ELEKTRONIK TIDNINGEN

Mike Beunder
vd för Vector Fabrics

Så analyserar man C-kod för parallellisering

Mike Beunder reder ut några av de subtila beroenden mellan programmets olika delar som kan ställa till problem när du får i uppgift att parallellisera ett seriellt program.

Redaktör
Jan Tångring
jan@etn.se
0734-17 13 09

EMBEDDED EXPERT

Så analyserar man C-kod för parallellisering

Mike Beunder är vd för Vector Fabrics, ett programverktygsföretag som han själv var med och startade i Eindhoven 2007. Under sin tidiga karriär var han VSLI-konstruktör och han har arbetat i framgångsrika nystartade och stora (”blue-chip”) halvledarföretag i Australien, Tyskland, Frankrike, USA och Nederländerna. Mike Beunder har en ingenjörsexamen i elektroteknik och en mastersexamen från Twente universitet i Nederländerna. Han har även en doktorsexamen från tyska Twente universitet och IMS Stuttgart.

Datorvetare och ingenjörer insåg för länge sedan – i princip vid datorvetenskapens födelse – att det är möjligt att snabba upp exekveringen av en beräkning genom att utnyttja dess inneboende parallellitet. Om det vore möjligt att identifiera de delar av ett program som kan utföras parallellt, omorganisera programflödet därefter och tillhandahålla hårdvara för varje parallell del, skulle den sammanlagda uppgiften kunna slutföras på kortare tid.

Allteftersom tiden gått har man insett att problemet i praktiken är rätt svårhanterligt. Under en lång tid fanns det inte på dagordningen att försöka höja prestanda genom att lösa det här problemet. Halvledarindustrin bjöd istället hela tiden på högre beräkningskraft genom allt snabbare CPU-klockhastigheter. Men den tiden är nu över och CPU-konstruktörer måste istället bygga arkitekturer med alltfler kärnor – multikärnor – för att kunna öka genomströmningen. Därför existerar nu allt fler parallella hårdvaruplattformar, vilket återigen riktar uppmärksamheten på det önskvärda i att effektivt kunna parallellisera programflöden.

Hurpass möjligt det är att bryta isär ett sekventiellt program och köra det i parallella delar, avgörs av hur beroende delarna är av varandra. Detta beror på beroenden inom programmet, och det är kritiskt att förstå denna typ av beroenden om man vill kunna skapa effektiva parallella implementeringar som är funktionellt identiska med den sekventiella versionen.

Sådan programstrukturanalys har huvudsakligen varit en intresse för universitetsforskare, eftersom kommersiella programmerare har kunnat tillgodoräkna sig allt snabbare processorer. Men den tiden är nu försvunnen; dessa en gång så svårbegripliga koncept har nu konsekvenser för samtliga program som skrivs inom industrin.

Amdahls lag gäller fortfarande

Enligt den sats som kallas Amdahls lag gäller att gränsen för den prestandaförbättring som kan fås genom parallellisering sätts av de delar av programmet som inte kan parallelliseras – som måste köras i en viss ordning. För *n* processorer är det bara möjligt att få programmet att gå *n* gånger snabbare om det kan delas upp i *n* ömsesidigt oberoende delar av exakt samma tidslängd. Och detta är nästan aldrig möjligt. Vilket betyder att programmerare nästan alltid måste nöja sig med en hastighetsökning som är något mindre än en faktor *n* på en *n*-kärnig multikärna. Konstruktören får nöja sig med att försöka öka uppsnabbningen så mycket som möjligt.

I ett givet program finns det ofta vissa steg vars prestanda är kritiska och andra steg som är mindre kritiska. Att få de kritiska delarna att gå så snabbt som möjligt är detsamma som att hålla de icke-kritiska delarna ur vägen. Effektiv parallellisering bygger på att man först tar bort alla beroenden som överhuvudtaget kan tas bort, och sedan försöker ta sig runt återstående beroenden.

Att räkna ut var alla dessa beroenden finns är inte lätt. Om beroenden förbises kan det resultera i en parallell version av ett program som skiljer sig funktionellt från originalet.

