

Genetische algoritmes en parallelisme

Auteur	: Martin van de Goor
Studentnummer	: 0349291
Begeleider	: Dick van Leijenhorst
Datum	: 17 juni 2006

Inhoud

1	INLEIDING	4
1.1	PROBLEEMSTELLING	4
1.2	MOTIVATIE	5
2	GENETISCHE ALGORITMES.....	6
2.1	WAT ZIJN GENETISCHE ALGORITMES?	6
2.2	BEGRIPPEN	7
2.2.1	Achtergrond	7
2.2.2	Gebruik in context GAs.....	8
2.2.3	Terminologie.....	8
2.3	CODERINGEN	8
2.3.1	Binaire codering	9
2.3.2	Niet-binaire coderingen.....	9
2.3.3	Keuze voor een codering	9
2.4	HET BASISALGORITME.....	9
2.5	HOE WERKEN GENETISCHE ALGORITMEN?	11
2.5.1	Schema's	12
2.5.2	Schema Theorem.....	12
2.6	GENETISCHE OPERATOREN	15
2.6.1	Selectie	15
2.6.2	Cross-over.....	17
2.6.3	Mutatie	17
2.7	VERVANGING HUIDIGE POPULATIE.....	18
2.7.1	Generational GAs	18
2.7.2	Steady-state GAs	18
2.7.3	Elitisme.....	18
3	PARALLELLISME EN GENETISCHE ALGORITMES.....	19
3.1	DEFINITIE PGA	19
3.2	WAAROM PARALLEL?.....	20
3.2.1	Waarom parallel computing?.....	20
3.2.2	Waarom PGAs?	21
3.3	TAXONOMIE VAN COMPUTERARCHITECTUREN	22
3.3.1	Flynns taxonomie.....	22
3.3.2	Topologieën.....	25
3.4	PARALLELLISATIESTRATEGIEËN	28
3.4.1	Low-level parallelisme	28
3.4.2	Decompositie.....	29
3.4.3	Threads.....	29
3.5	TAXONOMIE VAN PGAS	29
3.6	GLOBALE PARALLELLISATIE	32
3.7	MULTIPLE-DEME PGAS.....	33
3.7.1	Onderzoek	33
3.7.2	Migratie.....	35
3.8	COARSE-GRAINED PARALLELLISME	35
3.8.1	Onderzoek	35
3.8.2	Topologieën.....	36
3.9	FINE-GRAINED PARALLELLISME	36
3.9.1	Onderzoek	37
3.9.2	Topologieën.....	37
3.10	SUPERLINEARITEIT VAN PGAS	38
3.10.1	Definities	38
3.10.2	Analyse parallelle algoritmen	43
3.10.3	Oorzaken van superlineariteit	44
3.10.4	Claims van superlineariteit.....	46
4	VOORBEELDPROBLEEM	48

4.1	BESCHRIJVING	48
4.2	KEUZE PARALLEL GENETISCH ALGORITME	48
4.2.1	<i>Codering</i>	49
4.2.2	<i>Selectie</i>	49
4.2.3	<i>Cross-over</i>	50
4.2.4	<i>Mutatie</i>	51
4.2.5	<i>Computerarchitectuur</i>	52
4.2.6	<i>PGA</i>	54
4.3	IMPLEMENTATIE	55
4.4	PRESTATIEANALYSE	55
4.4.1	<i>Simulatie PGA</i>	55
4.4.2	<i>Verwachte versnelling</i>	55
5	CONCLUSIE	57
6	REFERENTIES	59

Afbeeldingen

AFBEELDING 1. TAXONOMIE VAN ZOEKTECHNIKEN UIT [ALBA02A].	6
AFBEELDING 2. EEN BESCHRIJVING VAN EEN PARALLEL GENETISCH ALGORITME NAAR [ALBA02A].	19
AFBEELDING 3. EEN TAXONOMIE VAN PARALLELE COMPUTERARCHITECTUREN NAAR [TANEN99].	23
AFBEELDING 4. TWEE MIMD-MACHINES NAAR [BARNE05]. (A) EEN MULTIPROCESSOR MET 4 PROCESSORS. (B) EEN MULTICOMPUTER MET 4 PROCESSORS.	24
AFBEELDING 5. EEN HYBRIDE SYSTEEM [BARNE05]: ZOWEL SHARED ALS DISTRIBUTED MEMORY ARCHITECTUREN WORDEN TOEGEPAST.	24
AFBEELDING 6. EEN PIRAMIDENETWERK VAN GROOTTE 16 NAAR [QUINN87]. DE APEX IS DE TOP EN DE BASE DE BODEM.	26
AFBEELDING 7. VERSCHILLENDE TOPOLOGIEËN NAAR [TANEN99]. (A) EEN STER. (B) EEN COMPLETE TOPOLOGIE. (C) EEN BOOM. (D) EEN RING. (E) EEN RASTER. (F) EEN DUBBELE TORUS. (G) EEN KUBUS. (H) EEN HYPERKUBUS.	27
AFBEELDING 8. EEN TAXONOMIE VAN PGAS VOLGENS [NOWOS99].	31
AFBEELDING 9. VERSCHILLENDE MODELLEN VAN EEN PGA, NAAR [ALBA02A]. (A) GLOBALE PARALLELLISATIE. (B) COARSE GRAIN. (C) FINE GRAIN. (D) COARSE EN FINE GRAIN. (E) COARSE GRAIN EN GLOBALE PARALLELLISATIE. (F) COARSE GRAIN EN COARSE GRAIN.	32
AFBEELDING 10. EEN SCHEMATISCHE WEERGAVE VAN EEN MASTER-SLAVE PARALLEL GENETISCH ALGORITME.	33
AFBEELDING 11. SCHEMATISCHE WEERGAVE VAN EEN MULTIPLE-DEME PGA UIT [CANTU97]. ELK PROCES IS EEN SIMPEL GA, EN ER VINDT (AF EN TOE) COMMUNICATIE PLAATS TUSSEN DE POPULATIES.	34
AFBEELDING 12. WAAR DE SUPERLINEAR SPEEDUP LINE DE GRENS TUSSEN TWEE GEBIEDEN KRUIST, KAN SUPERLINEAIRE VERSNELLING OPTREDEN VOLGENS [GUSTA90].	45
AFBEELDING 13. EEN VOORBEELD VAN EEN HANDELSREIZIGERSPROBLEEM MET VIJF STEDEN. DE AFSTANDEN ZIJN WEGGELATEN.	48
AFBEELDING 14. AFBEELDING NAAR [TANEN99]. (A) EEN 8x8 CROSSBAR SWITCH. (B) EEN OPEN CROSSPOINT. (C) EEN GESLOTEN CROSSPOINT.	53
AFBEELDING 15. AFBEELDING NAAR [TANEN99]. DE SUN ENTERPRISE 10000 SERVER. HET SYSTEEM ONDERSTEUNT MAXIMAAL 64 PROCESSORS EN 64 GB AAN GEHEUGEN.	53

Tabellen

TABEL 1. TERMEN EN HUN BIJBEHORENDE GEBRUIK IN DE CONTEXT VAN GENETISCHE ALGORITMES.	8
TABEL 2. POPULATIE MET INFORMATIE OVER FITNESS EN HET SELECTIEPROCES.	11
TABEL 3. POPULATIE NA EEN GENERATION.	11
TABEL 4. VOOR- EN NADELEN VAN SHARED EN DISTRIBUTED MEMORY SYSTEMS.	25
TABEL 5. TAXONOMIE VAN SPEEDUP MEASURES NAAR [ALBA02B].	41
TABEL 6. EDGE RECOMBINATION VAN DE ROUTES 13564287 EN 14236578. STAD 1 IS HET BEGIN- EN EINDPUNT. HET RESULTAAT VAN DE KRUISING IS 18756423.	51

1 Inleiding

Wetenschappers kijken veel van de natuur af. Een mooi recent voorbeeld hiervan is het onderzoek naar de manier waarop een gekko – een soort hagedis – aan bijv. glas blijft kleven. Een mogelijke toepassing is het maken van opnieuw bruikbare, sterk klevende materialen. Beter goed gejat dan slecht bedacht, zou je kunnen zeggen. John Holland, de uitvinder van genetische algoritmes, moet hetzelfde gedacht hebben. Holland liet zich namelijk ook door de natuur inspireren. Zijn onderzoek naar genetische algoritmes is gebaseerd op de evolutietheorie van Darwin. Deze algoritmes hebben als doel het vinden van oplossingen voor optimalisatieproblemen. Voordat we deze techniek verder gaan onderzoeken in het volgende hoofdstuk, formuleren we in de probleemstelling *wat* we precies willen onderzoeken. Het *waarom* komt daarna aan bod in de motivatie.

1.1 Probleemstelling

Uit de literatuur blijkt dat genetische algoritmes geschikt zijn voor uitvoering op parallele computerarchitecturen. Maar waarom zijn ze hiervoor geschikt, en welke voordelen zijn hieraan verbonden? Deze vraag vormt de hoofdvraag die we zullen gaan beantwoorden.

Hoofdvraag

Welke voordelen biedt het parallel boven het serieel uitvoeren van genetische algoritmes en waardoor worden deze veroorzaakt?

Deelvragen

- Welke genetische algoritmes maken optimaal gebruik van parallele processors en waarom?
- Kunnen we een concreet voorbeeld geven van een genetisch algoritme dat significant beter presteert door het parallel uitvoeren ervan?

Als we de vragen zien, kunnen we meteen een aantal nieuwe vragen stellen om bovenstaande te kunnen beantwoorden, zoals:

- Wat zijn genetische algoritmes? Hoe werken ze? Welke soorten zijn er?
- Welke voordelen heeft het parallel uitvoeren van algoritmen in het algemeen ten opzichte van serieel? Welke computerarchitecturen zijn geschikt voor parallele berekeningen?

In het tweede hoofdstuk zullen we voornamelijk de eerste vragen beantwoorden. Uitgelegd wordt wat de oorsprong van genetische algoritmen is, wat de gedachte achter de werking ervan is, welke terminologie wordt gebruikt en hoe het basisalgoritme eruitziet. Het derde hoofdstuk zal ingaan op wat de voordelen zijn van parallel (genetische) algoritmes uitvoeren. Ook zal gekeken worden naar wat parallele genetische algoritmes zijn, welke typen er bestaan en welke computerarchitecturen geschikt zijn voor de uitvoering ervan. Hoofdstuk vier zal een concreet probleem aanpakken aan de hand van de eerder beschreven theorie. Gekeken zal worden of met parallele technieken betere resultaten behaald kunnen worden dan met seriële, gegeven hetzelfde probleem. Er zal wel moeten worden vastgelegd wat ‘beter presteren’ inhoudt.

Hoofdstuk vijf is de conclusie, waarin de deelvragen en de hoofdvraag worden beantwoord. Het laatste hoofdstuk somt de gebruikte referenties op.

1.2 Motivatie

Mijn interesse in het onderwerp werd gewekt door een artikel waarin gezegd werd dat een genetisch algoritme een vergelijkbare oplossing had gevonden voor een probleem dat door mensen eerder was gepatenteerd. Met name het feit dat – mits de juiste voorbereidingen worden getroffen – een machine een ‘uitvinding’ kan doen, sprak mij erg aan. Inmiddels zijn er tientallen gevallen bekend waarin genetische algoritmen een vergelijkbare of betere oplossing voor een probleem hebben bedacht dan de mens.

Naarmate ik me meer in de materie verdiepte, kwam ik een verrassende bewering tegen: een genetisch algoritme zou op een parallelle computer onevenredig veel sneller zijn dan op een normale! Een uitdrukking die daarbij aangehaald werd, luidt: “*The whole is greater than the sum of the parts*”, oftewel: het geheel is meer dan de som van de afzonderlijke delen. Maar wanneer geldt dit, en belangrijker, waarom? Hoe steken die algoritmes in elkaar? En wat kunnen 10 kleine computers beter dan 1 grote? Antwoorden op dergelijke vragen hoop ik aan het einde te kunnen geven.

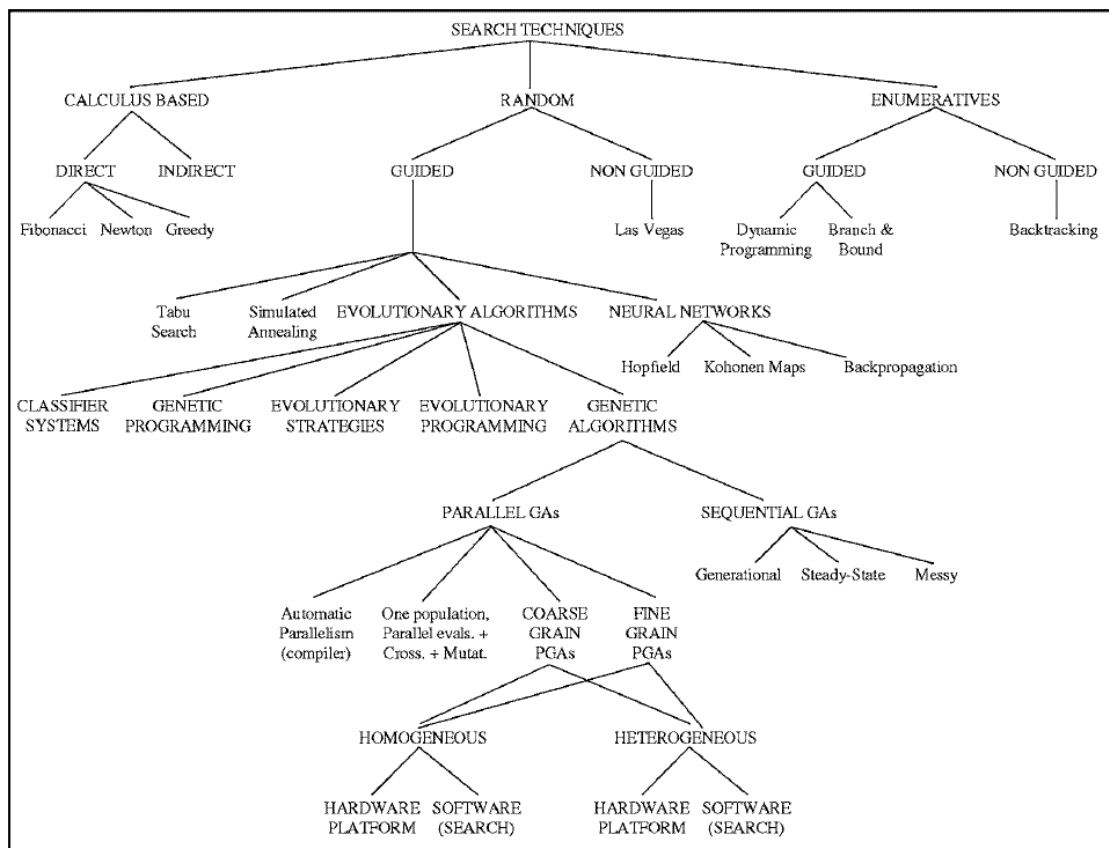
De genoemde bewering riep ook een aantal vragen bij mij op die ik *niet* in deze scriptie ga beantwoorden. Vragen als: Wat kan dit betekenen voor het internet, als je het beschouwt als een grote parallelle computer, als een *grid network*? Hoe goed begrijpen we parallelle algoritmes? Wat is hun toekomstpotentieel? Al deze vragen kunnen je motiveren bij het doen van onderzoek, ook al krijg je er geen antwoord op. Je krijgt echter wel inzicht in de werking van parallelle algoritmes. Met het oog op de toekomst kan dit zeker van pas komen, omdat ze een belangrijke(re) rol kunnen gaan vervullen in de wereld van de algoritmes. En daar draait het in de informatica tenslotte om: algoritmes.

2 Genetische algoritmes

Allereerst zullen we bekijken wat genetische algoritmes zijn, wat hun oorsprong is en waarvoor ze gebruikt worden (2.1). Daarna bekijken we welke enkele begrippen die we vaak zullen tegenkomen (2.2). De problemen die door genetische algoritmes worden aangepakt, moeten van te voren worden geformuleerd. In 2.3, dat over coderingen gaat, bespreken we hoe je dit doet. Vervolgens gaan we in op het echte algoritme (2.4). Later, in 2.6 en 2.7, bestuderen we bepaalde stappen van het algoritme in meer detail. Tussendoor bekijken we in 2.5 hoe genetische algoritmes werken door een technische analyse te geven.

2.1 Wat zijn genetische algoritmes?

Een *genetisch algoritme* (GA) is een zoektechniek voor het vinden van een oplossing voor combinatorische optimalisatieproblemen – problemen waarbij het gaat om het vinden van de beste combinatie van een aantal variabelen. Genetische algoritmes zijn een *guided random* zoektechniek, en vormen een onderdeel van de evolutionaire algoritmes (zie Afbeelding 1). Deze zijn gebaseerd op de evolutietheorie van Darwin en maken gebruik van haar technieken zoals overerving, mutatie, natuurlijke selectie en recombinitie. Deze begrippen zullen we later nader bekijken (zie 2.2).



Afbeelding 1. Taxonomie van zoektechnieken uit [ALBA02a].

Genetische algoritmes zijn bedacht door John Holland in de jaren '60 en zijn ontwikkeld door hem en zijn collega's. Zijn originele doel was om de manier waarop evolutie zoals die in de natuur voorkomt te bestuderen, en te bekijken hoe de mechanismen van natuurlijke aanpassing naar computersystemen vertaald zouden kunnen worden. Hollands boek *Adaptation in Natural and Artificial Systems* uit 1975 presenteerde het genetische algoritme als een abstractie van biologische evolutie. Maar waarom zou je evolutie willen gebruiken voor het oplossen van problemen? In [MITCH98] worden een aantal redenen hiervoor genoemd.

Bij veel problemen is het nodig om een enorm aantal oplossingsmogelijkheden te verkennen. Het is handig om in zo'n gigantische zoekruimte parallel te kunnen zoeken, zodat verschillende mogelijkheden tegelijk bekeken kunnen worden. Bovendien is het wenselijk dat een programma dat zich aanpast aan de situatie en op eenvoudige regels is gebaseerd.

Evolutie is een belangrijke bron van inspiratie voor het oplossen van dit soort problemen, en is in feite een methode om te zoeken in enorm aantal mogelijke oplossingen. In de biologie zijn de mogelijkheden alle combinaties van onze genen, en de oplossingen worden gevormd door gespecialiseerde organismen. Daarnaast is evolutie een gestuurde parallelle zoektechniek: in plaats van één organisme tegelijk te bestuderen, worden miljoenen soorten parallel getest. Ten slotte zijn de 'regels' van GAs betrekkelijk eenvoudig: genetische operatoren (recombinatie, mutatie, etc.) zorgen voor variatie in de soorten, waarna natuurlijke selectie ervoor zorgt dat alleen de best aangepaste organismen overleven.

2.2 Begrippen

Omdat genetische algoritmes geïnspireerd zijn door de evolutiebiologie, zullen we vaak termen uit de biologie tegenkomen. We beschouwen daarom de achtergrond van enkele kernbegrippen. Daarna bekijken we hoe deze kunnen plaatsnemen in de context van genetische algoritmen. We sluiten af met een overzicht van de gebruikte terminologie.

2.2.1 Achtergrond

Alle levende organismen bestaan uit cellen, en elke cel bevat *chromosomen* – strengen van DNA – die als het ware als 'blauwdruk' voor het organisme fungeren. De verzameling van alle chromosomen wordt het *genoom* van een organisme genoemd. Een chromosoom kan worden verdeeld in *genen*, die elk voor een specifiek eiwit coderen. Een gen draagt informatie voor een specifieke erfelijke eigenschap met zich mee. Van elk gen zijn twee of meer uitvoeringen mogelijk. Elke uitvoering wordt een *allel* genoemd. De *locus* is de plaats in het chromosoom waar een bepaald gen ligt.

Het *genotype* is de erfelijke informatie over een bepaalde eigenschap van een organisme. Het genotype in combinatie met de omgeving bepaalt het *fenotype*: de fysieke en mentale eigenschappen, zoals oogkleur en intelligentie.

Organismen waarbij de chromosomen in paren van twee voorkomen, worden *diploïd* genoemd; organismen die slechts één stel chromosomen per cel bezitten worden *haploïd* genoemd. Bij voortplanting vindt *recombinatie* (of *cross-over*) plaats. Er wordt ook wel van *kruising* gesproken. Bij diploïde organismen – zoals de mens – vindt eerst in elke ouder genenuitwisseling tussen elk chromosomenpaar plaats, wat leidt tot een *gameet*: een haploïde cel die uitsluitend dient voor de voortplanting. Daarna fuseren de gameten van de twee ouders tot een diploïde cel. Bij haploïde seksuele reproductie worden genen uitgewisseld tussen de enkelvoudige chromosomen van de ouders. Tijdens het kopiëren van het DNA kan *mutatie* optreden, waarbij nucleotiden (de bouwstenen van DNA) gewijzigd kunnen worden.

De *fitness* van een organisme is vaak gedefinieerd als de kans dat een organisme overleeft totdat het kan voortplanten, of als een functie van het aantal jongen dat het krijgt. Dit begrip hangt samen met dat van *natuurlijke selectie*: organismen die beter zijn aangepast aan hun omgeving hebben meer kans om nakomelingen te krijgen dan minder goed aangepaste organismen, zodat het type van het best aangepaste organisme overblijft.

2.2.2 Gebruik in context GAs

Chromosomen zijn binnen de context van GAs abstracte representaties van mogelijke oplossingen (*candidate solutions*), vaak gecodeerd in de vorm van een rij binaire getallen (*bitrij* of *bit string*). Een candidate solution wordt ook wel een *individu* genoemd. Alle individuen samen vormen de *populatie*. De genen zijn of losse bits, of korte blokken van aangrenzende bits die voor een bepaald element van een mogelijke oplossing coderen. Een allel in een bit string is 0 of 1. Bij grotere alfabetten zijn meerdere allelen mogelijk per locus. Paragraaf 2.3 gaat uitgebreider in op coderingen.

Het genotype van een individu komt overeen met de configuratie van de gebruikte codering. Er is vaak geen sprake van een fenotype bij GAs. De meeste toepassingen van genetische algoritmen gebruiken haploïde individuen met één chromosoom.

Cross-over bestaat typisch uit het uitwisselen van genetisch materiaal tussen twee haploïde ouders. Mutatie vindt plaats door op een willekeurige locus een ander teken uit het alfabet te plaatsen. Selectie gebeurt op basis van een zogenaamde fitnessfunctie (*fitness function*), soms ook wel de *objective function* genoemd. Deze functie vervangt natuurlijke selectie, en bepaalt op basis van het doel welke individuen in de huidige populatie uitgekozen wordt om te reproduceren. We zullen later terugkomen op mutatie, cross-over, en verschillende methoden van selectie.

2.2.3 Terminologie

We zijn verschillende termen tegengekomen uit de biologie. Uit de onderstaande tabel valt af te lezen wat we bedoelen met een term als we hem gebruiken in de context van GAs. Soms zullen we ook de term zelf gebruiken. Tussen haakjes is de Engelse term vermeld. Bij gebrek aan een goede vertaling voor het woord *fitness* zullen we hieraan vasthouden.

<i>Term</i>	<i>Gebruik in context GAs</i>
chromosoom	bitrij ¹ (<i>bit string</i>)
gen	kenmerk (<i>feature</i>), karakter (<i>character</i>)
allel	(karakter)waarde (<i>feature value</i>)
locus	positie in bitreeks (<i>string position</i>)
fitnessfunctie	fitnessfunctie (<i>objective function</i>)

Tabel 1. Termen en hun bijbehorende gebruik in de context van genetische algoritmes.

2.3 Coderingen

Als je een probleem wilt oplossen met genetische algoritmes, zul je eerst het probleem moeten formuleren. De codering bepaalt op welke manier je dit doet, en is cruciaal voor het al dan niet slagen van je GA. Vaak wordt een bit string van vaste lengte en volgorde gebruikt

¹ Eigenlijk is een chromosoom een karakterrij, maar omdat we meestal werken met een bitcodering, spreken we bijna altijd van een bitrij of een (bit) string.

om candidate solutions vast te leggen. Er zijn echter ook andere coderingen mogelijk. Een keuze voor een codering maken is niet zo eenvoudig, maar een richtlijn is wel te geven.

2.3.1 Binaire codering

Binaire codering houdt in dat een chromosoom gerepresenteerd worden door een reeks van l bits (een bit string), waarbij l de lengte van de bit string is. Een voorbeeld van een bitrij van lengte 5 is 10001. Er wordt vaak gekozen voor deze codering om een aantal redenen [MITCH98]:

- Historie: John Holland en zijn studenten concentreerden zich op de binaire codering en in de praktijk werd dit voorbeeld gevolgd.
- De bestaande theorie over GAs is gebaseerd op de aanname dat de binaire codering met een vaste lengte en volgorde wordt gebruikt. Hoewel die theorie ook op niet-binaire coderingen toegepast kan worden, is deze niet zo ver ontwikkeld. [HENDR91] schrijft dat bij niet-binaire coderingen zelden een schema-analyse (door eerst terug te vertalen naar een binaire codering) wordt uitgevoerd. In plaats daarvan vertrouwt men op empirische tests om te controleren of de gekozen representatie voldoet.
- Er zijn heuristieken ontwikkeld voor geschikte parameterinstellingen (bijv. de kans op cross-over of mutatie) van de binaire codering.

