for (h=0; h<HIDDEN; h++) {
    for (i=0; i<INPUT; i++)
      cin2hid[i][h]+=errhid[h]*example.input[i];
    cin2hid[INPUT][h]+=errhid[h];
  }
}

/* adjust weights */

void adjustWeights(/* double cin2hid[INPUT+1][HIDDEN],
```

```
                 double chid2out[HIDDEN+1][OUTPUT] */)
{
  int i,h,o;

  for (o=0; o<OUTPUT; o++) {
    for (h=0; h<HIDDEN; h++)
      hid2out[h][o]+=ETA*chid2out[h][o];
    hid2out[HIDDEN][o]+=ETA*chid2out[HIDDEN][o];
  }
  for (h=0; h<HIDDEN; h++) {
    for (i=0; i<INPUT; i++)
      in2hid[i][h]+=ETA*cin2hid[i][h];
    in2hid[INPUT][h]+=ETA*cin2hid[INPUT][h];
  }
}

int main()
{
  int exampleCount, threadCount, epokCount, lineCount, checkCount, testCount;

  fprintf(stderr,
  "Nät: %d-%d-%d Epoker: %d Antal exempel: %d Serialiserad version\n",
  INPUT, HIDDEN, OUTPUT, NOOFEPOKS, NOOFEXAMPLES);

  init();   /* Initialise weights */

  for (epokCount = 0; epokCount < NOOFEPOKS; epokCount++) {
    resetError();

    for (exampleCount = 0; exampleCount < NOOFEXAMPLES; exampleCount++) {
      feedforward(examples[exampleCount].input);
      calcError(examples[exampleCount]);
    }

    adjustWeights();
  }

  /* the test phase consists of setting input values, calculating the
     actual output and printing errors on the screen, for each example. */

  for (exampleCount = 0; exampleCount < NOOFEXAMPLES; exampleCount++) {
    feedforward(examples[exampleCount].input);
    for (lineCount = 0; lineCount < OUTPUT; lineCount++) {
      printf("% 5.3lf ",
             output[lineCount]-examples[exampleCount].target[lineCount]);
    }
    printf("\n");
  }

  #ifdef NOOFCHECKS
    printf("\n");

    for (checkCount = 0; checkCount < NOOFCHECKS; checkCount++) {
      feedforward(checks[checkCount].input);
      for (lineCount = 0; lineCount < OUTPUT; lineCount++) {
        printf("% 5.3lf ",
               output[lineCount]-checks[checkCount].target[lineCount]);
      }
      printf("\n");
    }
  #endif

  #ifdef NOOFTESTS
    printf("\n");

    for (testCount = 0; testCount < NOOFTESTS; testCount++) {
      feedforward((tests+testCount)->input);
      for (lineCount = 0; lineCount < OUTPUT; lineCount++) {
        printf("% 5.3lf ",
               output[lineCount]);
      }
      printf("\n");
    }
```

```
  #endif

  return 0;
}
```

Appendix C: Skript för generering av körbart ANN

build.sh:

```
#!/bin/sh

NETFILE=$1     # FILE CONTAINING NETWORK LAYOUT AND EXAMPLES
SERIAL=$2      # ENTER "_serial" as second argument for serial version

CC -lm -lpthread -O3 -DNETFILE=\"$NETFILE\" -o \
    ${NETFILE}${SERIAL}.exe epokbp$SERIAL.c

#gcc -lm -lpthread -O3 -multrasparc -DNETFILE=\"$NETFILE\" \
#    -o ${NETFILE}${SERIAL}.exe epokbp$SERIAL.c

#CC -fast -xO4 -depend -xchip=ultra -xarch=v8plus -lm -lpthread \
#   -DNETFILE=\"$NETFILE\" -o ${NETFILE}${SERIAL}.exe epokbp$SERIAL.c
```

Appendix D: Simulering av XOR

XOR.net:

```
#ifdef NETHEAD

/*#define LOGFETCH*/

#define NOOFEPOKS 10000

#define NOOFTHREADS 2

#define ETA    0.2 /* learning rate      */

#define INPUT  2   /* no of input units  */
#define HIDDEN 2   /* no of hidden units */
#define OUTPUT 1   /* no of output units */

#else

#define NOOFEXAMPLES 4

Example examples[NOOFEXAMPLES] = {
 { { 0.0, 0.0 }, { 0.0 } },
 { { 0.0, 1.0 }, { 1.0 } },
 { { 1.0, 0.0 }, { 1.0 } },
 { { 1.0, 1.0 }, { 0.0 } }
};

#endif
```

XOR.capture:

```
( 0.030- 0.000 =  0.030)
( 0.967- 1.000 = -0.033)
( 0.967- 1.000 = -0.033)
( 0.041- 0.000 =  0.041)
```

Appendix E: Simulering av Nencode 8

Nencode8.net:

```
#ifdef NETHEAD

/*#define LOGFETCH*/

#define NOOFTHREADS 4

#define NOOFEPOKS 10000

#define ETA    0.2 /* learning rate      */

#define INPUT  8   /* no of input units  */
#define HIDDEN 3   /* no of hidden units */
#define OUTPUT 8   /* no of output units */

#else

#define NOOFEXAMPLES 8

Example examples[NOOFEXAMPLES] = {
 { { 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },
   { 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 } },
 { { 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },
   { 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 } },
 { { 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0 },
   { 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0 } },
 { { 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0 },
   { 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0 } },
 { { 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0 },
   { 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0 } },
 { { 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0 },
   { 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0 } },
 { { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0 },
   { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0 } },
 { { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0 },
   { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0 } }
};

