Parallell Slump
Om å parallellisera genetiske algoritmar i Haskell

Hans Georg Schaathun*†
⟨georg@schaathun.net⟩
Høgskolen i Ålesund, Institutt for Ingeniør- og Realfag
Postboks 1517, 6025 Ålesund

Samandrag

Slumptalsgenerering er eit av dei mest fundamentale og klassiske problema i informatikk. Der er ei lang rekkje kjende gode slumptalsgeneratorar for sekvensielle program. Parallell programmering stiller nye krav til slumptalsgeneratorane, og ulike parallelliseringsparadigme har ulike behov. Når me krev at programmet skal vera deterministisk utelukker me mange av dei vanlegaste løysingane. Trass i at parallellisering av slumptalsgeneratorar har vore kjend som ei utfordring i 30 år, er der få generelle løysingar i literaturen. I denne artikkelen gjev me eit overblikk over kjende løysingar og viser korleis splittbare slumptalsgeneratorar kan brukast til ein deterministisk, dataparellell implementasjon av genetiske algoritmar.

1 Innleiing

Genetiske algoritmar er ein populær metode for å løysa optimeringsproblem. Dei imiterer evolusjonsprosessar i naturen. Moglege løysingar vert kalla *kromosom*, og ein startar algoritmen med ein *populasjon* av tilfeldig valde kromosom. Evolusjonsprosessen er slumpmessig der kromosoma vert muterte og dei beste kromosoma vert parra for å danna nytt avkom.

Evolusjonen er i utgangspunktet godt egna for parallellisering, sidan både parring og mutasjon opererer på kromosompar eller einskildkromosom heilt uavhengig av resten av populasjonen. Slumptalsgenerering på den andre sida, er ikkje like lett å parallellisera. Velkjende slumptalsalgoritmar er grunnleggjande sekvensielle tilstandsmaskiner.

Funksjonelle språk (som Lisp og Haskell) er rekna for å vera velegna for å uttrykka genetiske algoritmar på ein ryddig og strukturerert måte. Eit reint funksjonelt programmeringsparadigme (som i Haskell), utan global tilstand eller funksjonar med sideeffektar, har også store føremonar for parallellisering. Dei same eigenskapane som forenklar parallellisering vil derimot òg stilla strenge krav til slumptalsalgoritmen som vert brukt.

*Arbeidet er delfinansiert av *Regionalt Forskningsfond Midt-Norge* under prosjekt nr. ES504913 med tittelen *Dynamic Resource Allocation with Maritime Application* (DRAMA).

†Forfattaren ynskjer å takka Robin Bye for å ha delt spanande forskingsproblem og villig drøfta idéar innanfor og utanfor eigen ekspertise.

Denne artikkelen vart presentert på konferansen NIK-2014; sjå http://www.nik.no/.

I denne artikkelen skal me ta for oss korleis probabilistiske algoritmar kan parallelliserast i reine funksjonelle språk. For å illustrera problemstillingane med eit konkret døme, vil me ta for oss ein parallell implementasjon av genetiske algoritmar i Haskell [13], noko som har vore lekk i ein studie av dynamisk optimalisering av slepebåtplassering [2]. Hovudutfordringa ligg i å parallellisera sjølve slumptalsgenereringa, og me gjev eit oversyn over sokalla *splittbare slumptalsgeneratorar*. Me merker oss spesielt at slumptalsgeneratoren i standardbiblioteket åt Haskell ikkje gjev uavhengig fordelte slumptal, og me skal gje eit nytt, kort algebraisk argument som viser det.

2 Slumptalsgeneratorar

Slumptalsgenerering er eit av dei klassiske problema i informatikk [7]. Mange kvardagslege og vitskaplege bruksområde krev slumptal, t.d. (Monte Carlo-)simulering, spel, kryptografi og mange optimeringsalgoritmar. Både røynlege slumptal og pseudo-slumptal vert brukt. Sistnemnde vert genererte av ein deterministisk algoritme og er ikkje tilfeldige i det heile. Algoritmen vert designa slik at slumptala *ser tilfeldige ut*. Røynlege slumptal må hentast frå ein fysisk kjelde, og mange operativsystem har i dag støtte for det. Krava til slumptalsgeneratoren varierer frå bruksområde til bruksområde. Fylgjande krav er vanlege:

1. Det må vera uråd å skilja ei generert slumptalsfylgja frå ei ekte tilfeldig talfylgje.
2. Tilstrekkeleg mange slumptal må vera tilgjengeleg.
3. Slumptalsfylgja må vera reproduserbar.
4. Algoritmen må vera tilstrekkeleg rask.

Når me ser på dei to fyrste to krava, er det tydelge at røynlege slumptal er idealet og pseudo-slumptal er eit kompromiss. Røynlege slumptalskjelder gjev per definisjon ei ekte tilfeldig og uendeleg lang fylgje (føresett at me har uendeleg lang tid). Pseudo-slumptal vil alltid gje ei periodisk gjentakande fylgje, og det gjeld at perioden er tilstrekkeleg lang for den aktuelle bruken.

I kryptologien vert krav nr. 1 formalisert på ulike måtar [11]. Ein kan krevja at ein kvar polynomisk algoritme som observerer ei talfylgje og gissar på om ho er tilfeldig eller ikkje i, vil ta feil tilnærma annankvar gong. Ekvivalent kan ein krevja at nestebitsprøva vert bestått, dvs. at der ikkje finst nokon algoritme som i polynomisk tid kan forutseia det $n$te elementet i slumptalsfylgja med sannsyn vesentleg større enn 50% etter å ha observert $n-1$ element.