Voor sommige problemen is een binaire codering echter niet praktisch of geschikt. In dat geval moet een speciale codering voor dat probleem worden bedacht. Bovendien kan de ordening van de kenmerken in een bitrij soms arbitrair zijn. Later, in paragraaf 2.5 om precies te zijn, zullen we zien waarom deze ordening van belang is.

2.3.2 Niet-binaire coderingen

Omdat de binaire codering niet voor alle problemen geschikt is, kiest men ook wel eens voor een niet-binaire codering. Voor veel applicaties is het logisch om te kiezen voor een alfabet dat bestaat uit meerdere karakters of de reële getallen. Een ander voorbeeld is het gebruik van een boomcodering (*tree encoding*) [MITCH98]. Een voordeel hiervan is dat de zoekruimte niet begrensd is: elke denkbare boom is mogelijk. Nadelen zijn er ook: de ongecontroleerde groei van een boom kan ertoe leiden dat er geen gestructureerde individuen ontstaan. Ook kunnen de resulterende bomen erg groot en derhalve lastig te begrijpen zijn. Dit zijn slechts een aantal veelvoorkomende coderingen, vele andere zijn bedacht en gebruikt. Omdat een codering vaak probleemspecifiek is, gaan we er niet verder op in.

2.3.3 Keuze voor een codering

Hoe kiezen we een geschikte codering voor een probleem? In [MITCH98] vinden we een richtlijn die opgesteld is door Lawrence Davis, een onderzoeker met veel ervaring op het gebied van toepassen van GAs op praktische problemen. Davis zegt dat je moet kiezen voor de codering die het meest natuurlijk is voor jouw probleem, en vervolgens een bijpassend genetisch algoritme toe te passen dat die codering gebruikt.

Zolang de theorie van coderingen van GAs niet verder ontwikkeld is, is dit wellicht de beste oplossing.

2.4 Het basialgoritme

De eenvoudigste vorm van een genetisch algoritme maakt gebruik van de volgende operatoren: (*fitness-proportionate*) selectie, (*single-point*) cross-over en mutatie.

- Selectie selecteert de chromosomen uit de populatie voor reproductie. Des te hoger de fitness van het chromosoom, des te groter de kans is dat deze uitgekozen wordt. Dit wordt fitness-proportionate selection genoemd (voor een gedetailleerdere uitleg, zie 2.6.1.1).
- Cross-over kiest willekeurig een locus, waarna uitwisseling plaatsvindt tussen twee chromosomen van de delen na die locus, met twee nieuwe individuen als resultaat. Bij *single-point* cross-over vindt op één plaats kruising plaats, bij *multi-point* op meerdere plaatsen. De strings 100001 en 011111 zouden bijvoorbeeld éénmaal (single-point) kunnen kruisen na de derde locus, met als resultaat de strings 100111 en 011001.
- Mutatie draait willekeurig de waarden van bits om in een chromosoom, met een gegeven, relatief kleine kans (bijv. 0,001). Als de string 100000 zou muteren op de laatste positie, levert dit 100001 op.

Gegeven een op te lossen probleem en een bit-stringrepresentatie, werkt het basialgoritme als volgt:

1. Start met een willekeurig gekozen populatie van n chromosomen, die elk l bits lang zijn.
2. Bereken de fitnessfunctie $f(x)$ van elk chromosoom x uit de populatie.
3. Herhaal de volgende stappen totdat er n nakomelingen zijn:
 - a. Selecteer een paar van individuen uit de huidige populatie d.m.v. fitness-proportionate selection. Een chromosoom kan meer dan een keer worden geselecteerd.
 - b. Pas single-point cross-over op het paar toe met waarschijnlijkheid p_c (de kans op cross-over). Als er geen cross-over plaatsvindt worden de twee nakomelingen een exacte kopie van de ouders.
 - c. Muteer de twee individuen op elke locus met waarschijnlijkheid p_m (de kans op mutatie), en plaats beide nakomelingen in de nieuwe populatie.

Als n oneven is, kan een willekeurig individu uit de populatie worden verwijderd.
4. Vervang de huidige populatie door de nieuwe.
5. Ga naar stap 2, tenzij de stopconditie waar is.

Iedere iteratie van het algoritme wordt een *generation* genoemd. Het totaal aan generations wordt een *run* genoemd.

De procedure die hierboven is beschreven, is de basis voor de meeste toepassingen van GAs. Hij komt – op een aantal details na – overeen met het *simpele genetische algoritme* (SGA), dat beschreven staat in [GOLDB89].

De volgende twee paragrafen behandelen de stappen 3 en 4 uitgebreider. Zo behandelt 2.6 selectie (3a), cross-over (3b) en mutatie (3c) in detail. Paragraaf 2.7 gaat dieper in op vervangingsstrategieën (stap 4).

Een aantal zaken dienen nog ingevuld te worden, namelijk de definitie van de fitnessfunctie $f(x)$, de grootte van de populatie n , de lengte l van de strings, de kans op cross-over p_c resp. mutatie p_m , en de stopconditie. Gebruikelijke stopcondities zijn:

- Een vast aantal generaties is bereikt.
- Een bepaalde hoeveelheid rekentijd is verstreken.
- Een individu met de gewenste eisen is gevonden.

- Het individu met de hoogste fitness heeft een niveau bereikt waarna verdere berekeningen niet meer leiden tot een beter resultaat.

Het volgende voorbeeld is naar [MITCH98] en is bedoeld ter illustratie: het is niet representatief voor een gemiddelde GA-applicatie. Stel $f(x)$ is gelijk aan het aantal enen in een bit string x , $l = 8$, $n = 4$, $p_c = 0,7$ en $p_m = 0,001$. De stopconditie laten we even buiten beschouwing, omdat we slechts één iteratie van het algoritme bekijken.

<i>Label</i>	<i>String</i>	<i>Fitness</i>	<i>Kans op selectie</i>	<i>Verwacht aantal keren geselecteerd</i>	<i>Aantal keren geselecteerd</i>
A	00000110	2	2/12	8/12	0
B	11101110	6	6/12	24/12	2
C	00100000	1	1/12	4/12	1
D	00110100	3	3/12	12/12	1

Tabel 2. Populatie met informatie over fitness en het selectieproces.

De initiële populatie wordt willekeurig gegenereerd. We maken gebruik van fitness-proportionate selection. Dit houdt in dat naarmate de fitness van een individu hoger is, deze gemiddeld genomen vaker gekozen wordt. De kans dat chromosoom B wordt gekozen is gelijk aan 0.5. Omdat $n = 4$ bedraagt en de populatie constant wordt gehouden, wordt er vier keer een individu geselecteerd. In dit geval worden B en D en B en C als ouders uitverkozen (zie Tabel 2).

Vervolgens gaan we single-point cross-over toepassen met kans p_c . Stel dat B en D worden gekruist na positie 1, met als gevolg E = 10110100 en F = 01101110. Bij B en C vindt geen cross-over plaats; de kinderen zijn identiek aan hun ouders.

Daarna vindt mutatie op elke locus plaats met kans p_m . Het muteren van E op positie 6 levert E' = 10110000, F en C worden niet aangepast, en B vormt na mutatie B' = 01101110.

<i>Label</i>	<i>String</i>	<i>Fitness</i>
E'	10110000	3
F	01101110	5
C	00100000	1
B'	01101110	5

Tabel 3. Populatie na een generation.

Hoewel de string met de hoogste fitness (string B) uit de oude populatie verdwenen is, is het fitnessgemiddelde gestegen van 12/4 naar 14/4. Het herhalen van de procedure zal uiteindelijk leiden tot een string met allemaal enen.

2.5 Hoe werken genetische algoritmen?

In deze paragraaf proberen we de concepten van de werking van genetische algoritmes uit te leggen. De traditionele theorie (voor het eerst geformuleerd door Holland) zegt dat GAs werken door het ontdekken, benadrukken en combineren van “bouwstenen” van oplossingen [MITCH98]. Met bouwstenen worden combinaties van bitwaarden bedoeld die zorgen voor een hoge fitness bij strings waarin ze voorkomen. Om dit preciezer uit te kunnen leggen, moeten we eerst een aantal begrippen introduceren die te maken hebben met schema's. Daarna bekijken we het *Schema Theorem*.

2.5.1 Schema's

Een *schema* is een verzameling bit strings die beschreven kunnen worden met het alfabet dat bestaat uit nullen, enen en sterretjes (waarbij een sterretje een 0 of een 1 kan zijn). Een schema beschrijft dus bit strings die overeenkomsten vertonen op bepaalde loci. Bijv. het schema $S = 1****0$ stelt alle strings van lengte 6 voor die beginnen met een 1 en eindigen op een 0.

De breedte (*defining length*) is de afstand tussen de buitenste gedefinieerde bits (d.w.z. een bit die 0 of 1 is, maar geen sterretje). We bereken dit met een functie d . De breedte van schema S wordt dan: $d(S) = 5$. Stel $S' = 0*****$, dan is $d(S') = 0$, omdat het enige gedefinieerde bit de eerste en de laatste is.

De orde (*order*) van een schema – genoteerd door $o(S)$ – geeft het aantal gedefinieerde bits dat erin voorkomt. Bijvoorbeeld: $o(0*1*1) = 3$.

Een instantie (*instance*) is een concrete invulling van een schema. De functie i levert bij een gegeven schema S de verzameling van alle instanties van S op. Als $S'' = *01$, dan is $i(S'') = \{001, 101\}$. De bitrij 001 is een voorbeeld van een instantie van schema S'' .

Een willekeurige string van lengte l is een instantie van 2^l verschillende schema's. Ga maar na: de string 11 is een instantie van $**$, $1*$, $*1$ en 11 . Dus een populatie van n strings bevat instanties van tussen 2^l en $n \cdot 2^l$ verschillende schema's. Als alle bitrijen identiek zijn, zijn er exact 2^l , anders is het aantal schema's gelijk of kleiner aan $n \cdot 2^l$. De vraag is nu: hoeveel schema's worden er effectief verwerkt in elke generation? Om deze vraag te beantwoorden, bekijken we het effect van reproductie, cross-over en mutatie op de groei of vernietiging van belangrijke schema's.

Reproductie zorgt ervoor dat er naar verloop van tijd meer en meer instanties van de 'beste' schema's voorkomen in de populatie, omdat bitrijen met een hoge fitness een hoge kans hebben op selectie. Reproductie alleen kan echter niet zorgen voor nieuwe punten in de zoekruimte. Cross-over kan hier wel voor zorgen, mits het plaatsvindt (normaliter geldt: $p_c < 1$). Als het optreedt, dan is de kans aanwezig dat een schema 'verstoord' wordt. Verstoren betekent in dit geval dat gedefinieerde bits van elkaar gescheiden worden. In het algemeen betekent verstoring dat een schema wezenlijk wordt gewijzigd. Bits die relatief ver van elkaar verwijderd zijn in een string, worden door cross-over eerder gescheiden worden van elkaar dan bits die dicht bij elkaar zijn. Hierdoor worden schema's met een korte defining length niet snel aangetast door cross-over.

Mutatie is in staat een schema te verstoren (d.w.z. een gedefinieerd bit wijzigen), maar zal dit niet snel doen omdat p_m normaal gesproken erg klein is (in de orde van 10^{-3}).

Intuïtief voelen we aan dat schema's met een korte defining length en met instanties die gemiddeld een hoge fitnesswaarde hebben, ook wel *building blocks* (bouwstenen) genoemd, generatie op generatie worden doorgegeven in de populatie. De instanties van deze schema's worden immers niet verstoord, en hun aantal neemt bovendien exponentieel toe bij elke iteratie van het algoritme. Het blijkt dat het aantal nuttig bekeken schema's in de orde van n^3 ligt, als de populatie uit n strings bestaat [GOLDB89]. Dit verhoudt zich gunstig tot het aantal functie-evaluaties, te weten n . Dit effect wordt *impliciet parallelisme* genoemd.

2.5.2 Schema Theorem

We kunnen deze intuïtie proberen te formaliseren. Dit zullen we doen aan de hand van [GOLDB89], [HENDR91] en [MITCH98]. Het doel is om het verwachte aantal instanties van een schema in de volgende generaties te berekenen. Eerst introduceren we een functie om deze instanties te tellen. Daarna gaan we achtereenvolgens bekijken welke (gecombineerde) effecten reproductie, cross-over en mutatie hebben op het aantal instanties.

Stel $m(S, t)$ is een functie die het aantal instanties van schema S op tijdstip t bepaalt. We bekijken het voorbeeld van paragraaf 2.4, waarbij Tabel 2 en Tabel 3 de populatie voorstellen op resp. tijdstip t en $t+1$. Ter illustratie: $m(000****, 0) = 1$, $m(000****, 1) = 0$ en $m(011****, 1) = 2$. Schema's kunnen dus in verschillende generaties een verschillend aantal instanties hebben. Het doel kunnen we nu herformuleren; we wensen de verwachtingswaarde van $m(S, t+1)$ uit te drukken in termen van $m(S, t)$.

We beginnen met het effect van de selectieoperator. Het aantal instanties van een schema op tijdstip $t+1$ is afhankelijk van het aantal dat gekozen wordt. De kans dat een schema de selectieronde overleeft, is gelijk aan de gemiddelde fitness van de instanties van dat schema, gedeeld door de totale fitness van de populatie. De fitness van een schema S op tijdstip t berekenen we door de fitnesswaarden van alle instanties van schema S bij elkaar op te tellen, en dit vervolgens te delen door het aantal instanties van S :

$$f'(S, t) = \frac{\sum_{j \in i(S)} f(j, t)}{\#i(S)},$$

waarbij $f(j, t)$ de fitnessfunctie van instantie j op tijdstip t is.

De gezochte verwachtingswaarde wordt dan:

$$(2.1) \quad E(m(S, t+1)) = m(S, t) \cdot n \cdot \frac{f'(S, t)}{\sum_{j=1}^n f(j, t)} = m(S, t) \cdot \frac{f'(S, t)}{\bar{f}(t)},$$

waarbij n de populatiegrootte is en $\bar{f}(t)$ de gemiddelde fitness van de populatie op tijdstip t :

$$\bar{f}(t) = \frac{\sum_{j=1}^n f(j, t)}{n}.$$

Hier kunnen we uit afleiden dat het verwachte aantal schema-instanties groeit (daalt) van een schema S als de fitness van S groter (kleiner) is dan het fitnessgemiddelde van de populatie. Dit komt overeen met ons vermoeden dat reproductie ervoor zorgt dat er naar verloop van tijd (naar verwachting) meer instanties van de 'beste' schema's voorkomen in de populatie. We kunnen nog meer te weten komen over de groeisnelheid van de instanties. Stel dat een schema S gedurende de hele run $c \cdot \bar{f}_g$ (met c een constante) boven het gemiddelde fitnessniveau blijft. Dan kunnen we vergelijking (2.1) herschrijven² tot:

$$E(m(S, t+1)) = m(S, t) \cdot \frac{\bar{f}(t) + c \cdot \bar{f}(t)}{\bar{f}(t)} = m(S, t) \cdot (1 + c).$$

Voor tijdstip $t = 0$ kunnen we vervolgens schrijven:

$$E(m(S, t+1)) = m(S, 0) \cdot (1 + c)^t.$$

² Hoewel [HENDR91] juist opmerkt dat "in de literatuur" – waarschijnlijk refererend naar [GOLDB89] – $m(S, t+1)$ in plaats van $E(m(S, t+1))$ wordt geschreven als het om de verwachtingswaarde gaat, voert hij deze notatie niet consequent door. Dit zullen wij wel proberen te doen.

Hieruit volgt dat reproductie het aantal instanties van schema's met een constant bovengemiddelde fitnesswaarde exponentieel doet toenemen. De exponentiële toename van die instanties verklaart voor een deel het succes van genetische algoritmen.

We zijn nu aangekomen bij (single-point) cross-over. Voordat we vergelijking (2.1) kunnen uitbreiden, moeten we eerst bekijken hoe schema's door recombinatie verstoord kunnen worden. Stel we hebben een schema $S = 1***0*$. Deze kan op $d(S) = 4$ manieren verstoord worden. Het aantal plaatsen waar cross-over kan plaatsvinden is gelijk aan $l-1 = 5$, waarbij l de lengte van de string is. De kans op 'verstoring' is dus $4/5$, en op overleven derhalve $1/5$. De kans O_c dat een schema cross-over overleeft kunnen we in het algemeen als volgt noteren:

$$O_c(S) \geq 1 - p_c \cdot \frac{d(S)}{l-1},$$

waarbij p_c de kans op cross-over is. De kans op overleven is dus groter naarmate de breuk in de formule kleiner is. Aangezien alle schema's dezelfde lengte hebben, bepaalt de defining length de overlevingskans. Des te kleiner $d(S)$ is, des te groter is O_c . Smallere schema's hebben dus een grotere kans om te overleven.

Merk op dat er 'groter dan of gelijk aan' staat in de vergelijking.³ Sommige recombinaties zullen het schema namelijk niet verstoren, bijvoorbeeld als twee identieke strings met elkaar kruisen.

Het gecombineerde effect van reproductie en cross-over schrijven we als volgt:

$$(2.2) \quad E(m(S, t+1)) \geq m(S, t) \cdot \frac{f'(S, t)}{f(t)} \cdot \left(1 - p_c \cdot \frac{d(S)}{l-1}\right).$$

De laatste operator is mutatie. Een schema overleeft als geen van de gedefinieerde bits wordt gewijzigd door mutatie.⁴ De kans op mutatie noemen we p_m . Een schema S overleeft dus als elke ingevulde positie, in totaal $o(S)$, niet muteert:

$$O_m(S) \geq (1 - p_m)^{o(S)}.$$

De overlevingskans is dus groter voor schema's met een lage orde. Ook hier staat het $\geq$ -teken er om een reden: het kan zo zijn dat de mutatie van een schema een eerder verstoord schema oplevert. We herschrijven vergelijking (2.2) zodat de invloeden van selectie, cross-over en mutatie worden meegenomen in de definitieve vergelijking:

$$(2.3) \quad E(m(S, t+1)) \geq m(S, t) \cdot \frac{f'(S, t)}{f(t)} \cdot \left(1 - p_c \cdot \frac{d(S)}{l-1}\right) \cdot [(1 - p_m)^{o(S)}]$$

Vergelijking (2.3) staat bekend als het *Schema Theorem*. De interpretatie ervan is dat van smalle schema's van een lage orde en een bovengemiddeld fitnessniveau een exponentieel toenemend aantal instanties wordt geëvalueerd in volgende generaties. Deze analyse gaat overigens alleen op voor het basialgoritme (besproken in 2.4), waarin gebruik gemaakt wordt van een binaire codering en eenvoudige genetische operatoren.

³ In [HENDR91] wordt foutief $\leq$ in plaats van $\geq$ geschreven.

⁴ Een schema hoeft dus maar op één plaats gemuteerd te worden. In [HENDR91] wordt beweerd dat de kans dat het schema $0***1*$ verstoord wordt $(p_m)^2$ is. Ga na dat dit $-(p_m)^2 + 2 \cdot p_m$ moet zijn.

2.6 Genetische operatoren

2.6.1 Selectie

Stap 3a van het simpele genetische algoritme selecteert paren van individuen uit de huidige populatie op basis van de fitness. De bedoeling is de individuen zo uit te kiezen dat hun opvolgers uiteindelijk een hogere fitness hebben dan henzelf. Selectie moet gebalanceerd worden samen met mutatie en cross-over. Als te streng wordt geselecteerd op fitness nemen suboptimale individuen de populatie over en is er te weinig diversiteit om verdere vooruitgang mogelijk te maken. Als de balans naar de andere kant doorslaat, zal de populatie erg langzaam evolueren.

Hieronder zal ik een aantal selectiemethoden bespreken. In het algemeen is het lastig te zeggen voor welke problemen een bepaalde methode geschikt is; meer onderzoek hiernaar is nog nodig.

2.6.1.1 Fitness-proportionate selection

Hollands originele GA maakte gebruik van *fitness-proportionate selection*, waarbij het aantal keren dat een individu verwacht wordt te reproduceren (de verwachtingswaarde) gelijk is aan de fitness van het individu gedeeld door de gemiddelde fitness van de populatie. In formulevorm:

$$E(i, t) = n \cdot p(i, t) = n \cdot \frac{f(i, t)}{\sum_{j=1}^n f(j, t)} = \frac{f(i, t)}{\sum_{j=1}^n \frac{f(j, t)}{n}}$$

waarbij $E(i, t)$ de verwachtingswaarde, $p(i, t)$ de kans om geselecteerd te worden en $f(i, t)$ de fitnessfunctie van individu i op tijdstip t is, en n de populatiegrootte is.

Des te hoger de fitness van een individu, des te vaker deze gemiddeld genomen gekozen wordt. De kans om gepakt te worden is dus proportioneel met de fitness van een individu. Deze methode heeft een groot nadeel. Normaal gesproken is in de beginfase van het zoekproces de fitnessvariantie hoog en heeft een klein deel van de populatie een relatief hoge fitness. Fitness-proportionate selection zorgt ervoor dat individuen met een hoge fitness worden uitgekozen, waardoor de populatie snel overgenomen wordt door hen en hun nakomelingen. Hierdoor wordt verkenning van andere mogelijkheden uitgesloten. Dit wordt *premature convergence* genoemd: convergentie naar een oplossing vindt in een te vroeg stadium van het zoekproces plaats. Later tijdens het zoeken, als alle individuen van de populatie op elkaar lijken, is de fitnessvariantie erg laag, waardoor er nauwelijks verschillen zijn om uit te buiten, en evolutie praktisch stopt. De snelheid van evolutie is dus afhankelijk van de variantie van de fitness in de populatie.

2.6.1.2 Sigma scaling

Er zijn verschillende methodes ontwikkeld die de fitnesswaarden schalen naar verwachtingswaarden om zo te voorkomen dat premature convergence optreedt. Een voorbeeld hiervan is *sigma scaling*. Deze selectiemethode zorgt ervoor dat de *selection pressure* (selectiedruk, bijv. de mate waarin individuen met een hoge fitness gekozen worden), redelijk gelijk blijft gedurende het zoekproces in plaats van afhankelijk te zijn van de fitnessvariantie van de populatie. Een mogelijke formule zou zijn:

$$E(i, t) = 1 + \frac{f(i, t) - \bar{f}(t)}{c \cdot \sigma(t)} \quad \text{als } \sigma(t) \neq 0, \text{ anders } E(i, t) = 1.0$$

waarbij $\sigma(t)$ de standaarddeviatie van de fitness van de populatie op tijdstip t voorstelt, $\bar{f}(t)$ de gemiddelde fitness van de populatie op tijdstip t , en c een constante (tussen 1 en 3). Des te hoger c is, des te lager is de selectiedruk (elk individu heeft een redelijke kans op reproductie). Een individu met een fitnesswaarde van één standaardafwijking boven het gemiddelde heeft een verwachtingswaarde van 1,5 nakomelingen als $c = 2$. Als $E(i,t) < 0$, dan wordt de waarde arbitrair op 0 gezet, of op een lage waarde (net boven 0) zodat individuen met een lage fitness toch een kleine kans op reproductie hebben. Aan het begin van een run, wanneer de standaarddeviatie van fitnesswaarden vaak hoog is, zijn de individuen met een hoge fitness niet veel standaardafwijkingen boven het gemiddelde en vormen ze niet een te groot deel van de populatie die mag reproduceren. Later is de standaarddeviatie normaal gesproken lager, en zullen de individuen met een hogere fitness meer uitsteken boven de rest, zodat evolutie kan doorgaan.

2.6.1.3 Boltzmann selection

Sigma scaling houdt de selection pressure constant gedurende een run. Vaak is het echter wenselijk om de selectiedruk te variëren in de loop van het zoekproces. In het begin kan het goed zijn om niet te streng te selecteren, zodat individuen met een lage fitness ook de kans krijgen om te reproduceren. Op deze manier blijft de populatie lange tijd gevarieerd van samenstelling. Later kan de selectiedruk dan worden opgevoerd, zodat de beste individuen worden uitgekozen, ervan uitgaande dat de diversiteit aan het begin ervoor gezorgd heeft dat op de juiste plaats in de zoekruimte wordt gezocht.

Een methode om dit voor elkaar te krijgen is de *Boltzmann selection*. Hierbij is er sprake van een continu veranderende ‘temperatuur’ die de selectie beïnvloedt. De temperatuur begint hoog, wat een lage selectiedruk tot gevolg heeft. Daarna wordt deze parameter geleidelijk verlaagd, waardoor de selection pressure toeneemt. Het genetisch algoritme krijgt zo de kans de zoekruimte nader te bestuderen op de juiste plaats, terwijl er toch voldoende variatie in de populatie behouden wordt. Een voorbeeld van een implementatie is:

$$E(i,t) = \frac{e^{f(i,t)/T}}{\sum_{j=1}^n \frac{e^{f(j,t)/T}}{n}},$$

waarbij T de temperatuur is. Naarmate T daalt neemt het verschil tussen de verwachtingswaarden van individuen met een hoge en lage fitness toe. De bedoeling is om dit langzaam maar zeker tijdens het zoeken te laten verlopen door de temperatuur op een geplande manier te verlagen.

