Universität Leipzig

Fakultät für Mathematik und Informatik
Institut für Informatik

Entwicklung, Untersuchung und Implementierung von Parallelen Evolutionären Algorithmen für die Modellpartitionierungskomponente parallelMAP

DIPLOMARBEIT

Leipzig, 10. August 1998

vorgelegt von
Hendrik Schulze

Meinem Vater.

Zusammenfassung

Die vorliegende Arbeit untersucht Möglichkeiten der Parallelisierung von Evolutionären Algorithmen, welche hier zur Partitionierung von Daten für die parallele Logiksimulation benutzt werden. Neben einer allgemeinen Einführung in Grundbegriffe und Methoden von Evolutionären Algorithmen, Parallelverarbeitung, Logiksimulation und Datenpartitionierung wird das im Rahmen dieser Diplomarbeit entwickelte Programmpaket pga vorgestellt, sowie auf die darin benutzten Parallelisierungsmethoden und Kommunikationsstrukturen eingegangen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

1. Einleitung

Wissenschaft erwächst aus dem menschlichen Bestreben, die Welt zu verstehen und zu verändern. So wie die Erfindung der Dampfmaschine die Industrielle Revolution auslöste, hat die Entwicklung des Computers eine Revolution verursacht, welche die Gesellschaft nicht weniger verändern wird, als die erste genannte Umwälzung 150 Jahre zuvor. Bewirkte die erste Industrielle Revolution, daß der Mensch seine physischen Kräfte vergrößern konnte, so ist es die dem Computer geschuldete zweite Industrielle Revolution, welche die geistigen Kräfte des Menschen immer mehr vergrößern wird. Mit Hilfe der Mikroelektronik können Probleme bearbeitet werden, deren Komplexität bei weitem das überschreitet, was ein Mensch allein in seinem Leben bewältigen kann, Informationen können in Sekunden über die gesamte Welt verteilt und immense Datenmengen verwaltet werden.

Diese Revolution bewirkt, daß nicht nur in der Forschung, sondern auch in Bereichen der Dienstleistung, Produktion und des täglichen Lebens jährlich immer mehr Rechenleistung benötigt wird. Für die Hersteller von Mikroprozessoren und anderen Computerschaltkreisen bedeutet dies, in immer kürzerer Zeit immer leistungsfähigere Mikrochips herzustellen. Bei den IBM S/390 Großrechnern vervierfacht sich die Anzahl der integrierten Transistoren mit jeder Prozessorgeneration (aller drei Jahre). Eine kurze Produktfertigungszeit und ein frühes Erkennen und Beheben von Designfehlern ist essentiell für das Überleben einer Firma in diesem hart umkämpften Markt. Darum werden bei allen Herstellern von Mikroprozessoren Simulatoren benutzt, um frühzeitig die korrekte Funktion des zu entwickelnden Schaltkreises zu überprüfen. Solche Simulationen stellen eine große Herausforderung für die vorhandene Hardware dar. Da ein Simulator bis zu 10 Millionen mal langsamer arbeitet als der fertige Prozessor können Simulationen durchaus einige Tage dauern.

Eine Möglichkeit, die Entwicklungszeit zu verkürzen, und somit Kosten zu reduzieren, ist die ***Parallelisierung der Logiksimulation***, indem sich mehrere Prozessoren die anfallende Simulationsarbeit teilen. Die Qualität eines parallelen Programmes hängt vor allem von der geschickten Verteilung der Arbeit auf die einzelnen Prozessoren ab. Nur so kann gewährleistet werden, daß der Verwaltungsaufwand und der Datenaustausch zwischen den Prozessoren die erreichte Beschleunigung nicht wieder aufbraucht. Diese Verteilung, Partitionierung genannt, ist ein wesentlicher Bestandteil bei der Entwicklung und Benutzung paralleler Programme.

Ein Verfahren zur Partitionierung sind ***Evolutionäre Algorithmen***, eine Klasse von Optimierungs- und Suchverfahren, die sich sehr stark an der biologischen Evolution orien-

tieren. Mit Hilfe von Replikation, Variation und Selektion ist es möglich, Optimierungen auf Gebieten durchzuführen, über die man kein konkretes Wissen besitzt. Das Grundprinzip ist sehr einfach. Man verändert und kombiniert bestehende Lösungen zufällig. Nach einem Vergleich zwischen den Ausgangswerten und den neuen Lösungen, verwirft man die schlechteren, und beginnt von vorn. Die Erfolge Evolutionärer Algorithmen zeigen, daß es sich hier um eine Methode handelt, deren Potential bei weitem noch nicht ausgeschöpft ist.