#endif
```

Nencode8.capture:

```
( 0.949- 1.000 = -0.051) ( 0.038- 0.000 =  0.038) ( 0.042- 0.000 =  0.042) (
0.000- 0.000 =  0.000) ( 0.000- 0.000 =  0.000) ( 0.000- 0.000 =  0.000) ( 0.026-
0.000 =  0.026) ( 0.000- 0.000 =  0.000)
( 0.027- 0.000 =  0.027) ( 0.949- 1.000 = -0.051) ( 0.000- 0.000 =  0.000) (
0.000- 0.000 =  0.000) ( 0.044- 0.000 =  0.044) ( 0.029- 0.000 =  0.029) ( 0.000-
0.000 =  0.000) ( 0.000- 0.000 =  0.000)
( 0.025- 0.000 =  0.025) ( 0.000- 0.000 =  0.000) ( 0.948- 1.000 = -0.052) (
0.028- 0.000 =  0.028) ( 0.047- 0.000 =  0.047) ( 0.000- 0.000 =  0.000) ( 0.000-
0.000 =  0.000) ( 0.000- 0.000 =  0.000)
( 0.000- 0.000 =  0.000) ( 0.000- 0.000 =  0.000) ( 0.036- 0.000 =  0.036) (
0.956- 1.000 = -0.044) ( 0.001- 0.000 =  0.001) ( 0.000- 0.000 =  0.000) ( 0.028-
0.000 =  0.028) ( 0.042- 0.000 =  0.042)
( 0.000- 0.000 =  0.000) ( 0.026- 0.000 =  0.026) ( 0.024- 0.000 =  0.024) (
0.000- 0.000 =  0.000) ( 0.929- 1.000 = -0.071) ( 0.000- 0.000 =  0.000) ( 0.000-
0.000 =  0.000) ( 0.014- 0.000 =  0.014)
( 0.000- 0.000 =  0.000) ( 0.035- 0.000 =  0.035) ( 0.000- 0.000 =  0.000) (
0.000- 0.000 =  0.000) ( 0.001- 0.000 =  0.001) ( 0.955- 1.000 = -0.045) ( 0.029-
0.000 =  0.029) ( 0.043- 0.000 =  0.043)
( 0.040- 0.000 =  0.040) ( 0.000- 0.000 =  0.000) ( 0.000- 0.000 =  0.000) (
0.031- 0.000 =  0.031) ( 0.000- 0.000 =  0.000) ( 0.030- 0.000 =  0.030) ( 0.949-
1.000 = -0.051) ( 0.000- 0.000 =  0.000)
( 0.000- 0.000 =  0.000) ( 0.000- 0.000 =  0.000) ( 0.000- 0.000 =  0.000) (
0.024- 0.000 =  0.024) ( 0.049- 0.000 =  0.049) ( 0.024- 0.000 =  0.024) ( 0.000-
0.000 =  0.000) ( 0.947- 1.000 = -0.053)
```

Appendix F: Tider för testkörningar

Tidtagning av testkörningar har skett med zsh-kommandot time. Nedan följer utskrifter från alla testkörningar.

```
--- Master Thread som separat tråd ---

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Serialiserad version
Nencode32.net_serial.exe > /dev/null  42.45s user 0.05s system 99% cpu 42.525
total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 2
Nencode32.net.exe > /dev/null  45.47s user 2.46s system 192% cpu 24.937 total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 3
Nencode32.net.exe > /dev/null  48.64s user 4.78s system 256% cpu 20.828 total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 4
Nencode32.net.exe > /dev/null  51.32s user 12.42s system 309% cpu 20.595 total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 5
Nencode32.net.exe > /dev/null  51.53s user 13.61s system 330% cpu 19.715 total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 6
Nencode32.net.exe > /dev/null  53.20s user 18.11s system 292% cpu 24.409 total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 7
Nencode32.net.exe > /dev/null  57.57s user 22.45s system 302% cpu 26.439 total

--- Efter Master inbakad i tråd 0 ---

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Serialiserad version
Nencode32.net_serial.exe > /dev/null  42.45s user 0.05s system 99% cpu 42.525
total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 2 LOGFETCH
Nencode32.net.exe > /dev/null  44.96s user 1.79s system 189% cpu 24.731 total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 3 LOGFETCH
Nencode32.net.exe > /dev/null  47.82s user 4.20s system 252% cpu 20.638 total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 4 LOGFETCH
Nencode32.net.exe > /dev/null  50.13s user 10.89s system 316% cpu 19.252 total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 5 LOGFETCH
Nencode32.net.exe > /dev/null  49.87s user 13.14s system 331% cpu 19.018 total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 6 LOGFETCH
Nencode32.net.exe > /dev/null  52.86s user 14.78s system 306% cpu 22.069 total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 7 LOGFETCH
Nencode32.net.exe > /dev/null  58.78s user 19.88s system 310% cpu 25.366 total

---

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Serialiserad version
Nencode32.net_serial.exe > /dev/null  42.45s user 0.05s system 99% cpu 42.525
total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 2 ej LOGFETCH
Nencode32.net.exe > /dev/null  43.12s user 1.15s system 189% cpu 23.375 total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 3 ej LOGFETCH
Nencode32.net.exe > /dev/null  44.39s user 1.44s system 263% cpu 17.368 total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 4 ej LOGFETCH
Nencode32.net.exe > /dev/null  45.62s user 2.86s system 325% cpu 14.888 total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 5 ej LOGFETCH
Nencode32.net.exe > /dev/null  45.93s user 4.10s system 362% cpu 13.786 total
```