Literaturen om Monte Carlo-simuleringar og liknande felt er som regel nøgd med eit svakare krav. I praksis er ein som regel nøgd dersom generatoren består eit rimeleg utval av kjende slumptalstestar [7]. Det er det tredje og fjerde kravet som understrekar behovet for pseudo-slumptal. Røynlege slumptal er ikkje reproduserbare, og dei kan heller ikkje produserast raskt nok for alle bruksområde.

Kravet om at slumptalsfylgja skal vera reproduserbar er omdiskutert [14]. To føremonar vert framhaldne når reproduksjon er mogleg. Det eine er ved Monte Carlo-simuleringar, der reproduksjon gjev betre høve til å gjenta og validera eksperimentet. Den andre, som i mindre grad er omdiskutert, er at reproduserbare slumptal mogleggjer regresjonstesting (og einingstesting (*unit testing*)), og det er ein openberr føremon ved utvikling av samansette dataprogram.

Det verkar rimeleg å gå ut frå at der finst eit behov for reproduserbare slumptalsgeneratorar, sjølv om der òg er mange bruksområde som klarer seg utan. I resten av artikkelen vil me sjå utelukkande på reproduserbare pseudo-slumptalsgeneratorar.

Slumptalsgeneratoren – ei tilstandsmaskin

Ein slumptalsgenerator er ein tilstandsmaskin, som me kan uttrykka med ein funksjon,

$$\mathsf{next} : \mathcal{S} \to \mathcal{R} \times \mathcal{S}, \tag{1}$$

som definerer tilstandsovergangane med $\mathcal{S}$ som tilstandsrom og $\mathcal{R}$ som utfallsrom for slumptala. Me kan sjå på ein generator g som eit element $g \in \mathcal{S}$. Funksjonen $\mathsf{next}\, g = (r, g')$ gjev soleis ein ny generator g', ved sidan av slumptalet r. Slumptalsgeneratoren må initialiserast med eit *frø* (*seed*), g_0, og definerer då slumptalsfylgja $[r_1, r_2, \ldots]$ ved rekursjonen $\mathsf{next}\, g_{i-1} = (r_i, g_i)$. I dei fleste implementasjonar vil tilstanden g vera skjult, og API-et gjev berre ein funksjon som returnerer eit slumptal, utan argument.

Utfallsrommet $\mathcal{R}$ er lite vesentleg. Der er mange gode teknikkar for å omsetja slumptal frå eit utfallsrom til eit anna. I dei døma som me skal sjå, vil $\mathcal{R}$ vera eit interval $(\mathcal{R}_{\min}, \mathcal{R}_{\max})$ av heiltal, med $\mathcal{R}_{\min} \in \{0, 1\}$. Generatorar som returnerer flyttal vil typisk berre dela på $\mathcal{R}_{\max}$ for å få eit tal i intervallet $(0, 1)$.

Sidan slumptalsgeneratoren er ein endeleg tilstandsmaskin, er det openbert at han vil gjenta seg sjølv. Perioden til slumptalsfylga $[r_i]$ er definert som den minste verdien $t > 0$ slik at $r_i = r_{i+t}$ for all $i \geq k$ for some k. Perioden gjev ei øvre skranke på kor mange slumptal ein kan bruka, og er soleis ein kritisk ytingsparameter.

Den mest kjende og brukte slumptalsgeneratoren er truleg lineære kongruensgeneratorar, ogso kjend som Lehmers algoritme, som definerer

$$\begin{aligned}(g_i, g_i) &= \mathsf{next}\, g_{i-1}, \quad \text{der} \\ g_i &= a \cdot g_{i-1} + c \bmod m,\end{aligned}$$

der det returnerte slumptalet r_i er lik tilstanden g_i. Det er viktig å velja gode koeffisientar a og c for å få tilfredsstillande fordeling på slumptala. Modulusen m er vanlegvis eit primtal litt mindre enn maksimumsverdien for heiltal på maskina generatoren vert designa for.

Kryptografien kan tilby slumtalsgeneratorar med langt betre ganrantiar for den statistiske fordelinga av slumptala. Eit kvart blokksiffer, t.d. AES, kan køyrast i sokalla *counter mode*, som gjev rekursjonen

$$(e_k(g_i), g_i + 1) = \mathsf{next}\, g_i,$$

der e_k er krypteringsfunksjonen med nykjel k. Med AES er resultatet uniformt fordelte 128-bits heiltal.

Tradisjonelt har kryptografiske slumptalsgeneratorar vore lite brukte utanfor kryptografien, gjerne grunngjeve med at dei krev for mykje reknekraft. Med raskare maskiner og maskinvareimplementasjon av vanlege kryptoalgoritmar (t.d. i Intels CPU-ar) er det tvilsomt om dette lenger er ein relevant grunn. Likevel er vert dei ofra liten plass i literaturen innanfor Monte Carlo-simulering og *Computational Statistics*.

Parallell slump

Tilstandsmaskina (1) er grunnleggjande sekvensiell, og problemstillinga med slumptalsgenerering for parallelle program tok til å dukka opp på 1980-talet, særleg innanfor simuleringar av partikkelfysikk (t.d. [15, 4, 14, 5]).