In Zusammenarbeit mit dem IBM FORSCHUNGS- UND ENTWICKLUNGSLABOR BÖBLINGEN wurde an der UNIVERSITÄT LEIPZIG der Logiksimulator TEXSIM parallelisiert. An der UNIVERSITÄT LEIPZIG wird im Rahmen des DFG-Projektes "PARTIONIERUNGSALGORITHMEN FÜR MODELLDATENSTRUKTUREN ZUR PARALLELEN COMPILERGESTEUERTEN LOGIKSIMULATION" [3] an der Entwicklung von optimalen Partitionierungsstrategien gearbeitet.

Diese Diplomarbeit ist in dieses Projekt eingebettet und beschäftigt sich mit dem Einsatz von Parallelen Evolutionären Algorithmen um bestehende Datenpartitionierungen für PARALLELTEXSIM und dessen Weiterentwicklung PARALLELMVLSIM zu optimieren. Mit Hilfe des Programmes pga, das im Rahmen dieser Diplomarbeit entwickelt wurde, ist es möglich, die Simulationszeit von deterministisch erzeugten Partitionen um bis zu 40% zu verbessern. Dazu wurden spezielle genetische Verfahren entwickelt. Um pga für den täglichen Einsatz auf stark belasteten Systemen zu optimieren, ist ein neuartiges Parallelisierungskonzept entwickelt und implementiert worden.

1.1. Danksagung

Diese Arbeit entstand an der UNIVERSITÄT LEIPZIG in Zusammenarbeit mit der IBM FORSCHUNGS- UND ENTWICKLUNGS GMBH in Böblingen.

Ich möchte mich bei meinen Betreuern DR. HERING, DR.HAUPT und HERRN PETRI für das interessante Thema und die gute Betreuung der Arbeit bedanken und bei H.W. ANDERSON für die Betreuung seitens IBM.

Mein Dank gilt PROF.SPRUTH, da durch ihn die Zusammenarbeit mit IBM-Böblingen erst möglich wurde.

Besonders danke ich meinen Kommilitonen HILMAR HENNINGS, JORK LÖSER, DANIEL LUCKE, ROBERT REILEIN und THOMAS SIEDSCHLAG für die interessante Zusammenarbeit.

Ich bedanke mich bei meiner Freundin, KAREN HERRMANN, für das Verständnis und die Unterstützung.

Weiterhin bedanke ich mich bei meiner Mutter, ohne deren Hilfe ich wohl nie so weit gekommen wäre.

1.2. Verschiedenes

Aus Gründen der besseren Lesbarkeit werden in dieser Arbeit folgende typografischen Standards benutzt: zur Darstellung von Personen- und Eigennamen werden ” SMALL CAPS” benutzt und Hervorhebungen *kursiv* dargestellt. Der Schriftstil `Typewriter` findet Verwendung, um Programm- und Dateinamen, Befehle und Listings hervorzuheben.

Geschützte Namen und eingetragene Warenzeichen wurden nicht als solche kenntlich gemacht. Aus dem Fehlen der Markierung ^{TM}, © u.a. kann nicht geschlossen werden, daß die Bezeichnung ein freier Warenname ist . Ebensowenig wurde auf Patente und Gebrauchsmusterschutz hingewiesen.

Ein wichtiger Bestandteil dieser Diplomarbeit ist die Implementierung des Programmpaketes pga . Darum befindet sich im Anhang A eine kurze Dokumentation zu diesem Programm.