```
Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 6 ej LOGFETCH
Nencode32.net.exe > /dev/null  47.62s user 5.51s system 392% cpu 13.546 total

Nät: 32-4-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 7 ej LOGFETCH
Nencode32.net.exe > /dev/null  49.65s user 7.45s system 321% cpu 17.775 total

---

Nät: 32-20-32 Epoker: 10000 Antal exempel: 32 Serialiserad version
Nencode32.net_serial.exe > /dev/null  126.98s user 0.02s system 99% cpu 2:07.06
total

Nät: 32-20-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 2 LOGFETCH
Nencode32.net.exe > /dev/null  134.85s user 1.28s system 192% cpu 1:10.72 total

Nät: 32-20-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 3 LOGFETCH
Nencode32.net.exe > /dev/null  138.05s user 2.88s system 273% cpu 51.567 total

Nät: 32-20-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 4 LOGFETCH
Nencode32.net.exe > /dev/null  143.32s user 7.22s system 348% cpu 43.181 total

Nät: 32-20-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 5 LOGFETCH
Nencode32.net.exe > /dev/null  155.98s user 8.12s system 383% cpu 42.840 total

Nät: 32-20-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 6 LOGFETCH
Nencode32.net.exe > /dev/null  152.83s user 15.08s system 422% cpu 39.788 total

Nät: 32-20-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 7 LOGFETCH
Nencode32.net.exe > /dev/null  160.30s user 14.99s system 325% cpu 53.869 total

---
```


```
Nät: 32-20-32 Epoker: 10000 Antal exempel: 32 Serialiserad version
Nencode32.net_serial.exe > /dev/null  126.98s user 0.02s system 99% cpu 2:07.06
total
```


```
Nät: 32-20-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 2 ej LOGFETCH
Nencode32.net.exe > /dev/null  131.30s user 0.95s system 191% cpu 1:09.12 total

Nät: 32-20-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 3 ej LOGFETCH
Nencode32.net.exe > /dev/null  137.08s user 1.77s system 259% cpu 53.487 total

Nät: 32-20-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 4 ej LOGFETCH
Nencode32.net.exe > /dev/null  138.16s user 3.56s system 336% cpu 42.127 total

Nät: 32-20-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 5 ej LOGFETCH
Nencode32.net.exe > /dev/null  141.44s user 4.41s system 365% cpu 39.950 total

Nät: 32-20-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 6 ej LOGFETCH
Nencode32.net.exe > /dev/null  145.78s user 4.62s system 391% cpu 38.435 total

Nät: 32-20-32 Epoker: 10000 Antal exempel: 32 Antal Trådar: 7 ej LOGFETCH
Nencode32.net.exe > /dev/null  150.38s user 7.81s system 326% cpu 48.496 total

---

Nät: 32-150-32 Epoker: 3000 Antal exempel: 32 Serialiserad version
Nencode32.net_serial.exe > /dev/null  266.39s user 0.03s system 99% cpu 4:26.47
total

Nät: 32-150-32 Epoker: 3000 Antal exempel: 32 Antal Trådar: 2 LOGFETCH
Nencode32.net.exe > /dev/null  288.03s user 0.60s system 191% cpu 2:31.11 total

Nät: 32-150-32 Epoker: 3000 Antal exempel: 32 Antal Trådar: 3 LOGFETCH
Nencode32.net.exe > /dev/null  300.68s user 1.34s system 277% cpu 1:48.99 total

Nät: 32-150-32 Epoker: 3000 Antal exempel: 32 Antal Trådar: 4 LOGFETCH
Nencode32.net.exe > /dev/null  299.30s user 1.69s system 363% cpu 1:22.75 total