Den mest naïve løysinga er å dela éin global slumptalsgenerator, anten ved felles global tilstand eller vha. ein eigen tråd som leverer slumptal til andre trådar. Denne løysinga har to openberre lytar. For det fyrste kan slumptalsgeneratoren lett verta ein

flaskehals ved intensiv bruk av slump. For det andre vil der oppstå kapplaup mellom trådane slik at systemet ikkje kan vera deterministisk.

Ei anna vanleg løysing er å instansiera éin slumptalsgenerator per tråd (evt. per fysisk prosessor) på starten av køyringa. I tillegg til å velja ulikt frø for kvar instans, kan ein òg variera parametrar i algoritmen, t.d. koeffisientane a og c i Lehmers algoritme. Mange forfattarar har studert velegna koeffisientar for slik parallellisering, t.d. [10]. Denne løysinga er godt egna dersom arbeidet er deterministisk fordelt mellom trådane, og talet på trådar er kjent på førehand. Der er to tilfelle der løysinga er uegna. Dersom trådane dynamisk hentar oppgåver frå ein kø, får me kapplaup mellom trådane og systemet vert ikkje deterministisk. Den andre situasjonen er der nye trådar vert oppretta dynamisk under køyringa.

Den tredje, og mest generelle, løysinga som me har funne i literaturen er *splittbare slumptalsgeneratorar*. Me seier at ein slumptalsgenerator er splittbar dersom han i tillegg til next-operasjonen (1) òg har ein splitt-operasjon:

$$\mathsf{split} : \mathcal{S} \to \mathcal{S} \times \mathcal{S}.$$

Denne splittoperasjonen er svært fleksibel og opnar for ulike formar for parallellisering. T.d. kan ein bruka split kvar gong ein opprettar ein ny tråd. Kallet $(g_1, g_2) = \mathsf{split}\, g$ gjev to nye slumptalsgeneratorar for einkvan tilstand g. Den nye tråden kan få g_1 mens den gamle bruker g_2, og dei to kan køyre i parallell, uavhengig av kvarandre. Me skal sjå andre bruksmåtar etter kvart. API-et med split og next vart fyrst definert av Burton *et al.* [1], og dette API-et vert implementert i Haskells System.Random-bibliotek.

Der er få konkrete konstruksjonar av splittbare slumptalsgeneratorar i literaturen, og det viser seg at Haskells implementasjon ikkje har tilfredsstillande statistiske eigenskapar. Dokumentasjonen [12] skriv:

> *Until more is known about implementations of split, all we require is that split deliver generators that are (a) not identical and (b) independently robust (...).*

Claessen og Pałka [3] viste eit konkret døme på eit program for automatisk testing der moglege testfall ikkje vart testa med uniform fordeling, slik dei skulle ha vorte med ein velegna slumptalsgenerator.

3 Konstruksjon av splittbare slumptalsgeneratorar

Der klassiske slumptalsgeneratorar dannar ei fylgje av generatorar $[g_1, g_2, \ldots]$, dannar splittbare slumptalsgeneratorar eit tre, der kvar generator g har anten eitt barn g_N generert av next eller to barn (g_L, g_R) generert av split. Denne idéen dukker fyrst opp på 1980-talet [15] og vart kjend som Monte Carlo-tre.

Den fyrste konstruksjonen som me har funne er Lehmer-tre [4] der ein bruker Lehmers algoritme med ulike koeffisientar, slik at ein får splittfunksjonen

$$\mathsf{split}\, x = (a_L x + c_L \bmod m, a_R x + c_R \bmod m).$$

Haskells standardbibliotek System.Random nyttar i prinsippet Lehmer-tre, men med små variasjonar og i kombinasjon som ein multippel kongruensgenerator. Dvs. at han er bygd opp av to deltilstander der kvar deltilstand er definert ved Lehmers algoritme, men med ulike koeffisientar og moduli.

Me kan visa at Lehmer-tre ikkje gjev uavhengig fordelte slumptal. Definer

$$\begin{aligned}(g_L, g_R) &= \mathsf{split}\, g,\\ (g_{LL}, g_{LR}) &= \mathsf{split}\, g_L,\\ (g_{RL}, g_{RR}) &= \mathsf{split}\, g_R.\end{aligned}$$

Då kan me skriva

$$\begin{aligned}g_{LR} &= a_R(a_L g + c_L) + c_R \bmod m = a_R a_L g + a_R c_L + c_R \bmod m, , \quad \text{og}\\ g_{RL} &= a_L(a_R g + c_R) + c_L \bmod m = a_L a_R g + a_L c_R + c_L \bmod m,\end{aligned}$$

Tek me differansen mellom dei to likningane, har me

$$g_{LR} - g_{RL} = (a_R c_L + c_R) - (a_L c_R + c_L) \bmod m,$$

som viser eit affint avhenge mellom g_{LR} og g_{RR}. Det same avhenget oppstår òg i slumptalsgeneratoren i Haskell, unnateke når ein av deltilstandane er 0, noko som skjer med sannsyn om lag 2^{-30}.