2. Grundlagen

2.1. Parallelverarbeitung

Grundgedanke der Parallelverarbeitung ist, die Rechenleistung mehrerer Prozessoren auszunutzen, um ein Problem schneller bearbeiten zu können. Die heute am meisten verbreiteten Parallelrechner sind, entsprechend der Klassifizierung von FLYNN, MIMD-Computer[1]. MIMD-Computer bestehen aus mehreren Prozessoren, die unabhängig voneinander Instruktionen ausführen. Je nachdem, ob sich Parallelrechner gemeinsamen Speicher teilen oder nicht, werden sie in *Shared-Memory* oder *Distributed-Memory* Maschinen unterteilt. In Shared-Memory Computern teilen sich die integrierten Prozessoren Instruktionen und Daten in einem gemeinsamen Speicher. Zusätzlich dazu kann jedem Prozessor auch noch lokaler Speicher zur Verfügung stehen. Weil die Hardwarestrukturen für die gemeinsame Speicherverwaltung sehr aufwendig sind, besitzen Shared-Memory Computer im Normalfall wenige Prozessoren, haben aber den Vorteil, daß Austausch von Daten aufgrund des gemeinsamen Speichers sehr effizient möglich ist. Distributed-Memory Maschinen bestehen aus mehreren Knoten, die über eine Kommunikationsstruktur miteinander verbunden sind. Jeder Knoten kann als eigener sequentieller Computer betrachtet werden. Besteht für eine Menge von Computern die Möglichkeit der kooperativen Bearbeitung eines Problemes, kann ein Workstationcluster, bestehend aus einzelnen Computern, als Parallelrechner[2] aufgefaßt werden. Distributed-Memory Computer sind wesentlich besser skalierbar und preislich günstiger, als Shared-Memory-Maschinen. In der Praxis existieren Distributed-Memory Rechner mit bis zu mehreren hundert Knoten.

Der Nutzen von Parallelrechnern kann an den Werten des *Speedup* und der *Effizienz* abgelesen werden. Der Speedup eines Parallelen Programmes berechnet sich aus dem Verhältnis der parallelen Laufzeit zur sequentiellen Laufzeit:

$$S = \frac{t_{par}}{t_{seq}} \; . \tag{2.1}$$

[1] Multiple Instruction Multiple Data

[2] 1998 war erstmals ein Workstationcluster, bestehend aus 70 Rechnern mit Alphaprozessoren, unter den Top 500 der Supercomputer. Der Rechner des Los Alamos National Center erreichte 19.7 GFlops und kostet mit 150000 US$ ein Zehntel von vergleichbaren Parallelrechnern. Die einzelnen Knoten sind mit Fast Ethernet vernetzt und arbeiten unter LINUX.

Die Effizienz ist das Verhältnis Speedup zu Zahl der eingesetzten Knoten:

$$E = \frac{S}{n_{par}} . \tag{2.2}$$

Vernachlässigt man Effekte, wie superlinearen Speedup durch verkleinerte Problemgrößen, so gilt, daß der Speedup nie größer werden kann, als die Zahl der eingesetzten Knoten ($S \leq n_{par}$) und die Effizienz nie größer als eins wird:

$$E \leq 1 . \tag{2.3}$$

Weil sich viele sequentielle Probleme nicht in gleichgroße Teilprobleme zerlegen lassen, Knoten auf Zwischenergebnisse von anderen Knoten warten müssen und weil Datenaustausch zwischen den Prozessoren durchgeführt werden muß, ist eine Effizienz von ≈ 1 eher die Ausnahme. Die in der Praxis erzielten Werte liegen zum Teil erheblich darunter[3]. Neben der Ausführung von Parallelprogrammen auf einer angemessenen Architektur, ist es wichtig die Aufgaben und Daten so zu verteilen, daß alle Knoten gleichmäßig belastet sind, und die Kommunikation zwischen ihnen möglichst gering ist. Diese Verteilung, Partitionierung genannt, kann bei bestimmten Problemen sehr komplex werden, und muß nicht immer optimal lösbar sein. Die in dieser Arbeit betrachteten Probleme beziehen sich auf einen Einsatz auf Distributed Memory Computern.

2.2. Logiksimulation

Die Logiksimulation ist eine Möglichkeit, um digitale Schaltkreise zu verifizieren. Mit ihrer Hilfe lassen sich Entwurfsfehler frühzeitig finden und beheben. Somit verkürzt sich die Entwicklungszeit und Kosten können gespart werden. Für das Design eines Schaltkreises betrachtet man verschiedene Absraktionsebenen[13], die mit unterschiedlichen Simulationsmethoden in Verbindung stehen. Die Logiksimulation ist auf der Gate- und Register-Transferebene angesiedelt (siehe Abbildung 2.1). Je höher der Abstraktionslevel, desto geringer ist die Komplexität und somit der Entwurfs- und Simulationsaufwand. Leider gehen damit Informationen verloren, so daß man Gefahr läuft, bestimmte Probleme auf niedrigerem Niveau zu übersehen.