2.6.1.4 Tournament selection

Bij *tournament selection* wordt een ‘tournament’ (een toernooi) georganiseerd waar een aantal (N_{tour} , de *tournament size*) willekeurig geselecteerde individuen aan meedoen. Met kans p (met $0 < p \leq 1$) wordt het individu met de hoogste fitness uitgekozen. Het individu met de één na hoogste fitnesswaarde heeft kans $p \cdot (1-p)$ om gekozen te worden, de twee na hoogste heeft een kans van $p \cdot (1-p)^2$, etc. Als $p = 1$ wordt altijd voor het beste individu gekozen. Na selectie kan het gekozen individu eventueel worden verwijderd om te voorkomen dat het meer dan een keer te selecteren is. De selectieprocedure wordt n keer uitgevoerd als er n individuen nodig zijn.

Tournament selection lijkt qua selectiedruk op rank selection – een andere populaire selectiemethode die we niet zullen bespreken – maar heeft een lagere complexiteit. Zowel

tournament als rank selection voorkomen premature convergence door te vermijden dat een kleine groep gespecialiseerde individuen de populatie overneemt.

Bij de methoden waarbij de kans op selectie proportioneel toeneemt met de fitness moet er tweemaal per generation door de populatie worden gelopen: een keer om het fitnessgemiddelde van de populatie te berekenen (en, bij sigma scaling, de standaardafwijking), en een keer om de verwachtingswaarde van het individu te berekenen.

Rank selection vereist dat de populatie gesorteerd is: een mogelijk tijdrovende bezigheid.

Sorteren op basis van vergelijking (*comparison sort*) heeft als complexiteit $O(n \log n)$, waarbij n de populatiegrootte is [LEIJE04]. Tournament selection heeft als voordeel dat het een relatief lage complexiteit heeft. Omdat de looptijd van het algoritme van orde $O(n \cdot N_{\text{tour}})$ is, en normaal gesproken N_{tour} veel kleiner is dan n , is tournament selection lineair in n (als $p = 1$) [KRATI01]. Maar ook als $p < 1$ gaat dit op: het uitkiezen van een individu uit N_{tour} elementen kost constante tijd en beïnvloedt de complexiteit dus niet wezenlijk.

Ook is de selection pressure eenvoudig aan te passen door de tournament size te wijzigen. Als N_{tour} verhoogd wordt krijgen zwakke individuen een kleinere kans om gekozen te worden.

Daarnaast is tournament selection geschikt voor parallele implementatie omdat het selecteren simultaan kan gebeuren: meerdere toernooien kunnen tegelijkertijd worden gehouden.

2.6.2 Cross-over

Cross-over gaat over het kruisen van twee chromosomen en kent drie vormen: single-point, multi-point, en parametrized uniform cross-over.

Single-point cross-over is de eenvoudigste vorm: een willekeurige cross-over positie wordt gekozen en de delen van de twee ouders na deze plek worden verwisseld, zodat twee kinderen worden gevormd. Een nadeel is dat niet alle mogelijke strings opgeleverd kunnen worden. We kunnen namelijk inzien dat single-point cross-over instanties van $11^{*****}1$ en $****11^{**}$ nooit zo kan combineren dat $11^{**}11^*1$ opgeleverd wordt.

Daarnaast is er sprake van een effect dat *positional bias* wordt genoemd: bits die relatief ver van elkaar verwijderd zijn in een string zullen eerder gescheiden worden. De volgorde van de genen in het chromosoom is dus belangrijk, maar het probleem is dat van tevoren onbekend is welke bits gegroepeerd moeten worden. De volgorde is immers probleemspecifiek.

Een derde nadeel is dat de voorkeur aan bepaalde loci wordt gegeven: de uiteindes van de strings worden altijd uitgewisseld (als cross-over plaatsheeft).

Om deze problemen te ondervangen wordt veelal two-point cross-over toegepast. Hierbij worden willekeurig twee posities gekozen, waarna de tussenliggende segmenten worden uitgewisseld. Het voordeel hiervan is dat bits de volgorde waarin de genen staan minder van belang is, en dat de uitgewisselde segmenten niet altijd de stringuiteinden bevatten. Het eerste probleem – dat niet alle mogelijke strings opgeleverd kunnen worden – wordt echter niet geheel opgelost.

Een derde methode is *parametrized uniform* cross-over. Op elke bitpositie kan uitwisseling plaatsvinden met kans p (vaak: $0,5 < p \leq 0,8$). Er is geen sprake van positional bias: elke mogelijke string kan worden gemaakt. Dit heeft echter het nadeel dat er misschien nooit allelen ontstaan die zich aan elkaar aanpassen, omdat deze vorm van cross-over de strings wellicht te veel door elkaar husselt.

2.6.3 Mutatie

Waar cross-over zorgt voor voldoende variatie en innovatie, regelt mutatie dat de populatie niet blijft zitten met een gefixeerde waarde op een bepaalde locus. Stel dat alle strings er als

volgt uitzien: ***1**. Het zal duidelijk zijn dat cross-over er nooit voor zal kunnen zorgen dat de 1 op de vierde positie verandert in een 0. Dit is precies waar de mutatie-operator van pas komt. Mutatie zorgt dus voor voldoende variatie in de populatie. Te veel veranderen is echter niet wenselijk (net zoals bij cross-over). Daarom is de kans op mutatie normaal gesproken redelijk laag, bijv. $p_m = 0,001$. Het is de kunst om de juiste balans tussen selectie, cross-over en mutatie te vinden.

2.7 Vervanging huidige populatie

Stap 4 van het algoritme bepaalt welke individuen van de huidige populatie vervangen worden door individuen van de nieuwe populatie.

Allereerst moet worden gekeken welk deel van de huidige populatie aangemerkt wordt voor vervanging. We hebben in Afbeelding 1 gezien hoe de GAs worden onderverdeeld in twee categorieën: parallel en sequentieel. We bespreken hier de twee meest populaire sequentiële GAs: *generational* en *steady-state* GAs. Parallele genetische algoritmes komen in een volgend hoofdstuk uitgebreid aan bod.

Daarna bekijken we hoe individuen van de nieuwe populatie het zojuist gekozen deel van de huidige populatie gaan vervangen. Dit wordt bepaald door de vervangingsstrategie (*replacement strategy/policy*) die we toepassen. Een bekende replacement strategy is elitisme, die we derhalve zullen behandelen.

2.7.1 Generational GAs

De meeste GAs die in de literatuur worden besproken zijn *generational*. Dit houdt in dat elke generation de huidige populatie in zijn geheel wordt vervangen door de nieuw gevormde populatie. Het kan zijn dat bepaalde individuen in de populatie aanwezig blijven, omdat er geen kruising of mutatie heeft plaatsgevonden. In 2.4 is een voorbeeld van een generational GA gegeven.

2.7.2 Steady-state GAs

Bij *steady-state* GAs worden slechts een paar individuen vervangen elke generation. Normaal wordt een klein aantal individuen (met de laagste fitness) vervangen door nageslacht van de individuen met de hoogste fitness. Steady-state GAs gebruikt men vaak in lerende regelgebaseerde systemen, waarbij het voortbouwen op eerder vergaarde kennis van belang is, en de leden van de populatie samen i.p.v. onafhankelijk van elkaar het probleem oplossen.

2.7.3 Elitisme

Elitisme (*elitism*) zorgt ervoor dat de beste individuen tot nu toe bewaard blijven. Ze kunnen namelijk verloren gaan als ze niet geselecteerd worden om te reproduceren of als ze vernietigd worden door cross-over of mutatie. Stel dat individu i het beste individu is tot op tijdstip t . Als i niet in de populatie van tijdstip $t + 1$ voorkomt, zorgt elitisme ervoor dat i alsnog wordt opgenomen in de nieuwe populatie. Het is niet noodzakelijk dat de nieuwe populatie even groot is als de huidige. Als op tijdstip $t + 1$ namelijk de populatie $n + 1$ chromosomen telt, zorgt stap 3 ervoor dat er toch maar n nakomelingen zijn. Hoewel het de prestatie van een GA significant kan verbeteren, zijn er ook problemen bekend waarbij het toepassen van elitisme negatief uitpakt. Bovendien wordt het lokaal zoeken naar een oplossing gestimuleerd ten koste van een globaal optimum.

3 Parallellisme en genetische algoritmes

Dit hoofdstuk besteedt aandacht aan parallellisme en genetische algoritmes. We beginnen met de definitie van een parallel genetisch algoritme (3.1). Daarna geven we in 3.2 een aantal redenen waarom je überhaupt de moeite zou nemen om een GA parallel uit te voeren. Op welke computerarchitecturen je dit kunt doen, komt aan bod in 3.3.

Vervolgens bekijken we op welke manieren parallellisme en GAs gecombineerd kunnen worden. Dit doen we door op twee manieren te kijken naar dit probleem. Als eerste beschrijven we welke soorten parallellisatiestrategieën er mogelijk zijn voor genetische algoritmes beschouwd als zoektechniek (3.4). Als tweede beschouwen we een taxonomie van parallelle genetisch algoritmes (3.5). In de daaropvolgende paragrafen bespreken we de belangrijkste categorieën uit deze taxonomie.

In 3.10 wordt bekeken of genetische algoritmes onevenredig veel sneller zijn op een parallelle computer dan op een seriële.

3.1 Definitie PGA

Een *parallel genetisch algoritme* (PGA) is een GA dat geparallelliseerd wordt. Met *parallelliseren* bedoelen we geschikt maken voor uitvoering op een parallelle computerarchitectuur. Om te profiteren van een dergelijke architectuur moeten een of meerdere stappen in het algoritme gelijktijdig kunnen worden uitgevoerd. We zullen een daarom een uitbreiding van het basisalgoritme (zie 2.4) beschouwen.

In [ALBA02a] wordt beweerd dat de pseudocode in Afbeelding 2 een algemene beschrijving is van een PGA. Er is geen andere literatuur bij mij bekend waarin een definitie voor een PGA wordt gegeven.

```

t := 0;
initialize:    P(0) := {a1(0), ..., aμ(0)} ∈ Iμ;
evaluate:      P(0) : {Φ(a1(0)), ..., Φ(aμ(0))};
while ι(P(t)) ≠ true do //Reproductive plan
    select:     P'(t) := sΘs(P(t));
    recombine:  P''(t) := ⊗Θc(P'(t));
    mutate:     P'''(t) := mΘm(P''(t));
    evaluate:   P'''(t) : {Φ(a1'''(t)), ..., Φ(aλ'''(t))};
    replace:    P(t + 1) := rΘr(P'''(t) ∪ Q);
    <communication>
    t := t + 1;
end while

```

Afbeelding 2. Een beschrijving van een parallel genetisch algoritme naar [ALBA02a].

Zonder in te gaan op de notatie, zien we dat er een aantal stappen zijn: *initialisatie*, *evaluatie*, *selectie*, *recombinatie*, *mutatie*, *vervanging* en *communicatie*. Als we dit algoritme vergelijken⁵ met die van paragraaf 2.4 komen alle stappen, op communicatie na, in beide

⁵ Merk op dat de beschrijving van de GA in een *do...until*-vorm staat, terwijl die van de PGA in een *while*-vorm gegoten is.

algoritmen voor. Daarnaast wordt nog opgemerkt dat in een PGA meerdere ‘reguliere’ GAs (een aantal van) de stappen in de while-lus op hun eigen subpopulatie $P^i(t)$ uitvoeren (zie ook 3.7). Dit zijn de twee belangrijkste verschillen.

Communicatie kan in dit geval van alles zijn: van het uitwisselen van strings met andere processors, tot de uitwisseling van statistieken. Ook het uitvoeren van evaluaties op andere processors valt hieronder [ALBA02a].

Elke subpopulatie $P^i(t)$ kan uit vele honderden strings, maar ook uit één string bestaan. Als een deelpopulatie uit tientallen chromosomen bestaat noemen we het PGA *coarse grained* (zie 3.8), en vindt er af en toe interactie met andere deelpopulaties plaats. In het geval van één string per subpopulatie hebben we een *fine-grained* PGA (zie 3.9) en vindt er meer communicatie met aangrenzende deelpopulaties plaats. Bij *globale parallelisatie* (zie 3.6) wordt de evaluatie (de stap *evaluate*) – en mogelijk het kruisen en de mutatie (*recombine* en *mutate*) – parallel uitgevoerd. *Automatische parallelisatie* door de compiler (zie Afbeelding 1) komt nauwelijks voor, omdat het algoritme op de een of andere manier automatisch geparalleliseerd moet worden door de compiler [ALBA02a].

Een punt van kritiek op het gegeven algoritme is dat de communicatiestap op een arbitraire positie is neergezet: communicatie kan net zo goed vlak voor de evaluatiestap plaatsvinden omdat de evaluatie van individuen op meerdere processors tegelijkertijd plaatsheeft. Het is zelfs mogelijk dat communicatie vaker tussendoor optreedt.

Ten tweede is het opvallend dat er maar weinig verschillen zijn met het basisalgoritme. De toegevoegde waarde van het algoritmisch definiëren van een PGA is dus laag, wat ook verklaart waarom weinig auteurs dit hebben gedaan. Wel weten we nu dat communicatie en deelpopulaties een rol spelen bij een parallel genetisch algoritme.

3.2 Waarom parallel?

Voordat we in gaan op het antwoord op deze vraag, moeten we eerst een duidelijk onderscheid maken tussen het soort algoritme dat we toepassen, en de implementatie daarvan. Het uitvoeren van een GA op een parallelle computerarchitectuur staat los van het begrip parallel genetisch algoritme. Een sequentieel GA kan op meerdere computers uitgevoerd worden, net zo goed als een PGA op een computer met één processor gedraaid kan worden. We stellen daarom opnieuw twee vragen:

- Waarom zouden we een probleem op een parallelle computerarchitectuur willen oplossen?
- Waarom willen we een genetisch algoritme paralleliseren, d.w.z. geschikt maken voor uitvoering op een parallelle computerarchitectuur?

Parallel computing wordt door [QUINN87] gedefinieerd als het proces om problemen op een parallelle computerarchitectuur op te lossen. Hier gaan we als eerste op in.

Een aantal artikelen in de literatuur noemen redenen voor het toepassen van een PGA. Deze geven antwoord op de tweede vraag, en komen als tweede aan de beurt.

3.2.1 Waarom parallel computing?

Volgens [BARNE05] zijn er twee hoofdredenen om parallel computing toe te passen:

- Tijdsbesparing (“wall clock time”).
- Het oplossen van omvangrijke problemen.

Daarnaast zijn er grenzen aan *serial computing* – zowel fysieke als praktische redenen beperken de ontwikkeling van snellere seriële computers:

- Overdrachtssnelheden: de snelheid van een seriële computer is afhankelijk van hoe snel informatie getransporteerd kan worden. Het is onmogelijk om elektrische signalen sneller dan het licht te laten reizen, en de transmissiesnelheid in koperdraad ligt ook nog eens lager. Hogere snelheden vereisen dus dat componenten dichter op elkaar komen te zitten.
- Grenzen aan miniaturisatie: hoewel geavanceerdere technologie ervoor zorgt dat steeds meer transistors op een chip geplaatst kunnen worden, komt er een moment dat er een benedengrens is bereikt wat betreft de grootte van een component.
- Economische overwegingen: het is duur om één processor steeds sneller te maken. Het gebruik van een groot aantal relatief goedkope processors om hetzelfde resultaat te bereiken, is financieel vaak aantrekkelijker.

Ook moet vooruit worden gekeken. De afgelopen 10 jaar zijn netwerken alsmaar sneller geworden, en hebben gedistribueerde systemen en multiprocessor computerarchitecturen opkomst gemaakt. Dit suggereert dat parallel computing de toekomst heeft.

3.2.2 Waarom PGAs?

In de literatuur worden verscheidene redenen genoemd om PGAs te gebruiken. Zo worden in [ALBA02a] een aantal (positieve) kenmerken van parallelle GAs ten opzichte van sequentiële GAs genoemd:

1. Minder tijd nodig om een oplossing te vinden (sneller algoritme).
2. Reductie van het aantal functie-evaluaties.
3. Mogelijkheid om grotere populaties te gebruiken dankzij de parallelle computerarchitecturen waarop de algoritmes worden uitgevoerd.
4. Betere oplossingen als resultaat.
5. Minder gevoelig voor premature convergence.

In [NOWOS99] worden situaties genoemd waarin PGAs in plaats van de gewone variant geschikt zijn om te gebruiken:

1. Voor bepaalde problemen moet de populatie groot zijn en is er dus een grote behoefte aan geheugen. In sommige gevallen is het ondoenlijk een applicatie te draaien op een enkele computer, waardoor het gebruiken van een PGA onvermijdelijk is.
2. Het evalueren van fitnessfuncties is tijdrovend. In de literatuur zijn reketijden van meer dan 1 ‘processorjaar’ bericht voor één run. Het ligt voor de hand om dit soort berekeningen parallel uit te voeren.
3. ‘Gewone’ GAs kunnen vast komen te zitten in een suboptimaal gebied van de zoekruimte, waardoor de hoop op betere oplossingen vervliegt. PGAs daarentegen kunnen in verschillende deelgebieden van de zoekruimte zoeken.

De laatste opmerking sluit aan bij Goldbergs bevindingen, maar is enigszins controversieel. Goldberg heeft in 1991 laten zien dat een gewoon GA niet per se een optimale oplossing vindt, en dat de effectiviteit ervan sterk afhankelijk is van de populatiegrootte [CANTU95]. In later werk van Cantú-Paz ([CANTU97], begeleid door Goldberg zelf) wordt deze opmerking echter genuanceerd. Als de populatie te klein is, is er wellicht een tekort aan goede building blocks, waardoor het lastig is goede oplossingen aan te wijzen. Als de populatie te groot is, wordt te veel tijd besteed aan het onnodig evalueren van individuen. Eenzelfde balans moet echter gevonden worden bij PGAs.

De implementatie van PGAs op parallelle computerarchitecturen (zie 3.3) heeft duidelijk voordelen, zoals de grotere rekenkracht en geheugenruimte. Een ander voordeel van PGAs is dat ze betere oplossingen leveren dan GAs die gebruik maken van één populatie. De reden hiervoor is dat meerdere populaties ertoe leiden dat *speciation* (het ontstaan van soorten) kan plaatsvinden. Elke deelpopulatie ontwikkelt zich in een verschillende richting, zodat verschillende optima (pieken in de grafiek) bekeken worden. Dit komt het vinden van een betere oplossing ten goede [NOWOS99]. Desalniettemin kan een bepaalde soort de gehele populatie overnemen als er niet specifiek voor gezorgd wordt dat dit niet gebeurt. Zogenaamde speciation GAs zijn hier precies op gericht [ALBA02a].

3.3 Taxonomie van computerarchitecturen

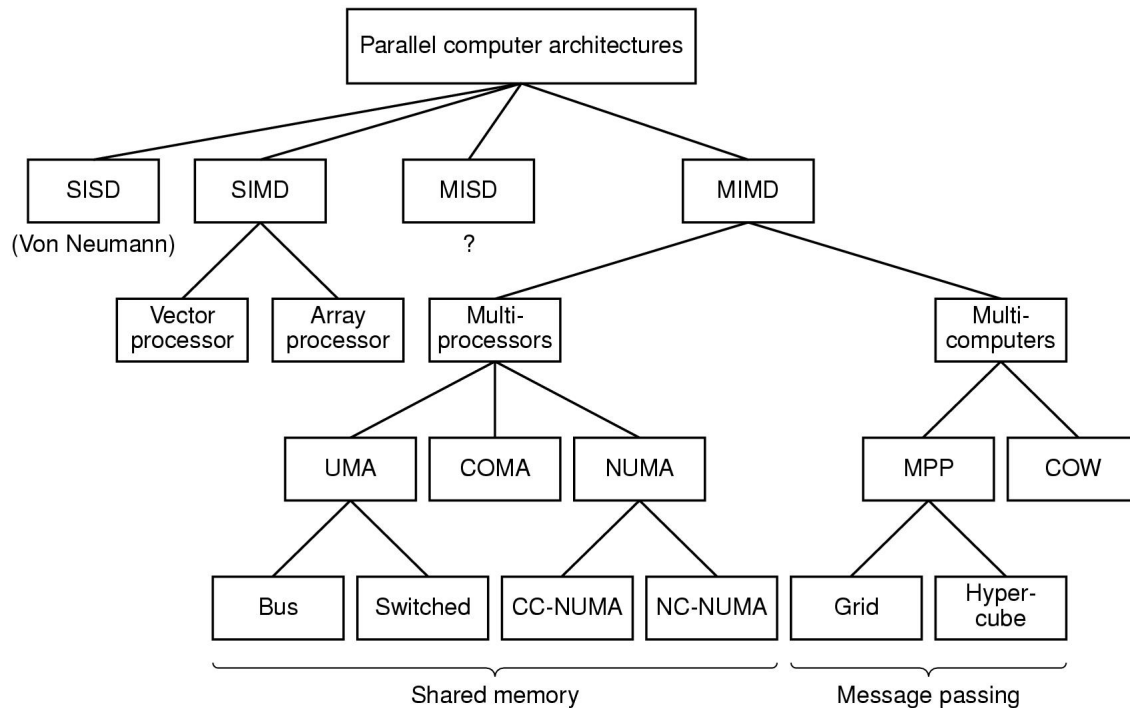
PGAs worden ontwikkeld voor een bepaalde parallelle computerarchitectuur. Voordat we gaan praten over deze algoritmes, maken we eerst een overzicht van parallelle computers op basis van (een uitbreiding van) Flynn's taxonomie. Vervolgens komen verschillende topologieën aan bod die gebruikt kunnen worden bij de toepassing van parallelle genetische algoritmes.

3.3.1 Flynn's taxonomie

Een relatief bekende taxonomie van computerarchitecturen is die van Flynn, die hij in 1972 opstelde. Zijn classificatie berust op twee concepten: instruction streams en data streams. Een *instruction stream* is een reeks instructies die uitgevoerd worden door een computer; een *data stream* is een reeks data die gebruikt worden voor de uitvoering van een instruction stream. Deze 'streams' kunnen enkelvoudig of meervoudig zijn, zodat er vier klassen van architecturen te onderscheiden zijn [QUINN87]:

- *Single-Instruction stream, Single-Data stream (SISD).*
De klassieke Von Neumann-computer is hier een voorbeeld van: een instruction stream werkt op een data stream. Ook hedendaagse computers met één processor vallen in deze categorie.
- *Single-Instruction stream, Multiple-Data stream (SIMD).*
Een enkelvoudige instruction stream werkt op meerdere data-elementen tegelijk. Gebruikelijk is dat er meerdere aritmetische verwerkingseenheden zijn die elk hun eigen data kunnen opvragen en manipuleren.
- *Multiple-Instruction stream, Single-Data stream (MISD).*
Computers die meerdere instructies op hetzelfde stuk data laten werken zijn er niet in pure vorm, hoewel sommigen machines met een *pipeline* beschouwen als MISD-computers [TANEN99].
- *Multiple-Instruction stream, Multiple-Data stream (MIMD).*
Deze categorie bevat de meeste systemen met meerdere processors, en zal voornamelijk onze aandacht hebben.

Hoewel dit een aardig overzicht is, is het allesbehalve volledig. Tanenbaum heeft een uitbreiding gemaakt op Flynn's taxonomie (zie Afbeelding 3), waarbij de categorieën SIMD en MIMD verder zijn uitgewerkt [TANEN99].



Afbeelding 3. Een taxonomie van parallelle computerarchitecturen naar [TANEN99].

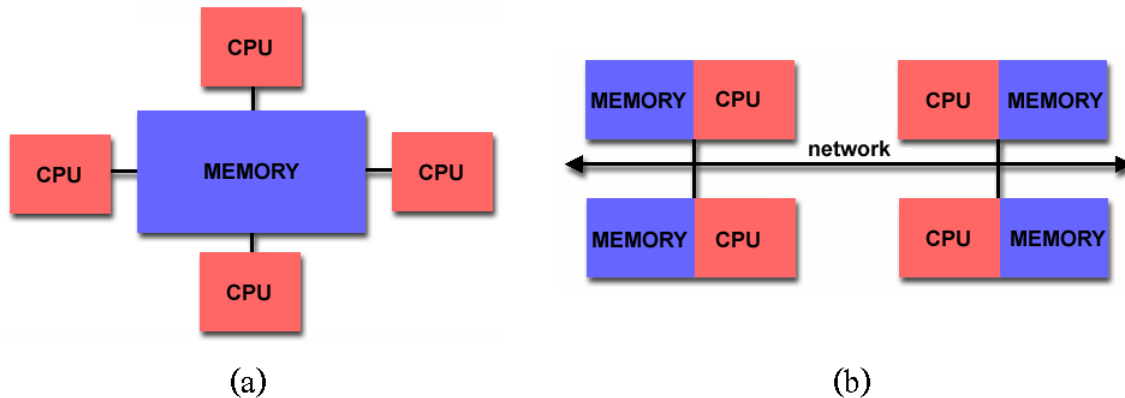
Bij SIMD-computers worden *array* en *vector processors* onderscheiden. Dergelijke computers worden gebruikt bij rekenintensieve problemen in een wetenschappelijke context waarbij er regelmatige datastructuren zoals vectoren en rijen gebruikt worden. We zullen hier niet verder op ingaan.