Splittbar slumptalsgeneratorar som omgrep skriv seg frå Burton *et al* [1]. Dei føreslog tre konstruksjonar for å splitta ein eksisterande, sekvensiell slumptalsfylgje $[r_1, r_2, \ldots, r_n]$. *Leapfrog* splittar i

$$[r_1, r_3, r_5, \ldots] \quad \text{og} \quad [r_2, r_4, r_6, \ldots], \tag{2}$$

medan jamnt delte segment splittar i

$$[r_1, r_2, \ldots, r_{\lfloor n/2 \rfloor}] \quad \text{og} \quad [r_{\lfloor n/2 \rfloor + 1}, r_{\lfloor n/2 \rfloor + 2}, \ldots, r_n]. \tag{3}$$

Den siste metoden, med tilfeldig delte segment, splittar i

$$[r_1, r_2, \ldots] \quad \text{og} \quad [r_{r_1}, r_{r_1+1}, \ldots]. \tag{4}$$

Tilsynelatande krev alle desse tre metodane at ein genererer store delar av slumptalsfylgja før ho kan splittast. Det kan vera riktig for nokre algoritmar, men det er ikkje tilfellet for t.d. Lehmers algoritme, som tillet hopp i talfylgja i konstant tid. Fleire andre forfattarar har studert desse metodane med tanke på parallell instantiering av eit endeleg tal på generatorar. Me har ikkje funne arbeid som analyserer dei med tanke på at generatoren skal kunne splittast dynamisk og vilkårleg under køyring.

Splitting med dei tre metodane frå [1] krev svært lang periode. Ved sekvensiell bruk kan ein bruka $O(n)$ tal frå ein slumptalsfylgje av lengde m. Derimot kan ein berre splitta $\log n$ gongar før ein får segment av lengd 1. Splitting bruker dermed opp perioden i eksponensiell takt.

Claessen og Pałka [3] definerte ein splittbar slumptalsgenerator basert på kryptografiske spreidefunksjonar (*hashing*). I prinsippet er det ei generalisering av *counter mode*, som me har definert tidlegare. Tilstandsrommet er mengda av alle binære strengar $\{0,1\}^*$, der split kan skrivast som

$$\mathsf{split}\, \mathbf{x} = (\mathbf{x}||0, \mathbf{x}||1),$$

og $||$ tyder konkatenering, og next som

$$\mathsf{next}\, \mathbf{x} = (h_k(\mathbf{x}), \mathbf{x}||1),$$

der h er spreidefunksjonen med nykjel k. Der *counter mode* legg 1 til eit heiltal i tilstandsovergangen, vil Claessen og Pałkas generetor leggja til ein ny bit i ein streng. Spreidefunksjonen som dei nyttar er Merkle-Damgård-konstruksjonen. og det gjer at dei ikkje treng å lagra heile strengen $\mathbf{x}$. Strengen vert delt i blokkar på 256 bits, og det rekk å lagra den siste (ofte ufullstendige) blokken saman med spreideverdien av alle dei føregåande blokkane, slik at køyretida og lagringsbehovet er konstante. Den endelege konstruksjonen åt Claessen og Pałka omfattar nokre variasjonar og triks for å gjera implementasjonen meir effektiv, men prinsippet er som skildra over. Dei har publisert ein Haskell-implementasjon i pakka tf-random.

4 Funksjonell programmering og Haskell

Funksjonell programmering og funksjonar er til dels tvetydige omgrep, der ulike programmeringsspråk kan ha strenge eller liberale tolkningar av dei same omgrepa. Funksjonar er kjenneteikna av at dei returnerer ein verdi, i motsetjing til prosedyrar som kommuniserer med andre delar av programmet vha. argument og/eller globale variablar. Funksjonsuttrykk kan dermed inngå i komplekse aritmetriske uttrykk, noko som ofte gjev ei formulering lik det me ville brukt for å forklara framgangsmåten for menneskelege lesarar.

I matematikken er funksjonar strengare definert. Funksjonsuttrykket $f(\mathbf{x})$ gjev alltid same verdi, uansett kvar og når det vert brukt. Funksjonsuttrykket kan erstattast med verdien, utan å endra tydinga av dette uttrykket eller andre uttrykk i eit lengre argumentet. Denne eigenskapen vert kalla *referentiell transparens*, og funksjonane vert då kalla *reine funksjonar*.

Motsatsen til reine funksjonar er funksjonar med bieffektar, t.d. ved at dei til liks med prosedyrar endrar global tilstand (globale variablar), og dei gjev ingen referentiell transparens. Om ein då byter ut funksjonsuttrykket med verdien, vert bieffekten undertrykt og semantikken vert endra. Slike bieffektar kan vera vanskelege å analysera, og generelt er referentiell transparens ein føremon for å analysera og forstå programmet.

I parallelle program har reine funksjonar den fordelen at dei er trådsikre. Utan bieffektar er der ingen verknad utover funksjonen, og dermed ingen verknad på andre trådar. Dersom språket forbyd bieffektar og garanterer referentiell transparens, slik som reine funksjonelle språk gjer, kan kompilatoren i prinsippet parallellisera automatisk.

Det er verdt å merkja seg at funksjonell programmering ikkje føreset eit funksjonelt språk, og i alle fall ikkje eit reint funksjonelt språk. Me taler gjerne om funksjonell programmering når ein i stor grad bruker reine funksjonar og unngår bieffektar so langt det er råd. Hybridspråk som python og scala er vorte svært populære i dei seinare åra, med syntaktisk støtte for mange funksjonelle konstruksjonar ved sidan av imperativ og objekt-orientert støtte. Det legg godt til rette for å velja ein funksjonell stil der det gjer koden lettlest, men det gjev ingen garantiar om referentiell transparens som kompilatorane kan bruka til optimalisering eller automatisk parallellisering.

map-funksjonen

Funksjonelle programmeringsspråk har ofte ein eigen kjærleik til lister som datatype, med ei lang rekkje innebygde operasjonar på lister. Ein av dei grunnleggjande listefunksjonane er `map`, som i Haskell har typedeklarasjonen

```
map :: ( a -> b ) -> [a] -> [b]
```

Dvs. `map` tek to argument, ein funksjon frå ein vilkårleg type `a` til type `b`, samt ei liste over type `a`. Resultatet er ei liste over type `b`. Eit kall til `map` ser slik ut

```
map f xs
```

Det som `map` gjer er å bruka funksjonen `f` på kvart element i lista `xs` $= [x_1, x_2, \ldots]$ for å returnera lista [`f` x_1, `f` $x_2, \ldots$]. Prinsippet ved `map` er egna for alle typar samlingar (mengder, tre, vektorar, osv.), ikkje berre for lister.