Vilket programmeringsspråk som används för att skriva programmet påverkar i princip inte analysen. Det ideala parallella språket (vilket inte finns) skulle klara sig utan analysverktyg – det skulle framhäva alla beroenden explicit. ANSI C

För ytterligare information och läsarfrågor:

Bryon Moyer, marknadschef, Vector Fabrics
Tel: +1 408 981 2029
E-post: bryon@vectorfabrics.com
Hemsida: www.vectorfabrics.com

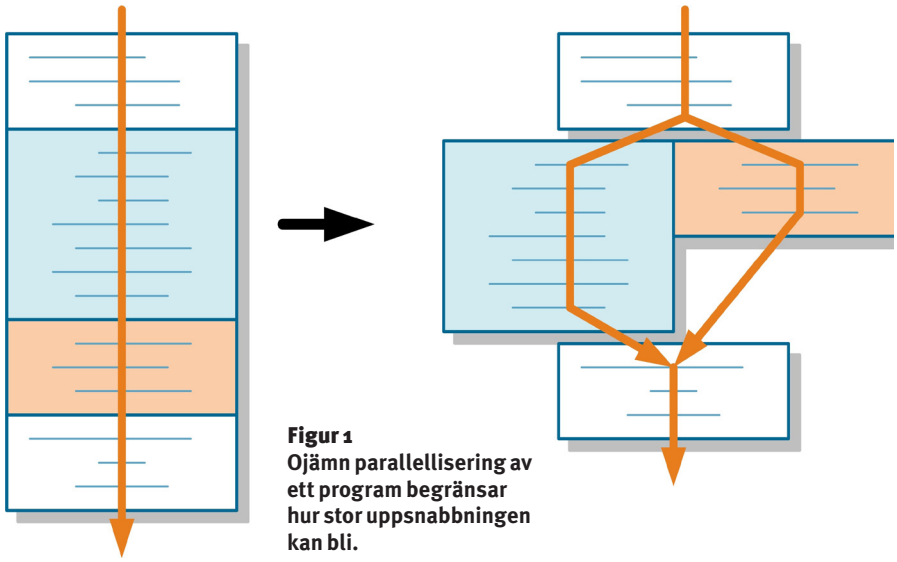
är långt ifrån detta idealspråk, men det är det klart vanligaste språket för programmering av inbyggda system (liksom C++ är i företagssystem) och det kommer troligen att förbli det. Detta faktum innebär också att det i praktiken inte är så stor skillnad som man kanske kan tro på att parallellisera ny och gammal kod.

Parallellisering innebär att man skapar programsektionser som ska exekvera parallellt med varandra. Sådana delar kan implementeras som trådar eller processer. Trådar är underordnade en viss process, och delar ett gemensamt minnesutrymme. Processer är helt oberoende av varandra. Ett program för en multikärna kan implementeras i en enda process med flera trådar eller i en grupp entrådsprocesser i asymmetrisk multiprocessning (AMP, asymmetric multiprocessing) där processerna kommunicerar sinsemellan (IPC, inter-process communication). Eller i en kombination av båda.

Parallellitet finns på alla nivåer

På vilken nivå bör möjligheter till parallell exekvering undersökas? På lägsta nivå kan parallellitet på instruktionsnivå utnyttjas för att omorganisera och parallellisera beräkningar. Möjligheten beror dock till stor del på processorns arkitektur och är något som kompilatorer redan är byggda för att kunna göra på egen hand – det är svårt att hitta mer att optimera på denna nivå. Det andra extremfallet är att leta parallellism i stora programdelar – dessa visar sig nästan alltid vara ömsesidigt beroende om beräkningen rör något annat än det mest triviala.

En mer användbar approach är att sikta in sig på gränsen mellan processor och minne. Medan processorn i sig och dess register styrs av kompilatorn, berörs minnesarkitekturen inte uttryckligen av kompilatorn. På denna nivå kan det löna sig att undersöka relationer. De kritiska operationerna är lagring och laddning. Kompilatorn tar hand om variabler i processorregister, men så snart deras värden lagras i eller laddas från minnet har vi möjlighet att styra de beroendeförhållanden som uppkommer. Konceptet kan utökas till att inkludera kringutrustning (som ofta är minnesmappade) – närhelst en variabel lämnar processorns sfär (och kompilatorns) blir den av intresse.



Figur 1
Ojämn parallellisering av ett program begränsar hur stor uppsnabbningen kan bli.

Den andra frågan som uppkommer är hur stora kodblock som det är meningsfullt att analysera. Det är mycket vanligt att man analyserar programmets beteende på funktionsnivå. Mycket av beräkningsarbetet, framför allt i inbyggda system, sker i programloopar, och de är av kritiskt intresse när man avgör hur ett program ska parallelliseras.

Man kan få reda på mycket om ett programs beteende genom inspektion, eller genom statisk analys. Denna kan dock missa kritiskt beteende, särskilt beträffande hur pekare och andra minnesrelaterade realtidsfunktioner uppför sig, vilket endast kan snappas upp med dynamisk analys, som observerar vad som händer medan ett program utför beräkning med en viss uppsättning data.

