PARALLELLISM INOM PROGRAMMERINGSSPRÅKET FORTRESS

Jonatan Wiik

Kandidatavhandling i datateknik
Handledare: Mats Aspnäs
Institutionen för informationsteknologi
Åbo Akademi
29 mars 2010

Referat

I den här avhandlingen presenteras Fortress – ett nytt parallellt programmeringsspråk ämnat för vetenskapliga beräkningar. Språket och dess egenskaper och särdrag, så som dess matematiska syntax och utvidgningsbara biblioteksystem, presenteras först allmänt. Avhandlingen koncentrerar sig därefter på språkets implicita parallella egenskaper och hur dessa förverkligas i språket genom att bygga in uttryck som automatiskt exekveras parallellt. Språkets minneshantering med stöd för transaktioner presenteras också. Till slut ges ett kort exempelprogram för att illustrera språkets kodskrivningsprocess och syntaktiska egenskaper.

Nyckelord: Fortress, programmeringsspråk, parallellprogrammering, implicit parallellism

Innehåll

1. Inledning

Den ständigt pågående utvecklingen av datorer har lett till att vi idag har processorer med flera kärnor och klockfrekvenser på flera gigahertz. I dag kan en hemdator uppnå cirka 15 gigaflops i beräkningskapacitet med en tvåkärnig processor. För att få ut bästa prestanda ur det växande antalet högeffektiva datorer krävs det nya, mera sofistikerade programmeringsspråk.

Defense Advanced Research Projects Agency (DARPA) i USA har förutsett behovet av bättre parallella programmeringsspråk och har därför bland annat finansierat projektet High Productivity Computing Systems. Projektets mål är att skapa en ny generation av högproduktiva datorsystem [1]. Ett av de huvudsakliga målen med projektet är att skapa ett högproduktivt programmeringsspråk, som involverar: hög prestanda för effektiv användning av datasystem, hög programmerbarhet för effektiv utveckling av programvara och hög portabilitet för att lätt kunna köra programvaran på olika plattformar. Under det här projektet har tre nya programmeringsspråk utvecklats: Chapel, X10 och Fortress. DARPA slutade finansiera utvecklingen av Fortress efter fas två av projektet, men utvecklingen har ändå fortsatt i form av ett open source-projekt under ledning av Programming Language Research Group vid Sun Microsystems.

Den här avhandlingen presenterar Fortress. Programmeringsspråket beskrivs på ett allmänt plan där dess särdrag och grundbegrepp presenteras,varefter avhandlingen koncentrerar sig på språkets parallella egenskaper och den funktionalitet som stöder dessa egenskaper. Till sist presenteras ett kort exempelprogram för att demonstrera språkets egenskaper och funktionalitet.

Utvecklingen av Fortress är ännu i ett tidigt skede och all planerad funktionalitet är därför ännu inte implementerad i den nuvarande versionen av språket. Det här innebär att vissa funktioner som beskrivs i den här avhandlingen inte nödvändigtvis ännu är implementerade och kan komma att ändras på ett betydande sätt i framtiden.

2. Fortress som programmeringsspråk

Fortress är ett modernt, komponentbaserat programmeringsspråk, som är designat för att producera högeffektiv mjukvara med hög programmerbarhet. Program skrivna i språket ska kunna köras på stora datorsystem och dess huvudsakliga användningsområde är framför allt vetenskapliga beräkningar, men det lämpar sig även för mer allmänna tillämpningar. Namnet Fortress kommer från att ett av huvudmålen med språket är att skapa ett ”säkert Fortran”, men trots detta är språket helt nytt och har inte mycket gemensamt med Fortran förutom det avsedda användningsområdet och det har inte gjorts några försök att göra Fortress bakåtkompatibelt med Fortran [2].

2.1 Egenskaper och särdrag

Fortress skiljer sig i flera avseenden från många andra programmeringsspråk, men en av av språkets viktigaste egenskaper är dess stöd för parallell exekvering. Till skillnad från de flesta andra programmeringsspråk, som utvidgats med stöd för parallell exekvering, så sker ekveringen parallellt som standard i Fortress.

En annan egenskap som skiljer Fortress från andra programmeringsspråk är syntaxen, som påminner väldigt långt om matematisk notation. Genom att skapa en sådan syntax har man försökt göra det lättare att formulera matematiska problem i form av programkod och på så vis göra språket attraktivt för tillämpningar relaterade till vetenskapliga beräkningar [2].

Utvecklarna av Fortress har haft som mål att skapa ett så öppet och utvidgbart språk som möjligt. Därför är all funktionalitet, förutom grundläggande funktioner och datatyper, implementerade i form av bibliotek istället för att bygga in dem i kompilatorn [3].

2.2 Syntax

Eftersom Fortress är ett programmeringsspråk främst ämnat för vetenskapliga tillämpning så har man skapat en syntax som påminner till stor utsträckning om matematisk notation. Syftet med syntaxen är att den skall vara lättläst, samtidigt som steget mellan algoritm och källkod är så litet som möjligt. Syntaxen passar utmärkt för vetenskapliga beräkningar, men kan vara något begränsande i mer allmänna tillämp-

```
for i <- 1:n, j <- 1:n do
  a[i,j] =
    SUM [k <- 1:n] b[i,k] c[k,j]
end
```

```
for i <- 1:n, j <- 1:n do
  a[i,j] =
    ∑ [k <- 1:n] b[i,k] c[k,j]
end
```

for $i \leftarrow 1:n, j \leftarrow 1:n$ do

$$a_{i,j} = \sum_{k \leftarrow 1:n} b_{i,k}\, c_{k,j}$$

end

Figur 2.1: Illustration av de olika formaten som man kan presentera Fortress-kod på, i ordningen: ASCII, Unicode, LaTeX [2].

ningar.