Bei der Komplexität moderner Schaltkreise ist es nicht möglich, einen Schaltkreis 'ad hoc' zu entwerfen. Stattdessen werden bereits vorhandene und getestete Bausteine einer Ebene zu neuen Komponenten der nächsten Ebene kombiniert. Diese Grundbausteine werden als Primitive bezeichnet und sind in Bibliotheken verfügbar.

Die für diese Arbeit betrachteten Simulatoren TEXSIM und MVLSIM [11] [4] setzen auf dem Register-Transfer-Gate-Level auf[4]. Mittels einer Hochsprache für den Schaltkreisent-

[3] Die Effizienz von "Deep Blue", dem IBM-Parallelrechner, der als erstes einen Schachweltmeister besiegte, lag bei 0.25.

[4] In der Praxis können Register-Transfer-Level und Gate-Level nicht immer eindeutig getrennt werden. TEXSIM und MVLSIM können über beiden Ebenen arbeiten.

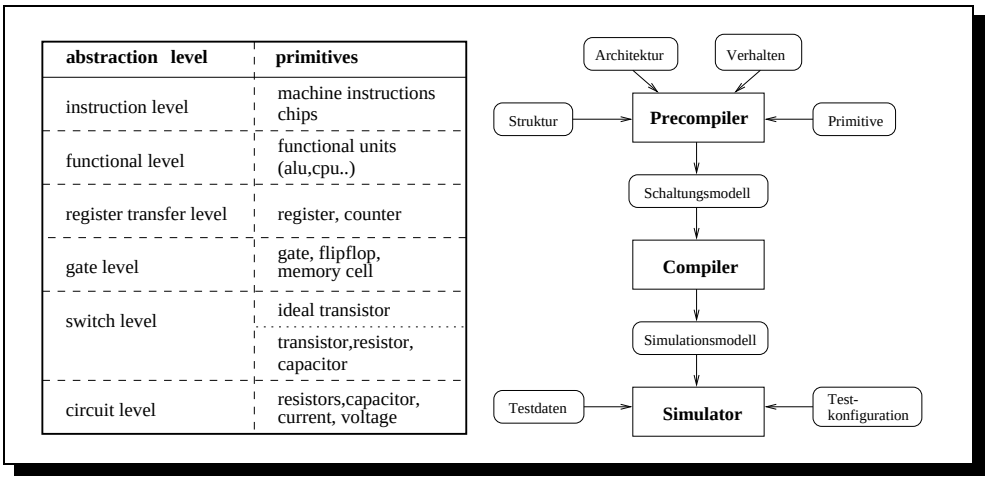

Abbildung 2.1.: links: Abstraktionsstufen im VLSI Design, rechts: Logiksimulationsumgebung

wurf[5] werden Verhalten, Struktur und Architektur festgelegt und mit den zugrunde liegenden Primitiven verbunden. Über mehrstufige Compilerläufe wird ein Simulationsmodell erzeugt, über dem der Simulator arbeiten kann.

2.3. Parallele Logiksimulation

Um die Komplexität der Logiksimulation meistern zu können, ist Parallelisierung eine geeignete Vorgehensweise. Voraussetzung dafür ist, daß die vorhandene Simulationsarbeit optimal auf alle involvierten Instanzen verteilt wird, damit sich der Parallelisierungsaufwand und der Einsatz von teurer Hardware, trotz des unvermeidlichen Overheads rentiert. Ein wichtiges Problem ist, eine Struktur zu finden, die eine effiziente Parallelisierung zuläßt.

Der Parallelisierung von TEXSIM [4] und MVLSIM liegt die modellinhärente Parallelität der Prozessormodelle zugrunde. Hierbei wird das Modell in Teilmodelle (Blöcke) zerlegt, die parallel simuliert werden. Eine solche Zerlegung bezeichnet man als Partitionierung. Diese hat einen sehr großen Einfluß auf die Effizienz der Parallelisierung.

Um den Datenaustausch zwischen den Blöcken zu begrenzen, kommunizieren PARALLEL-TEXSIM und PARALLELMVLSIM nur nachdem ein voller Taktschritt des Prozessormodelles simuliert worden ist. Sind die Daten zwischen den Blöcken ausgetauscht, wird das Modell mit Inputwerten und den im vorherigen Simulationsschritt berechneten Daten neu initialisiert und der nächste Takt simuliert.