Beroendeförhållanden

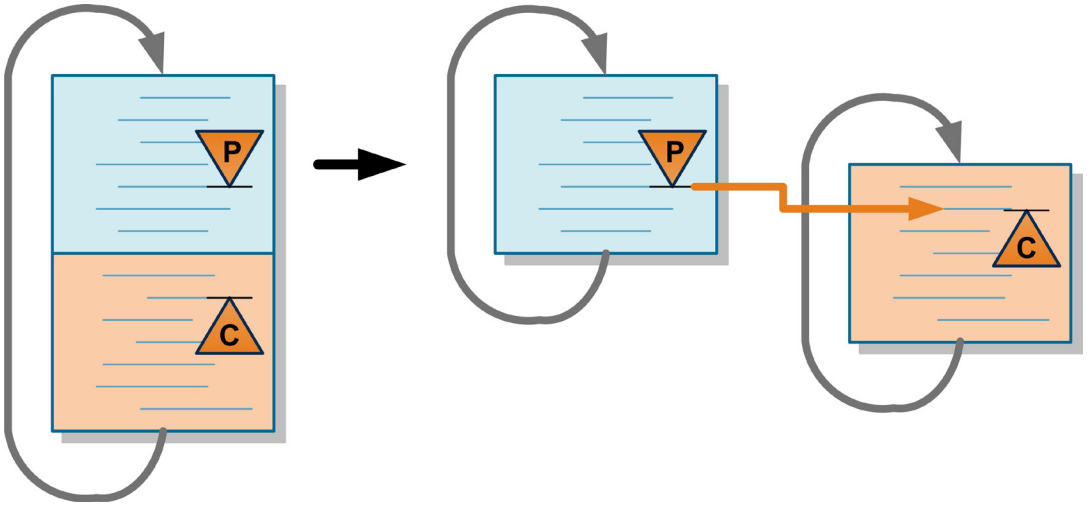
Nyckeln till effektiv parallellisering av sekventiell kod är beroenden, som förekommer i olika skepnader. Om en beräkning använder resultat från en annan beräkning, så måste den vänta tills den första beräkningen är klar innan den kan fortsätta. Detta är ett beräkningsberoende eftersom det drivs av programmets beräkningsflöde. Ett användbart sätt att se på händelser är att ett beräkningsberoende startar med en laddningsfunktion, där ett värde hämtas som ska bearbetas, tills sekvensen slutar med en lagringsoperation. Denna typ av beroenden identifieras lätt med statisk analys.

Nästa typ av beroende är inte lika uppenbart, då det utgörs av den svårspårade aktivitet som är förknippad med framför allt pekare. C-språket, i synnerhet, möjliggör odisciplinerad, ostrukturerad användning av pekare på ett sätt som – beroende på synvinkel – kan uppfattas som allt från kreativt till slarvigt. Sådan användning är den enskilt största källan till oväntade, svårfunna, subtila buggar och problem, speciellt vid försök att parallellisera ett sekventiellt program. De kan inte hanteras av kompilatorer eller vanliga profilerare.

Ett tidigt försök att hantera pekarproblematiken var att helt enkelt förbjuda pekarna. Det lägger en enorm omskrivningsbörda på ingenjören, som ofta inte är kodens ursprungsförfattare och som därmed kanske inte ens känner till strategin bakom kodens formulering.

Ett bättre tillvägagångssätt är att separat analysera beteendet hos pekarna och identifiera de kritiska beroenden som de ger upphov till. På grund av att de är knutna till pekare och minnesaccess kallas dessa beroenden för *minnesberoenden*. De börjar med en lagringsoperation, som normalt (men inte uteslutande) är en skrivning via pekare (pointer write), och avslutas med en eller flera laddningsoperationer, normalt via pekarreferenser.

Det som gör att dessa beroenden inte är statiskt analyserbara är det faktum att det kan finnas många pekare som refe-



Figur 2
Parallellisering av en slinga där data produceras (P) och förbrukas (C) i parallella trådar. Den andra tråden måste invänta tillgängligheten av data innan den kan gå vidare.

rerar till samma address vid vilken given punkt som helst under programmets exekvering, vardera under olika namn. Därmed behövs dynamisk analys för att identifiera vem som läser från och skriver till specifika adresser.

Eftersom denna lagring av data för senare laddning i själva verket är kommunikation av data från en del av programmet till en annan, innebär felaktig parallellisering att vissa delar av programmet inte kommer att ha rätt data. Om ett program parallelliseras utan att rätt hänsyn visas dessa beroenden kommer beteendet att kraftigt störas på sätt som kan vara mycket svåra att avlusa.

Dessa två huvudgrupper av beroenden kretsar kring behovet för data att vara tillgänglig för användning, vilket innebär väntan på att läsa datan. Den motsatta situationen kan också vara sann: en bit data som kommer att skrivas över kan vara tvungen att behålla sitt nuvarande värde tills dess att värdet inte längre behövs. Detta innebär att en skrivfunktion försenas tills all läsning som behövs fullbordats, en situation kallad *antiberoende* (anti-dependency).