Denne `map`-funksjonen er eit glimrande grunnlag for parallellisering. So lenge `f` er utan bieffektar, speler det inga rolle i kva rekkjefylgje me reknar ut elementa `f` x_i som skal returnerast. Me kan gjerne gjera det parallellt. Denne idéen er òg grunnlaget for map/reduce-paradigmet som Google lanserte for skybasert utrekning.

Parallellisert `map` er eit døme på dataparallell programmering; dvs. at alle trådane køyrer same instruksjonar (funksjonen `f`) på ulike data. Dermed er parallell `map` egna på GPU-ar og andre SIMD-arkitekturar (*Single Instruction, Multiple Data*), der det ikkje er råd å køyra ulike instruksjonar i dei ulike trådane.

Slumptal i funksjonelle språk

Me har skildra slumptalsgeneratorar som tilstandsmaskiner, med ein funksjon $\mathsf{next} : \mathcal{S} \to \mathcal{R} \times \mathcal{S}$, som definerer tilstandsovergangane. Frå imperative språk er me vande med at tilstanden $s \in \mathcal{S}$ er skjult og lagra i ein global, statisk eller delt variabel. Me får tilgang til ein funksjon, sei `rnd`, som returnerer eit tilfeldig tal frå $\mathcal{R}$. Denne funksjonen vil kalla $(r, g') = \mathsf{next}\, g$, oppdatera den globale tilstanden $g \leftarrow g'$, og returnera r.

I reine funksjonelle språk finst ingen global tilstand, og slumptalshandteringa må ordnast på ein annan måte. Me ser lett at `rnd`, som returnerer eit nytt tal kvar gong, ikkje er referentielt transparent. Dersom to parallelle trådar bruker `rnd` med den same globale tilstanden g, so vil resultatet avhenga av kva tråd som rekk å kalla `rnd` fyrst, og programmet vert dermed ikkje deterministisk.

Utan global tilstand må ein kvar probabilistisk funksjon f få slumptalsgeneratoren g som argument, og dei må returnera den oppdaterte tilstanden g' etter å ha kalla $(r, g') = \mathsf{next}\, g$ saman med den eigentlege returverdien. Neste gong ein probabilistisk funksjon vert brukt, må ein bruka den nye tilstanden g'. Dersom g vert brukt fleire gongar får ein gjentekne, og ikkje tilfeldige tal.

Ein slik probabilistisk funksjon som tek slumptalsgeneratoren som ekstraargument, kan ikkje brukast som argument til `map`. For kvart element x_i som vert handsama, får me ein ny slumptalsgenerator g_i' som skal brukast ved handsaming av neste element x_{i+1}. Dét er eit problem som `map` ikkje er definert for.

Heile tanken med å senda slumptalsgeneratorar rundt mellom funksjonar kan synast tungvint. So er det ogso mogleg å unngå å gjera det eksplisitt, ved å bruka monadar. Me skal ikkje gå nærare inn på kva monadar er her. Lat oss berre understreka at monadar gjer det mogleg å bruka probabilistiske funksjonar som skildra her saman med `map`, men berre ved sekvensiell køyring. Monaden vil tvinga fram ein sekvensiell ordning som definitivt hindrar parallellisering, og moglegvis også hindrar ein del kompilatoroptimalisering. Når me skal parallellisera ein probabilistisk prosess, tarv me altso ei anna løysing.

Dataparallell programmering med slumptal

Der er ei enkel løysing som opnar for dataparallelle, probabilistiske program. Sei at me har ei liste eller vektor $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ over ein eller annan type A og at me har ein

probabilistisk funksjon $f : A \to B$. Som nemnd over, kan me omsetja f til ein rein funksjon $f' : A \times \mathcal{S} \to B \times \mathcal{S}$, der me har eit ekstra argument inn og ut, for slumptalsgeneratoren.

Dersom me har ein splittbar slumptalsgenerator, so kan me danna oss ei liste med uavhengige generatorar $[g_1, g_2, \ldots, g_n]$ ved rekursjonen $(g_i, g_i') = \mathsf{split}\, g_{i-1}'$. Desse generatorane kan so parrast med elementa frå $\mathbf{x}$ og danna lista $\mathbf{x}' = [(x_1, g_1), (x_2, g_2), \ldots, (x_n, g_n)]$. No kan me anvenda f på kvart element i $\mathbf{x}$ ved uttrykket `map` f' $\mathbf{x}'$. Sidan f' er ein rein funksjon, er dette uttrykket lovleg i rein funksjonell programmering og kan parallelliserast ukritisk.