Fortress tillåter att Unicode-tecken, så som $\sum$, används i koden. Varje tillåtet Unicode-tecken har också en motsvarande ASCII-variant. Fortress distribueras dessutom med ett verktyg, kallat Fortify, som gör det möjligt att konvertera Fortress-kod till LaTeX-format [4]. Fortress-kod kan alltså presenteras i tre olika format: ASCII-, Unicode- och LaTeX-format. En illustration av dessa format finns i figur 2.1. LaTeX-formatet har den fördelen att koden blir lättläst och lätt att förstå ur matematisk synvinkel. Formatet kan dock skapa problem ifall man vill använda sig av programkod från ett exempel givet i LaTeX-format, eftersom det inte alltid är uppenbart hur man ska representera vissa tecken i ASCII-format.

Som man ser i figur 2.1 så representeras multiplikation i Fortress-kod av juxtaposition, precis som i matematiken. Juxtaposition är i själva verket en operator som alla andra och kan också överbelastas (overload), som alla andra operatorer i Fortress.

2.3 Typsystem och grundbegrepp

Fortress stöder flera olika programmeringsparadigmer med hjälp av ett välutvecklat typsystem. Det är främst ett objektorienterat språk, men har också stöd för funktionell programmering.

2.3.1 Objekt och traits

Fortress typsystem baserar sig på så kallade *traits*, som kan jämföras med gränssnitt i Java, och *objekt*, som kan jämföras med klasser i Java. Traits skiljer sig från Java-gränssnitt på det viset att de kan innehålla programkod. Dessutom stöder traits multipel arv, det vill säga, ett trait eller objekt kan ärva flera traits [3]. Kommunikation mellan objekt sker endast via traits, ett objekt kan alltså inte direkt kommunicera med ett annat objekt.

2.3.2 Komponeneter och API:n

Fortress-program organiseras i *komponenter* (components) och *API:n*. Ett API beskriver en komponent och den funktionalitet den tillhandahåller, så som objekt, traits och funktioner. Alla externa referenser i en komponent är till API:n, som importeras av den refererande komponenten [4].

Komponenter kan importera och exportera API:n. Att exportera ett API betyder att komponenten implementerar alla de begrepp som definieras i API:t [4]. Ett exempel är det här ”Hello world”-programmet:

```
component HelloWorld
export Executable
  run():() = println(”Hello, world!”)
end
```

Programmet består av en komponent `HelloWorld`, som exporterar API:t Executable, som definierar en enda funktion, `run`. Funktionen implementeras i det här exemplet så att den skriver ut strängen ”Hello, world!”.

Att importera ett API innebär att komponenten kan använda sig av de begrepp som definieras i API:t [4]. Till exempel kan man importera ett API för att få tillgång till funktionalitet så som listor.

2.3.3 Variabler och aggregatuttryck

Fortress har stöd för både föränderliga (mutable) och oföränderliga (immutable) variabler [4]. En oföränderlig variabel deklareras på något av följande två sätt:

```
namn:Typ = uttr
namn = uttr
```

I det senare fallet så blir variabelns typ samma som typen på `uttr`, medan det i det andra fallet räcker med att `uttr` är en subtyp av `Typ`. Föränderliga variabler deklareras i sin tur på något av följande två sätt:

Tabell 2.1: Sammanställning av några olika aggregatuttryck i Fortress

Typ	Syntax
Räcka, Vektor	`[0 1 2 3 4]`
2-dimensionell räcka, Matris	`[1 2; 3 4]`
Mängd	`{0,1,2,3,4}`
Associativ räcka	`{"a" |-> 0, "b" |-> 1, "c" |-> 2}`
Lista	`<|0,1,2,3,4|>`

```
var namn:Typ = uttr
namn:Typ := uttr
```

Numeriska datatyper kan förses med enheter och dimensioner, som finns inbyggda i Fortress. Fortress kan då exempelvis garantera att resultatet från en funktion har rätt enhet, som i följande exempel:

```
kinetiskEnergi(m:RR64 kg_,
               v:RR64 m_/s_
              ):RR64 kg_ m_^2/s_^2
  = (m v^2)/2
```

I likhet med matematisk notation har Fortress syntaktiskt stöd för att skriva ner flera typer av datasamlingar, så som räckor, matriser, vektorer, mängder och listor [4]. En sammanställning av några av de vanligaste datasamlingarna finns i tabell 2.1.

2.4 Utvidgbart språk

Utvecklarna av Fortress har haft som mål att skapa ett så öppet och utvidgbart språk som möjligt. Därför är all funktionalitet, utom grundläggande funktioner och datatyper, implementerad i form av bibliotek istället för i kompilatorn. Således består Fortress av en liten ”kärna” och en stor mängd bibliotek, som kan implementera så grundläggande funktionalitet som 32-bitars heltal eller for-loopar [3].

För att man skall kunna definiera grundläggande uttryck och datatyper i bibliotek krävs det att bibliotekutvecklare ges omfattande kontroll över både språkets syntax och semantik, vilket stöds av Fortress välutvecklade typsystem. Dessutom ges bibliotekutvecklare kontroll över strukturer som normalt sätt är interna i en kompilator, så som det abstrakta syntaxträdet (AST) som fås när man parsar ett program [3]. Dessa strukturer kan användas bland annat till optimeringar, som i de flesta språk sköts av kompilatorn, men som i Fortress måste göras i biblioteken.