In de MIMD-categorie is er een opsplitsing gemaakt tussen *multiprocessors* en *multicomputers*. Het grote verschil tussen deze twee is dat bij multiprocessors elke processor het geheugen kan aanspreken, terwijl dit niet kan bij multicomputers, waardoor communicatie in het laatste geval noodzakelijk is. De eerstgenoemde zijn zogenaamde *shared memory systems*: meerdere processors maken gebruik van een gedeeld geheugen (zie Afbeelding 4(a)). De laatstgenoemde computers worden wel *message-passing* of *distributed memory systems* genoemd (zie Afbeelding 4(b)), omdat de gegevens verspreid staan over meerdere geheugens en er derhalve informatie tussen de verschillende knopen in het netwerk uitgewisseld moet worden.

Er zijn meerdere soorten multiprocessors. Ze verschillen in de manier waarop ze omgaan met het gedeelde geheugen. Bij grote multiprocessors is er sprake van een variabele snelheid waarmee opgevraagde gegevens uit het geheugen beschikbaar zijn, omdat het geheugen opgesplitst is in meerdere modules [TANEN99]. De vraag is hoe je het beste hiermee om kunt gaan. Hierdoor zijn er verschillende implementaties ontstaan (UMA, COMA, NUMA), die qua naam logischerwijs allemaal eindigen op de letters MA (Memory Access).

In [QUINN87] wordt onderscheid gemaakt tussen *tightly* en *loosely coupled* multiprocessors. In het eerste geval is er sprake van een globaal gedeeld geheugen, terwijl in het tweede geval de lokale geheugenmodules van elke computer worden gebruikt. Tanenbaum noemt systemen (in het algemeen) *loosely coupled* als ze een klein aantal grote, onafhankelijke processors

hebben die communiceren over een relatief langzame verbinding, en tightly-coupled⁶ als de componenten kleiner zijn, dichter op elkaar zitten en de beschikking hebben over een snelle verbinding [TANEN99]. Als we deze begrippen gebruiken is de betekenis dus contextafhankelijk.



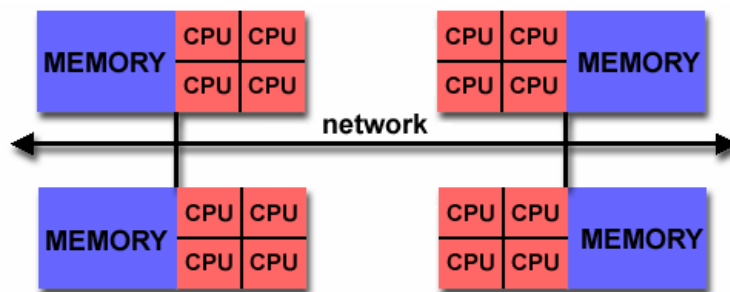
Afbeelding 4. Twee MIMD-machines naar [BARNE05]. (a) Een multiprocessor met 4 processors. (b) Een multicomputer met 4 processors.

Multicomputers kunnen grofweg in twee categorieën worden ingedeeld.

De eerste is die van de *Massively Parallel Processors*, kortweg *MPP*. Dit zijn vaak dure, tightly-coupled supercomputers met vele processors. Twee topologieën, een grid en een hypercube, worden genoemd. Deze, en meer, zullen we bekijken in 3.3.2 als we topologieën gaan beschouwen.

De andere categorie bestaat uit normale pc's of werkstations die via een standaardnetwerk worden verbonden. Pc's in een netwerk worden soms aangeduid met *NOW* (Network of Workstations) of *COW* (Cluster of Workstations) [TANEN99]. Een architectuur is *homogeen* als alle werkstations in het netwerk hetzelfde zijn; anders wordt van *heterogeen* gesproken.

Hoewel er theoretisch geen grote verschillen zijn, zijn die er in de praktijk wel: COWs kosten slechts een fractie van een MPP-systeem. Bovendien verschillen ze in de manier waarop ze gebouwd en gebruikt worden. MPPs gebruiken state-of-the-art technologie, terwijl een cluster van werkstations



Afbeelding 5. Een hybride systeem [BARNE05]: zowel shared als distributed memory architectureën worden toegepast.

bestaat uit betaalbare standaardtechnologie. Een supercomputer bouw je meestal voor een speciaal doel, terwijl je een aantal pc's in een netwerk misschien al een COW zou kunnen noemen. Het doel van deze uitwijding is om aan te geven dat de verdeling in twee categorieën enigszins arbitrair is aangezien het geen technische, maar praktische scheiding is.

⁶ Bewust of onbewust: Tanenbaum schrijft *loosely coupled* los, terwijl *tightly-coupled* aaneen wordt geschreven; hiermee wordt het verschil in begrip benadrukt.

In Tabel 4 staan voor- en nadelen van shared en distributed memory systems volgens [BARNE05]. Dit geeft een ietwat praktischere benadering van multiprocessors en –computers. Hoewel als een voordeel genoemd wordt dat geheugentoegang bij een shared memory systeem in constante tijd kan plaatsvinden, strookt dit niet met de taxonomie uit Afbeelding 3. Multiprocessors kunnen namelijk ook computers zijn waarbij deze tijd niet constant is, te weten NUMA (NonUniform Memory Access). De beweringen over de schaalbaarheid van de systemen worden wel ondersteund door [TANEN99]. Er wordt namelijk door Tanenbaum opgemerkt dat het implementeren van een systeem met gedeeld geheugen een uitdaging is voor een paar honderd processors, terwijl het bouwen van een multicomputer met 10.000 CPUs of meer redelijk rechttoe rechtaan is. Daar staat tegenover dat multiprocessors makkelijker te programmeren zijn dan multicomputers. Het ideale ontwerp is schaalbaar (*scalable*), d.w.z. blijft goed presteren als er processors worden toegevoegd. Een manier om dit te doen is door het bouwen van hybride systemen. In Afbeelding 5 is een dergelijk systeem afgebeeld.

	<i>Shared memory system</i>	<i>Distributed memory system</i>
Voordelen	▪ Globale geheugenruimte is gebruiksvriendelijk vanuit het perspectief van een programmeur. ▪ Het delen van gegevens is snel en kan in een vaste hoeveelheid tijd gebeuren vanwege de nabijheid van het geheugen tot de processors.	▪ Geheugen is schaalbaar met het aantal processors. De geheugengrootte neemt proportioneel toe met het aantal processors. ▪ Elke processor kan snel en zonder onderbrekingen zijn eigen geheugen benaderen. ▪ Kosteneffectief: gemeengoed processors en netwerken kunnen worden gebruikt.
Nadelen	▪ Niet (eenvoudig) schaalbaar: het toevoegen van processors leidt tot meer verkeer op het communicatiekanaal tussen het geheugen en de processors. Het wordt daardoor steeds moeilijker en duurder om een ontwerp te maken met een toenemend aantal processors. ▪ De programmeur is verantwoordelijk voor de synchronisatie die voor correcte toegang tot het geheugen zorgt.	▪ De programmeur is verantwoordelijk voor allerlei details aangaande de communicatie tussen processors. ▪ Het kan lastig zijn de bestaande datastructuren (die uitgaande van globaal geheugen) te gebruiken. ▪ De toegang tot geheugen kan niet in vaste hoeveelheid tijd plaatsvinden.

Tabel 4. Voor- en nadelen van shared en distributed memory systems.

3.3.2 Topologieën

We hebben gezien wat multiprocessors en –computers zijn. Nu gaan we bekijken op welke manier de communicatiekanalen tussen de componenten kunnen worden gelegd. De topologie bepaalt de manier waarop dit gebeurt. Een topologie kunnen we beschrijven met een graph. Eerst bepalen we hoe we een topologie interpreteren, en introduceren we kort de benodigde graphterminologie. Daarna bekijken we een aantal topologieën en hun kenmerkende eigenschappen.

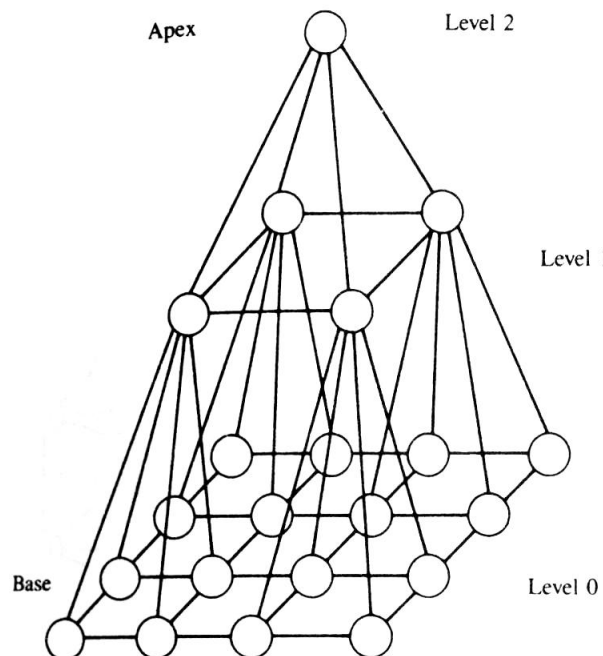
Als we een topologie als een graph beschouwen zijn er verschillende interpretaties mogelijk. In [QUINN87] wordt van *processor organizations* – methoden om processors in een parallelle computer te verbinden – gesproken. De lijnen zijn de verbindingen en de knopen de processors. Deze interpretatie is helder en eenvoudig, maar lijkt geheugen buiten beschouwing te laten en verliest daarmee zijn algemene toepasbaarheid.

In [TANEN99] wordt over de topologie van *interconnection networks* gesproken. Een interconnection network bestaat uit maximaal vijf componenten: processors, geheugenmodules, interfaces, links en switches. De lijnen in de graph worden *links* genoemd en de knopen *switches*; de overige componenten worden niet weergegeven. Normaal gesproken zouden de processors en geheugens via interfaces aan de switches gekoppeld worden. De links zijn de fysieke verbindingen en de switches routeren *packets*. Een packet bestaat uit een verzoek van de processor om data uit geheugen dat elders is. Deze interpretatie heeft als voordeel dat zij zowel op multiprocessors als op multicomputers van toepassing is, en zullen we daarom gebruiken. Bij multiprocessors komt dit neer op hoe de processors en de geheugenmodules aan elkaar worden gekoppeld, en de modules onderling. Bij multicomputers bepaalt het de manier waarop de computers onderling communiceren.

We gaan nu een graph en samenhangende begrippen in woorden omschrijven. Een graph G heeft *knopen* en *lijnen*. Elke knoop in de graph heeft een aantal verbindingen met andere knopen. Dit aantal wordt de *graad* genoemd. Een *pad* tussen twee verbonden knopen a en b is de kortste route tussen a en b . De *diameter* is het maximum van de lengtes van alle paden. Dit begrip is belangrijk voor het versturen van berichten: hoeveel tijd kost het in het slechtste geval? Een andere belangrijke eigenschap heeft te maken met de hoeveelheid data die verstuurd kan worden. Een maat hiervoor is de *bisection bandwidth* [TANEN99]. Om dit te berekenen moeten we het netwerk in twee gelijke delen opsplitsen (wat betreft knopen) door denkbeeldig een aantal lijnen te verwijderen. Vervolgens berekenen we de totale bandbreedte van de verwijderde lijnen. Deze opsplitsing kan op meerdere manieren, maar we zijn geïnteresseerd in het minimum. Het idee erachter is dat in het slechtste geval de communicatie beperkt wordt door de bisection bandwidth.

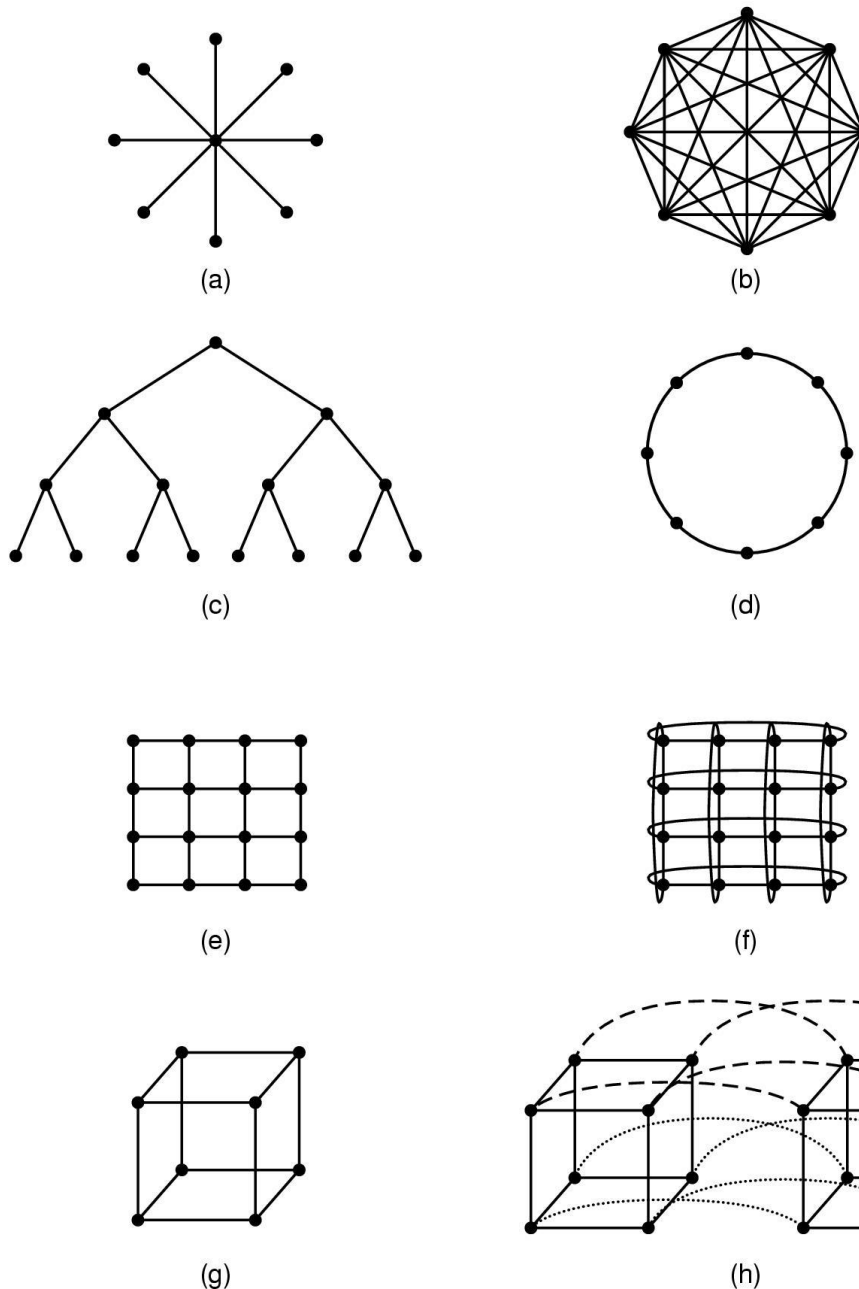
In Afbeelding 7 zien we verscheidene topologieën, waarvan we sommige bespreken aan de hand van [TANEN99].

In de ster (*star*) zouden aan de buitenste knopen processors met geheugen hangen, terwijl de middelste alleen packets routeert, zie Afbeelding 7(a). Deze configuratie is eenvoudig qua ontwerp, maar de centrale knoop zou bij grote systemen een bottleneck vormen. Bovendien is dit ontwerp foutgevoelig, omdat het uitvallen van de centrale knoop leidt tot het ter aarde storten van het hele systeem. Bij de complete topologie (*full interconnect*) in Afbeelding 7(b) zijn alle knopen met elkaar verbonden. Dit ontwerp lijkt alleen maar voordelen te



Afbeelding 6. Een piramidenetwerk van grootte 16 naar [QUINN87]. De *apex* is de top en de *base* de bodem.

hebben: de bisection bandwidth wordt gemaximaliseerd, de diameter is minimaal en het is extreem robuust (er kunnen zes willekeurige verbindingen uitvallen). Helaas neemt het aantal verbindingen v sterk toe met het aantal knopen k , omdat $v = k*(k - 1)/2$. Dit loopt uit de hand als k groot wordt.



Afbeelding 7. Verschillende topologieën naar [TANEN99]. (a) Een ster. (b) Een complete topologie. (c) Een boom. (d) Een ring. (e) Een raster. (f) Een dubbele torus. (g) Een kubus. (h) Een hyperkubus.

Een derde topologie is de boom (*tree*), te zien in Afbeelding 7(c). Vergeleken met de complete topologie verhoudt het aantal verbindingen v zich redelijk tot het aantal knopen k : voor een volle boom geldt $v = k - 1$. Ook de diameter is redelijk gunstig, deze is in de orde van $\log_2(k)$ [CHEMI94]. Een probleem bij dit ontwerp is dat de bisection bandwidth gelijk is

aan de capaciteit van de verbindingen. Omdat het berichtenverkeer intensief is in de bovenste knopen in de boom, zal hier een bottleneck ontstaan. Om de bisection bandwidth te verhogen kun je de knopen in de top van de boom meer bandbreedte geven, wat een *fat tree* wordt genoemd. Een andere vorm die je wel eens tegenkomt is het piramiden netwerk (*pyramid network*, zie Afbeelding 6), wat in feite een volle boom met ariteit vier is, waarvan de knopen op elk niveau met elkaar zijn verbonden in een *mesh network* [QUINN87]. Merk op dat dit geen echte boom is, omdat er cycli (*cycles*) in voorkomen.

Een ander woord voor mesh network is een raster (*grid*), te zien in Afbeelding 7(e). Dit ontwerp is regelmatig van opzet, schaalbaar en heeft een diameter die slechts toeneemt met de wortel van het aantal knopen. Een variant hierop is de dubbele torus (*double torus*), waarbij ook de eerste en de laatste knoop van een kolom resp. rij met elkaar zijn verbonden (zie Afbeelding 7(f)). De foutgevoeligheid neemt hiermee af, net als de diameter.

Een andere regelmatig opgebouwde structuur is de kubus, geïllustreerd in Afbeelding 7(g). Interessanter is de hyperkubus (*hypercube*), wat een n -dimensionale kubus is (met $n \geq 4$). In Afbeelding 7(h) is een 4D kubus afgebeeld. Veel parallelle computers gebruiken deze topologie omdat de diameter relatief langzaam groeit met het aantal knopen. De diameter is namelijk de $\log_2$ van het aantal knopen. Een 10-dimensionale hypercube heeft 1024 knopen, maar slechts een diameter van 10. Een grid heeft bij een gelijk aantal knopen zes keer zo veel verbindingen nodig. Het aantal verbindingen ligt wel redelijk hoog door de lage diameter, maar de hypercube is toch de gebruikelijke keuze voor systemen die goed moeten presteren.

3.4 Parallellisatiestrategieën

In deze paragraaf gaan we parallellisatiestrategieën bekijken voor gestuurde willekeurige zoektechnieken in het algemeen. We zitten daarmee op het niveau van *tabu search* en *simulated annealing* (zie Afbeelding 1). In [CRAIN98] worden deze zoektechnieken, zoals genetische algoritmen, onderverdeeld in drie typen:

1. Het parallelliseren van operaties binnen een iteratie van een oplossingsmethode. Deze strategie, ook wel *low-level parallelism* genoemd, is er alleen op gericht berekeningen te versnellen, zonder de zoekruimte intensiever te verkennen of op zoek te gaan naar betere oplossingen. Uiteindelijk wordt hetzelfde resultaat opgeleverd als in het geval dat sequentieel gewerkt zou worden, alleen gebeurt het sneller.
2. *Decompositie* van het probleemdomein of de zoekruimte. Deze methode is gebaseerd op het principe dat rekenkracht toegewezen kan worden toegewezen aan het oplossen van een reeks kleine problemen, waaruit een totale oplossing van het probleem kan worden geconstrueerd. Het zoektraject van de parallelle aanpak is verschillend van die van de sequentiële methode.
3. Meerdere zoekacties (*threads*) tegelijk starten, met verschillende mate van synchronisatie en samenwerking. Deze strategie probeert de zoekruimte grondiger te verkennen door gelijktijdig te zoeken door het domein d.m.v. meerdere threads.

We behandelen deze drie types achtereenvolgens, en bekijken hoe ze toegepast kunnen worden op GAs.

3.4.1 Low-level parallelism

Low-level parallelism houdt in dat sommige of alle operaties die onderdeel uitmaken van een iteratie van een sequentiële methode, gelijktijdig uitgevoerd worden. Strategieën op dit

niveau zijn erop gericht de looptijd van een bepaalde methode te verlagen. In de context van PGAs wordt gesproken van *globale parallelisatie* (zie 3.6).

3.4.2 Decompositie

Decompositie splitst het probleem op in (niet-overlappende) deelproblemen. Deze deelproblemen worden onafhankelijk van elkaar en gelijktijdig opgelost. Hieruit kan vervolgens een oplossing voor het gehele probleem worden gemaakt. Volgens [CRAIN98] is er geen parallel genetisch algoritme bekend die gebruik maakt van decompositie. Er wordt wel opgemerkt dat PGAs waarbij populaties worden opgesplitst tot het derde type parallelisme behoren. De genetische operatoren maken namelijk de deelpopulaties (zogenaamde *demes*) homogeen. De zoekruimte wordt niet verdeeld: elke deme kan namelijk individuen creëren uit de gehele populatie. Hierdoor zijn de deelproblemen overlappend geworden, en voldoen ze niet meer aan de definitie van decompositie.

Alle implementaties die bekeken werden maken gebruik van het “natuurlijk” parallelisme dat inherent is aan evolutionaire technieken. Impliciet parallelisme speelt hier een belangrijke rol in, en is besproken in paragraaf 2.5.

3.4.3 Threads

De derde parallelisatiemethode is het gebruiken van threads door meerdere zoekacties te starten. Hierbij wordt de zoekruimte gelijktijdig doorzocht. Er kan vanuit dezelfde initiële oplossing worden uitgegaan, maar dit hoeft niet. Bij een zoekstrategie als deze moeten een aantal belangrijke parameters worden bepaald volgens [CRAIN98], namelijk:

- de topologie, die definieert hoe de threads verbonden zijn met elkaar;
- de communicatiemethode (broadcast, propageren, één centrale opslagplaats, etc.);
- de threads waartussen informatie wordt uitgewisseld;
- het type communicatie (synchroon of asynchroon);
- het tijdstip waarop uitwisseling van informatie plaatsvindt;
- de uit te wisselen informatie.

Communicatie is *synchroon* als de threads tegelijkertijd informatie uitwisselen; *asynchroon* als dit niet op hetzelfde moment gebeurt. Dit staat los van het tijdstip waarop informatieuitwisseling plaatsvindt, wat op een vast of willekeurig tijdstip kan zijn.

Binnen de context van GAs zijn er twee belangrijke parallelisatiestrategieën: *coarse-grained* en *fine-grained* parallelisme. Deze zijn ontstaan uit het onderzoek dat is gedaan naar *multiple-deme* PGAs. Al deze onderwerpen komen later in dit hoofdstuk aan bod.

3.5 Taxonomie van PGAs

We hebben gezien hoe zoektechnieken gebruik kunnen maken van parallelisatie. Nu gaan we dieper de boom in van Afbeelding 1, en zijn we aangekomen op het niveau van parallelle genetische algoritmes. In [NOWOS99] is een taxonomie voorgesteld voor parallelle genetische algoritmen waarin (vergeleken met andere classificaties) veel verschillende klassen expliciet worden onderscheiden. Deze bestuderen we om een zo breed mogelijke kijk te krijgen op PGAs. Vervolgens kunnen we specifieke onderdelen nader bestuderen.

De manier waarop GAs geparalleliseerd kunnen worden, hangt af van:

- hoe de fitnessfunctie geëvalueerd wordt en hoe mutatie toegepast wordt;
- of een enkele of meerdere deelpopulaties worden gebruikt;

- hoe individuen worden uitgewisseld (als meerdere deelpopulaties worden gebruikt);
- hoe selectie plaatsvindt (globaal of lokaal).

Er wordt onderscheid gemaakt tussen acht (min or meer generieke) klassen:

1. *Master-slave parallelisatie*

Dit algoritme maakt gebruik van een enkele populatie. De evaluatie van de individuen en/of de toepassing van de genetische operatoren wordt parallel uitgevoerd. Master-slave parallelisatie staat ook wel bekend als gedistribueerde fitnessevaluatie (*distributed fitness evaluation*) en *globale parallelisatie*, en wordt besproken in 3.6.

2. *Statische deelpopulaties met migratie*

Kenmerkend voor deze klasse is het gebruik van meerdere deelpopulaties (*demes*) en de aanwezigheid van een migratie-operator. Multiple-deme PGAs, de gebruikelijke naam voor ‘statische deelpopulaties met migratie’, worden veel toegepast, en worden besproken in 3.7. Migratie komt aan bod in 3.7.2. Coarse en fine-grained parallelisme vallen in deze klasse, welke in resp. 3.8 en 3.9 worden behandeld.