Denne framgangsmåten gjev oss ei deterministisk kopling mellom slumptalsgeneratoren g_i og dataobjektet x_i der han vert brukt. Denne komplinga er heilt naudsynt for å få deterministisk oppførsel i dataparallell programmering, der me ikkje veit korleis data vert fordelt mellom trådane ved køyring, eller kva rekkjefylgje dei vert handsama i. Dels avheng det av talet på trådar som vert bestemt ved køyring, og dels er det opp til kompilatoren å fordela arbeidet. Prinsippet for eit deterministisk, pseudo-tilfeldig, dataparallelt program er altso at *slumptalsgeneratoren må fylgja dataobjekta, og ikkje eksekusjonstråden*.

5 Parallell GA

Genetiske algoritmar er eit godt problem for å illustrera probabilistisk, parallell programmering. Algoritmen er relativt enkel å forstå, og parallelliteten er openberr.

Genetiske algoritmar

Genetiske algoritmer (GA) løyser optimeringsproblem, dvs. problem på formen

$$\min_{\mathbf{x}} f(\mathbf{x}).$$

Funksjonen f kaller me *kostnadsfunksjonen*. Dersom me formulerer problemet som eit maksimeringsproblem, vert tilsvarande funksjon gjerne kalla *fitness*, som høver godt med røtene som algoritmen har i evolusjonsteorien. Her fylgjer me [6] og bruker kostnadsfunksjonen.

Ei mogleg løysing $\mathbf{x}$ i domenet åt kostnadsfunksjonen f vert kalla eit *kromosom*. Den genetiske algoritmen startar med ein *populasjon* av tilfeldig trekte kromosom, og imiterer ein evolusjonsprosess der kromosom med høg kostnad døyr ut, medan dei med låg kostnad formerer seg.

Genetiske algoritmar finst i ulike variantar. Den som me tek for oss her vert kalla generasjonsbasert GA. Algoritmen er iterativ og kvar runde vert kalt ein *generasjon*, der heile eller mesteparten av populasjonen vert skifta ut for kvar iterasjon. Me får altso ei fylgje $[p_1, p_2, ...]$ av populasjonar, der p_1 er tilfeldig trekt, og p_i vert rekna ut som ein probabilistisk funksjon av p_{i-1}. Denne funksjonen er sjølve evolusjonsprosessen. Det er vanleg å anta at talet på kromosom er konstant frå generasjon til generasjon.

Evolusjonsprosessen består av fylgjande steg:

1. Kostnadsberekning. Funksjonen $f(\mathbf{x})$ vert rekna ut for kvar $\mathbf{x}$.
2. Utval. Me vel ut (1) kromosom som skal overleva til neste generasjon, og (2) kromosom som skal formera seg. Utvalet kan gjerast på ulike måtar, so lenge kromosom med låg kostnad hev stort sannsyn for å verta valde.
3. Kryssing (*crossover*). Dei kromosoma som skal formera seg vert ordna i par, og for kvart par vert to nye kromosom rekna ut vha. *crossover*-funksjonen. Dei to nye kromosoma inngår i populasjonen i neste generasjon.

4. Mutasjon. Kvart kromosom, med eit vist sannsyn, vert utsett for ei lita, tilfeldig endring før det inngår i den nye generasjonen. Det er mogleg å halda dei beste kromosoma utanfor mutasjonssteget. Dette vert kalla *elitisme*.

For kvart av stega 2, 3 og 4 er der mange moglege framgangsmåtar, som er egna for ulike optimeringsproblem. Seinare i avsnitt 5 presenterer me dei funksjonane som me brukte i slepebåtproblemet [2], meint som eit døme.

Dataparallell programmering med Repa

Repa er eit av fleire bibliotek for parallell programmering i Haskell. Me skal gje eit kort innblikk i dei eigenskapane me har brukt, men for vidare detaljar anbefaler me [9].

Repa tilbyr ein parametrisert datatype `Array r sh e` for vektorar og matriser. Parametrane er henhaldsvis representasjonen `r`, dimensjonen `sh` og datatype for elementene `e`. Dei to siste skulle vera greie å forstå. Representasjonen bestemmer korleis elementa er lagra. Dei viktigaste representasjonane er `U` for *unboxed*, `V` for *boxed* og `D` for *deferred*. Boksa data (`V`) er lagra indirekte gjennom ei peikar, og treng ikkje vera ein utrekna verdi. «Boksen» kan innehalda eit uttrykk som ikkje vert utrekna før verdien trengst. Ein uboksa verdi (`U`) er derimot lagra kompakt og må vera ferdig utrekna.

Den tredje representasjon `D` vil seia at matrisa ikkje er manifestert. Me kan ikkje henta ut data frå ein slik matrise, men me kan be Repa om å rekna ut matrisa slik at me får ein ferdig boksa eller uboksa matrise. Me kan be om å få denne utrekninga gjort anten sekvensielt (`computeS`) eller parallelt (`computeP`). Denne `D`-representasjonen er viktig for ytinga, fordi fleire matriseoperasjonar kan setjast saman utan at Repa treng laga ein manifestert matrise mellom kvart steg, og kompilatoren får meir fridom til å optimalisera.