Biblioteken stöder målsättningen att kunna utvidga språket. Eftersom funktionalitet implementeras i form av bibliotek istället för att byggas in i kompilatorn kan programmerare på ett enkelt sätt anpassa och förbättra Fortress, eller lägga till ny funktionalitet. Det här kan förenkla processen att ta i bruk Fortress som teknologi och stöder programmerares behov att utvecklas.

3. Parallellism

En av de största skillnaderna mellan Fortress och många andra programmeringsspråk är hur parallellism hanteras. I motsats till många andra programmeringsspråk, som utvidgats med stöd för parallell exekvering, så är Fortress designat redan från början för att exekvering ska ske parallellt. Det här betyder att exekvering automatiskt sker parallellt när det är möjligt. Den här typen av parallellism kallas för *implicit parallellism*.

3.1 Implicit parallellism

Fortress är implicit parallellt, vilket betyder att kompilatorn eller programtolken automatiskt avgör när exekvering ska ske parallellt. Det här låter programmeraren koncentrera sig på problemet han skall lösa som sådant, istället för att koncentrera sig på hur det kan parallelliseras. En annan fördel med implicit parallellism är att parallelliseringen separeras från själva algoritmen. Dessutom medför implicit parallellism också minskad mängd programkod och ökad produktivitet [5].

Fortress specificerar när evaluering kan ske parallellt, men kräver inte att det i själva verket sker parallellt. Om det exempelvis inte finns någon ledig processor eller om det förbrukar mera resurser att distribuera problemet än att lösa det så kan exekveringen av parallella uppgifter ske sekventiellt [6]. I de fall då det är nödvändigt kan programmeraren explicit ange att exekvering ska ske parallellt ifall det finns behov för det.

3.2 Trådar

Det finns två olika typer av trådar i Fortress: *implicita* och *explicita* [4]. Explicita trådar skapas av programmeraren med funktionen `spawn`, medan implicita trådar skapas automatiskt av programmet vid behov. En tråd i Fortress har fem olika tillstånd: inte startad, exekveras, suspenderad, normalt avslutad och abrupt avslutad.

3.2.1 Implicita trådar

Det finns en mängd olika uttryck i Fortress som är implicit parallella. Med ett implicit parallellt uttryck avses ett uttryck som ger upphov till parallell exekvering. Varje implicit parallellt uttryck skapar en grupp av implicita trådar. Alla trådar i en grupp skapas tillsammans och varje tråd i gruppen måste avslutas innan gruppen

som helhet avslutas. Med andra ord tillämpas så kallad ”fork-join”-metod. Om en av trådarna i en grupp avslutas abrupt, det vill säga på ett onormalt sätt, så avslutas också gruppen som helhet abrupt. En programmerare kan inte på något vis operera på implicita trådar, de sköts helt och hållet av Fortress-implementationen. I det här kapitlet presenteras och illustreras de uttryck, som ger upphov till implicita trådar.

Tuppeluttryck

Varje element i ett tuppeluttryck evalueras i en separat implicit tråd. Ett tuppeluttryck kan typiskt se ut på följande sätt:

```
(a,b) :=
  (doSomething(0), doSomething(1))
```

Also do-block

Parallell exekvering kan också uttryckas med hjälp av `also do`-block. Block av kod separerade med also do exekveras parallellt. Nedan finns ett exempel på användning av `also do`:

```
do
  sum += doSomething(0)
also do
  sum += doSomething(1)
also do
  sum += doSomething(2)
end
```

Metod- och funktionsanrop

Evalueringen av metod- och funktionsanrop tillsammans med dess attribut sker parallellt i implicita trådar. Ett metodanrop i Fortress består av en mottagare följt av en punkt och ett metodnamn och eventuella argument. Ett typiskt metodanrop ser ut på följande sätt:

```
myString.replace(”foo”, ”few”)
```

I ett metodanrop evalueras mottagaren, som i detta fall är `myString`, tillsammans med metodargumenten parallellt i implicita trådar. Efter att trådgruppen som utgörs av dessa har avslutats normalt så exekveras själva metoden.

Funktionsanrop fungerar på motsvarande sätt. Ett funktionsanrop består av två delar: ett uttryck som identifierar funktionen och ett argumentuttryck. Dessa två uttryck evalueras parallellt innan själva funktionen exekveras.

Generatorer

For-loopar och en del andra loop-uttryck, så som summeringar, är parallella i Fortress. Parallellismen i dessa uttryck definieras av så kallade generatorer. Generatorer behandlas mera ingående i kapitel 3.3.

Extremumuttryck

Fortress har stöd för så kallade extremumuttryck. Ett extremumuttryck påminner om ett vanligt case-uttryck, men väljer istället ut ett extremvärde som erhålls när uttrycken jämförs med en given operator. Ett exempel på ett extremumuttryck:

```
case most > of
  1 mile => ”1 mile är mer än 1 kilometer”
  1 kilometer => ”1 km är mindre än 1 mile”
end
```

Koden ovan evalueras till: ”1 mile är mer än 1 kilometer”

Villkorsuttrycken, som i fallet ovan är 1 mile och 1 kilometer, evalueras parallellt i implicita trådar. Uttrycken jämförs därefter parvis med operatorn som angetts i uttrycket, som i det här fallet är >. Även jämförelsen sker parallellt.