Damit ein voller Takt bei einem Modellblock ohne Datenaustausch mit anderen Blöcken simu-

[5] z.B. VHDL, BDLS-3, DSL-1 (die beiden letztern sind IBM-interne Entwicklungen)

liert werden kann, muß die Schaltung so zerlegt werden, daß alle zur Evaluierung des Blockes, während eines Taktschrittes, benötigten Daten in diesem Block erzeugt werden können. Dazu werden *fan-in Cones* betrachtet, welche ausgehend von einem Kopfelement (Latch oder Outputbox) alle Elemente beinhalten, die eine Verbindung zu dem Kopfelement haben und selbst kein Kopfelement sind. Somit sind in dem fan-in Cone eines Conekopfes alle Elemente (Boxen) enthalten, die zur Evaluierung des Kopfes relevante Daten beitragen. Abbildung 2.2 zeigt ein solches Cone (schattiert). Latche sind spezielle Boxen, welche die Daten speichern, die für den nächsten Simulationsschritt wieder benötigt werden. Outputboxen speichern den Output eines simulierten Modelles.

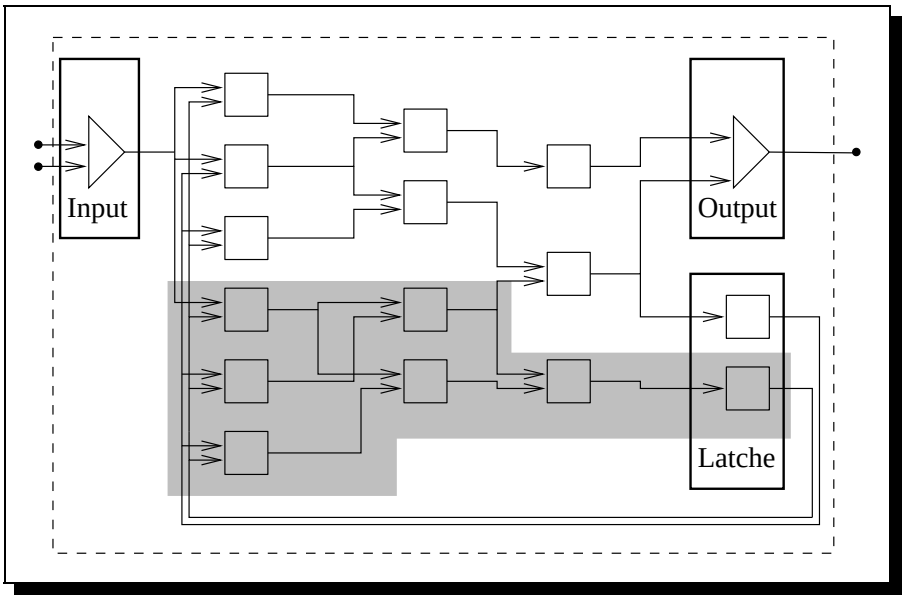

Abbildung 2.2.: Fan-In Cone in einem schematischen Prozessormodell.

Augenfällig ist die Überlappung der meisten Cones. Werden zwei sich überlappende Cones auf unterschiedlichen Knoten evaluiert, so bedeutet dies, daß gemeinsame Teile der Logik mehrfach berechnet werden, da während eines Taktes keine Daten ausgetauscht werden können. Auf die genauen Eigenschaften des Hardwaremodelles und die Überlappung der Cones wird in [9] und [10] eingegangen.

2.4. Partitionierung von Prozessormodellen

Ziel der Partitionierung ist eine Reduktion der parallelen Simulationszeit $\hat{T}_p$. Eine untere Schranke läßt sich aus Gleichung 2.3 folgern:

$$\hat{T}_p \geq \frac{\hat{T}_s}{n_p}\,, \tag{2.4}$$

wobei n_p die Knotenzahl und $\hat{T}_s$ die sequentielle Simulationszeit ist.

Eine Modellpartition ist auffaßbar als eine Zuordnung von Teilmodellen zu Prozessoren, auf denen diese simuliert werden. Wie in 3.1.2 noch näher gezeigt wird, läßt sich eine Partition als ein Vektor $\vec{p}_i$ darstellen. Die Zeit für einen parallelen Simulationszyklus T_p einer Modellpartition $\vec{p}_i$ läßt sich mit:

$$T_p(\vec{p}_i) = \max_{0<j\leq n_p} (T_j + K_j + R_j + W_j) \tag{2.5}$$

abschätzen. Dabei ist T_j die Evaluierungszeit der Logik, und K_j ist die Kommunikationszeit des Knotens j. R_j und W_j sind Zeiten, die für auszutauschende Daten zum Auslesen aus dem Simulationsmodell bzw. zum Schreiben in dieses benötigt werden. ist die Zeit, um die Daten, die ausgetauscht werden müssen, aus dem Simulationsmodell auszulesen und empfangene Daten in dieses zu schreiben. Weil die Zeit für die Parallele Simulation im Wesentlichen von der Simulationszeit und dem Kommunikations- und Synchronisationsoverhead bestimmt wird, müssen für eine gute Partitionierungsstrategie folgende Punkte berücksichtigt werden:

- gleichmäßige Lastverteilung
- minimale Mehrfachauswertung von Logik
- minimale Kommunikation zwischen den einzelnen Knoten nach jedem Takt.

Das Problem bei der Modellpartitionierung ist, daß die betrachteten Prozessormodelle bis zu 10^7 Elemente enthalten und Größen von einigen hundert Megabyte aufweisen. In [9] und [10] wird gezeigt, daß sich auf Grundlage der Überlappung der Cones und deren Kommunikationsbeziehungen ein Hypergraph konstruieren läßt, der das Prozessormodell repräsentiert. Das Problem der Modellpartitionierung läßt sich somit auf das Problem der Graphpartitionierung abbilden, welches NP-vollständig ist. Daher ist es mit heutiger Rechentechnik nicht möglich, eine optimale Lösung in akzeptabler Zeit zu berechnen.

Eine suboptimale Lösung läßt sich nur durch eine Reduktion der Daten erreichen, um die Problemgröße auf ein handhabbares Maß zu verkleinern. Dazu wurde an der Universität Leipzig eine Hierarchische Partitionierungsstrategie entwickelt [11]. Diese Strategie beginnt mit sehr einfachen und schnellen Vorpartitionierungsalgorithmen, welche die Problemgröße von 10^5-10^6 Cones auf 10^3-10^4 reduziert. Hierbei werden die Cones zu Supercones zusammengefaßt. Genau wie Cones überlappen sich die Supercones und müssen nach jedem Zyklus Daten austauschen, haben aber mehrere Coneköpfe. Auch auf Basis der Supercones kann ein Hypergraph konstruiert werden, der die Überlappung der Supercones und die Kommunikationsbeziehungen widerspiegelt. Aufbauend auf den Supercones und dem Hypergraphen arbeiten Partitionierungsalgorithmen der 2. Stufe und erstellen Partitionen, mit denen eine Logiksimulation möglich ist. Die Qualität der so erzeugten Partitionen kann durch Optimierungsverfahren zum Teil deutlich[6] verbessert werden. Evolutionäre Algorithmen sind ein solches

[6] Bei dem "Monet"-Modell, daß auf 15 Blöcke partitioniert wird, kann der Speedup mittels Evolutionärer Algorithmen von 8 auf 11.5 verbessert werden.

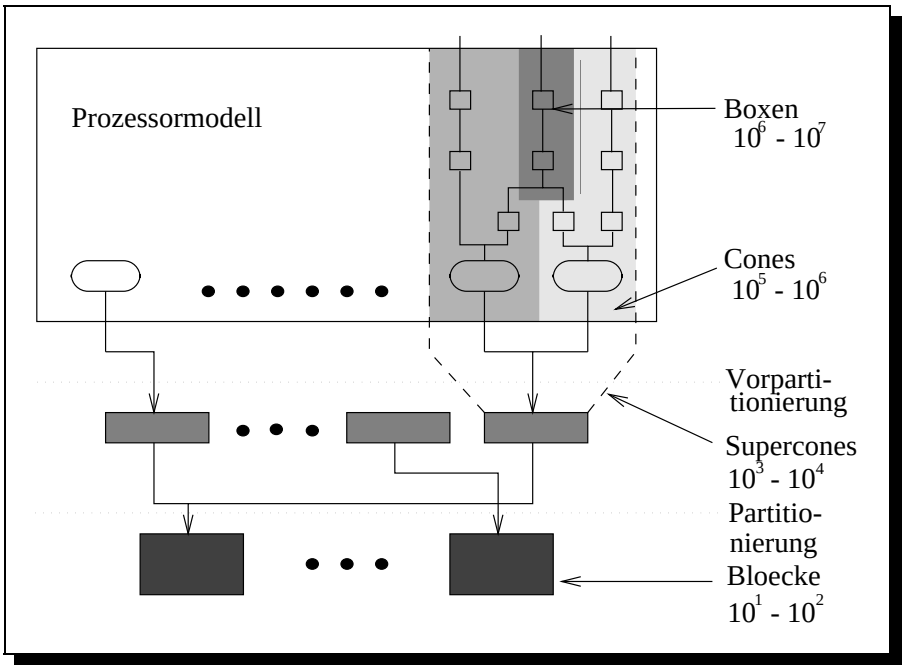

Abbildung 2.3.: Reduktion der Problemgröße durch hierarchische Partitionierung.