Repa egnar seg på mange måtar best med uboksa typar, og det var ei utfordring at slumptalsgeneratorane ikkje utan vidare har nokon uboksa representasjon. Boksa representasjon tillet Haskell i sin latskap å utsetja alle utrekningar, og dermed vil ikkje `computeP` tvinga fram utrekningane i parallell slik ein skulle forventa. Me fann fylgjande triks [8] for å løysa dette problemet:

```
import Control.DeepSeq (NFData(..))
force :: (Monad m,Shape sh, NFData e) =>
         Array D sh e -> m Array V sh e
force a = do
    r  <- computeP a
    !b <- computeUnboxedP (Repa.map rnf r)
    return r
```

Funksjonen `force` tek ei matrise i `D`-representasjon og returnerer ei boksa matrise. Det fyrste han gjer er å be om parallell utrekning for å få ei boksa matrise `r`, men fordi Haskell er lat, skjer der ingen umiddelbar utrekning. Neste line bruker `rnf`-funksjonen som reduserer eit uttrykk til normal form. Det vil i praksis seia at innhaldet i boksen vert utrekna. Vha. `Repa.map` reduserer me kvart einskild element til normal form, og returverdien er ei matrise over ein tom type. Denne matrisa kan me so be om å få utrekna parallelt som ei uboksa matrise, og fordi returverdien er uboksa og vert bunden til ein striks variabel (`b`) er det ikkje lenger råd å utsetja utrekninga.

Genetiske Algoritmar i Repa

Populasjonen vert representert som ein éin-dimensjonal Repa-tabell, der elementa er tuplar (x, g, y). Det fyrste elementet x er kromosomet, g er slumptalsgeneratoren

(tilstanden), og y er kostnaden $c(x)$ (type Double). Under mellomrekning kan y ha andre tolkingar, men er alltid type Double.

Instantiering

For å instantiera ein ny populasjon treng me ein funksjon r som tek ein slumptalsgenerator g og returnerer $(x, g', f(x))$ der x er eit tilfeldig kromosom vald vha. g og g' er den nye generatortilstanden. Me startar med å generera slumptalsgeneratoren g med eksternt frø. Dernest lagar me ein Repa-tabell P_g med generatorar g_i generert ved rekursjonen $(g_i, g_i') = \mathsf{split}\, g_{i-1}'$ og $g_0' = g$. Ved å *mappa* r over P_g får me den fyrste populasjonen P_0.

Seleksjon

Til utvalssteget har me brukt roulette med rangvekting. Det krev at populasjonen fyrst vert sortert etter kostnad. Her utnyttar me at Repa bruker muterbare *arrays* internt, og bruker ein sekvensiell sorteringsalgoritme frå Vector-biblioteket i Haskell. Før sortering må tabellen manifesterast, noko me gjer parallelt vha. trikset frå avsnitt 5. Tabellen vert delt i to; dei N_{keep} beste kromosoma dannar mengda P_{keep}, og resten dannar komplementet $P_{\text{keep}}^{\text{C}}$. Kromosoma frå P_{keep} overlever til neste generasjon og har ein sjanse til å få born. Kromosoma i $P_{\text{keep}}^{\text{C}}$ vert erstatta av nye born.

Rangvekting vil seia at sannsynet p_i for at kromosom $\mathbf{x}_i$ skal verta foreldre er proposjonal med plasseringa i kostnadsorden. Dersom $\mathbf{x}_i$ står på plass r_i i den sorterte lista i P_{keep}, reknar me ut

$$p_i = r_i \cdot \frac{2}{n(n+1)},$$

og let $y = p_i$ erstatta kostnaden i tuppelen for kvart element i P_{keep}.

Neste steg er å erstatta kromosoma i komplementet $P_{\text{keep}}^{\text{C}}$ med tilfeldig trekte foreldre ihht. sannsynsfordeling p_i. For å kunna parallellisera dette steget bruker me slumptalsgeneratorane frå dei eksisterande kromosoma i $P_{\text{keep}}^{\text{C}}$. Me definerer ein funksjon

$$s(P_{\text{keep}}, (\mathbf{x}, g, y)) = (\mathbf{x}', g', y),$$

som bruker slumptalsgeneratoren g til å dra eit tilfeldig kromosom $\mathbf{x}'$ frå P_{keep} ihht. sannsynsfordeling definert over. Det siste elementet y i returverdien vert ikkje brukt. Foreldra vert generert ved å *mappa* funksjonen $s(P_{\text{keep}}, \cdot)$ over $P_{\text{keep}}^{\text{C}}$, slik at me får ein tabell P_{parens}.

Kryssing

I kryssingssteget bruker me *single-point* crossover. Gjeve to foreldre $\mathbf{x} = (x_1, x_2, \ldots, x_m)$ og $\mathbf{y} = (y_1, y_2, \ldots, y_m)$, dreg me eit tilfeldig heiltal $c \in \{0, 1, \ldots, m\}$ og lat $\mathbf{x}' = (y_1, y_2, \ldots, y_c, x_{c+1}, \ldots, x_m)$ og $\mathbf{y}' = (x_1, x_2, \ldots, x_c, y_{c+1}, \ldots, y_m)$. Sidan me òg må ta vare på slumptalsgeneratoren, definerer me ein kryssingsfunksjon

$$k((\mathbf{x}, g_1), (\mathbf{y}, g_2)) = ((\mathbf{x}', g_1'), (\mathbf{y}', g_2)),$$

der g_1 vert brukt til å generera c og den nye tilstanden g_1', og $\mathbf{x}'$ og $\mathbf{y}'$ er definert som over.

Tabellen $P_{\text{keep}}^{\text{C}}$ må delast i to like delar, slik at me får foreldra i par. Når det er gjort kan me anvenda k på kvart par (vha. operasjonen `Repa.zipWith`). Resultatet er en tabell med par av avkom, der para kan splittast for å få ein ny tabell som saman med P_{keep} dannar den nye populasjonen før mutasjon.