3. *Statische overlappende deelpopulaties zonder migratie*

Dit model lijkt sterk op het vorige, in de zin dat het ook demes kent. Het verschil is echter dat er geen sprake is van migratie (het uitwisselen van individuen), maar van overlappende gebieden (*overlapping areas*). Sommige individuen behoren tot meerdere deelpopulaties en worden in meerdere selectie- en cross-overoperaties meegenomen. Deze methode kan toegepast worden op shared memory systems door de gehele populatie in het gedeelde geheugen te plaatsen en verschillende threads parallel op de deelpopulaties uit te voeren. Hierbij is enige synchronisatie nodig voor de overlappende gebieden als genetische operatoren worden toegepast. Verschillende computerarchitecturen die een geschikte ruimtelijke verdeling van de demes toestaan zijn geschikt, zoals een 2D- of 3D-mesh (zie Afbeelding 7(e)).

4. *Grootschalige parallelle genetische algoritmes*

Als we het aantal demes in het statische overlappende model verhogen en het aantal individuen in elke deme verlagen krijgen we een (fine-grained) *massively parallel genetic algorithm*. Omdat de kenmerken van deze soort algoritmes behoorlijk verschillen van het overlappende (coarse-grained) model, en omdat het algoritme normaliter wordt gebruikt in MPP-systemen (zie Afbeelding 3), is een aparte klasse in de PGA-taxonomie ingericht. Er is maar één populatie bij dit algoritme. De interactie tussen de individuen wordt beperkt door de ruimtelijke structuur van de computerarchitectuur die wordt gebruikt. Een individu kan namelijk alleen kruisen met zijn burens. Het is gebruikelijk om de individuen in een grid te plaatsen, omdat MPP-systemen vaak een dergelijke topologie aannemen. Andere interconnection networks, zoals een ring, torus, kubus of hyperkubus (zie Afbeelding 7), zijn ook mogelijk. Het ideale geval is om voor elke individu een PE (*Processing Element*) te hebben, wat goed past bij massively parallel computers. Het model kan ook worden gesimuleerd op een cluster of workstations (zie 3.3.1), maar in dit geval neemt dat hoge communicatiekosten met zich mee, wat deze oplossing minder aantrekkelijk maakt.

5. *Dynamisch overlappende deelpopulaties*

Dynamic demes is een combinatie van globale parallelisatie met een coarse-grained GA. Er is geen rol weggelegd voor de migratie-operator, omdat er één populatie is en

informatie tussen individuen wordt uitgewisseld door een dynamische reorganisatie van de demes gedurende een run. Het idee erachter is om het wachten op de traagste verwerking van individuen te verminderen door de deelpopulaties dynamisch te splitsen. Bovendien is het algoritme schaalbaar omdat het aantal individuen per deme is in te stellen, en kan het zowel op shared als op distributed memory systems worden geïmplementeerd.

6. *Parallele steady-state genetische algoritmes*

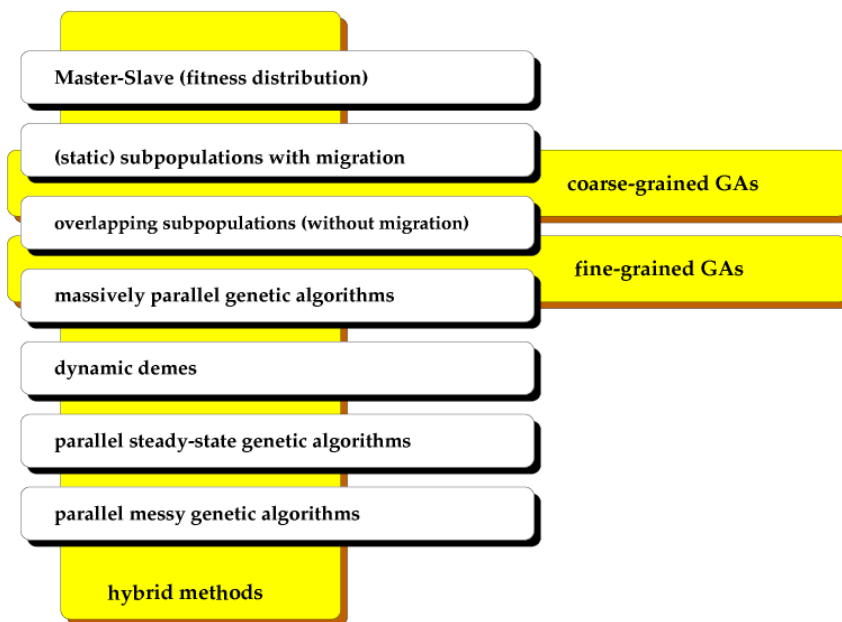
Bij steady-state GAs is het relatief eenvoudig bepaalde genetische operatoren te paralleliseren, omdat hierbij de populatie continu wordt geüpdatet. Selectie en het vervangen van individuen moet sequentieel gebeuren, maar de andere genetische operatoren en fitnessevaluatie kan parallel gebeuren.

7. *Parallele messy genetische algoritmes*

Hoewel we messy GAs niet hebben bekeken, gaan we er toch even kort op in. Het doel ervan is om de prestaties van functieoptimalisatie te verbeteren door steeds langere strings op te bouwen uit building blocks [MITCH98]. Messy GAs bestaan uit drie fasen. In de initialisatiefase en de *primordial phase* wordt de initiële populatie gecreëerd. De eerste fase kan parallel worden uitgevoerd omdat communicatie niet noodzakelijk is. De primordial phase neemt het leeuwendeel van de rekentijd in beslag, waardoor deze fase in aanmerking komt voor parallelisatie van selectie en fitnessevaluatie. In de laatste fase, de *juxtapositional phase*, worden deeloplossingen uit de vorige fase met elkaar gecombineerd.

8. *Hybride methodes*

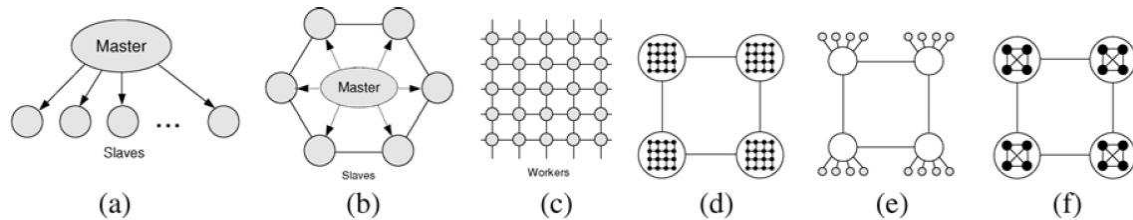
Om GAs te paralleliseren is een combinatie van bovenstaande methoden ook een mogelijkheid. Het combineren van technieken kan resulteren in een algoritme dat de voordelen van de verschillende onderdelen heeft, maar toch betere prestaties laat zien dan elke component apart.



Afbeelding 8. Een taxonomie van PGAs volgens [NOWOS99].

Uit Afbeelding 8 worden de relaties tussen coarse en fine-grained GAs en de bovenstaande klassen duidelijk.

In andere overzichten van parallelle genetische algoritmes, zoals in [CANTU97] en [ALBA02a], wordt onderscheid gemaakt tussen globale parallelisatie, coarse-grained GAs, fine-grained GAs en hybride algoritmes (zie Afbeelding 9). In [CANTU97] worden hybride algoritmes ook wel *hierarchical* PGAs genoemd, omdat er sprake is van meerdere niveaus.



Afbeelding 9. Verschillende modellen van een PGA, naar [ALBA02a]. (a) Globale parallelisatie. (b) Coarse grain. (c) Fine grain. (d) Coarse en fine grain. (e) Coarse grain en globale parallelisatie. (f) Coarse grain en coarse grain.

3.6 Globale parallelisatie

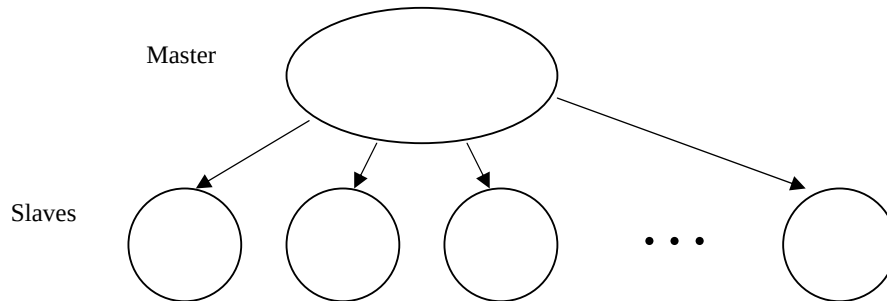
In deze klasse van PGAs wordt de evaluatie van individuen en de toepassing van genetische operatoren expliciet geparalleliseerd. Er is sprake van één globale populatie. Alle individuen kunnen met elkaar kruisen. De semantiek van de operatoren blijft dezelfde. Implementatie is relatief eenvoudig, waarbij een versnelling proportioneel met het aantal processors verwacht kan worden.

De evaluatie wordt parallel uitgevoerd door een deelverzameling van individuen aan elke processor toe te wijzen. Er is geen communicatie tussen de processors gedurende het evaluatieproces, omdat de fitness van elk individu onafhankelijk is van de anderen. Communicatie vindt alleen plaats als de individuen worden verzonden en als de fitnesswaarden ervan worden teruggestuurd. Als het algoritme wacht totdat alle resultaten binnen zijn, dan wordt het *synchroon* genoemd. Een synchroon parallel genetisch algoritme heeft dezelfde eigenschappen als het basisalgoritme, met als enige verschil de snelheid. Bij de asynchrone versie wordt er niet gewacht op langzame processors, en gaat de vergelijking met het simpele genetische algoritme niet meer op.

Op een shared-memory systeem (zie Afbeelding 4(a)) kunnen de individuen in het gedeelde geheugen worden opgeslagen. Elke processor kan vervolgens zijn individuen lezen, evalueren, en de resultaten ervan naar het geheugen schrijven zonder dat er een conflict optreedt. De versnelling is proportioneel (maar waarschijnlijk sublineair) met het aantal processors. Superlineariteit (zie 3.10) valt niet te verwachten, vanwege de verdeel-en-heers-aanpak [KONFR04].

Op een computer met gedistribueerd geheugen (zie Afbeelding 4(b)) kan de populatie in één processor worden opgeslagen om de toepassing van genetische operatoren te vereenvoudigen. De 'master'-processor is verantwoordelijk voor het sturen van individuen naar 'slave'-processors die de fitness ervan uitrekenen en de operatoren erop loslaten om zo de nieuwe generatie te produceren. Deze vorm wordt wel *master-slave parallelization* genoemd (zie Afbeelding 10).

In [CANTU97] staat een onderzoek van Marin, Trelles-Salazar en Sandoval beschreven. Zij gebruikten het master-slave model, waarbij de slaves een GA uitvoerden en hun beste (deel)resultaten naar het master-proces stuurden. Vervolgens koos de master de beste individuen tot dan toe uit en stuurde deze naar alle slaves. Experimentele resultaten laten zien dat op een klein netwerk van zes workstations een bijna lineaire versnelling kan worden bereikt. Deze aanpak zou volgens de auteurs schaalbaar moeten zijn omdat de communicatie maar af en toe plaatsvindt.



Afbeelding 10. Een schematische weergave van een master-slave parallel genetisch algoritme.

Je kunt met globale parallelisatie nog verder gaan door de toepassing van genetische operatoren parallel uit te voeren. We bekijken de operatoren van het basialgoritme. De selectieoperator vereist meestal bepaalde statistische gegevens (zoals de gemiddelde fitness van de populatie) waardoor er een bottleneck ontstaat door sequentiële berekening hiervan. Bij tournament selection (zie 2.6.1.4) daarentegen is het voldoende als van een kleine deelverzameling van de populatie de fitnesswaarde bekend is. Cross-over kan op $n/2$ paren van individuen tegelijkertijd worden toegepast. Mutatie kan op elk individu worden uitgevoerd, onafhankelijk van de anderen.

Het is echter de vraag of het de moeite loont de stappen van genetische operatoren te paralleliseren. Ze zijn vrij eenvoudig en de benodigde tijd voor communicatie kan langer zijn dan de tijd die nodig is voor berekeningen. Zeker in machines met gedistribueerd geheugen, waarbij de kosten van communicatie relatief hoog kunnen zijn, speelt dit een rol.

Al met al zijn master-slave PGAs relatief eenvoudig te implementeren en is het een efficiënte parallelisatiemethode als de evaluatie van de fitness een tijdrovende bezigheid is. Omdat de manier van zoeken niet wezenlijk anders is, kan de beschikbare theorie voor het basialgoritme ook toegepast worden [CANTU97].

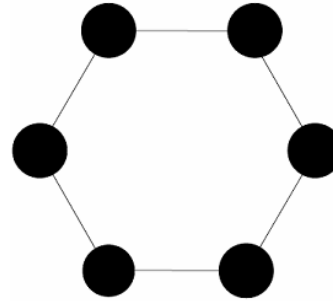
3.7 Multiple-deme PGAs

Multiple-deme parallel GAs maken gebruik van een aantal relatief grote deelpopulaties (demes) en van migratie (zie Afbeelding 11). Migratie (*migration*) houdt in dat er individuen worden uitgewisseld tussen de verschillende processen. We zullen eerst een aantal onderzoeksresultaten aan bod laten komen, waarin enkele begrippen aan de orde komen. Daarna bespreken we wat er nog meer komt kijken bij migratie.

3.7.1 Onderzoek

In [CANTU97] wordt een uitgebreide uiteenzetting van invloedrijke papers gegeven. Relevante stukken hiervan worden hieronder besproken.

Grosso was een van de eersten die PGAs met meerdere populaties systematisch heeft onderzocht. Hij experimenteerde met vijf demes en een vast migratiepercentage (*migration rate*, het deel van de deelpopulatie dat migreert), en ondervond dat de snelheid waarmee de gemiddelde fitness van de populatie toenam, hoger was dan in een enkele grote populatie. Dit bevestigt een principe wat in de populatiegenetica bestaat, namelijk dat een goede eigenschap zich sneller verspreidt in een kleine populatie dan in een grote. Hij merkte echter ook op dat als de demes geïsoleerd waren, de stijging van de fitness op een lager niveau stopte dan bij de grote populatie. Anders gezegd, de kwaliteit van de oplossing na convergentie was lager. Met een laag migratiepercentage waren de resultaten vergelijkbaar met het geval van de geïsoleerde demes. Een gemiddeld percentage leverde echter oplossingen die gelijkwaardig waren aan het geval waarbij een enkele populatie werd gebruikt. Deze observaties laten zien dat er een kritieke migration rate is waaronder de prestaties negatief worden beïnvloed vanwege de isolatie van de demes, en waarboven een gelijkwaardige prestatie wordt neergezet t.o.v. de enkelvoudige populatie. Latere onderzoeken bevestigen dat het gebruiken van migratie beter werkt om een globaal optimum te vinden dan het niet toepassen ervan.



Afbeelding 11. Schematische weergave van een multiple-deme PGA uit [CANTU97]. Elk proces is een simpel GA, en er vindt (af en toe) communicatie plaats tussen de populaties.

Uit onderzoek van Tanese, waarbij migratie op vaste tijdstippen plaatsvond, blijkt o.a. dat als de frequentie waarmee individuen worden uitgewisseld – het *migration interval* – te hoog of te laag is, het algoritme ook slechter gaat presteren. De migration rate is dus niet de enige parameter die van belang is.

Cohoon, Hedge, Martin en Richards vonden overeenkomsten tussen de evolutie van oplossingen bij PGAs en de theorie van *punctuated equilibria*. Deze theorie zegt dat populaties het merendeel van de tijd in een equilibrium (evenwicht) zijn – er zijn geen wezenlijke veranderingen in de genetische samenstelling – maar dat veranderingen in de omgeving een snelle ontwikkeling in gang kunnen zetten. Een belangrijke onderdeel van de omgeving van een populatie is zijn eigen samenstelling, omdat individuen daarin constant elkaar beconcurreren. De aankomst van individuen uit andere populaties kan dit evenwicht doorbreken (punctuate) en evolutionaire krachten aan het werk zetten. In hun onderzoeken kwamen Cohoon et al. tot de conclusie dat er weinig gebeurde tussen migraties door, maar dat nieuwe oplossingen gevonden werden kort nadat uitwisseling van individuen had plaatsgevonden.

Multiple-deme PGAs zijn erg populair, en er zijn talloze onderzoeken aan gewijd. Er zijn een aantal redenen te bedenken hiervoor [CANTU97]:

- Het recept is eenvoudig: neem een aantal conventionele GAs, draai ze op een parallelle computerarchitectuur, en wissel op vooraf gekozen tijden individuen uit.
- Het kost weinig moeite een GA om te vormen in een multiple-deme PGA. Het grootste gedeelte van het bestaande programma hoeft niet veranderd te worden: er hoeven alleen een aantal subroutines aan te worden toegevoegd.
- Coarse-grain parallelle computers zijn vaak beschikbaar en zo niet, dan zijn ze eenvoudig te simuleren met een netwerk van werkstations.

3.7.2 Migratie

In deze paragraaf bekijken we nieuwe begrippen en parameters uit [CRAIN98] vanwege de introductie van de migratie-operator.

Met de komst van meerdere deelpopulaties moeten beslissingen genomen worden t.a.v. het aantal deelpopulaties, en het aantal individuen per deelpopulatie. We hebben reeds kennisgemaakt met de *migration rate* – het deel van de deelpopulatie dat migreert – en het *migration interval* – de frequentie waarmee individuen worden uitgewisseld. De *emigration policy* bepaalt hoe individuen worden geselecteerd voor migratie: de beste, willekeurig, willekeurig uit de betere individuen, etc. De *immigration policy* geeft aan welke individuen worden vervangen in de deelpopulatie aan de ontvangende kant: de slechtste, willekeurige selectie, etc.

In de volgende paragrafen bekijken we hoe multiple-deme PGAs verder ontwikkeld en verfijnd werden.

3.8 Coarse-grained parallellisme

Bij deze vorm van parallellisme wordt de populatie verdeeld in een aantal deelpopulaties, die relatief geïsoleerd zijn ten opzichte van elkaar. Het aantal demes is relatief laag, maar er zitten veel individuen in een deme. Coarse-grained PGAs worden gekenmerkt door hun relatief lange verwerkingstijd van een deme en door de geregelde communicatie voor het uitwisselen van individuen. De migratie-operator, die voor deze uitwisseling zorgt, hebben we besproken in 3.7.2. Soms worden deze algoritmes ook wel *distributed algorithms* genoemd, omdat ze normaal gesproken op distributed memory systems (zie Afbeelding 4(b)) worden geïmplementeerd.

Eerst kijken we naar interessant onderzoek in dit gebied op basis van [CANTU97].

Vervolgens bespreken we welke topologieën geschikt zijn en waarmee bij de keuze ervan rekening moet worden gehouden.

3.8.1 Onderzoek

Een belangrijke vraag in het onderzoek naar PGAs is of, en onder welke omstandigheden, een parallel GA een betere oplossing kan vinden dan een serieel GA. Starkweather, Whitley, en Mathias hebben twee belangrijke observaties gedaan omtrent deze vraag. Als eerste blijken relatief geïsoleerde demes vaak tot verschillende oplossingen te leiden, en kunnen migratie en recombinitie deeloplossingen combineren. Ze speculeerden dat een PGA met een lage migration rate mogelijk betere oplossingen vindt voor opsplitsbare functies. Hun tweede observatie was dat als deze recombinitie leidt tot individuen met een lagere fitness (doordat bijv. de functie niet op te splitsen is), het seriële genetische algoritme in het voordeel is.

Bij het overgrote deel van de multiple-deme PGAs vindt migratie gesynchroniseerd plaats, dus op een vooraf afgesproken moment. Er zijn ook voorbeelden waar bij migratie asynchroon gebeurt. Braun liet bijv. migratie pas plaatsvinden nadat de demes geen vooruitgang meer boekten. Hij probeerde de diversiteit in de deelpopulaties te herstellen, om zo premature convergence te voorkomen.

De vraag die hierop volgt is: wanneer is het de juiste tijd om te migreren?

Als migratie te vroeg plaatsheeft, is het aantal goede building blocks in de migranten wellicht te laag om de zoektocht in de goede richting te sturen. Communicatie is in dat geval alleen maar verspilling van moeite en tijd.

3.8.2 Topologieën

In het algemeen is een coarse-grained PGA geschikt om te draaien op een loosely coupled systeem met een relatief lage bandbreedte [KOZA98]. Coarse-grained PGAs zijn bovendien geschikt voor heterogene netwerken [NOWOS99]. Omdat er in verhouding veel individuen in een deme zitten, en coarse-grained PGAs worden gekenmerkt door hun relatief lange verwerkingstijd van een deme, ligt het ook voor de hand dat ze vaak op multicomputers worden geïmplementeerd.

De topologie is een belangrijke factor in de prestaties van een PGA, omdat de keuze hiervan bepaalt hoe snel of langzaam goede oplossingen zich verspreiden naar de andere deelpopulaties [CANTU97]. Als de knopen sterk met elkaar verbonden zijn – d.w.z. de topologie heeft een hoge gemiddelde graad – of de gekozen topologie heeft een lage diameter (of allebei), dan zullen goede oplossingen zich verspreiden naar alle demes en de gehele populatie overnemen. Als het omgekeerde waar is (lage gemiddelde graad en/of hoge diameter), dan zullen strings zich minder snel verspreiden en zijn de deelpopulaties geïsoleerd ten opzichte van elkaar, wat tot gevolg heeft dat verschillende oplossingen zich kunnen ontwikkelen. We zagen in 3.2.2 dat dit speciation genoemd werd.

Uit een studie die onderzoek naar dit onderwerp verrichtte, bleek dat PGAs met een topologie met een hoge gemiddelde graad minder functie-evaluaties vereiste dan waarbij de gemiddelde graad lager was. Er werd gekeken naar de complete topologie, een 4D hyperkubus, een torus en naar ringen waarin verkeer in een of twee richtingen was toegestaan (zie Afbeelding 7).

Daarnaast moet er rekening worden gehouden met migratie. Als individuen naar een willekeurige deme kunnen migreren, spreken we van het *island model*. Als alleen aangrenzende demes ('buren') in aanmerking komen, wordt gebruik gemaakt van het *stepping stone model* [NOWOS99]. De kosten van migratie moeten bij de keuze van een topologie ook worden meegenomen. Des te meer en vaker je individuen laat migreren, des te hoger de communicatiekosten.

In [CANTU97] wordt notie gemaakt van statische en dynamische topologieën. Bij PGAs met statische topologieën maken onderzoekers vaak gebruik van de topologie waarin de computers zijn georganiseerd, zoals hypercubes en ringen. Bij een dynamische organisatie wordt migratie vanuit een deme niet beperkt tot een vast aantal demes, maar worden de migranten gestuurd naar demes die aan bepaalde criteria voldoen. Het doel hiervan is om ze ergens naartoe te sturen waar ze ook effect hebben. Een voorbeeld van zo'n criterium is de diversiteit van de populatie.

3.9 Fine-grained parallelisme

Fine-grained PGAs zijn min of meer tegenovergesteld aan coarse-grained PGAs. Een groter aantal processors is vereist omdat de populatie verdeeld is in een groot aantal kleine demes. Ze hebben één globale populatie die ruimtelijk gestructureerd is, waardoor de interactie tussen de individuen wordt beperkt door de structuur. Interactie tussen de deelpopulaties geschiedt door middel van een migratie-operator (zie 3.7.2) of het gebruik van overlappende demes (zie 3.5) [NOWOS99].

Als migratie gebruikt wordt, zal het migration interval hoog liggen zodat communicatie frequent plaatsvindt. Er zit immers maar een klein aantal strings (of maar één string) in een deme, zodat zich daarin weinig ontwikkeling kan voordoen.

Als deelpopulaties overlappen is er sprake van een *diffusion* model [ALBA02a]. Elk subalgoritme wacht totdat zijn burens klaar zijn (er is dus sprake van synchrone communicatie), waarna een deel van de populatie vervangen kan worden. Selectie en voortplanting wordt beperkt tot een bepaald gebied (*neighborhood*), waardoor deze stappen in het algoritme van 3.1 parallel kunnen plaatsvinden.

3.9.1 Onderzoek

In [CANTU97] worden een aantal interessante onderzoeksresultaten besproken die we nu de revue laten passeren. Een aantal papers hebben betrekking op de topologie, andere vergelijken coarse- en fine-grained PGAs met elkaar.

Manderick en Spiessens implementeerden een fine-grained PGA met een gedistribueerde populatie op een 2D-grid. Selectie en voortplanting was alleen lokaal mogelijk. Geobserveerd werd dat de prestaties van het algoritme afnamen naarmate de grootte van de *neighborhood* toenam. Als de *neighborhood* groot was genoeg was, kwam het parallel GA overeen met een enkele ongestructureerde populatie.

Shapiro en Naveta gebruikten een fine-grained PGA voor het voorspellen van de secundaire structuur van RNA. Er werd een topologie gebruikt waarbij een processor met acht andere verbonden was. Verschillende logische topologieën werden getest: alle acht dichtstbijzijnde burens, de vier dichtstbijzijnde burens en alle burens binnen een bepaalde afstand d . De resultaten voor $d > 2$ waren het slechtst en het vierburens-schema was beter dan de topologie met acht burens.

Verschillende onderzoeken vergelijken fine- en coarse-grained PGAs, maar er is geen eenduidig antwoord op de vraag welke van de twee beter presteert. Het probleem is dat de vergelijking afhangt van onze criteria: willen we bijv. de “wall clock time” minimaliseren of willen we de beste oplossing opleveren? Welk algoritme het beste gekozen kan worden is dus afhankelijk van de situatie.