Nät: 32-150-32 Epoker: 3000 Antal exempel: 32 Antal Trådar: 5 LOGFETCH
Nencode32.net.exe > /dev/null  308.72s user 3.08s system 403% cpu 1:17.24 total
```

Nät: 32-150-32 Epoker: 3000 Antal exempel: 32 Antal Trådar: 6 LOGFETCH
Nencode32.net.exe > /dev/null 333.39s user 4.02s system 416% cpu 1:20.94 total

Nät: 32-150-32 Epoker: 3000 Antal exempel: 32 Antal Trådar: 7 LOGFETCH
Nencode32.net.exe > /dev/null 333.15s user 4.25s system 326% cpu 1:43.18 total

\---

Nät: 32-150-32 Epoker: 3000 Antal exempel: 32 Serialiserad version
Nencode32.net_serial.exe > /dev/null 266.39s user 0.03s system 99% cpu 4:26.47 total

Nät: 32-150-32 Epoker: 3000 Antal exempel: 32 Antal Trådar: 2 ej LOGFETCH
Nencode32.net.exe > /dev/null 277.17s user 0.33s system 192% cpu 2:24.21 total

Nät: 32-150-32 Epoker: 3000 Antal exempel: 32 Antal Trådar: 3 ej LOGFETCH
Nencode32.net.exe > /dev/null 291.29s user 0.65s system 268% cpu 1:48.92 total

Nät: 32-150-32 Epoker: 3000 Antal exempel: 32 Antal Trådar: 4 ej LOGFETCH
Nencode32.net.exe > /dev/null 293.96s user 0.85s system 343% cpu 1:25.95 total

Nät: 32-150-32 Epoker: 3000 Antal exempel: 32 Antal Trådar: 5 ej LOGFETCH
Nencode32.net.exe > /dev/null 307.04s user 1.88s system 375% cpu 1:22.30 total

Nät: 32-150-32 Epoker: 3000 Antal exempel: 32 Antal Trådar: 6 ej LOGFETCH
Nencode32.net.exe > /dev/null 328.08s user 1.75s system 357% cpu 1:32.37 total

Nät: 32-150-32 Epoker: 3000 Antal exempel: 32 Antal Trådar: 7 ej LOGFETCH
Nencode32.net.exe > /dev/null 325.45s user 2.42s system 323% cpu 1:41.50 total

\---

Nät: 32-300-32 Epoker: 1500 Antal exempel: 32 Serialiserad version
Nencode32.net_serial.exe > /dev/null 268.86s user 0.03s system 99% cpu 4:28.96 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 2 LOGFETCH
Nencode32.net.exe > /dev/null 273.53s user 0.46s system 195% cpu 2:20.46 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 3 LOGFETCH
Nencode32.net.exe > /dev/null 305.95s user 0.84s system 257% cpu 1:59.05 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 4 LOGFETCH
Nencode32.net.exe > /dev/null 321.46s user 1.13s system 350% cpu 1:31.99 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 5 LOGFETCH
Nencode32.net.exe > /dev/null 317.27s user 1.84s system 414% cpu 1:17.01 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 6 LOGFETCH
Nencode32.net.exe > /dev/null 358.37s user 1.94s system 425% cpu 1:24.61 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 7 LOGFETCH
Nencode32.net.exe > /dev/null 360.86s user 2.77s system 339% cpu 1:47.26 total

\---

Nät: 32-300-32 Epoker: 1500 Antal exempel: 32 Serialiserad version
Nencode32.net_serial.exe > /dev/null 268.86s user 0.03s system 99% cpu 4:28.96 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 2 ej LOGFETCH
Nencode32.net.exe > /dev/null 276.03s user 0.35s system 188% cpu 2:26.78 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 3 ej LOGFETCH
Nencode32.net.exe > /dev/null 297.45s user 0.46s system 260% cpu 1:54.40 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 4 ej LOGFETCH
Nencode32.net.exe > /dev/null 314.85s user 0.70s system 343% cpu 1:31.95 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 5 ej LOGFETCH
Nencode32.net.exe > /dev/null 313.44s user 0.72s system 366% cpu 1:25.82 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 6 ej LOGFETCH

```
Nencode32.net.exe > /dev/null  353.49s user 1.14s system 353% cpu 1:40.44 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 7 ej LOGFETCH
Nencode32.net.exe > /dev/null  355.74s user 1.78s system 299% cpu 1:59.36 total

---

Nät: 32-600-32 Epoker: 1500 Antal exempel: 32 Serialiserad version
Nencode32.net_serial.exe > /dev/null  611.75s user 0.06s system 99% cpu 10:11.85
total

Nät: 32-600-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 2 LOGFETCH
Nencode32.net.exe > /dev/null  647.91s user 0.49s system 193% cpu 5:35.61 total

Nät: 32-600-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 3 LOGFETCH
Nencode32.net.exe > /dev/null  702.55s user 1.00s system 266% cpu 4:24.14 total

Nät: 32-600-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 4 LOGFETCH
Nencode32.net.exe > /dev/null  725.82s user 1.31s system 349% cpu 3:27.85 total

Nät: 32-600-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 5 LOGFETCH
Nencode32.net.exe > /dev/null  745.35s user 1.95s system 403% cpu 3:05.11 total

Nät: 32-600-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 6 LOGFETCH
Nencode32.net.exe > /dev/null  753.37s user 2.22s system 314% cpu 3:59.96 total

Nät: 32-600-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 7 LOGFETCH
Nencode32.net.exe > /dev/null  782.68s user 3.06s system 327% cpu 3:59.87 total

---

Nät: 32-600-32 Epoker: 1500 Antal exempel: 32 Serialiserad version
Nencode32.net_serial.exe > /dev/null  611.75s user 0.06s system 99% cpu 10:11.85
total

Nät: 32-600-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 2 ej LOGFETCH
Nencode32.net.exe > /dev/null  644.85s user 0.30s system 191% cpu 5:37.75 total

Nät: 32-600-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 3 ej LOGFETCH
Nencode32.net.exe > /dev/null  726.92s user 0.32s system 261% cpu 4:37.72 total

Nät: 32-600-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 4 ej LOGFETCH
Nencode32.net.exe > /dev/null  729.78s user 0.67s system 322% cpu 3:46.82 total

Nät: 32-600-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 5 ej LOGFETCH
Nencode32.net.exe > /dev/null  718.21s user 1.18s system 340% cpu 3:31.07 total

Nät: 32-600-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 6 ej LOGFETCH
Nencode32.net.exe > /dev/null  760.32s user 1.23s system 294% cpu 4:18.46 total

Nät: 32-600-32 Epoker: 1500 Antal exempel: 32 Antal Trådar: 7 ej LOGFETCH
Nencode32.net.exe > /dev/null  788.75s user 1.67s system 288% cpu 4:33.99 total

---

Nät: 32-900-32 Epoker: 1000 Antal exempel: 32 Serialiserad version
Nencode32.net_serial.exe > /dev/null  635.72s user 0.03s system 99% cpu 10:35.79

Nät: 32-900-32 Epoker: 1000 Antal exempel: 32 Antal Trådar: 2 LOGFETCH
Nencode32.net.exe > /dev/null  689.23s user 0.46s system 195% cpu 5:52.38 total

Nät: 32-900-32 Epoker: 1000 Antal exempel: 32 Antal Trådar: 3 LOGFETCH
Nencode32.net.exe > /dev/null  726.53s user 0.79s system 268% cpu 4:30.68 total