Test

Fortress har inbyggt stöd för att skapa så kallade enhetstest för att verifiera att programkod fungerar korrekt. Test körs parallellt i Fortress.

3.2.2 Explicita trådar

Explicita trådar skapas av programmeraren med uttrycket `spawn`, som returnerar en instans av typen `Thread[T]` där `T` är typen på det uttryck som getts som argument till spawn. Explicita trådar kan kontrolleras med kommandona `wait`, `ready` och `stop`. Uttrycket som getts som argument till `spawn` kommer att beräknas i en skild tråd, varefter resultatet fås med funktionen `val`. Funktionen `val` blockeras tills beräkningen av resultatet är klar. Ifall det inte finns resurser tillgängliga för att köra tråden parallellt så försöker Fortress köra uttrycket i den explicita tråden innan exekveringen fortsätter.

3.3 Generatorer

Många uttryck i Fortress använder så kallade *generatorer* (generators) för att kontrollera och åstadkomma parallell exekvering. En generator bestämmer om en exe-

Tabell 3.1: Exempel på några vanliga generatorer i Fortress

Generator	Funktionalitet
`j:k`	Heltal från j till k
`j#n`	n på varandra följande heltal startande från j
`[0,1,2,3,4]`	Elementen i en räcka, lista eller mängd
`a.indices`	Index för en räcka a
`sequential(g)`	En sekventiell version av en annan generator g

kvering kommer att ske parallellt och hur exekveringen i så fall delas upp i trådar. Generatorer påminner delvis om iteratorer i till exempel Java eller C++. Precis som nästan all funktionalitet i Fortress så specifieras också generatorer i form av bibliotek, vilket betyder att en programmerare själv kan definiera egna generatorer.

En generator binder en serie variabler till värden som produceras av en serie objekt, som ärver traitet Generator$[\![E]\!]$, där E är den typ av objekt som generatorn genererar [4]. Man kan referera till en instans av en generator direkt, men i praktiken refererar man oftast till så kallade generatorbindningar. En generatorbindning består av ett antal identifierare följt av `<-` och ett generatoruttryck. Några exempel på vanliga generatoruttryck finns i tabell 3.1. Fortress har tre olika huvudsakliga syntaxformer som använder sig av generatorer: for-loopar, comprehensioner och reduktionsuttryck.

3.3.1 For-loopar

For-loopar i Fortress använder sig av generatorer för att uttrycka iteration. Således är looparna parallella om de inte explicit görs sekventiella. En for-loop i Fortress kan exempelvis se ut på följande sätt:

```
for i <- 1:5 do
  print(i " ")
  print(i " ")
end
```

En provkörning av den här loopen gav följande utskrift:

```
4 1 4 1 5 2 5 2 3 3
```

Som man ser så skrivs siffrorna ut utan inbördes ordning. Det här är en av de viktigaste aspekterna med generatorer: En generator kan utföra beräkningar i vilken ordning den vill och kör dem oftast parallellt. Programmerare måste beakta det här och vid behov explicit definiera loopen som sekventiell genom att använda funktionen `sequential`.

Exempel på en reduktion:

$$s = \sum_{i \leftarrow 1:6} i^2$$

Exempel på en comprehension:

$$\langle i^2 \mid i \leftarrow 1:10 \rangle$$

Figur 3.1: Exempel på användningen av generatorer och reduktioner och comprehensioner i Fortress.

3.3.2 Reduktionsuttryck och comprehensioner

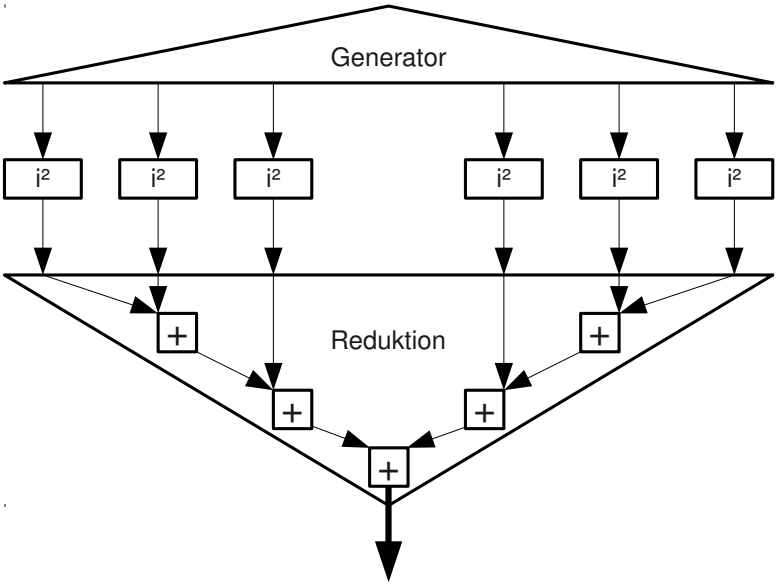
Generatorer används också i comprehensioner och reduktionsuttryck. Typiskt för reduktioner är att de beräknar någon funktion för varje genererat värde och sedan kombinerar ihop dessa till ett slutligt resultat. Exempel på några reduktioner och comprehensioner ges i figur 3.1. Precis som for-loopar beräknas också dessa uttryck parallellt [7].