3.9.2 Topologieën

We zagen in Afbeelding 8 dat fine-grained PGAs een combinatie zijn van statische overlappende deelpopulaties zonder migratie en grootschalige parallelle genetische algoritmes. Bij deze klassen wordt gekozen voor een grid. Verder blijkt uit het bovenstaande onderzoek dat het bovendien een goed presterende topologie is. Bij fine-grained PGAs wordt dus bijna altijd voor een grid/mesh (zie Afbeelding 7(e)) als topologie gekozen. De computerarchitectuur wil daarentegen nog wel eens verschillen.

In [ALBA02a] wordt gezegd dat fine grain PGAs eenvoudig te paralleliseren zijn op SIMD-computers. Vermoedelijk wordt verwezen naar array processors, die gebruikmaken van een groot aantal identieke processors die dezelfde reeks instructies op verschillende data uitvoeren. Dit type computer maakt vaak gebruik van een rechthoekig grid omdat dit goed past bij tweedimensionale problemen die te maken hebben met matrices of afbeeldingen. Daarnaast is dit ontwerp schaalbaar omdat automatisch meer bandbreedte beschikbaar komt als er processors worden toegevoegd [TANEN99].

In [NOWOS99] en [CANTU97] wordt gesproken over de verwantschap tussen fine-grained PGAs en grootschalige parallelle genetische algoritmes (zie 3.5). De laatstgenoemde soort wordt vaak geïmplementeerd op MPP-systemen (zie Afbeelding 3), en valt dus in de MIMD-categorie. Het kan ook zijn dat het gebruik verschoven is van SIMD naar MIMD: in

[TANEN99] lezen we dat MPPs de rol van vector supercomputers en array processors hebben overgenomen, althans in “de top van de digitale voedselketen”. Het model kan ook worden gesimuleerd op een cluster of workstations (zie 3.3.1), maar in dit geval neemt dat hoge communicatiekosten met zich mee omdat er erg vaak informatie wordt uitgewisseld.

3.10 Superlineariteit van PGAs

In verschillende onderzoeken wordt geclaimd dat parallelle genetische algoritmes zouden zorgen voor een meer dan lineaire versnelling ten opzichte van traditionele GAs. Dit verschijnsel wordt superlineariteit (*super-linearity*) genoemd. Tot op heden lijkt er geen consensus te zijn over hoe superlineariteit van PGAs gedefinieerd moet worden. In [KONFR04] wordt geprobeerd de claims van superlineariteit te weerleggen. Daarentegen wordt in [ALBA02b] beweerd dat PGAs wel een superlineaire versnelling kunnen bereiken. We gaan bekijken hoe in beide artikelen de versnelling en efficiëntie van (parallelle) genetische algoritmes wordt gedefinieerd (3.10.1). Daarnaast geven we een algemene analyse van parallelle algoritmen (3.10.2). Vervolgens bespreken we onder welke omstandigheden een superlineaire versnelling al dan niet mogelijk is (3.10.3), en welke claims gerechtvaardigd zijn en waarom (3.10.4).

3.10.1 Definities

Om vast te kunnen stellen of bij een algoritme sprake is van superlineariteit zullen we eerst moeten vastleggen wanneer er sprake van is.

We beschrijven daarom achtereenvolgens twee definities van superlineariteit voor PGAs, de eerste komt uit [KONFR04], de ander uit [ALBA02b]. Vervolgens vergelijken we ze en maken we een beargumenteerde keuze hiertussen.

3.10.1.1 Definitie Konfršt

Stel $T(N, p)$ is de tijd die nodig is voor p processors om een probleem van grootte N op te lossen. $T(N, 1)$ is de scherpste bovengrens voor de looptijd van het snelste sequentiële algoritme. De bovengrens voor de rekentijd noemen we $SU(N)$, waarbij $T(N, 1) \sim SU(N)$.

De versnelling S van een parallel algoritme $S(N, p)$ is als volgt gedefinieerd:

$$(3.1) \quad S(N, p) = \frac{SU(N)}{T(N, p)}.$$

Als $S(N, p) < p$, dan is er sprake van een minder dan lineaire versnelling (*sub-linear speedup*). Als $S(N, p) = p$ of $S(N, p) > p$, dan heeft het algoritme een lineaire resp. superlineaire versnelling. De versnelling kan niet hoger zijn dan lineair, want anders kunnen we een nieuwe bovengrens $SU'(N)$ verzinnen (door de p processors sequentieel te laten werken), met $p \cdot T(N, p) \leq SU'(N)$ en $SU'(N) < SU(N)$, wat in strijd is met de definitie van de bovengrens.

Efficiëntie E is gedefinieerd als de versnelling per processor:

$$(3.2) \quad E(N, p) = \frac{S(N, p)}{p}.$$

De efficiëntie houdt dus direct verband met de versnelling. We zien dat als $E(N, p) \leq 1$ dan $S(N, p) \leq p$, en als $E(N, p) > 1$ dan $S(N, p) > p$.

Nu gaan we een definitie opstellen voor (P)GAs, om uitspraken te kunnen doen over de versnelling ervan. Allereerst wederom een aantal definities:

- p is het aantal processors;
- δ is de topologie van de deelpopulaties (zie 3.8.2 en 3.9.2);
- n_d is de grootte van de deelpopulaties (*deme size*);
- N is de populatiegrootte bij het GA;
- N' is de populatiegrootte bij het PGA, met $N' = \sum_{i=1}^r n_d$ (met r het aantal demes);
- $Q(N, 1)$ is de kwaliteit van de oplossing (*solution quality*) van het GA;
- $Q'(N', p)$ is de kwaliteit van de oplossing van het PGA;
- ξ is de frequentie waarmee $Q(N, 1)$ behaald wordt;
- ξ' is de frequentie waarmee $Q'(N', p)$ behaald wordt;
- $f(\cdot)$ is de te optimaliseren functie.

De kwaliteiten Q en Q' geven aan hoe dicht de oplossing tot de optimale oplossing is genaderd ($\max[f(\cdot)]$ of $\min[f(\cdot)]$). De optimale oplossing weet je niet, dus Q zul je op de een of andere manier moeten vastleggen. Hier zegt Konfrst niets over, maar je zou bijv. naar het individu met de hoogste fitnesswaarde kunnen kijken. Ook legt hij niet uit wat hij met frequentie bedoelt. Vermoedelijk houdt hij rekening met het feit dat GAs probabilistisch zijn en niet altijd binnen een gegeven tijd een oplossing (van kwaliteit Q) leveren. Als vier van de vijf keer een oplossing wordt opgeleverd, is de frequentie 80%.

Dan zijn er nog de populatiegroottes N en N' . Cantú-Paz toonde aan dat een parallel genetisch algoritme een oplossing van dezelfde kwaliteit kan bereiken als een GA, maar met een kleiner aantal individuen, afhankelijk van de topologie δ van het PGA. Anders gezegd: bij dezelfde populatiegroottes bereikt een PGA optimale oplossingen sneller dan een GA. In [CANTU99] wordt dit uitgebreid behandeld.

Om een GA te kunnen vergelijken met een PGA, moeten zowel de frequentie als de kwaliteit van de oplossing van de verschillende algoritmes aan elkaar gelijk zijn, als volgt:

$$\xi = \xi'; \quad Q(N, 1) = Q'(N', p).$$

De taken moeten uitgevoerd worden onder identieke omstandigheden: dezelfde hardware, topologie, optimalisatiefunctie $f(\cdot)$, etc. Daarnaast moeten de metingen nauwkeurig zijn. GAs zijn probabilistisch, wat inhoudt dat een oplossing vinden voor een gegeven probleem een variërende hoeveelheid tijd in beslag neemt. Meetfouten leiden ook tot ongewenste afwijkingen. Daarom moeten de tests, onafhankelijk van elkaar, vaker dan honderd keer⁷ worden gedraaid, zodat daarna gemiddeld kan worden over de gemeten tijden. De gemiddelde rekentijden $\bar{T}(N, 1)$ en $\bar{T}(N, p)$ kunnen we als volgt berekenen:

$$\bar{T}(N, 1) = \frac{\sum_{i=1}^j T_i(N, 1)}{j}; \quad \bar{T}(N, p) = \frac{\sum_{i=1}^j T_i(N', p)}{j},$$

met $j > 100$.

⁷ Het aantal honderd lijkt arbitrair gekozen: er is geen achterliggende reden of theorie die dit verantwoordt.

De versnelling van een parallel genetisch algoritme is:

$$(3.3) \quad S(N', p) = \frac{\bar{T}(N, 1)}{\bar{T}(N', p)},$$

waarbij $N > N'$, $Q(N', 1) = Q'(N', p)$ en $\zeta = \zeta'$.

De efficiëntie van een parallel genetisch algoritme wordt dan:

$$E(N', p) = \frac{S(N', p)}{p},$$

met $N' < N$.

3.10.1.2 Definitie Alba

In [ALBA02b] wordt begonnen met een overzicht waarin op verschillende manieren de versnelling kan worden gemeten (zie Tabel 5).

Type I (*strong speedup*) geeft aan dat de looptijd van het PGA⁸ vergeleken moet worden met die van het beste sequentiële algoritme dat bekend is. Dit algoritme hoeft niet per se een GA te zijn. Omdat het een lastige taak om het efficiëntste algoritme te vinden voor een bepaald probleem, wordt deze manier van meten van de versnelling bijna niet gebruikt.

In plaats daarvan gebruiken onderzoekers vaak hun eigen sequentiële versie van hun algoritme (type II, *weak speedup*). We zouden het algoritme kunnen draaien met een van tevoren vastgestelde stopconditie (type II.B), bijv. een maximum aantal evaluaties, maar een vast aantal stappen is geen geschikte maat, omdat het resultaat afhankelijk is van het machinemodel.

Om versnelling goed vast te kunnen stellen, moet de kwaliteit van de oplossing gedefinieerd zijn voor zowel het sequentiële als het parallelle genetische algoritme, zodat we de looptijden kunnen vergelijken (type II.A).

We zouden het PGA dat draait op m processors kunnen vergelijken met een enkele processor die een *panmictic* (globale, ongestructureerde) populatie gebruikt (type II.A.1). Alba verwijst naar [PUNCH98], waarin gezegd wordt dat dezelfde hoeveelheid werk gedaan moet worden, dus dat uni- en multi-processor implementaties aan elkaar gelijk moeten zijn. Aangezien een lager aantal functie-evaluaties nodig is bij een PGA dan een GA (zie 3.10.1.1), wordt niet hetzelfde werk verzet. Het lijkt daarom beter een maat te definiëren waarbij hetzelfde algoritme op 1 en op m processors wordt gedraaid en de code identiek is (type II.A.2). De versnelling s_m van een PGA is:

$$(3.4) \quad s_m = \frac{E[T_1]}{E[T_m]},$$

waarbij $E[T_1]$ de gemiddelde rekentijd op een enkele processor is en $E[T_m]$ de gemiddelde rekentijd op m processors.

Met deze definitie kunnen we onderscheid maken tussen sub-linear speedup ($s_m < m$), linear speedup ($s_m = m$), en super-linear speedup ($s_m > m$). Efficiëntie e_m wordt dan:

$$e_m = \frac{s_m}{m} \times 100\%.$$

⁸ Hoewel in [ALBA02b] van PEAs (*Parallel Evolutionary Algorithms*) wordt gesproken, zijn deze haast niet te onderscheiden van PGAs. Bovendien is zijn beschrijving van een PEA praktisch gelijk is aan die van een PGA, zoals gegeven in 3.1 en afkomstig uit [ALBA02a].

Om betrouwbare metingen te verrichten moeten gemiddelde en niet absolute tijden worden gebruikt vanwege de non-deterministische aard van PGAs. Ten tweede moeten de implementaties voor een systeem met één processor en die met meerdere processors exact hetzelfde zijn, zodat we speedup measure type II.A.1 vermijden. Ten derde moet een oplossing voor het probleem worden gevonden om van een betekenisvolle versnelling te kunnen spreken. Deze eisen zorgen ervoor dat versnelling een bruikbare waarde geeft bij het bepalen van het voordeel van het toevoegen van meer processors.

-
- I. Strong speedup
 - II. Weak speedup
 - A. Speedup with solution-stop
 - 1. Versus panmixia
 - 2. Orthodox
 - B. Speedup with predefined effort
-

Tabel 5. Taxonomie van *speedup measures* naar [ALBA02b].

3.10.1.3 Vergelijking definities

Nu we beide definities hebben beschouwd, zetten we de overeenkomsten en verschillen op een rij.

Konfršt en Alba vergelijken algoritmes allebei op basis van de kwaliteit van de oplossing. Daarnaast zijn ze het erover eens dat resultaten gemiddeld moeten worden over meerdere runs, aangezien genetische algoritmes non-deterministisch zijn.

Een verschil is de manier waarop omgegaan wordt met een oplossing. Alba schrijft dat voor een betekenisvolle versnelling überhaupt een oplossing voor het probleem gevonden moet worden. Konfršt gaat daar anders mee om door te kiezen voor een frequentie waarmee een bepaalde oplossingskwaliteit wordt bereikt (zie vergelijking 3.3). Het lijkt het erop dat hij niet in alle gevallen een oplossing verwacht. Omdat Alba, in tegenstelling tot Konfršt, zijn bewering kan onderbouwen met een verwijzing naar een onderzoeksresultaat, nemen we die van hem serieuzer. Bovendien wordt in [KONFR04] niet duidelijk beschreven wat met frequentie bedoeld wordt.

Een tweede verschil is dat met Konfršts definitie een GA met een PGA vergeleken wordt, terwijl die van Alba PGAs met elkaar vergelijkt. In het eerste geval kan dit problemen geven volgens [KONFR04]: in sommige gevallen bestaat er geen sequentieel algoritme, bijv. bij het fine-grained PGA, waarbij elk individu verbonden is met een processor. De individuen wisselen informatie uit met hun burens. Maar hoe construeren we een sequentiële versie ervan, hoe meten we de tijd, en hoe bewijzen we dat het algoritme equivalent is met de parallelle versie? Als we deze vragen niet kunnen beantwoorden, kunnen we geen betrouwbare prestatievergelijking uitvoeren, zo zegt Konfršt.

We kunnen ons afvragen hoe we überhaupt een sequentiële versie van een PGA kunnen construeren, en hoe we bewijzen we dat het algoritme equivalent is met de parallelle versie. En wat bedoelt Konfršt met ‘equivalent’? Hij zegt – afgezien van vergelijking 3.3 – dat een GA equivalent is aan een PGA als “dezelfde taken onder dezelfde omstandigheden (identieke hardware, topologie, optimalisatiefunctie)” uitgevoerd worden. Maar wat zijn “dezelfde taken”? We nemen een coarse-grained PGA als voorbeeld, waarbij migratie ook een taak is. Om een equivalent GA te construeren, moet onderscheid gemaakt worden tussen de individuen in de populatie om zo groepen te creëren, maar zonder expliciet deelpopulaties te maken. Deelpopulaties komen immers niet voor in de definitie van een GA (zie 3.10.1.1). Een mogelijkheid is om meerdere oplossingen in één chromosoom te stoppen. Het

chromosoom stelt de deme voor; het aantal oplossingen kunnen we zien als de deme size. De genetische operatoren moeten worden gewijzigd, zodat met deze structuur gewerkt kan worden. Verder moet er een migratiestap aan het algoritme worden toegevoegd, waarin individuen worden uitgewisseld tussen de chromosomen. Ook is er meer administratie vereist, omdat stukje bij beetje aan de opbouw van een chromosoom wordt gewerkt. Je kunt echter een aantal vraagtekens zetten bij deze aanpak:

- Lijkt de beschreven methode nog voldoende op het basisalgoritme in 2.4? Is de migratieoperator niet een wezenlijke wijziging? Worden dezelfde stappen als bij een PGA uitgevoerd?
- Waarom zouden we veel moeite steken in het construeren van een algoritme dat werkt als een PGA, maar het meest lijkt op een GA? Waarom gebruiken we dan niet meteen een PGA, net zoals Alba?

De bovenstaande interpretatie van equivalentie lijkt niet overeen te komen met die van Konfršt, om een tweetal redenen:

1. De populatiegrootte van het sequentiële GA groter moet zijn dan die van het PGA ($N > N'$) in vergelijking 3.3. In het bovenstaande geval moet echter $N = N'$ gelden, omdat (impliciet) van deelpopulaties gebruik gemaakt wordt.
2. De prestatievergelijkingen die in [KONFR04] worden bekeken (zie 3.10.4) implementeren simpelweg een SGA, en proberen niet equivalentie op bovenstaande manier te bewijzen.

Het lijkt er dus op dat Konfršt uitgaat van een SGA (zie 2.4). Maar dan snijdt de opmerking “in sommige gevallen bestaat er geen sequentieel algoritme” geen hout, omdat hij het begrip ‘equivalent’ niet goed definieert. Immers: als je niet kunt zeggen wanneer een PGA dezelfde taken uitvoert als een GA, kun je ook niet beweren dat je geen GA kunt construeren dat equivalent is aan een PGA. Een redelijke aanname is dat onder dezelfde taken dezelfde genetische operatoren, dezelfde vervangingsstrategie en dezelfde optimalisaties wordt verstaan.

Het belangrijkste verschil tussen Alba’s en Konfršts definitie is de mate waarin de algoritmes die vergeleken worden van elkaar verschillen. Alba zegt dat de uni- en multiprocessor implementaties exact hetzelfde moeten zijn, en door een PGA op één processor te simuleren wordt dit zo goed mogelijk nagestreefd. In het geval van Konfršt moet zowel een PGA als een GA worden geconstrueerd. De verschillen tussen de te vergelijken algoritmes zijn groter, met name de manier waarop ze werken.

3.10.1.4 Keuze definitie

In hoofdstuk 4 gaan we een voorbeeldprobleem bespreken en een genetisch algoritme dat dit probleem oplost implementeren. Daar wordt in de praktijk bekeken “welke voordelen het parallel uitvoeren van genetische algoritmes biedt boven het serieel uitvoeren ervan” (zie 1.1). Om in 4.4 uitspraken te doen over deze voordelen, onder andere de versnelling van het algoritme, maken we nu een keuze uit de verschillende definities die we hebben beschouwd.

In [ALBA02b] worden verschillende speedup measures gedefinieerd (zie Tabel 5). Strong speedup is geen praktische methode: hoe weet je zeker dat je het snelste sequentiële algoritme hebt gevonden? Bovendien willen we genetische algoritmes met elkaar vergelijken, en niet een PGA met het beste sequentiële algoritme dat er bestaat. We willen ook geen measure die afhankelijk is van de machine waarop je test (type II.B). Dan blijft type II.A over. Deze zegt dat dezelfde solution quality gedefinieerd moet zijn voor zowel het sequentiële als het parallelle GA. Ook in [KONFR04] wordt geëist dat de frequentie en kwaliteit van de oplossingen aan elkaar gelijk zijn (zie vergelijking 3.3).

We gaan nu de types II.A.1 en II.A.2 nader beschouwen. Het eerste type, II.A.1, vergelijkt een GA met een PGA. Volgens Alba moet voor een ‘eerlijke’ vergelijking dezelfde hoeveelheid werk gedaan worden. Hij kiest daarom voor II.A.2, waarbij een PGA op 1 en op m processors gedraaid, waarna de gemiddelde rekestijden op elkaar worden gedeeld (zie vergelijking 3.4). Konfrst lost dit anders op, namelijk door in vergelijking 3.3 te eisen dat de populatiegrootte van het sequentiële GA groter moet zijn dan die van het PGA ($N > N'$, zie halverwege pagina 39). Het nadeel is dat met benaderingen en vrij complexe formules – beschreven in [CANTU99] – gewerkt moet worden om een schatting voor N en N' te vinden. Daarom, en omdat Alba exact dezelfde algoritmes vergelijkt, kiezen we voor de definitie van Alba: vergelijking 3.4.

3.10.2 Analyse parallele algoritmen

Quinn beschrijft de versnelling (*speedup*) van een parallel algoritme als “de tijd die nodig is voor het uitvoeren van het beste sequentiële algoritme dat probleem P oplost, gedeeld door de tijd die nodig is voor een parallel algoritme dat probleem P oplost”. Superlineaire versnelling treedt op als “de versnelling groter is dan het aantal processors dat wordt gebruikt”. Efficiëntie (*efficiency*) is gedefinieerd als “de verhouding tussen de versnelling en het aantal gebruikte processors” [QUINN87].

Amdahl’s law is een argument tegen het gebruik van parallele processors. In [KONFR04] staat dat Amdahl’s law zegt dat elk algoritme bestaat uit een sequentieel deel, dat door een enkele processor moet worden uitgevoerd, en een parallel deel, dat verdeeld kan worden over meerdere processors voor parallele uitvoering.

Stel dat $T(N, 1) = f_s + f_p = 1$, waarbij f_s en f_p resp. de relatieve fractie van het sequentiële en het parallele deel zijn. Bij gebruik van p processors is de parallele rekestijd

$$T(N, p) = f_s + \frac{f_p}{p},$$

en daarmee wordt de versnelling:

$$S(N, p) = \frac{1}{f_s + \frac{1-f_s}{p}} = \frac{p}{(p-1)f_s + 1}.$$

De efficiëntie is gedefinieerd als:

$$E(N, p) = \frac{1}{(p-1)f_s + 1}.$$

Voor de versnelling kunnen we de limiet berekenen als het aantal processors naar oneindig gaat:

$$\lim_{p \rightarrow \infty} S(N, p) = \lim_{p \rightarrow \infty} \frac{1}{f_s + \frac{1-f_s}{p}} = \frac{\lim_{p \rightarrow \infty} 1}{\lim_{p \rightarrow \infty} f_s + \frac{1-f_s}{p}} = \frac{1}{f_s}.$$

In [KONFR04] wordt (ten onrechte) beweerd dat de bovenstaande limiet naar 0 toe gaat.

Voor de efficiëntie kunnen we dezelfde limiet berekenen:

$$\lim_{p \rightarrow \infty} E(N, p) = \lim_{p \rightarrow \infty} \frac{1}{(p-1)f_s + 1} = \frac{\lim_{p \rightarrow \infty} 1}{\lim_{p \rightarrow \infty} (p-1)f_s + 1} = 0.$$

Stel $f_s = 0.05$ en $p = 21$, dan geldt $S(N, 21) = 10.5$ en $E(N, 21) = 0.5$. De maximale versnelling is $\lim_{p \rightarrow \infty} S(N, p) = 1/0.05 = 20$.

We kunnen hieruit concluderen dat er een limiet is aan de versnelling die kan worden bereikt door het toevoegen van extra processors. Bovendien is superlineaire versnelling per definitie onmogelijk.

Met de komst van massively parallel computers zijn er parallelle algoritmen verschenen waarbij het sequentiële rekendeel niet of nauwelijks een rol speelt (dus f_s ligt dicht bij nul). Om de versnelling in dit geval beter te kunnen beschrijven heeft Gustafson een nieuwe wetmatigheid beschreven, die – verrassend genoeg – door het leven gaat als “Gustafson’s law”. Deze wet zegt dat de versnelling onbeperkt is, en de efficiëntie naar 1 gaat als het aantal processors naar oneindig gaat. Op computers waarbij f_s klein is kunnen we een bijna lineaire versnelling met het aantal processors verwachten. Er is dus geen limiet meer aan de versnelling (althans bij een schaalbaar ontwerp, zie 3.3.1), maar superlineariteit is theoretisch nog steeds onmogelijk. Pas in later werk van Gustafson (zie [GUSTA90]) werd ook rekening gehouden met de probleemgrootte, de afstand tot het geheugen en het soort geheugen, waarna geconcludeerd werd dat superlineariteit wel mogelijk is in praktische situaties (zie Afbeelding 12).

3.10.3 Oorzaken van superlineariteit

Op basis van de conclusies die worden getrokken in [KONFR04] en [ALBA02a] omtrent de oorzaken van superlineariteit van PGAs geven we hiervan een overzicht (voor de verandering een keer geen taxonomie).

1. Implementatie

Het algoritme dat op één processor wordt uitgevoerd is op een bepaalde manier inefficiënt. Als het sequentiële algoritme bijvoorbeeld lineaire lijsten voor data gebruikt, is de parallelle versie sneller omdat kortere lijsten gebruikt worden die later worden samengevoegd. Bij toepassing van een binaire boom zou een dergelijke (superlineaire) versnelling niet optreden. Daarnaast daalt de complexiteit van de operaties selectie en cross-over als de populatie wordt verdeeld en in stukken behandeld wordt. Als het sequentiële algoritme een globale panmictic populatie gebruikt, kan de parallelle versie tot een superlineaire versnelling komen omdat het ook kan profiteren van oorzaken 2 en 3 [ALBA02b]. Ook in [KONFR04] wordt gesproken van problemen die efficiënt op te lossen zijn op een parallelle manier, maar extra stappen vereisen om een dergelijke implementatie te simuleren op een machine met één processor.