Nät: 32-900-32 Epoker: 1000 Antal exempel: 32 Antal Trådar: 4 LOGFETCH
Nencode32.net.exe > /dev/null  740.58s user 1.14s system 356% cpu 3:27.95 total

Nät: 32-900-32 Epoker: 1000 Antal exempel: 32 Antal Trådar: 5 LOGFETCH
Nencode32.net.exe > /dev/null  790.25s user 1.50s system 380% cpu 3:28.36 total

Nät: 32-900-32 Epoker: 1000 Antal exempel: 32 Antal Trådar: 6 LOGFETCH
Nencode32.net.exe > /dev/null  817.62s user 1.69s system 420% cpu 3:14.71 total
```

```
Nät: 32-900-32 Epoker: 1000 Antal exempel: 32 Antal Trådar: 7 LOGFETCH
Nencode32.net.exe > /dev/null  839.26s user 2.60s system 351% cpu 3:59.52 total

---

Nät: 32-900-32 Epoker: 1000 Antal exempel: 32 Serialiserad version
Nencode32.net_serial.exe > /dev/null  635.72s user 0.03s system 99% cpu 10:35.79

Nät: 32-900-32 Epoker: 1000 Antal exempel: 32 Antal Trådar: 2 ej LOGFETCH
Nencode32.net.exe > /dev/null  696.91s user 0.31s system 190% cpu 6:06.41 total

Nät: 32-900-32 Epoker: 1000 Antal exempel: 32 Antal Trådar: 3 ej LOGFETCH
Nencode32.net.exe > /dev/null  742.74s user 0.33s system 261% cpu 4:43.72 total

Nät: 32-900-32 Epoker: 1000 Antal exempel: 32 Antal Trådar: 4 ej LOGFETCH
Nencode32.net.exe > /dev/null  760.01s user 0.59s system 328% cpu 3:51.21 total

Nät: 32-900-32 Epoker: 1000 Antal exempel: 32 Antal Trådar: 5 ej LOGFETCH
Nencode32.net.exe > /dev/null  823.71s user 0.76s system 343% cpu 3:59.68 total

Nät: 32-900-32 Epoker: 1000 Antal exempel: 32 Antal Trådar: 6 ej LOGFETCH
Nencode32.net.exe > /dev/null  840.71s user 1.23s system 350% cpu 4:00.25 total

Nät: 32-900-32 Epoker: 1000 Antal exempel: 32 Antal Trådar: 7 ej LOGFETCH
Nencode32.net.exe > /dev/null  855.56s user 1.08s system 299% cpu 4:45.83 total

--- 128 exempel ---

Nät: 32-4-32 Epoker: 5000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null  85.07s user 0.08s system 99% cpu
1:25.30 total

Nät: 32-4-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 2 LOGFETCH
Nencode32_stor.net.exe > /dev/null  92.75s user 0.76s system 180% cpu 51.812
total

Nät: 32-4-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 3 LOGFETCH
Nencode32_stor.net.exe > /dev/null  85.39s user 2.12s system 277% cpu 31.556
total

Nät: 32-4-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 4 LOGFETCH
Nencode32_stor.net.exe > /dev/null  84.62s user 4.33s system 362% cpu 24.555
total

Nät: 32-4-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null  88.09s user 6.17s system 416% cpu 22.650
total

Nät: 32-4-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 6 LOGFETCH
Nencode32_stor.net.exe > /dev/null  91.22s user 6.82s system 373% cpu 26.237
total

Nät: 32-4-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 7 LOGFETCH
Nencode32_stor.net.exe > /dev/null  91.53s user 9.74s system 330% cpu 30.597
total

---

Nät: 32-4-32 Epoker: 5000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null  85.07s user 0.08s system 99% cpu
1:25.30 total

Nät: 32-4-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 2 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null  84.95s user 0.55s system 191% cpu 44.540
total

Nät: 32-4-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 3 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null  83.30s user 0.66s system 282% cpu 29.673
total