En naiv implementation av en reduktion beräknar delresultaten parallellt, men kombinerar ihop dem sekventiellt. I många fall är det dock möjligt att göra kombineringen mycket effektivare, till exempel om den kombinerande operationen är associativ eller kommutativ eller har andra gynnsamma egenskaper [3]. Reduktioner finns färdigt definierade i Fortress-biblioteken för olika datatyper och operationer. I figur 3.2 illustreras hur summeringen i figur 3.1 sker med hjälp av en generator och en reduktion. Eftersom addition är en associativ operation kan summeringen av delresultaten ske parallellt.

3.4 Parallelliseringsalgoritm

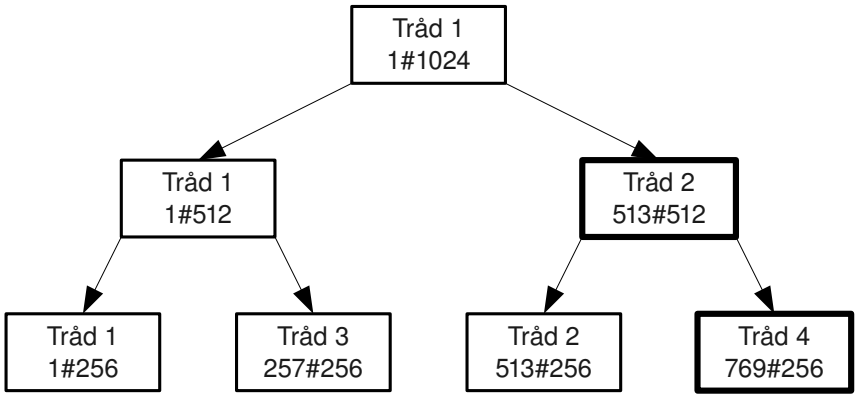
Sättet som Fortress hanterar parallellisering på kallas ”work stealing”, som är ett schema baserat på arbete gjort av Blumofe och Leiserson [9], men har utvidgats så att datastrukturen inte har några begränsningar vad beträffar storlek [10]. Schemat låter varje tråd upprätthålla en dubbeländad kö (deque) för jobb som skall utföras. Tråden utför själv jobb från bottnen på kön. Ifall en annan tråd blir utan jobb kan den stjäla jobb från toppen av trådens kö. För att minimera kommunikationen mellan trådar kan inte jobb som stulits från en tråd stjälas tillbaka av samma tråd [2].

I figur 3.3 illustreras ett exempel på hur ett program kan delas upp i fyra olika trådar. Rektanglarna med tjockare ram symboliserar jobb som stulits av tomma trådar.



Figur 3.2: En generator delar upp beräkningar i delar som kan utföras parallellt och som därefter kombineras ihop till ett slutligt resultat med en reduktion [8].

```
for <- 1#1024 do
  gorNagot(i)
end
```



Figur 3.3: Work stealing-algoritmen delar snabbt och effektivt upp en uppgift i olika trådar. Rektanglarna med tjockare kant representerar jobb som stulits av tomma trådar [11].

3.5 Minneshantering

3.5.1 Regioner

På stora datorsystem är distributionen av data viktig, eftersom kostnaden för minnesaccesser är varierande. Därför är det av avgörande betydelse att placera data ”nära” den processor som kommer att använda sig av dem. I Fortress kontrolleras det här genom att dela in datorsystem i *regioner* i vilka data sparas och beräkningar utförs [3].

Varje tråd, varje objekt och varje element i en räcka är associerad med en region. En region beskriver på ett abstrakt plan strukturen på det system som ett Fortress-program körs på, så som processor- och minnesresurser. Regionerna bildar en hierarki där löven är de mest lokala regionerna, så som processorkärnor, medan regionerna närmare roten är mera utspridda, så som kluster. Ett exempel på en regionhierarki finns illustrerat i figur 3.4. Regionen för ett objekt `o` fås genom att kalla på funktionen `o.region`. Den speciella regionen Global utgör alltid roten på regionhierarkin [4].

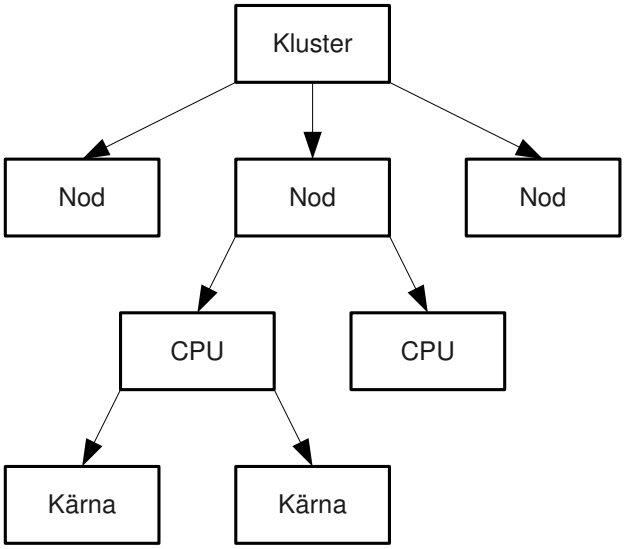
Med ett `at`-uttryck kan en tråd explicit placeras i en viss region, men det är inte nödvändigtvis möjligt för en tråd att befinna sig i en given region. Till exempel finns alltid en tråd som håller på att exekveras på exekveringsnivå, vilket oftast är löven i hierarkin [4]. Därför är *at*-kommandot endast en rekommendation, Fortress-implementationen kan fortfarande välja att exekvera tråden i en annan region, till exempel för att balansera exekveringen över de tillgängliga resurserna [4]. Följande kodexempel beräknar `a[i]` lokalt medan `a[j]` beräknas i den region där `j` finns:

```
(v,w) := (a[i],
  at a.region(j) do a[j] end)
```