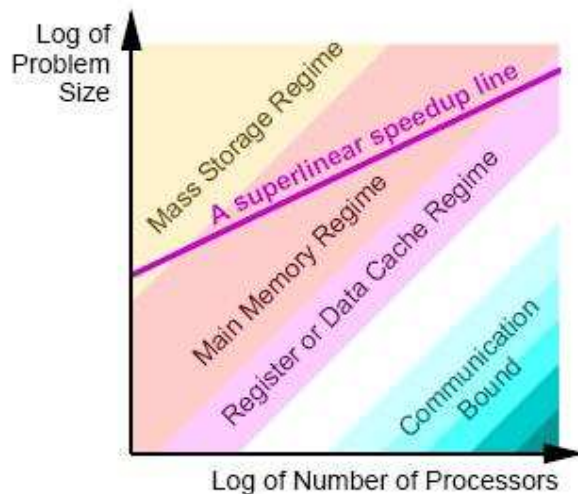
2. Numeriek

Omdat de zoekruimte normaal gesproken erg groot is, kan het lang duren voordat het sequentiële programma een groot deel heeft afgezocht voordat een oplossing wordt gevonden. Zijn parallelle tegenhanger kan daarentegen een andere zoekvolgorde (zie 3.4.3) hanteren waardoor sneller een oplossing gevonden kan worden. Heuristische zoekmethoden, zoals GAs, hebben een kans die groter is dan nul om een oplossing na T seconden te vinden, met $T > 0$. Door een algoritme op meerdere processors in plaats van op een enkele te draaien, vergroten we de kans op het vinden van een oplossing. Dit is

bewezen door Shonkwiler, die ook een numeriek model heeft opgesteld voor het specificeren van de versnelling van PGAs: $s_m = m \cdot a^{m-1}$, met m het aantal processors en superlineaire versnelling als $a > 1$ [ALBA02b].

3. Computerarchitectuur

Als van een sequentiële naar een parallelle computerarchitectuur wordt overgestapt, neemt dikwijls de processorkracht toe. Andere systeembronnen zoals geheugen (in de vorm van massaopslag, RAM-geheugen en cache) kunnen ook lineair toenemen met het aantal processors. Door van deze extra bronnen gebruik te maken kan een parallel algoritme een superlinear speedup verkrijgen [ALBA02b] (zie Afbeelding 12).



Afbeelding 12. Waar de *superlinear speedup line* de grens tussen twee gebieden kruist, kan superlineaire versnelling optreden volgens [GUSTA90].

4. Definitie

Superlineariteit valt of staat met de keuze van de definitie. Als superlineariteit niet of niet goed wordt gedefinieerd, is het claimen ervan gemakkelijk. Definities hebben we besproken in 3.10.1.

Soms wordt bijvoorbeeld niet meegenomen dat PGAs wezenlijk efficiënter zijn dan GAs, en minder functie-evaluaties nodig hebben (zie 3.2.2). We zagen o.a. in 3.10.1.1 dat bij dezelfde populatiegroottes een PGA sneller een optimale oplossing bereikt dan een GA. Volgens [KONFR04] is dit een van de oorzaken van super-linearity. Dat zo een superlineaire versnelling bereikt kan worden, blijkt uit testresultaten van [ALBA02b].

5. Meetmethode

Een vijfde mogelijke oorzaak van (claims van) superlineariteit is het op een onduidelijke manier meten of berekenen van prestaties. Dit hangt samen met de definitiekeuze. Zowel in [KONFR04] als in [ALBA02b] worden er eisen aan de testmethode gesteld. Belangrijk is dat de gemeten resultaten gemiddeld worden over meerdere runs omdat genetische algoritmes non-deterministisch van aard zijn. Dit non-determinisme heeft namelijk tot gevolg dat de rekentijd variabel kan zijn: de ene keer wordt sneller een oplossing gevonden dan de andere keer.

3.10.4 Claims van superlineariteit

In de literatuur zijn vele claims gedaan met betrekking tot de superlineariteit van genetische algoritme. In [KONFR04] worden de resultaten van deze onderzoeken op een rijtje gezet. We zullen daarbij terugverwijzen naar de onderliggende oorzaken, die beschreven zijn in 3.10.3.

Belding onderzocht een coarse-grained PGA met migratie en liet er *Royal Road*-functies op los [BELDI95]. Dit zijn functies die oorspronkelijk ontworpen zijn voor het beantwoorden van vragen rondom de verwerking van schema's [NIMWE99]. Ze testen hoe goed een genetisch algoritme in staat is building blocks te propagieren naar volgende generaties, Hiervoor moet aangegeven worden hoeveel 'bouwstenen' er zijn en wat de gewenste configuratie van elke steen is. In twee van de vier gevallen kon Belding berichten van een superlineaire versnelling. Hij maakte gebruik van een vast aantal functie-evaluaties (type II.B in Tabel 5). Zijn verklaring was dat het programma en de bijbehorende gegevens evenredig verdeeld waren over de (snelle) cachegeheugens van de processors. Bij het sequentiële genetische algoritme, die maar van één processor gebruik maakte, kon slechts een klein deel in de cache staan en het grootste gedeelte in het (langzame) grote geheugen. Dit geval hoort bij oorzaak nummer 3.

Een speciaal type multiple-deme PGA heeft geïsoleerde deelpopulaties, maar kent geen migratie. Dit algoritme gaat door het leven met de naam *Independent and Identical Processing* (IIP), omdat meerdere instanties van hetzelfde sequentiële algoritme onafhankelijk van elkaar op verschillende processors draaien. Alle berekeningen stoppen als een van de instanties een oplossing van een bepaalde kwaliteit heeft gevonden. Vaak is de speedup lineair met het aantal processors.

Shonkwiler bereikte met IIP enorme versnellingen (500 – 600) voor het *inverse fractal problem* (IFP) met 10 tot 20 processors [SHONK93]. Hij liet de algoritmes elk 100.000 iteraties doorlopen (type II.B in Tabel 5). IFP is een *deceptive problem*, wat wil zeggen dat als er goede building blocks ontstaan, dit het genetisch algoritme verwijdert van de objective function (het doel). Voor een exacte definitie, zie [SHONK91]. IIP is bijzonder geschikt voor dit probleem omdat de processen onafhankelijk van elkaar zijn en niet naar hetzelfde suboptimale punt convergeren. IFP is namelijk een probleem waarbij het vinden van een oplossing oneindig lang kan duren. [ALBA02b] verklaart deze versnelling met oorzaak 2, die specifiek op dit probleem slaat.

[KONFR04] houdt er de volgende redenering op na, die meer aansluit bij oorzaak 5. Omdat de looptijd oneindig kan zijn en er maar beperkt tijd beschikbaar is, kunnen maar een klein aantal tests worden uitgevoerd. Hierdoor kunnen grote verschillen ontstaan in de looptijd van de algoritmes, waardoor superlineaire versnelling niet verwonderlijk is. In [SHONK93] wordt inderdaad vermeld dat in 10 van de 24 gevallen er geen oplossing was gevonden na 100.000 iteraties van het algoritme, waardoor de gemiddelde tijd slechts een benadering is. Shonkwiler beweert echter dat de berekende versnelling hierdoor *lager* uitvalt dan de daadwerkelijke versnelling, vermoedelijk omdat hij verwacht dat het verschil tussen de looptijd van het sequentiële en het parallelle algoritme alleen maar zal toenemen.

In [BARAG95] werd bij de coarse-grained implementatie van het PGA een superlineaire versnelling geobserveerd. Hierbij werd het Traveling Salesman Problem (TSP, zie 4.1) als testprobleem gebruikt op een computer met 128 knopen en een hypercube-topologie (zie 3.3.2). Er werd vergeleken op basis van de looptijd en de fitnesswaarden na de evaluatie van 512.000 oplossingen, om zo een vergelijking met de fine-grained implementatie te kunnen maken. Hoe de versnelling berekend wordt, is echter onduidelijk. De superlineaire versnelling was te danken aan een *quick sort* algoritme dat (zover gedocumenteerd) niet

gebruikt werd bij de sequentiële versie. Daarnaast kostte het sorteren minder tijd naarmate het aantal knopen toenam, omdat de lijsten korter werden. Er werden dus twee algoritmes vergeleken, maar ze deden niet hetzelfde werk, aldus [KONFR04]. Dit geval van superlineariteit valt onder oorzaak 1: implementatie.

Bij een ander geval waarbij super-linearity werd geclaimd, was geen sprake van een eerlijke berekening. Het doel van het onderzoek was om de plaatsing van GSM-zendmasten te optimaliseren door het aantal masten te minimaliseren en het totale gebied waarin je met een mobiele telefoon bereik hebt te maximaliseren. Hoewel de resultaten prima waren, is het onduidelijk hoe de tijd $T(N, 1)$ – zie 3.10.1.1 – berekend was. Oorzaak 4 verklaart deze claim.

4 Voorbeeldprobleem

In dit hoofdstuk gaan we een concreet probleem uit de informatica aanpakken. We bekijken het ‘Traveling Salesman Problem’, afgekort TSP. In de literatuur is dit probleem al vaak bestudeerd. Volgens [FOGEL88] is hier een aantal redenen voor:

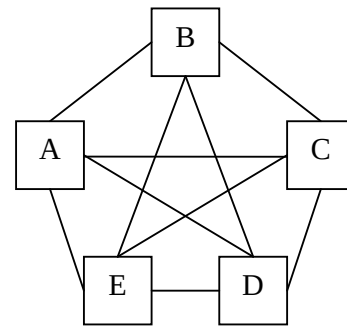
- de oplossing van het probleem is moeilijk, hoewel het probleem zelf eenvoudig te omschrijven is;
- de resultaten zijn toepasbaar in een breed scala van toepassingen;
- het probleem is een soort referentieprobleem (*benchmark problem*) geworden.

Hoewel het een relatief bekend probleem is, geven we toch allereerst een precieze beschrijving (4.1). Het kiezen van een geschikt algoritme voor het TSP komt daarna aan bod (4.2). De codering en de keuze van de genetische operators worden hierin besproken. Daarna komen resp. de implementatie (4.3) en de prestatieanalyse (4.4) aan de beurt.

4.1 Beschrijving

Het Traveling Salesman Problem (in het Nederlands: handelsreizigerprobleem) laat zich vrij eenvoudig formuleren: gegeven een aantal steden en de kosten om tussen twee willekeurige steden te reizen, wat is de goedkoopste route waarbij elke stad een keer wordt bezocht en in het startpunt wordt geëindigd? We kunnen dit ook met graphentheorie opschrijven.

Gegeven een complete ongerichte gewogen graph G zoeken we een Hamilton-circuit met de laagste kosten. Een Hamilton-circuit is een cykel waarbij elke knoop één keer wordt bezocht (met uitzondering van de startknoop). G wordt gerepresenteerd door het paar (K, d) , waarbij K de verzameling knopen is en d de afstandsfunctie die paren knopen afbeeldt op de natuurlijke getallen. Het aantal knopen noemen we N . Omdat de graph compleet is, d.w.z. alle knopen zijn met elkaar verbonden, representeren we de lijnen niet expliciet. In deze graph stellen de knopen de steden voor, de lijnen representeren de wegen en de gewichten staan voor de kosten of afstand van die weg. In



Afbeelding 13. Een voorbeeld van een handelsreizigersprobleem met vijf steden. De afstanden zijn weggelaten.

Afbeelding 13 is een voorbeeld weergegeven, met $K = \{ A, B, C, D, E \}$, $N = 5$ en $d(i, j) = 1$ voor alle i en j uit K . Ter wille van de overzichtelijkheid zijn de afstanden weggelaten. In dit geval hebben alle Hamilton-circuits dezelfde kosten. In het algemeen zullen de afstanden tussen de knopen echter variëren, en willen we $\sum d(i, j)$ minimaliseren, waarbij de paren (i, j) voorkomen in het Hamilton-circuit. Dit is in feite de fitnessfunctie van het genetische algoritme.

4.2 Keuze parallel genetisch algoritme

Nu we het probleem hebben geformuleerd, kunnen we een keuze gaan maken voor een parallel genetisch algoritme. We hebben hiermee twee doelen voor ogen. Ten eerste willen we een algoritme dat het handelsreizigersprobleem op een efficiënte manier aanpakt. Ten tweede zou het interessant zijn het onderzoek van [ALBA02b] te kunnen bevestigen. Onder andere daarom hebben we gekozen voor zijn definitie van superlineariteit (zie 3.10.1.4).

Om het TSP aan te pakken zullen we een speciale codering (4.2.1) met bijpassende cross-overoperator (4.2.3) moeten construeren. Daarnaast bekijken we ook de genetische operatoren selectie (4.2.2) en mutatie (4.2.4). Aan het einde bekijken we welke computerarchitecturen voorhanden zijn in 4.2.5, en welke PGAs hiervoor geschikt zijn (4.2.6).

4.2.1 Codering

4.2.1.1 Path representation

De eenvoudigste manier om een route (een opeenvolging van steden) te coderen is volgens de *path representation* [WILLE04]. Als een reiziger in Afbeelding 13 achtereenvolgens de steden A, B, D, E, C en daarna weer A aandoet, wordt de codering simpelweg: ABDEC. Een voordeel is dat je makkelijk aan de representatie kunt zien wat de route is. Het nadeel van deze methode is dat je niet de standaard cross-overoperator kunt gebruiken. Als we bijv. ABDEC kruisen met DACBE na de tweede locus, dan resulteert dit in ABCBE en DADEC. Hoewel we van tevoren twee correcte Hamilton-circuits hadden, is daar na toepassing van de genetische operator geen sprake meer van. We zullen dus een ander type cross-over moeten verzinnen, of voor een andere representatie kiezen.

4.2.1.2 Ordinal representation

De *ordinal representation* is bedacht door Grefenstette om gebruik te kunnen maken van de simpele cross-overoperator [FOGEL88]. De ordinal representation van de route A, C, E, D, B is 12321. Om deze string te verkrijgen moet een standaardvolgorde van de steden afgesproken worden, bijv. A, B, C, D, E. Vervolgens worden de steden uit de route (in dit geval A, C, E, D, B) één voor één uit de standaardvolgorde gehaald en hun positie genoteerd. A is dus de eerste stad (we noteren een 1) die we verwijderen, waarna de standaardvolgorde wordt: B, C, D, E. C is de tweede stad (we noteren een 2) in de rij, wat B, D, E overlaat. E is de derde stad (we noteren een 3), en B, D blijft over, etc. Uiteindelijk kom je zo dus op 12321 uit. De route is hieruit weer te construeren aan de hand van de representatie en de standaardvolgorde (ga na). Omdat een stad wordt verwijderd uit de standaardvolgorde na gepakt te zijn, voorkómt je dat er dubbele steden vóórkomen in de route. Kortom, kruising leidt zo altijd tot een geldige route.

4.2.1.3 Adjacency representation

De *adjacency representation* is ontworpen voor cross-overoperators die nakomelingen produceren die de meeste wegen van hun ouders overerven [POTVI96]. Deze representatie werkt als volgt. Stad j bezet positie i in het chromosoom als er een weg loopt van stad i naar stad j . Bijvoorbeeld, het chromosoom 38526417 (adjacency) codeert voor 13564287 (path). Stad 3 bezet positie 1, omdat er een weg loopt van 1 naar 3. Op eenzelfde manier zien we dat er een weg van 2 naar 8 loopt.

4.2.2 Selectie

In [POTVI96], [FOGEL88] en [WILLE04] wordt weinig tot geen aandacht besteed aan de keuze van het selectiemechanisme. In [POTVI96] wordt alleen opgemerkt dat in vele implementaties een vorm van selectie wordt gebruikt die de fitnesswaarde van een route op een bepaalde manier transformeert (zoals sigma scaling, zie 2.6.1.2). Waarschijnlijk wordt dit gedaan om premature convergence te voorkomen, waaraan fitness-proportionate selection lijdt (2.6.1.1). Omdat bij een PGA selectie vaak plaatsvindt in een neighborhood (d.w.z. aangrenzende demes), is het handig om onafhankelijk te zijn van de gehele populatie. Tournament selection (2.6.1.4) is hier geschikt voor. Bovendien heeft deze methode een

relatief lage complexiteit. Daarnaast wordt premature convergence tegengegaan door zwakkere individuen ook een kans op voortplanting te geven.

4.2.3 Cross-over

We zagen in 4.2.1 dat voor het handelsreizigersprobleem het van belang is dat de keuze van de cross-overoperator en de representatie op elkaar afgestemd zijn. We bekijken als eerste Hollands versie van cross-over, waarbij de ordinal-codering wordt toegepast. Het merendeel van de operatoren maakt echter gebruik van de path of adjacency representation. We bespreken edge recombination naar [POTVI96], omdat deze operator op een interessante manier wegen van zijn ouders overerft en ook nog redelijke resultaten oplevert. Dan is er nog de evolutionaire aanpak van [FOGEL88], waarbij geen cross-over wordt gebruikt.

4.2.3.1 Hollands cross-over

In 2.6.2 hebben we deze vorm van cross-over besproken. Hollands cross-over beschouwt de route als een chromosoom en elke stad als een gen. Het probleem is dat na kruising de resulterende routes niet op de oorspronkelijke routes lijken. Hierdoor is er geen binding tussen de originele routes en de nieuwe routes. Het idee achter evolutie is juist dat voortgebouwd wordt op eerdere resultaten. De resultaten van Hollands cross-over in combinatie met de ordinal representation bleken in de meeste gevallen niet beter dan een willekeurige zoektocht [FOGEL88].

4.2.3.2 Edge recombination

Edge recombination construeert een nieuwe route door de wegen (edges) van de routes van de ouders te recombineren. Er wordt een zogenaamde ‘edge map’ bijgehouden. Per stad is er een lijst van wegen die voorkomen in een van de routes van de ouders en die leiden naar steden die nog niet voorkomen in de op te leveren route. Als eerste wordt een stad als beginpunt gekozen. Vervolgens wordt gekeken welke stad (die met het beginpunt is verbonden) het minste aantal wegen heeft in de edge map. Als twee of meer steden hetzelfde aantal hebben, dan wordt een van hen willekeurig gekozen. Op deze manier wordt de kans verkleind dat een stad geselecteerd wordt die geen paden heeft naar niet-bezochte steden. In dat geval moet een willekeurige weg worden gekozen, waardoor de route minder zal lijken op zijn ouders. Dit willen we vermijden, zoals we zagen in 4.2.3.1. Daarna wordt de edge map aangepast, en wordt de tweede stap herhaald (waarbij de zojuist gekozen stad als beginpunt wordt beschouwd). De volgorde waarin de steden zijn geselecteerd, bepaalt het resulterende chromosoom. We geven een voorbeeld op basis van [POTVI96].

Stel we hebben de routes 13564287 en 14236578 (volgens de path representation). De initiële edge map is geval (a) van Tabel 6. Bij stad 1 schrijven we 3, 4, 7 en 8 op omdat in de eerste route er wegen lopen van 1 naar 3 en van 7 naar 1 en in de tweede route van 1 naar 4 en 8 naar 1. Dit doen we op een vergelijkbare manier voor de andere steden. In het algemeen is initieel het minimum aantal wegen van een stad in de edge map twee (als in beide routes dezelfde wegen van/naar die stad lopen), en het maximum vier (als in beide routes verschillende wegen van/naar die stad lopen).

We nemen als startpunt stad 1. Als wegen dubbel onderstreept zijn in Tabel 6, dan is de bijbehorende stad geselecteerd. We verkrijgen (b) in de edge map door de wegen te verwijderen die naar 1 leiden, en 1 zelf. Vervolgens bekijken we welke stad (waar een weg vanuit 1 naartoe loopt) het minste aantal wegen heeft. De wegen van de bekeken steden zijn enkel onderstreept in de tabel. Vanuit stad 1 lopen er wegen naar 3, 4, 7 en 8. In dit geval hebben de steden 4, 7 en 8 een gelijk aantal wegen; stad 8 wordt willekeurig geselecteerd. We zijn nu aanbeland bij geval (c). Van (b) weten we dat stad 8 wegen heeft lopen naar 2 en 7.

Stad 7 heeft het minste aantal wegen, en wordt geselecteerd. Deze procedure herhaalt zich totdat er geen steden meer geselecteerd kunnen worden. Uit Tabel 6 is af te lezen dat het resultaat 18756423 is.

Stad	1	2	3	4	5	6	7	8
(a)	<u>3 4 7 8</u>	3 4 8	1 2 5 6	1 2 6	3 6 7	3 4 5	1 5 8	1 2 7
(b)	-	3 4 8	<u>2 5 6</u>	<u>2 6</u>	3 6 7	3 4 5	<u>5 8</u>	<u>2 7</u>
(c)	-	<u>3 4</u>	2 5 6	2 6	3 6 7	3 4 5	<u>5</u>	-
(d)	-	3 4	2 5 6	2 6	<u>3 6</u>	3 4 5	-	-
(e)	-	3 4	<u>2 6</u>	2 6	-	<u>3 4</u>	-	-
(f)	-	3 4	<u>2</u>	<u>2</u>	-	-	-	-
(g)	-	<u>3</u>	2	-	-	-	-	-
(h)	-	-	=	-	-	-	-	-

Tabel 6. Edge recombination van de routes 13564287 en 14236578. Stad 1 is het begin- en eindpunt. Het resultaat van de kruising is 18756423.

Hoewel we in dit voorbeeld gebruik hebben gemaakt van de path representation, is de adjacency representation specifiek ontworpen voor operatoren zoals edge recombination (zie 4.2.1.3).

4.2.3.3 Geen cross-over

De evolutionaire aanpak van [FOGEL88] maakt geen gebruik van cross-over. Wel wordt een vorm van swap mutation toegepast (zie 4.2.4.1). Daarnaast wordt de nadruk sterker gelegd op de fitnesswaarden van de gevonden (tussen)oplossingen door competitie in de populatie te versterken. Iedere route moet het opnemen tegen 10% van de overige routes. De kans om te overleven neemt toe naarmate de route korter is ten opzichte van die van de 'tegenstander'. Iedere 5000 generaties neemt de populatie met een enkele route af, om te simuleren dat in de natuur de voedselbronnen naar verloop van tijd ook verminderen.

Fogel vergeleek zijn aanpak met de *partially-mapped crossover* (PMX), bedacht door Goldberg en Lingle. Er werd een probleem aan beide algoritmes voorgelegd ter grootte van 100 steden. De steden werden willekeurig verdeeld en 30 runs werden uitgevoerd. Na elke run werden de steden op een andere manier gerangschikt en elk algoritme mocht 20.000 nakomelingen produceren. Fogels algoritme had een gemiddelde routelengte van 2667,3 eenheden gevonden, ten opzichte van 4319,5 eenheden van PMX. Het verschil werd veroorzaakt doordat de PMX-operator te veel swap mutations op de route uitvoerde (gemiddeld meer dan 33 verwisselingen per mutatie). Hierdoor leken de resulterende routes na kruising niet (voldoende) op de oorspronkelijke routes. Dat dit tot matige resultaten leidt, zagen we reeds in 4.2.3.1.

4.2.4 Mutatie

Mutaties voor het TSP zijn gericht op het willekeurig genereren van nieuwe volgordes van steden in een route. Hieronder sommen we ze op.

4.2.4.1 Swap mutation

Swap mutation verwisselt twee steden willekeurig. Verwisseling kan ook beperkt worden tot twee verbonden steden. Deze operator lijkt het meeste op de originele mutatie-operator, omdat slechts een kleine wijziging aan de route wordt aangebracht [POTVI96]. Fogel maakte in [FOGEL88] gebruik van een variant op swap mutation. Hij kiest een stad en plaatst hem op een willekeurige maar verschillende positie in de route terug.

4.2.4.2 Inverse mutation

In [WILLE04] wordt *inverse mutation* genoemd. Deze mutatie-operator draait de volgorde van een willekeurig gekozen deel van de route om.

4.2.4.3 Scramble

Bij *scramble* worden twee willekeurige posities op de route uitgekozen, waarna de tussenliggende steden, wederom willekeurig, met elkaar worden verwisseld [POTVI96]. Het nadeel van deze operator is dat als de afstand tussen de posities groot is, de steden flink door elkaar worden gehusseld. Hierdoor heeft de route na mutatie weinig meer gemeen met de oorspronkelijke, met alle gevolgen van dien (zie 4.2.3.1).

4.2.4.4 Local hill-climbing

Local hill-climbing is erop gericht om lokale optima te vinden. Volgens [POTVI96] worden heuristieken, zoals *2-opt* en *3-opt*, waarbij wegen worden verlegd, vaak toegepast. Deze heuristiek wordt een gegeven aantal keer herhaald, of herhaald totdat een optimum is bereikt. In de conclusie wordt gezegd dat de combinatie van kruising en local hill-climbing noodzakelijk is voor het slagen van een genetisch algoritme. Het idee achter *2-opt* als volgt. Na crossover snijden wegen elkaar in de route. Door het aantal kruisende paden te verminderen, kan een kortere route kan worden geconstrueerd. Een uitgebreidere bespreking van *2-opt* is te vinden in [HENDR91].

4.2.5 **Computerarchitectuur**

In 3.3 hebben we een overzicht gezien van parallelle computerarchitecturen. Binnen de universiteit zijn een tweetal architecturen beschikbaar. Allereerst zijn er Sun Blade 100 workstations beschikbaar. Ten tweede is er de Sun Enterprise 10000 (E10K), ook wel ‘tengrand’ genoemd. We zullen beide opties bespreken en hun geschiktheid bespreken.