Nät: 32-4-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 4 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null  83.65s user 1.46s system 367% cpu 23.132
total
```

Nät: 32-4-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 86.14s user 3.02s system 446% cpu 19.958 total

Nät: 32-4-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 6 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 87.01s user 2.12s system 442% cpu 20.151 total

Nät: 32-4-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 7 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 87.95s user 3.61s system 336% cpu 27.247 total

---

Nät: 32-20-32 Epoker: 5000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 250.42s user 0.05s system 99% cpu 4:10.63 total

Nät: 32-20-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 2 LOGFETCH
Nencode32_stor.net.exe > /dev/null 256.04s user 1.01s system 195% cpu 2:11.76 total

Nät: 32-20-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 3 LOGFETCH
Nencode32_stor.net.exe > /dev/null 252.44s user 1.52s system 290% cpu 1:27.50 total

Nät: 32-20-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 4 LOGFETCH
Nencode32_stor.net.exe > /dev/null 257.82s user 3.55s system 374% cpu 1:09.71 total

Nät: 32-20-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 261.36s user 3.89s system 458% cpu 57.846 total

Nät: 32-20-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 6 LOGFETCH
Nencode32_stor.net.exe > /dev/null 259.86s user 9.09s system 517% cpu 51.954 total

Nät: 32-20-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 7 LOGFETCH
Nencode32_stor.net.exe > /dev/null 266.00s user 7.74s system 336% cpu 1:21.31 total

---

Nät: 32-20-32 Epoker: 5000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 250.42s user 0.05s system 99% cpu 4:10.63 total

Nät: 32-20-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 2 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 255.54s user 0.56s system 194% cpu 2:11.55 total

Nät: 32-20-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 3 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 251.93s user 0.79s system 292% cpu 1:26.48 total

Nät: 32-20-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 4 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 253.01s user 1.14s system 372% cpu 1:08.15 total

Nät: 32-20-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 257.44s user 2.14s system 454% cpu 57.118 total

Nät: 32-20-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 6 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 230.61s user 3.72s system 462% cpu 50.659 total

Nät: 32-20-32 Epoker: 5000 Antal exempel: 128 Antal Trådar: 7 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 257.27s user 4.23s system 339% cpu 1:16.96 total

---

Nät: 32-100-32 Epoker: 2000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 480.24s user 0.02s system 99% cpu 8:00.35 total

Nät: 32-100-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 2 LOGFETCH
Nencode32_stor.net.exe > /dev/null 473.43s user 0.50s system 197% cpu 3:59.63 total

Nät: 32-100-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 3 LOGFETCH
Nencode32_stor.net.exe > /dev/null 489.62s user 0.91s system 291% cpu 2:48.03 total

Nät: 32-100-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 4 LOGFETCH
Nencode32_stor.net.exe > /dev/null 486.76s user 1.81s system 383% cpu 2:07.33 total

Nät: 32-100-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 496.66s user 2.18s system 454% cpu 1:49.86 total

Nät: 32-100-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 6 LOGFETCH
Nencode32_stor.net.exe > /dev/null 492.40s user 2.12s system 495% cpu 1:39.89 total

Nät: 32-100-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 7 LOGFETCH
Nencode32_stor.net.exe > /dev/null 496.05s user 3.18s system 347% cpu 2:23.80 total

---

Nät: 32-100-32 Epoker: 2000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 480.24s user 0.02s system 99% cpu 8:00.35 total

Nät: 32-100-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 2 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 473.39s user 0.42s system 197% cpu 4:00.08 total

Nät: 32-100-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 3 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 487.75s user 0.57s system 288% cpu 2:49.33 total

Nät: 32-100-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 4 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 491.28s user 0.61s system 379% cpu 2:09.62 total

Nät: 32-100-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 497.04s user 1.07s system 452% cpu 1:50.00 total

Nät: 32-100-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 6 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 501.12s user 1.82s system 517% cpu 1:37.15 total

Nät: 32-100-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 7 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 493.60s user 1.82s system 348% cpu 2:22.34 total

---

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 723.03s user 0.13s system 99% cpu 12:03.24 total

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 2 LOGFETCH
Nencode32_stor.net.exe > /dev/null 729.15s user 0.39s system 195% cpu 6:13.69 total

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 3 LOGFETCH
Nencode32_stor.net.exe > /dev/null 751.62s user 1.15s system 292% cpu 4:16.94 total

```
Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 4 LOGFETCH
Nencode32_stor.net.exe > /dev/null  740.48s user 1.36s system 374% cpu 3:17.85
total

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null  743.56s user 2.62s system 452% cpu 2:44.83
total

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 6 LOGFETCH
Nencode32_stor.net.exe > /dev/null  773.16s user 2.62s system 490% cpu 2:38.07
total

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 7 LOGFETCH
Nencode32_stor.net.exe > /dev/null  777.71s user 3.25s system 357% cpu 3:38.48
total

---

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null  714.77s user 0.06s system 99% cpu
11:54.98 total

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 2 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null  732.77s user 0.35s system 197% cpu 6:10.78
total

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 3 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null  756.41s user 0.52s system 293% cpu 4:18.26
total

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 4 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null  726.40s user 1.13s system 381% cpu 3:10.76
total

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null  752.24s user 1.09s system 453% cpu 2:46.02
total

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 6 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null  769.28s user 1.51s system 453% cpu 2:49.81
total

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 7 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null  771.10s user 2.24s system 353% cpu 3:39.04
total