3.5.2 Delad och lokal data

Varje objekt i Fortress är antingen *delat* (shared) eller *lokalt* (local). Dessa anger huruvida ett objekt är synligt från en eller flera trådar. Ett lokalt objekt är transitivt nåbart från variablerna i högst en tråd, medan ett delat objekt kan nås från flera trådar [4]. Att accessera ett lokalt objekt är mindre resurskrävande än att accessera ett delat objekt. Delningen av data kontrolleras i första hand av Fortress-implementationen, men kan också till viss grad kontrolleras av programmerare för exempelvis optimeringssyften.

När ett objekt skapas är det lokalt. När en referens till ett lokalt objekt sparas i en delad datastruktur måste objektet *publiceras*, varvid all data som objektet refererar till också publiceras. Att publicera ett objekt kan vara en resurskrävande operation,



Figur 3.4: Ett exempel på en regionhierarki. Elementen närmare roten av trädet är mera utspridda, så som till exempel kluster, medan löven är de mest lokala elementen, som till exempel processorkärnor.

speciellt ifall datastrukturen som delas är stor. Det här kan orsaka att ett till synes kort uttryck tar lång tid att exekvera [4].

3.5.3 Distribuerade räckor

I Fortress är räckor, vektorer och matriser komplexa datastrukturer som antas vara utspridda över datorsystemet. Hur datastrukturen distribueras bestäms av Fortressbiblioteken och beror i allmänhet på storleken på datastrukturen och på datorsystemet som programmet körs på. Elementen i datastrukturen kan således befinna sig i olika regioner.

3.5.4 Transaktionellt minne

I parallella programmeringsspråk är det viktigt att kunna synkronisera läsningar och skrivningar till minnet för att undvika oväntade resultat. Istället för att använda sig av vanliga låsningar (locks) för synkronisering, så använder Fortress sig av så kallade transaktioner. En transaktion är en kodsekvens som utför en mängd läsningar och skrivningar till ett delat minne. Ur utomstående trådars synvinkel sker läsningarna och skrivningarna atomärt.

Transaktionellt minne föreslogs första gången 1993 av Maurice Herlihy och Eliot Moss [12]. Transaktioner orsakar inga större prestandaförluster på moderna datorsystem och dessutom löser de många problem associerade med låsningar [13].

Minneshanteringen i den nuvarande Fortress-implementationen använder DSTM2 [14] för att implementera transaktioner. DSTM2 är ett mjukvarubibliotek skrivet i Java ämnat för implementering av transaktionellt minne i mjukvara.

En transaktion kan antingen verkställas (commit), varvid alla operationer ser ut att ha utförts atomärt, eller förkastas (abort), varvid operationerna ser ut att inte alls ha utförts. Det kan uppstå konflikter mellan transaktioner när de försöker accessera samma objekt och minst en av dem försöker skriva till det. I sådana fall måste en av transaktionerna vänta på att den andra ska avslutas [14]. I Fortress är trådar numrerade och transaktioner skapade av tråden med lägst nummer har alltid förtur [11].

3.5.5 Atomära uttryck

Transaktioner skapas i Fortress med uttrycket `atomic`. Ett atomärt uttryck består av `atomic` följt av ett antal instruktioner. Alla läsningar och skrivningar som görs i det atomära uttrycket kommer att ses som ett enda steg av andra trådar. Ifall ett atomärt uttryck avslutas abrupt så förkastas alla läsningar som gjorts i uttrycket [4]. Atomära uttryck kan avbrytas med funktionen `abort`, som kan köras innuti ett atomärt uttryck. Det förkastar alla ändringar som gjorts i det atomära uttrycket. Ett typiskt atomärt uttryck ser ut på följande sätt:

```
atomic do
  x += 1
  y += 1
end
```

4. Exempelprogram

I det här kapitlet demonstreras ett kort exempelprogram skrivet i Fortress för att ge en mera komplett bild av hur ett Fortress-program är uppbyggt och hur Fortress skiljer sig från andra språk.

4.1 Programmet

Programmet löser en given ekvation iterativt med Newton-Raphson-metoden. Newton-Raphsons iterationsformel kan skrivas som [15]:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{4.1}$$

Genom att välja ett initialvärde x_0 och en önskad precision p och upprepa iterationsformeln tills dess att $|x_{n+1} - x_n| < p$ så fås ett nollställe med önskad noggrannhet.

4.2 Implementation

Programkoden finns given i sin helhet i bilaga A i både normalt ASCII-format och LaTeX-renderat format. Programmet beräknar nollställen för en funktion, som ges i programkoden som en normal Fortress-funktion. Också funktionens derivata, som behövs för beräkningen, ges i form av en Fortress-funktion. Programmet består av fem funktioner:

- `newton` – tar som argument en räcka med initialvärden och ett decimaltal som indikerar den önskade precisionen. Funktionen beräknar Newton-Raphsons iterationsformel parallellt för varje initialvärde och skriver ut resultatet.
- `newtonIter` – själva implementationen av Newton-Raphsons iterationsformel. Tar som argument ett initialvärde och önskad precision och itererar tills dess att önskad precision uppnåtts.
- `f` – den funktion på vilken Newton-Raphson-metoden skall tillämpas. Funktionen tar precis som en vanlig matematisk funktion ett argument `x`. I exempelprogrammet består funktionen av ett fjärde gradens polynom.
- `Df` – derivatan av funktionen `f`.
- `run` – funktion för att initialisera och köra programmet.