Optimierungsverfahren. Weil die parallele Hardware für die Simulation auch für die Partitionierung verfügbar ist, bietet sich eine verteilte Partitionierungsstrategie an, mit Parallelen Evolutionären Algorithmen als Optimierungsverfahren.

2.5. MPI - das Message Passing Interface

2.5.1. Einführung

Der Name MPI steht für Message Passing Interface und ist ein vom MPI FORUM entwickelter Standard. Ziel ist es, einen allgemein akzeptierten Standard zu schaffen, um Plattform unabhängig parallele Programme entwickeln zu können. Eigenschaften dieses Standards sollen Praktikabilität, Portabilität, Effizienz und Flexibilität sein. Um diese Ziele erreichen zu können, wurde 1992 das MPI FORUM [16] ins Leben gerufen, unter dessen Dach 40 Organisationen aus Wirtschaft und Wissenschaft zusammen den MPI Standard erarbeiteten. Statt ein bestehendes System zu adaptieren, wurden in MPI die Vorteile vieler bestehender Message Passing Systeme eingearbeitet. Nur so konnte ein allgemein akzeptiertes, theoretisch durchdachtes, als auch praktikables System entstehen. Stark beeinflußt wurde MPI aber trotzdem von den Arbeiten des IBM T.J.WATSON RESEARCH CENTER, INTELS NX/2, EXPRESS, NCUBE'S VERTEX, P4 und PARMACS.

Auf der SUPERCOMPUTING 93 konnte die erste Version des Standards präsentiert werden,

der als MPI 1.0 im Mai 1994 verabschiedet wurde. Im März 1995 erschien MPI 1.1 [17] um kleinere Fehler gegenüber Version 1.0 zu korrigieren und ein paar Verbesserungen einzubauen. MPI-2 [18] wurde April 1997 veröffentlicht. Zu Beginn dieser Arbeit lag noch keine stabile Implementierung dieses Standards vor.

MPI 1.1 Implementierungen hingegen gibt es für viele verschiedene Rechnerarchitekturen und deren Kommunikationssysteme[15]. So z.B:

- Workstationcluster (alle gängigen UNIX-Workstation incl. LINUX)
- Shared Memory Systeme
- nCUBE
- NEXUS
- MEIKO
- INTEL NX
- CRAY T3D / T3E
- IBM SP2 MIT HIGH PERFORMANCE SWITCH

Ein entscheidender Vorteil von MPI ist, daß es sehr einfach zu benutzen ist. So reichen 6 MPI-Befehle aus, um ein funktionsfähiges paralleles Programm zu schreiben. Darüber hinaus stehen für komplizierte Kommunikationsstrukturen sehr komplexe Befehle zur Verfügung (siehe Abb. 2.4). Auch wenn die MPI Kommunikationsbibliothek alle systemspezifischen Besonderheiten verdeckt, nutzen die Implementierungen von MPI dennoch das Potential spezieller Hardware aus. Somit ermöglicht MPI, portierbare parallele Programme bei maximaler Performance zu implementieren.

2.5.2. Das Kommunikationskonzept von MPI

MPI unterscheidet zwei große Gruppen der Kommunikation: Punkt-zu-Punkt Kommunikation und Kollektive Kommunikation.

Punkt-zu-Punkt Kommunikation

Punkt-zu-Punkt Kommunikation ist die gezielte Kommunikation zwischen zwei Knoten, auf denen ein paralleles Programm läuft. Hierbei wird noch einmal zwischen Blocking Communication und Nonblocking Communication unterschieden.