---

Nät: 32-300-32 Epoker: 1500 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null  1068.75s user 0.06s system 99% cpu
17:48.95 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 128 Antal Trådar: 2 LOGFETCH
Nencode32_stor.net.exe > /dev/null  1069.16s user 0.56s system 198% cpu 9:00.26
total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 128 Antal Trådar: 3 LOGFETCH
Nencode32_stor.net.exe > /dev/null  1170.17s user 1.10s system 273% cpu 7:07.72
total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 128 Antal Trådar: 4 LOGFETCH
Nencode32_stor.net.exe > /dev/null  1195.52s user 1.41s system 367% cpu 5:25.32
total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null  1156.82s user 2.15s system 463% cpu 4:10.22
total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 128 Antal Trådar: 6 LOGFETCH
Nencode32_stor.net.exe > /dev/null  1258.88s user 2.02s system 500% cpu 4:11.81
total
```

Nät: 32-300-32 Epoker: 1500 Antal exempel: 128 Antal Trådar: 7 LOGFETCH
Nencode32_stor.net.exe > /dev/null 1272.31s user 2.92s system 380% cpu 5:35.04 total

---

Nät: 32-300-32 Epoker: 1500 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 1068.75s user 0.06s system 99% cpu 17:48.95 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 128 Antal Trådar: 2 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 1070.45s user 0.38s system 196% cpu 9:05.37 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 128 Antal Trådar: 3 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 1141.06s user 0.40s system 277% cpu 6:52.06 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 128 Antal Trådar: 4 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 1175.25s user 0.79s system 366% cpu 5:21.32 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 1147.61s user 0.93s system 449% cpu 4:15.58 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 128 Antal Trådar: 6 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 1250.24s user 1.43s system 462% cpu 4:30.60 total

Nät: 32-300-32 Epoker: 1500 Antal exempel: 128 Antal Trådar: 7 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 1231.76s user 2.45s system 361% cpu 5:41.24 total

---

Nät: 32-600-32 Epoker: 1500 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 2420.37s user 0.05s system 99% cpu 40:20.58 total

---

Nät: 32-900-32 Epoker: 1000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 2557.37s user 12.06s system 99% cpu 42:49.57 total

---

Nät: 32-900-32 Epoker: 1000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 2557.37s user 12.06s system 99% cpu 42:49.57 total

Nät: 32-900-32 Epoker: 1000 Antal exempel: 128 Antal Trådar: 2 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 2731.08s user 6.22s system 196% cpu 23:09.76 total

--- Hur påverkar antalet epoker effektiviteten? ---

Nät: 32-20-32 Epoker: 500 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 25.33s user 0.07s system 99% cpu 25.473 total

Nät: 32-20-32 Epoker: 500 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 26.42s user 0.57s system 437% cpu 6.169 total

Nät: 32-20-32 Epoker: 500 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 26.05s user 0.46s system 433% cpu 6.114 total

-

Nät: 32-20-32 Epoker: 1000 Antal exempel: 128 Serialiserad version

Nencode32_stor.net_serial.exe > /dev/null 49.93s user 0.08s system 99% cpu 50.070 total

Nät: 32-20-32 Epoker: 1000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 53.21s user 1.12s system 449% cpu 12.096 total

Nät: 32-20-32 Epoker: 1000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 52.39s user 0.54s system 447% cpu 11.833 total

-

Nät: 32-20-32 Epoker: 2000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 100.79s user 0.03s system 99% cpu 1:40.95 total

Nät: 32-20-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 105.21s user 1.73s system 444% cpu 24.047 total

Nät: 32-20-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 103.82s user 0.98s system 448% cpu 23.375 total

-

Nät: 32-20-32 Epoker: 3000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 151.11s user 0.04s system 99% cpu 2:31.28 total

Nät: 32-20-32 Epoker: 3000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 158.21s user 3.05s system 454% cpu 35.509 total

Nät: 32-20-32 Epoker: 3000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 155.38s user 1.54s system 448% cpu 34.996 total

-

Nät: 32-20-32 Epoker: 10000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 500.87s user 0.05s system 99% cpu 8:21.01 total

Nät: 32-20-32 Epoker: 10000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 515.88s user 4.33s system 447% cpu 1:56.35 total

Nät: 32-20-32 Epoker: 10000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 521.17s user 8.51s system 445% cpu 1:58.79 total

---

Nät: 32-150-32 Epoker: 500 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 179.40s user 0.11s system 99% cpu 2:59.58 total

Nät: 32-150-32 Epoker: 500 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 184.65s user 0.72s system 455% cpu 40.687 total

Nät: 32-150-32 Epoker: 500 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 188.17s user 0.56s system 456% cpu 41.309 total

-

Nät: 32-150-32 Epoker: 1000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 358.88s user 0.11s system 99% cpu 5:59.09 total

Nät: 32-150-32 Epoker: 1000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 368.82s user 1.28s system 457% cpu 1:20.98 total

Nät: 32-150-32 Epoker: 1000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 373.82s user 0.75s system 455% cpu 1:22.31 total

-

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 714.77s user 0.06s system 99% cpu 11:54.98 total

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 743.56s user 2.62s system 452% cpu 2:44.83 total

Nät: 32-150-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 752.24s user 1.09s system 453% cpu 2:46.02 total

-

Nät: 32-150-32 Epoker: 3000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 1065.46s user 0.06s system 99% cpu 17:45.69 total

Nät: 32-150-32 Epoker: 3000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 1111.26s user 3.42s system 448% cpu 4:08.62 total

Nät: 32-150-32 Epoker: 3000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 1125.15s user 1.91s system 452% cpu 4:08.95 total

-

Nät: 32-150-32 Epoker: 10000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 3596.02s user 0.11s system 99% cpu 59:56.13 total

Nät: 32-150-32 Epoker: 10000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 3703.82s user 10.36s system 461% cpu 13:25.06 total

Nät: 32-150-32 Epoker: 10000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 3741.11s user 5.77s system 458% cpu 13:36.85 total

---

Nät: 32-300-32 Epoker: 500 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 359.28s user 0.11s system 99% cpu 5:59.45 total

Nät: 32-300-32 Epoker: 500 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 384.58s user 0.60s system 463% cpu 1:23.19 total

Nät: 32-300-32 Epoker: 500 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 384.60s user 0.36s system 445% cpu 1:26.42 total

-

Nät: 32-300-32 Epoker: 1000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 722.75s user 0.03s system 99% cpu 12:03.08 total

Nät: 32-300-32 Epoker: 1000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 770.33s user 1.05s system 461% cpu 2:47.02 total

Nät: 32-300-32 Epoker: 1000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 762.38s user 0.75s system 449% cpu 2:49.79 total

-

Nät: 32-300-32 Epoker: 2000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 1444.19s user 0.08s system 99% cpu 24:04.36 total

Nät: 32-300-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 1539.79s user 2.34s system 463% cpu 5:32.57 total

Nät: 32-300-32 Epoker: 2000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 1524.40s user 1.00s system 449% cpu 5:39.38 total

-

Nät: 32-300-32 Epoker: 3000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 2135.01s user 0.08s system 99% cpu 35:35.22 total

Nät: 32-300-32 Epoker: 3000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 2316.15s user 3.20s system 460% cpu 8:23.13 total

Nät: 32-300-32 Epoker: 3000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 2305.22s user 2.02s system 450% cpu 8:32.60 total

-

Nät: 32-300-32 Epoker: 10000 Antal exempel: 128 Serialiserad version
Nencode32_stor.net_serial.exe > /dev/null 7233.86s user 0.08s system 100% cpu 2:00:31.88 total

Nät: 32-300-32 Epoker: 10000 Antal exempel: 128 Antal Trådar: 5 LOGFETCH
Nencode32_stor.net.exe > /dev/null 7713.41s user 10.48s system 464% cpu 27:42.79 total

Nät: 32-300-32 Epoker: 10000 Antal exempel: 128 Antal Trådar: 5 ej LOGFETCH
Nencode32_stor.net.exe > /dev/null 7629.01s user 4.63s system 445% cpu 28:32.84 total

--- XOR ---

Nät: 2-2-1 Epoker: 10000 Antal exempel: 240 Serialiserad version
XOR_stor.net_serial.exe > /dev/null 21.80s user 0.03s system 99% cpu 21.839 total

Nät: 2-2-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 2 LOGFETCH
XOR_stor.net.exe > /dev/null 26.63s user 1.66s system 189% cpu 14.931 total

Nät: 2-2-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 3 LOGFETCH
XOR_stor.net.exe > /dev/null 29.79s user 5.18s system 254% cpu 13.716 total

Nät: 2-2-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 4 LOGFETCH
XOR_stor.net.exe > /dev/null 32.08s user 12.39s system 304% cpu 14.608 total

Nät: 2-2-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 5 LOGFETCH
XOR_stor.net.exe > /dev/null 31.69s user 15.52s system 307% cpu 15.350 total

Nät: 2-2-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 6 LOGFETCH
XOR_stor.net.exe > /dev/null 34.17s user 18.71s system 298% cpu 17.745 total

Nät: 2-2-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 7 LOGFETCH
XOR_stor.net.exe > /dev/null 30.04s user 27.30s system 293% cpu 19.504 total

---