Eftersom det nollställe man erhåller med Newton-Raphsons metod beror på vilket initialvärde man väljer kan man i programmet ange flera initialvärden. Programkoden är skriven så att Newton-Raphsons metod kan beräknas parallellt för varje enskilt initialvärde. Funktionen som utför de parallella beräkningarna ser ut på följande sätt:

```
newton(x_init:Array[\RR64,ZZ32\],prec:RR64):() = do
  for x <- x_init do
    (root, iterCount) = newtonIter(x, prec)
    println("Startvarde " x " gav resultatet " root)
  end
end
```

Parallelliseringen sker med hjälp av en for-loop och en generator, `x <- x_init`, som genererar alla element i räckan `x_init`. Funktionen kallar på funktionen `newtonIter` som är den funktion som utför Newton-Rahpsons metod. Funktionen ser ut på följande sätt:

```
newtonIter(x_init:RR64,prec:RR64):(RR64,ZZ32) = do
  iterCount:ZZ32 := 0
  x:RR64 := x_init
  x_prev:RR64 := x_init - 10000 prec

  while |x - x_prev| > prec do
    iterCount := iterCount + 1
    x_prev := x
    x := x - f(x)/Df(x)
  end
  (x, iterCount)
end
```

Funktionen tar som argument ett initialvärde `x_init` och en önskad precision, prec. Funktionen består av en loop, som itereras tills dess att ändringen på variabeln x värde mellan två iterationer är mindre än den önskade precisionen. Från kodexemplet ovan ser man att en funktion i Fortress kan returnera mer än ett värde. I det här fallet returneras ett decimaltal för värdet på x och ett heltal för antalet iterationer som krävdes för att komma till resultatet.

Funktionerna `f` och `Df`, som används för beräkningen definieras som vanliga Fortress-funktioner. Matematiska funktioner kan i Fortress skrivas så att de är nästan identiska med matematisk notation. Funktionerna som används i exempelprogrammet ser ut på följande sätt:

```
f(x:RR64):RR64 = -5 x^4 + 2 x^3 + 2 x^2 + 3 x + 5

Df(x:RR64):RR64 = -20 x^3 + 6 x^2 + 4 x + 3
```

5. Slutsatser

Fortress är ett av de första programmeringsspråken i sitt slag. Matematisk notation, utvidgbarhet och implicit parallellism är de mest innovativa delarna av språket. Språket är kanske inte lika flexibelt som exempelvis C, men konceptet med bibliotek kombinerat med dess vetenskapliga approach gör språket attraktivt inom dess avsedda användningsområde.

Fortress sätt att uppnå parallell exekvering är en av språkets viktigaste egenskaper. Språket stöder så kallad implicit parallellism, vilket betyder att så gott som alla tänkbara beräkningar som kan ske parallellt automatiskt parallelliseras i Fortress. Den här typen av approach kommer förmodligen att vara en nödvändighet för alla effektiva programmeringsspråk i framtiden, då antalet beräkningsenheter i datorsystem hela tiden ökar. Implicit parallellism gör att programmeraren befrias från problemet att parallellisera en algoritm och gör att han istället kan koncentrera sig på algoritmen som sådan. Förutom implicit parallellism har Fortress också en mängd andra egenskaper som stöder parallella beräkningar och gör språket lämpat för applikationer som körs på stora datorsystem. Till dessa egenskaper hör funktioner så som regioner och transaktioner.

På grund av syntaxen, som efterliknar matematisk notation, är Fortress-kod både lättläst och lätt att skriva. Det här gör språket produktivt och intuitivt även för mindre erfarna programmerare. Syntaxen passar utmärkt för vetenskapliga beräkningar, men kan vara något begränsande i mer allmänna tillämpningar.

Eftersom Fortress är ett nytt språk finns det mycket utvecklingsarbete kvar att göra innan språket blir användbart i verkliga tillämpningar. Den nuvarande implementationen är inte särskilt effektiv och saknar många av de planerade funktionerna. Fortress utvidgningsmöjligheter i kombination med det faktum att språket är ett open source-projekt gör dock att språket har goda möjligheter att utvecklas och bli ett attraktivt alternativ inom vetenskapliga beräkningar.

6. Litteraturförteckning

[1] DARPA TCTO. DARPA HPCS Project. `http://www.darpa.mil/tcto_hpcs.html`, Mars 2010.

[2] E. Uney och H.N. Dalfes. A new parallell programming language Fortress: Features and applications. I: *Fifth International Conference on Soft Computing, Computing with Words and Perceptions in System Analysis, Decision and Control*, 2009.

[3] E. Allen, D. Chase, C. Flood, V. Luchangco, J.W. Maessen, S. Ryu och G.L. Steele Jr. Project fortress: A multicore language for multicore processors. *Linux Magazine*, ss 38–43, 2007.

[4] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.W. Maessen, S. Ryu, G.L. Steele Jr och S. Tobin-Hochstadt. The Fortress language specification, version 1.0. *Sun Microsystems*, 2006.

[5] T. Haapasalo. Multi-core Programming: Implicit Parallelism. `http://www.cs.hut.fi/u/tlilja/multicore/slides/implicit-handouts.pdf`, April 2009.