4.2.5.1 Sun workstations

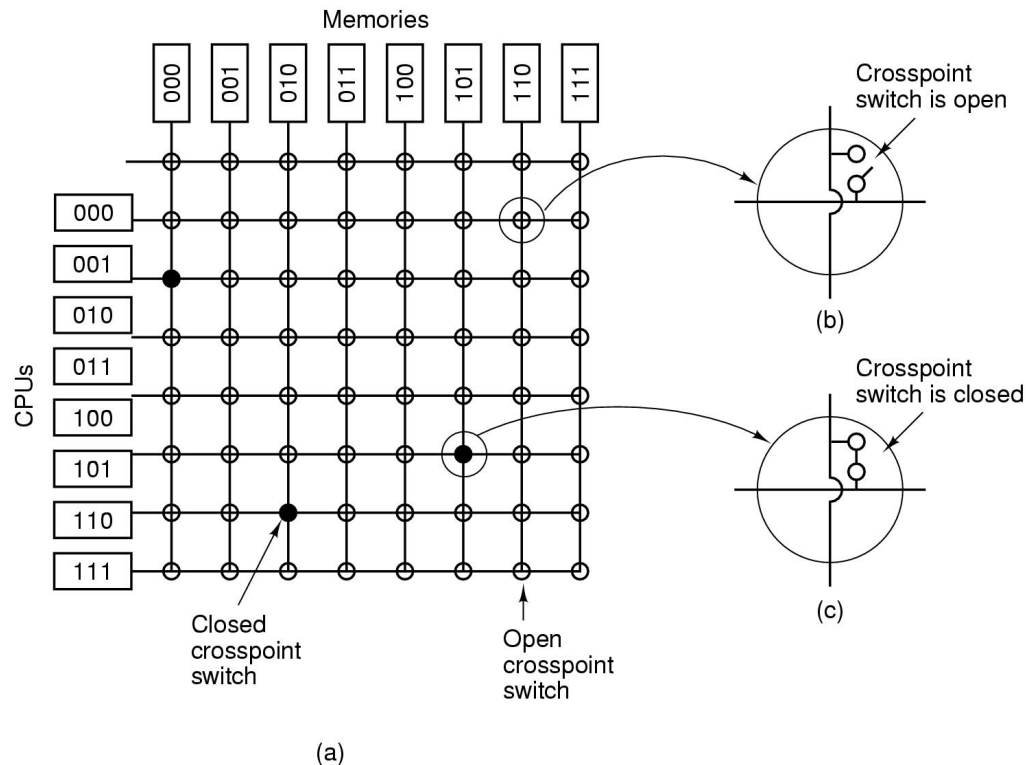
In terminalkamer 8 zijn 20 Sun Blade 100 workstations beschikbaar. Deze hebben elk een UltraSPARC-processor die op 500 MHz draait, 256 KB aan cache- en 256 MB aan SDRAM-geheugen. Het besturingssysteem is Unix Solaris 8. De computers zijn onderling verbonden middels netwerkkabels en 100Mbit-switches in een ster-vorm, zoals in Afbeelding 7(a). Omdat er meerdere switches zijn, zijn er meerdere sterren die aan elkaar gekoppeld zijn. Deze computerarchitectuur is een cluster of workstations, en behoort daarmee tot de categorie COW (zie Afbeelding 3).

4.2.5.2 Tengrand

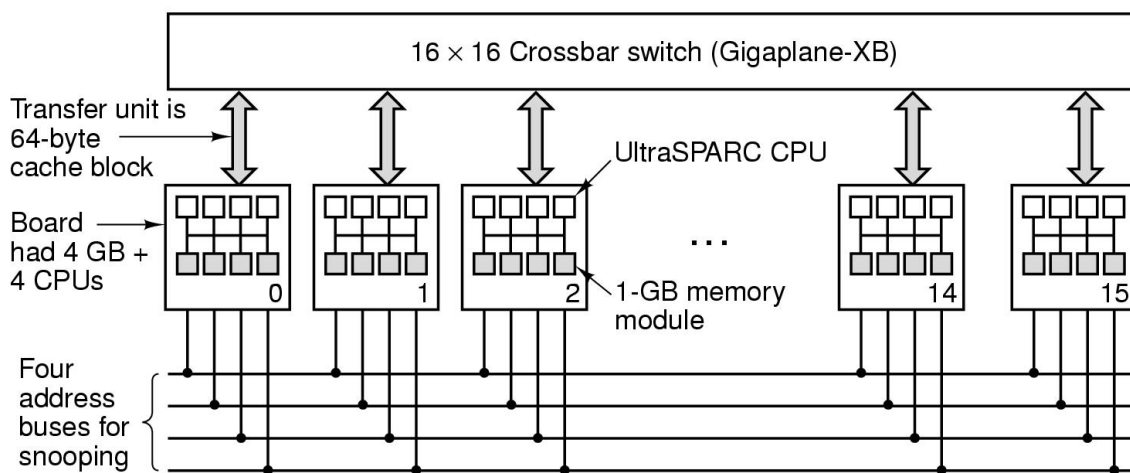
De tengrand is een krachtige server, bedoeld voor intensieve berekeningen. De machine bevat 36 UltraSPARC-processors die lopen op 333 MHz, en 36 GB aan geheugen. In [TANEN99] worden de Sun Enterprise 10000 in detail besproken. Wat betreft de topologie: de processors en geheugenmodules zijn verbonden via een zogenaamde *crossbar switch* (zie Afbeelding 14). In (a) is er een gelijktijdige verbinding tussen de (processor, geheugen) paren (001, 000), (101, 101) en (110, 010).

In de taxonomie in Afbeelding 3 is de tengrand in te delen als een switched UMA multiprocessor. Switched omdat hij van een crossbar switch gebruikmaakt, UMA (Uniform Memory Access) omdat elke processor dezelfde toegangstijd heeft tot elke geheugenmodule, en multiprocessor omdat er sprake is van een gedeeld geheugen. De identieke toegangstijd is te danken aan de snelle crossbar switch, Gigaplane-XB genaamd, en strenge ‘timing requirements’. In Afbeelding 15 is de Gigaplane-XB bovenin weergegeven, die de gegevens voor de geheugens en caches verplaatst. Er zijn 16 borden die elk plaats bieden aan 4

processors en 4 GB aan geheugen. Elke processor beschikt over een secondary cache van 4 of 8 MB. Deze architectuur is daarmee extra interessant, omdat met een dergelijk grootte cache het verschil tussen 1 processor (met 4/8 MB) of 36 processors (met 144/288 MB) duidelijk merkbaar zou moeten zijn (zie 3.10.3, oorzaak 3). Verder worden er vier buslijnen gebruikt om ervoor te zorgen dat een processor niet data uit zijn cache gebruikt die elders is gewijzigd. Kortom, deze *snooping caches* zorgen ervoor dat de data in de caches consistent blijft.



Afbeelding 14. Afbeelding naar [TANEN99]. (a) Een 8x8 crossbar switch. (b) Een open crosspoint. (c) Een gesloten crosspoint.



Afbeelding 15. Afbeelding naar [TANEN99]. De Sun Enterprise 10000 server. Het systeem ondersteunt maximaal 64 processors en 64 GB aan geheugen.

4.2.6 PGA

4.2.6.1 Sun workstations

Een voor de hand liggende keuze voor deze computerarchitectuur is een coarse-grained PGA (zie 3.8). Coarse-grained PGAs worden niet voor niets ook wel *distributed algorithms* genoemd, omdat ze vaak op distributed memory systems worden geïmplementeerd. Van de meeste nadelen van een dergelijke machine (zie Tabel 4) hebben we gelukkig geen last. Dit type genetisch algoritme gaat namelijk niet uit van een globaal geheugen. Elke processor neemt een deme voor zijn rekening, zodat er 20 deelpopulaties zijn. Alle operaties vinden lokaal plaats, dus er hoeven geen gegevens opgevraagd te worden uit geheugens van andere workstations. Hierdoor wordt er geen nadeel ondervonden van het feit dat de toegang tot geheugen niet in vaste hoeveelheid tijd kan plaatsvinden.

De enige communicatie die tussen de machines noodzakelijk is, is het uitwisselen van individuen. Alle demes kunnen dit met elkaar doen, omdat we het island model toepassen. De parameters van migratie (zie 3.7.2) zullen, naast de gebruikelijke zoals de kans op cross-over, bij de implementatie moeten worden geoptimaliseerd. Belangrijk is dat de communicatie asynchroon plaatsvindt (zie 3.4.3). Op het moment dat een processor klaar is het met een generation moet hij meteen verder kunnen met de volgende. Hiervoor moeten de migrerende individuen aan de zendende of aan de ontvangende kant worden gebufferd, totdat ze gebruikt kunnen worden. Op deze manier wordt de processorkracht optimaal benut.

4.2.6.2 Tengrand

De keuze voor een PGA voor de tengrand is niet eenvoudig. We bekijken als eerste fine-grained parallelisme. Een massively parallel genetic algorithm valt echter af: hiervoor zijn er te weinig processors. De tengrand is tenslotte ook een multiprocessor, en geen MPP-systeem. Bovendien zou elk individu een processor toebedelen (het ideale geval) een veel te kleine populatie tot gevolg hebben.

Een andere mogelijkheid is een coarse-grained PGA. Dit algoritme is echter meer geschikt voor een loosely coupled systeem met een relatief lage bandbreedte (zie 3.8.2). Het zou jammer zijn om het voordeel van een snel en in vaste tijd toegankelijk geheugen niet te benutten. Dit zou deels afgevangen kunnen worden door het migration interval (zie 3.7.2) relatief laag te kiezen, zodat vaak individuen worden uitgewisseld.

Een betere optie is statische overlappende deelpopulaties zonder migratie (zie 3.5). De gehele populatie kan in het gedeelde geheugen geplaatst worden. Verschillende threads kunnen vervolgens parallel op de deelpopulaties uitgevoerd worden. Het is wel belangrijk om een ruimtelijke structuur te kiezen, zodat speciation kan plaatsvinden (zie 3.2.2). Deze structuur is onafhankelijk van de topologie zoals die in de tengrand aanwezig is, omdat al het geheugen een vaste toegangstijd heeft. Omdat dit type PGA sterk lijkt op de coarse-grained versie, kunnen we ook gerelateerd onderzoek gebruiken bij onze keuze. In 3.8.2 zagen we dat een topologie met een hoge gemiddelde graad minder functie-evaluaties vereiste dan een waarbij de gemiddelde graad lager was. Dit zet ons aan tot het kiezen van een double torus of een 4D-hypercube voor de ruimtelijke structuur (zie 3.3.2).

Een ander aspect is synchronisatie. Sommige individuen behoren namelijk tot meerdere deelpopulaties en worden in meerdere selectie- en cross-overoperaties meegenomen. Afspraken over wanneer een thread een individu mag gebruiken zijn dus nodig.

4.3 Implementatie

Het zou een interessante praktijktest zijn om op basis van 4.2 een PGA te implementeren op een van de twee beschreven computerarchitecturen. Daar dit echter een aanmerkelijke hoeveelheid extra tijd zou vergen, is in deze scriptie de voorkeur gegeven aan de theoretische analyse van parallelle genetische algoritmes.

4.4 Prestatieanalyse

4.4.1 Simulatie PGA

In 3.10.1.4 hebben we gekozen voor de definitie die beschreven staat in vergelijking 3.4. Hiervoor is het noodzakelijk een PGA te simuleren op een processor. Dit is mogelijk door een programmeertaal te gebruiken die threads ondersteunt (zoals Java) en voor elke processor in het parallelle geval een thread te creëren. Dit neemt echter overhead met zich mee, omdat tussen threads gewisseld moet worden; dit kan een eerlijke vergelijking in de weg staan. Bovendien worden PGAs vaak geïmplementeerd in talen die (standaard) geen threads ondersteunen, zoals C++. Daarom beschrijven we hieronder een methode die de uitvoering van een PGA op een sequentiële manier aanpakt.

Bij de Sun werkstations was de keus gevallen op een coarse-grained PGA. Om dit algoritme op een machine met één processor te draaien wordt binnen elke deme net zolang generations doorlopen, totdat deze klaar is om individuen te laten migreren naar andere demes. Als de deme waarmee gestart is voor de tweede keer aan de beurt is, worden individuen van andere demes geïntegreerd in de populatie en herhaalt het bovenstaande proces zich.

Voor de tengrand hebben we gekozen voor statische overlappende deelpopulaties zonder migratie, waarbij verschillende threads parallel op de deelpopulaties uitgevoerd worden. Simulatie van dit PGA op een sequentiële computer kan door de (denkbeeldige) threads achtereenvolgens uit te voeren. In het parallelle geval is enige synchronisatie tussen de threads vereist. Om te zorgen dat beide algoritmes exact hetzelfde werk doen, moet dit ook worden gesimuleerd. Er moet rekening mee worden gehouden dat van een bepaalde topologie gebruik wordt gemaakt, die bepaalt welke deelpopulaties met elkaar in overlappende gebieden mogen voorkomen.

4.4.2 Verwachte versnelling

Op basis van de PGAs die we gekozen hebben voor de twee computerarchitecturen, kunnen we een voorspelling doen voor de versnelling. We nemen aan dat we de gehele architectuur kunnen gebruiken. Dit is waarschijnlijk geen realistische aanname, omdat er meerdere gebruikers op de machine zijn. Het is echter niet gezegd dat de voorspellingen niet geldig kunnen zijn als deze aanname niet waar is.

De tengrand is de interessantste architectuur. In deze machine is namelijk veel cachegeheugen aanwezig. Als we een programma maken dat in totaal tussen de 4 – 144 MB of 8 – 288 MB (afhankelijk van de cache per processor) groot is, dan verwachten we een superlineaire versnelling. Stel dat het programma 100 MB groot is. Met 1 processor zou het overgrote deel terecht komen in het RAM-geheugen. Met 36 processors zou alles in het cachegeheugen passen! Dit zou een merkbaar verschil op de prestaties moeten hebben. Hierbij wordt wel aangenomen dat het programma mooi verdeeld is over de cachegeheugens, wat niet per se het geval hoeft te zijn.

De Sun werkstations beschikken in totaal over 5 MB aan cache- en ruim 5 GB aan RAM-geheugen. Om een situatie te creëren die vergelijkbaar is met de bovenstaande, zou het programma tussen de 256 KB en 5 MB groot moeten zijn. Dit zou de grootte van de populatie sterk beperken, waardoor een kleiner probleem aangepakt zou moeten worden. Communicatie verloopt echter trager dan bij de tengrand. Dit kan ertoe leiden dat de processorcapaciteit niet optimaal benut wordt, waardoor de kans op een superlineaire versnelling kleiner wordt.

5 Conclusie

Eerst beantwoorden we de beide deelvragen. Aan de hand hiervan beantwoorden we daarna de hoofdvraag.

Welke genetische algoritmes maken optimaal gebruik van parallelle processors en waarom?
Allereerst bekijken we de ontwikkeling die de parallelisatie van genetische algoritmes heeft doorgemaakt. Daarna beantwoorden we de deelvraag.

De eerste stap richting het geschikt maken van genetische algoritmes voor parallelle processors was de berekening van fitnesswaarde van individuen – de meest tijdrovende stap in het algoritme – uit te besteden aan verschillende processors. Dit heet globale parallelisatie. De versnelling is bijna lineair met het aantal processors, mits de fitnessevaluatie praktisch alle rekentijd in beslag neemt.

De volgende voornaamste stap was het opdelen van de populatie in meerdere deelpopulaties (demes), zogenaamde multiple-deme PGAs. Dit zorgt ervoor dat de demes zich in verschillende richtingen kunnen ontwikkelen, waardoor speciation (het ontstaan van soorten) kan plaatsvinden. Zo worden verschillende optima (pieken in de grafiek) bekeken worden. Dit komt het vinden van een betere oplossing ten goede. Door af en toe individuen uit te wisselen (migratie), kunnen goede oplossingen (individuen met een hoge fitnesswaarde) zich verspreiden over de gehele populatie. Tegelijkertijd zorgt migratie ervoor dat de demes voldoende divers blijven, zodat ze zich verder kunnen ontwikkelen.

Daarna werd dit principe verfijnd en ontstonden onder andere coarse- en fine-grained parallelle genetische algoritmes. Verschillende onderzoeksresultaten lieten zien dat met deze PGAs in sommige gevallen een superlineaire versnelling te bereiken is. Na een analyse van wat superlineariteit betekent te hebben uitgevoerd, bleken er een aantal oorzaken hiervoor te zijn. Sommige claims waren terecht, andere niet. Op basis van de analyse kunnen we de conclusie trekken dat coarse-grained PGAs optimaal gebruik maken van parallelle processors. In het algemeen is dit niet per se de beste keus: het is afhankelijk van de computerarchitectuur welk parallel genetisch algoritme geschikt is. Hier zullen we in het antwoord op de volgende deelvraag ook op ingaan.

Kunnen we een concreet voorbeeld geven van een genetisch algoritme dat significant beter presteert door het parallel uitvoeren ervan?

We hebben verschillende gevallen uit de literatuur besproken waarmee een superlineaire versnelling is bereikt. Daarnaast hebben we voor twee parallelle computerarchitecturen een PGA voorgesteld. Deze voorstellen zijn gebaseerd op het onderzoek naar PGAs en de topologieën die er geschikt voor zijn. Bij de keuze ervan hebben we ervoor gezorgd dat superlineaire versnelling kan optreden, waarbij bewust is gekeken naar de oorzaken die ten grondslag liggen aan superlineariteit van PGAs.

Welke voordelen biedt het parallel boven het serieel uitvoeren van genetische algoritmes en waardoor worden deze veroorzaakt?

Het parallel uitvoeren van genetische algoritmes dwingt af dat we PGAs uitvoeren op een parallelle computerarchitectuur. Dit heeft een aantal voordelen tot gevolg:

- Multiple-deme PGAs bereiken bij dezelfde populatiegroottes sneller optimale oplossingen dan een GA. Zo kan in minder tijd een vergelijkbare oplossing gevonden worden, of in dezelfde tijd een betere oplossing. Het antwoord op de eerste deelvraag verklaart waarom.
- Parallelle architecturen hebben meer processorkracht. Dit komt van pas bij het evalueren van fitnessfuncties, wat de meest tijdrovende bezigheid is. In de literatuur zijn rekestijden van meer dan 1 ‘processorjaar’ bericht voor één run. Het uitvoeren van dergelijke

berekeningen op een parallelle architectuur levert een tijdsbesparing op in de vorm van *wall clock time*.

- Parallelle architecturen hebben meer geheugen (cache, RAM-geheugen, massaopslag). Hierdoor kan de populatie in een sneller type geheugen worden geplaatst, waardoor berekeningen erop sneller kunnen worden uitgevoerd. Dit heeft een tijdsbesparing tot gevolg. Er kan ook voor een grotere populatie worden gekozen om een omvangrijker probleem aan te pakken; dit doet het tijdsvoordeel echter teniet.
- Opgesplitste datastructuren kunnen efficiënter worden verwerkt. Als het sequentiële algoritme bijvoorbeeld lineaire lijsten voor het sorteren data gebruikt, is de parallelle versie sneller omdat kortere lijsten gebruikt worden die later worden samengevoegd. Daarnaast daalt de complexiteit van de operaties selectie en cross-over als de populatie wordt verdeeld en in stukken behandeld wordt.
- Heuristische zoekmethoden, zoals GAs, hebben een kans die groter is dan nul om een oplossing na T seconden te vinden, met $T > 0$. Bij bepaalde problemen, bijv. waarbij het vinden van een oplossing oneindig lang kan duren, vergroten we de kans op het vinden van een oplossing aanzienlijk door een algoritme op meerdere processors in plaats van op een enkele te draaien.

6 Referenties

- [ALBA02a] Enrique Alba en José M. Troya, *Improving flexibility and efficiency by adding parallelism to genetic algorithms*, Statistics and Computing, Volume 12, Issue 2, Apr 2002, pag. 91.
- [ALBA02b] Enrique Alba, *Parallel evolutionary algorithms can achieve super-linear performance*, Information Processing Letters, Elsevier 82(1): 7-13, 2002.
- [BARAG95] R. Baraglia en R. Perego, *Parallel Genetic Algorithms for Hypercube Machines*, Istituto CNUCE, Consiglio Nazionale delle Ricerche, Pisa, Italy, 13, 1995.
- [BARNE05] B. Barney, *Introduction to Parallel Computing*, Livermore Computing, 2005, http://www.llnl.gov/computing/tutorials/parallel_comp/.
- [BELDI95] T. Belding, *The distributed genetic algorithm revisited*, In Eshelman, L., editor, Proceedings of the Sixth International Conference on Genetic Algorithms, pages 114–121, San Francisco, CA, Morgan Kaufmann, 1995.
- [CANTU95] Erick Cantú-Paz, *A Summary of Research on Parallel Genetic Algorithms*, Technical Report 95007, Illinois Genetic Algorithms Laboratory, Department of General Engineering, University of Illinois, Urbana, Illinois, 1995.
- [CANTU97] Erick Cantú-Paz, *A Survey of Parallel Genetic Algorithms*, Technical Report 97003, Illinois Genetic Algorithms Laboratory, Department of General Engineering, University of Illinois, Urbana, Illinois, 1997.
- [CANTU99] Erick Cantú-Paz, *Designing Efficient and Accurate Parallel Genetic Algorithms*, PhD thesis, Graduate College of the University of Illinois at Urbana Champaign, 1999.
- [CHEMI94] Wasel Chemij, *Parallel Computer Taxonomy*, MPhil, Aberystwyth University, 1994, <http://www.gigaflop.demon.co.uk/comp/MPhil.htm>.
- [CRAIN98] T.G. Crainic en M. Toulouse, *Parallel metaheuristics*, Fleet management and logistics (T.G. Crainic and G. Laporte, eds.), 205-251, Kluwer, 1998.
- [FERNA03] Francisco Fernandez, Marco Tomassini en Leonardo Vanneschi, *An empirical study of multipopulation genetic programming*, Genetic Programming and Evolvable Machines, 4(1), pag. 21-51, 2003.
- [FOGEL88] D.B. Fogel, *An Evolutionary Approach to the Traveling Salesman Problem*, Biological Cybernetics, 60, Springer-Verlag, 1988, pag. 139 – 144.
- [GOLDB89] David E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.
- [GUSTA90] John L. Gustafson, *Fixed Time, Tiered Memory, and Superlinear Speedup*,

Proceedings of the Fifth Distributed Memory Computing Conference, 1990,
<http://www.scl.ameslab.gov/Publications/Gus/Superlinear/Superlinear.pdf>.

- [HENDR91] Chris Hendriks, *Genetische algoritmen*, Katholieke Universiteit Nijmegen, Nijmegen, 1991.
- [KONFR04] Zdeněk Konfršt, *On Super-linearity in Parallel Genetic Algorithms*, Proceedings of the 2nd IASTED International Conference on Neural Networks and Computational Intelligence, ACTA Press, USA, 2004, pag. 96-101.
- [KOZA98] J. R. Koza en D. Andre, *A parallel implementation of genetic programming that achieves superlinear performance*, J. Inform. Sci. 106 (1998), 201–218.
- [KRATI01] J. Kratica, D. Tošić, V. Filipovic en I. Ljubic, *Solving the simple plant location problem by genetic algorithms*, RAIRO - Operations Research, 35(1):127-142, 2001.
- [LEIJE04] D.C. van Leijenhorst, *T2 – Complexiteitstheorie: Een beknopte inleiding in 12 voordrachten*, Radboud Universiteit Nijmegen, Nijmegen, 2004, pag. 106.
- [MITCH98] M. Mitchell, *An introduction to genetic algorithms*, MIT Press, Cambridge, MA, 1998.
- [NIMWE99] E. van Nimwegen, J. Crutchfield en M. Mitchell, *Statistical Dynamics of the Royal Road Genetic Algorithm*, Theoretical Computer Science, Vol. 229, No. 1, pag. 41-102, 1999.
- [NOWOS99] M. Nowostawski en R. Poll, *Parallel Genetic Algorithm Taxonomy*, Proceedings of the Third International Conference on Knowledge-based Intelligent Information Engineering Systems KES'99, pages 88-92, IEEE Computer Society, 1999.
- [POTVI96] J. Potvin, *Genetic algorithms for the traveling salesman problem*, Annals of Operations Research 63, pag. 339-370, 1996.
- [PUNCH98] W.F. Punch, *How effective are multiple populations in genetic programming*, Proceedings of the Genetic Programming Conference, Morgan Kaufmann, Los Altos, CA, 1998, pp. 308 – 313.
- [QUINN87] M.J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, Singapore, 1987.
- [SHONK91] R. Shonkwiler, F. Mendivil en A. Deliu, *Genetic Algorithms for the 1-D Fractal Inverse Problem*, Proceedings of the Fourth Intl. Conf. on Genetic Algorithms, San Mateo, CA, pag. 495-501, 1991.
- [SHONK93] R. Shonkwiler, *Parallel Genetic Algorithm*, Proceedings of the Fifth Intl. Conf. on Genetic Algorithms, San Mateo, CA, pag. 199-205, 1993.

- [TANEN99] Andrew S. Tanenbaum, *Structured Computer Organization*, Fourth Edition, Prentice Hall, Upper Saddle River, NJ, 1999.
- [WILLE04] Christiaan Willemsen, *Traveling Saleperson Problem: benadering door genetische en evolutionaire algoritmen*, Seminarium Informatica 2003, KUN Informatica Grondslagen, febr. 2004.