	CPU-tid	Klokketid
1 CPU	19,29s	20,28s
4 CPU-ar	26,03s	10,02s

Tabell 1: Døme på køyretid.

Mutasjon

Me bruker ein enkel mutasjonsfunksjon $m(\mathbf{x}, g) = (\mathbf{x}', g')$. Kvart gen i kromosomet $\mathbf{x}$ vert erstatte med 10% sannsyn av eit uniformt fordelt tilfeldig gen. I mutasjonstrinnet held me utanfor dei N_{elite} beste kromosoma, som ikkje vert muterte. Merk at me ikkje ynskjer å rekalkulera kostnadene og sortera i dette steget, og difor bør $N_{\text{elite}} \leq N_{\text{keep}}$. Tilsvarande tidlegare steg kan me *mappa* mutasjonsfunksjonen m på populasjonstabellen der dei N_{elite} fyrste kromosoma er haldne utanfor. Resultatet er populasjonen P' for neste generasjon.

Me bruker eit svært enkelt stoppvilkår, der me køyrer m generasjonar for ein førehandsvald verdi av m.

Resultat

Me har implementert genetiske algoritmar i samsvar med framgangsmåten som skildra. Sjølv om arbeidet ikkje er avslutta, so er prototypen fullt funksjonell og løyser reelle optimeringsproblem. I forsøk får me resultat tilnærma like dei som me får med andre, sekvensielle implementasjonar. Arbeidet som gjenstår er fyrst og framst optimalisering av minne- og resursbruk, t.d. bruk av strikse variablar, uboksa typar og *inlining* av funksjonar. Koden treng refaktorisering for å verta meir lesbar.

Nokre innleiande testar med døme på køyretid er vist i tabell 1. I GA har me brukt 2000 generasjonar, 500 kromosomar à to gen, der $N_{\text{keep}} = 200$ kromosomar overlever og formerer seg, med ein elitisme på $N_{\text{elite}} = 20$. Me ser at der er ein tydeleg parallelliseringsgevinst, men òg tydeleg *overhead*.

6 Konklusjon

Me har utvikla ein parallell implementasjon av genetiske algoritmar i Haskell. I artikkelen har me diskutert grunnleggjande prinsipp for dataparallell implementasjon av probabilistiske prosessar. Parallelliseringa er heilt avhengig av ein *splittbar slumptalsgenerator*. Trass i at behovet for slike slumptalsgeneratorar har vore kjend lenge, er der få kjende konstruksjonar. Den einaste tillitsvekkjande konstruksjonen me finn kom so seint som i 2013 [3]. Konstruksjonen som vert brukt i standardbiblioteket for Haskell gjev ikkje uavhengige slumptal når splitting vert brukt.

Med ein splittbar slumptalsgenerator er det relativt enkelt å skriva ein dataparallell og deterministisk implementasjon av genetiske algoritmar. Me har brukt Haskell til vår prototyp, men prinsippa er overførbare til andre språk, både imperative og funksjonelle, so lenge der er støtte for dataparallellitet.

Der gjenstår mykje arbeid på implementasjonen vår, og ytinga er difor enno ikkje god. Likevel viser testane ein tydeleg parallelliseringseffekt som demonstrerer at metodane er gyldige. So langt i prosjektet har målet vore eit konseptprov og demonstrasjon av probabilistisk, dataparallell programmering. Når me tek omsyn til at der førebels ikkje er lagt arbeid i optimalisering av koden, er ytinga ikkje skuffande.

Referansar

[1] FW Burton and RL Page. Distributed random number generation. *Journal of Functional Programming*, 2(2):203–212, April 1992.

[2] Robin T. Bye and Hans Georg Schaathun. An improved receding horizon genetic algorithm for the tug fleet optimisation problem. In *Proceedings of the 28th European Conference on Modelling and Simulation*, 2014.

[3] Koen Claessen and Michał H. Pałka. Splittable pseudorandom number generators using cryptographic hashing. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, Haskell '13, pages 47–58, New York, NY, USA, 2013. ACM.

[4] Paul Frederickson, Robert Hiromoto, Thomas L. Jordan, Burton Smith, and Tony Warnock. Pseudo-random trees in monte carlo. *Parallel Computing*, 1(2):175 – 180, 1984.

[5] John H Halton. Pseudo-random trees: multiple independent sequence generators for parallel and branching computations. *Journal of Computational Physics*, 84(1):1–56, 1989.

[6] Randy L. Haupt and Sue Ellen Haupt. *Practical Genetic Algorithms*. John Wiley and sons, inc., second edition, 2004.

[7] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, third edition edition, 1998.

[8] Kosmikus (Andres Löh). Answer to 'parallel repa code doesn't create sparks', 2013. `http://stackoverflow.com/questions/16097418/parallel-repa-code-doesnt-create-sparks`.

[9] Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly, 2013.

[10] Michael Mascagni. Parallel linear congruential generators with prime moduli1. *Parallel Computing*, 24(5-6):923–936, 1998.

[11] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Inc., 1997.

[12] Ryan Newton. The random package, 2011. Haskell Random Number Library. Documentation. Available at `http://hackage.haskell.org/package/random-1.0.1.1`.

[13] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly, 2008.

[14] Ora E. Percus and Malvin H. Kalos. Random number generators for MIMD parallel processors. *Journal of Parallel and Distributed Computing*, 6(3):477 – 497, 1989.

[15] Tony T. Warnock. Synchronization of random number generators. *Congressus Numerantium*, 37:135–144, 1983.