[6] Project Fortress Community. Implicit Parallelism in Fortress. `http://projectfortress.sun.com/Projects/Community/wiki/ImplicitParallelismInFortress`, Januari 2008.

[7] Project Fortress Community. Using Generators in Fortress. `http://projectfortress.sun.com/Projects/Community/wiki/UseAGenerator`, Februari 2010.

[8] J.W. Maessen och S. Ryu. A Short Hands-On Introduction to Fortress. `http://research.sun.com/projects/plrg/Publications/MITtutorial2009.pdf`, April 2009.

[9] R.D. Blumofe och C.E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[10] D.R. Chase och Y. Lev. Dynamic circular work-stealing deque, Mars 2008. US Patent 7,346,753.

[11] Christine H. Flood. Project Fortress: A new programming language from Sun Labs. I: *JavaOne*, 2008.

[12] M. Herlihy och J.E.B. Moss. Transactional memory: Architectural support for lock-free data structures. I: *Proceedings of the 20th annual international symposium on Computer architecture*, s 300. ACM, 1993.

[13] Project Fortress Community. Transactions in Fortress. `http://projectfortress.sun.com/Projects/Community/wiki/TransactionsInFortress`, Februari 2008.

[14] M. Herlihy, V. Luchangco och M. Moir. A flexible framework for implementing software transactional memory. I: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, s 262. ACM, 2006.

[15] T. Gustafsson. MATEMATIK IV Numeriska metoder. 2004.

A. Exempelprogramkod

I den här bilagan listas källkoden för ett exempelprogram skrivet i Fortress. För jämförelse ges koden ges både i LaTeX-formaterad kod och i vanligt ASCII-format

A.1 LaTeX-formaterad programkod

```
component Newton
export Executable
```

⊛ Newtons iterationsformel

$newton\big(x_{\text{init}}\colon \text{Array}[\![\mathbb{R}64, \mathbb{Z}32]\!], prec\colon \mathbb{R}64\big)\colon () = \texttt{do}$

```
(*
```

Beräkna Newtons formel för varje startvärde parallellt.

```
*)
```

$\texttt{for } x \leftarrow x_{\text{init}} \texttt{ do}$

$(root, iterCount) = newtonIter(x, prec)$

$println(\text{“}\texttt{Startvarde }\text{”}\, x\, \text{“}\texttt{ gav resultatet }\text{”}\, root$
$\text{“}\texttt{ efter }\text{”}\, iterCount\, \text{“}\texttt{ iterationer}\text{”})$

$\texttt{end}$

$\texttt{end}$

$newtonIter(x_{\text{init}}\colon \mathbb{R}64, prec\colon \mathbb{R}64)\colon (\mathbb{R}64, \mathbb{Z}32) = \texttt{do}$

$iterCount\colon \mathbb{Z}32 := 0$

$x\colon \mathbb{R}64 := x_{\text{init}}$

$x_{\text{prev}}\colon \mathbb{R}64 := x_{\text{init}} - 10000\, prec$

$\texttt{while } |x - x_{\text{prev}}| > prec \texttt{ do}$

$iterCount := iterCount + 1$

$x_{\text{prev}} := x$

$x := x - \dfrac{f(x)}{\mathrm{Df}(x)}$

$\texttt{end}$

$(x, iterCount)$

$\texttt{end}$

⊛ Funktionen

$f(x\colon \mathbb{R}64)\colon \mathbb{R}64 = -5\,x^4 + 2\,x^3 + 2\,x^2 + 3\,x + 5$

⊛ Derivatan av funktionen

$\mathrm{Df}(x\colon \mathbb{R}64)\colon \mathbb{R}64 = -20\,x^3 + 6\,x^2 + 4\,x + 3$

✪ run-metod for testning

$run()\colon () =$ do

$prec\colon \mathbb{R}64 = 0.000000001$

✪ Startvarden som ska anvands

$x_{\text{init}}\colon \mathbb{R}64_2 = [2.0\ 0.0]$

$newton(x_{\text{init}}, prec)$

end

end

A.2 Programkod i ASCII-format

```
component Newton
export Executable

  (*) Newtons iterationsformel
  newton(x_init:Array[\RR64,ZZ32\],prec:RR64):() = do
    (*
     Beräkna Newtons formel för varje startvarde
     parallellt.
     *)
    for x <- x_init do
      (root, iterCount) = newtonIter(x, prec)
      println("Startvarde " x " gav resultatet " root
              " efter " iterCount " iterationer")
    end
  end

  newtonIter(x_init:RR64,prec:RR64):(RR64,ZZ32) = do
    iterCount:ZZ32 := 0
    x:RR64 := x_init
    x_prev:RR64 := x_init - 10000 prec
    while |x - x_prev| > prec do
      iterCount := iterCount + 1
      x_prev := x
      x := x - f(x)/Df(x)
    end
    (x, iterCount)
```

```
  end

  (*) Funktionen
  f(x:RR64):RR64 = -5 x^4 + 2 x^3 + 2 x^2 + 3 x + 5

  (*) Derivatan av funktionen
  Df(x:RR64):RR64 = -20 x^3 + 6 x^2 + 4 x + 3

  (*) run-metod for testning
  run():() = do
    prec:RR64 = 0.000000001
     (*) Startvarden som ska anvands
     x_init:RR64[2] = [2.0 0.0]
    newton(x_init,prec)
  end

end
```