```
Nät: 2-2-1 Epoker: 10000 Antal exempel: 240 Serialiserad version
XOR_stor.net_serial.exe > /dev/null  21.80s user 0.03s system 99% cpu 21.839
total

Nät: 2-2-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 2 ej LOGFETCH
XOR_stor.net.exe > /dev/null  26.73s user 0.75s system 191% cpu 14.369 total

Nät: 2-2-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 3 ej LOGFETCH
XOR_stor.net.exe > /dev/null  28.76s user 1.44s system 269% cpu 11.219 total

Nät: 2-2-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 4 ej LOGFETCH
XOR_stor.net.exe > /dev/null  31.17s user 2.81s system 311% cpu 10.908 total

Nät: 2-2-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 5 ej LOGFETCH
XOR_stor.net.exe > /dev/null  30.27s user 4.25s system 385% cpu 8.951 total

Nät: 2-2-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 6 ej LOGFETCH
XOR_stor.net.exe > /dev/null  29.87s user 5.44s system 348% cpu 10.120 total

Nät: 2-2-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 7 ej LOGFETCH
XOR_stor.net.exe > /dev/null  29.17s user 7.10s system 348% cpu 10.395 total

--

Nät: 2-100-1 Epoker: 10000 Antal exempel: 240 Serialiserad version
XOR_stor.net_serial.exe > /dev/null  626.84s user 0.03s system 100% cpu 10:26.87
total

Nät: 2-100-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 2 LOGFETCH
XOR_stor.net.exe > /dev/null  561.13s user 1.15s system 192% cpu 4:52.33 total

Nät: 2-100-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 3 LOGFETCH
XOR_stor.net.exe > /dev/null  561.73s user 2.53s system 290% cpu 3:14.30 total

Nät: 2-100-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 4 LOGFETCH
XOR_stor.net.exe > /dev/null  576.71s user 5.18s system 383% cpu 2:31.77 total

Nät: 2-100-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 5 LOGFETCH
XOR_stor.net.exe > /dev/null  581.82s user 9.65s system 466% cpu 2:06.82 total

Nät: 2-100-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 6 LOGFETCH
XOR_stor.net.exe > /dev/null  579.22s user 20.18s system 505% cpu 1:58.54 total

Nät: 2-100-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 7 LOGFETCH
XOR_stor.net.exe > /dev/null  599.89s user 13.58s system 338% cpu 3:01.10 total

--

Nät: 2-100-1 Epoker: 10000 Antal exempel: 240 Serialiserad version
XOR_stor.net_serial.exe > /dev/null  626.84s user 0.03s system 100% cpu 10:26.87
total

Nät: 2-100-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 2 ej LOGFETCH
XOR_stor.net.exe > /dev/null  570.60s user 0.72s system 191% cpu 4:58.49 total

Nät: 2-100-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 3 ej LOGFETCH
XOR_stor.net.exe > /dev/null  561.68s user 1.28s system 293% cpu 3:11.54 total

Nät: 2-100-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 4 ej LOGFETCH
XOR_stor.net.exe > /dev/null  567.94s user 2.01s system 383% cpu 2:28.47 total

Nät: 2-100-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 5 ej LOGFETCH
XOR_stor.net.exe > /dev/null  568.62s user 4.31s system 477% cpu 2:00.04 total

Nät: 2-100-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 6 ej LOGFETCH
XOR_stor.net.exe > /dev/null  578.11s user 5.84s system 550% cpu 1:46.01 total

Nät: 2-100-1 Epoker: 10000 Antal exempel: 240 Antal Trådar: 7 ej LOGFETCH
XOR_stor.net.exe > /dev/null  564.93s user 6.56s system 345% cpu 2:45.53 total
```