En annan typ av beroende handlar om *initiering* av programmet – tilldelning av variabelvärden för att ta det enklaste exemplet – innan det körs.

Också *biblioteksanrop* kan skapa beroenden som kan vara kritiska för korrekt programfunktion. Ett printf-anrop blir exempelvis viktigt om utskriftsordningen är av betydelse, men inte om det viktiga bara är att utskriften sker, och inte i vilken ordning.

Programslingor och beroenden som följer av dem

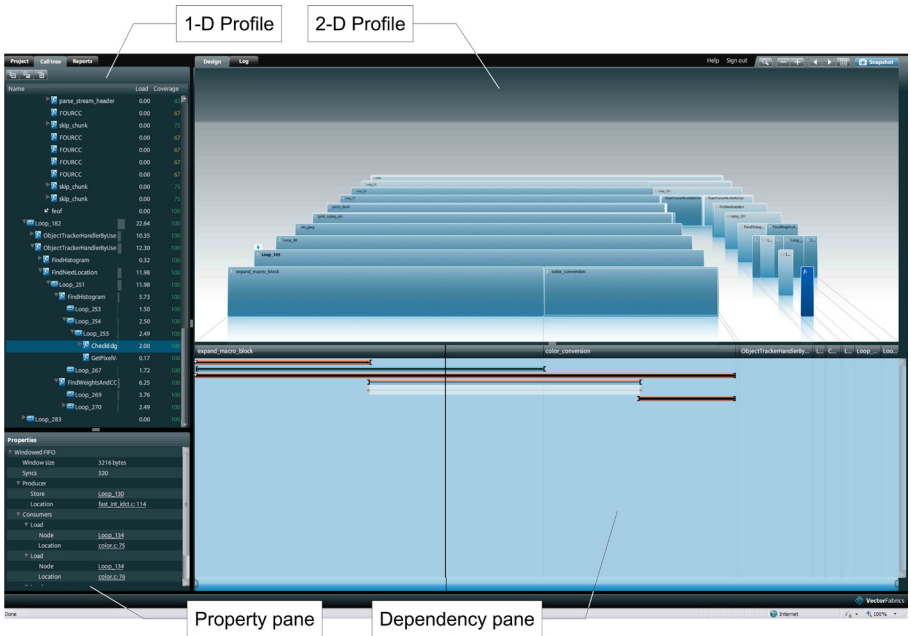
Loopar kan ofta vara parallelliserade för att förbättra prestandan; varje varv i en loop kan tilldelas en särskild uppgift. Om det inte finns några beroenden mellan varven kan alla varv utföras samtidigt. Här finns ett speciellt viktigt sorts minnesberoende eller antiberoende: konsumtionen under ett loopvarv av ett värde som beräknats i ett annat loopvarv.

Förekomsten av sådana loopberoenden gör inte denna del av ett program omöjlig att parallellisera, men det kräver vissa justeringar av analysen, inklusive att man introducerar koncept som *beroendeavstånd* – i huvudsak är detta det antal loopvarv som sker från det att programmet genererar data till att det använder det igen. *Nästlade loopar* lägger ytterligare komplexitet till analysen, som måste vara datacentrerad och följa produktion och konsumtion av varje enhet. Program kan mycket väl ha flera nästlade loopar, som överför värden mellan flera lager i hierarkin från den innersta till den yttersta loopen.

Ger meningsfull analys

Bara att beskriva informationsflödet, via dessa olika typer av beroenden som blottläggs med dynamisk analys, är en utmaning i sig; det är en viktig del av Vector Fabrics analysprogramvara *vfAnalyst*. Visualiseringen av beroenden och andra liknande kritiska delar förenklar väsentligt något som annars är en svår uppgift för ingenjörer eller programmerare – även om de är desamma som de som skrev, eller håller på att skriva, koden. De ger även en fingervisning om varför konventionella tekniker för programprofilering är, om inte helt oanvändbara, så åtminstone av mycket begränsat värde som parallelliseringsverktyg.

Det är lätt att skriva kod som, vid sin exekvering, ger upphov till extremt komplexa, invecklade utbyten av data. Detta kan fullständigt fördunkla, vid manuell inspektion, de möjligheter till parallellisering som kommer att bli mycket produktiva. Medan Vector Fabrics University Program gör denna produktivitet tillgänglig för den traditionella akademiska världen är det inom den kommersiella världen som de största vinsterna kommer att förverkligas. ■



Figur 3 Exempel på skärmbild från vfAnalyst, ett verktyg som analyserar program för samstämmighet och beroenden och kan föreslå strategier för parallellisering.