Bei der Blocking Communication muß eine Instanz den Sendeprozeß einleiten, und dabei u.a. den Empfänger, den Typ der Nachricht, die Zahl der gesendeten Blöcke, sowie ein Tag[7] festlegen. Der Empfänger verfährt analog[8]. Tabelle 2.1 zeigt die Syntax für diese Befehle. Nach der Abarbeitung eines Blocking Send/Receive Befehles, wird dem Nutzer garantiert, daß alle Daten ausgetauscht sind und die Puffer einen definierten Inhalt haben, oder aber ein erkennbarer Fehler aufgetreten ist. Bei jedem Datenaustausch warten die kommunizierenden Knoten aufeinander. Dies kann erwünscht sein, birgt aber den Nachteil, daß ein langsamer Knoten das gesamte parallele Programm behindert, womit ein deutlicher Geschwindigkeitsverlust verbunden sein kann. Um dies zu umgehen, gibt es die Möglichkeit der Nonblocking Communication.

Nonblocking Send/Receive-Befehle leiten nur die Kommunikation ein, warten aber nicht auf deren Abschluß, sondern geben die Kontrolle an den Programmierer zurück, der somit die Möglichkeit erhält, während der Kommunikation weitere Berechnungen durchzuführen. Es obliegt der Verantwortung des Programmierers, daß bis zum Abschluß der Kommunikation die Kommunikationspuffer nicht benutzt werden. Um den Erfolg eines Datenaustausches festzustellen, stehen weitere Befehle zur Verfügung (siehe Tabelle 2.2). Nonblocking Befehle ermöglichen es, effizienter zu programmieren, doch nur zum Preis von gefährlichen, potentiellen Fehlerquellen, und somit einer höheren Verantwortung beim Implementieren von parallelen Programmen.

Kollektive Kommunikation

Kollektive Kommunikation ist definiert als ein Datenaustausch, in dem eine Gruppe von Prozessen involviert ist. Typische Funktionen sind z.B.:

Barrier ermöglicht eine Synchronisation aller Prozesse.

Broadcast verschickt ein Datum an alle Knoten.

Gather sammelt Daten von allen Prozessen auf einem Knoten.

Scatter verschickt dedizierte Daten an alle Prozesse.

Der wichtigste Parameter für die kollektive Kommunikation ist der *Kommunikator*, mit dem eine Untermenge aller vorhandenen Knoten selektiert werden kann. Die größte Bedeutung kommt dabei dem vordefinierten Kommunikator MPI_COMM_WORLD zu, über den alle Prozessoren angesprochen werden können. Es ist möglich, Kommunikatoren dynamisch zu erzeugen oder zu verändern, und somit virtuelle Topologien zu erstellen und zu verwalten. Dies vereinfacht die Verwaltung der Kommunikationsstrukturen und erhöht die Flexibilität von MPI. Verwendung findet es zum Beispiel in der MAP [7] (siehe A.5).

[7] Ein Tag ist eine vom Nutzer festzulegende Id, welche die Art der Nachricht spezifiziert. Man kann somit verschiedene Typen von Nachrichten unterscheiden und entsprechend reagieren.

[8] Es ist allerdings möglich Wildcards zu benutzen und somit verschiedene Sender oder Tags zu akzeptieren.

int MPI_Send (void *buf, int cnt, MPI_Datatype dat, int dst, int tag, MPI_Comm com);

int MPI_Recv (void *buf, int cnt, MPI_Datatype dat, int src, int tag, MPI_Comm com, MPI_Status *sta);

buf Adresse des Sende/Empfangspuffers

cnt Anzahl der verschickten bzw. erwarteten Daten

dat Datentyp

src Spezifikation des Senders (Wildcard möglich)

dst Spezifikation des Empfängers

tag Spezifikation des Nachrichtentypes

com Spezifikation der Gruppe, in der kommuniziert wird

sta Status, der angibt, ob Kommunikation erfolgreich war

Tabelle 2.1.: Blocking Communication: Befehlssyntax

int MPI_ISend (void *buf, int cnt, MPI_Datatype dat, int dst, int tag, MPI_Comm com,MPI_Request *req);
Startet ein Nonblocking Send (I – immedeate)

int MPI_IRecv (void *buf, int cnt, MPI_Datatype dat, int src, int tag, MPI_Comm com, MPI_Status *sta, MPI_Request *req);
Startet ein Nonblocking Receive

MPI_Test (MPI_Request *rqst, int *flag, MPI_Status stat);
testet, ob Kommunikation abgeschlossen

MPI_Wait (MPI_Request *rqst, MPI_Status stat);
wartet, bis Kommunikation abgeschlossen

rqst Handle, mit dem jede Kommunikation identifiziert wird

flag Flag, welches anzeigt, ob Kommunikation erfolgreich

Tabelle 2.2.: Nonblocking Communication: Befehlssyntax

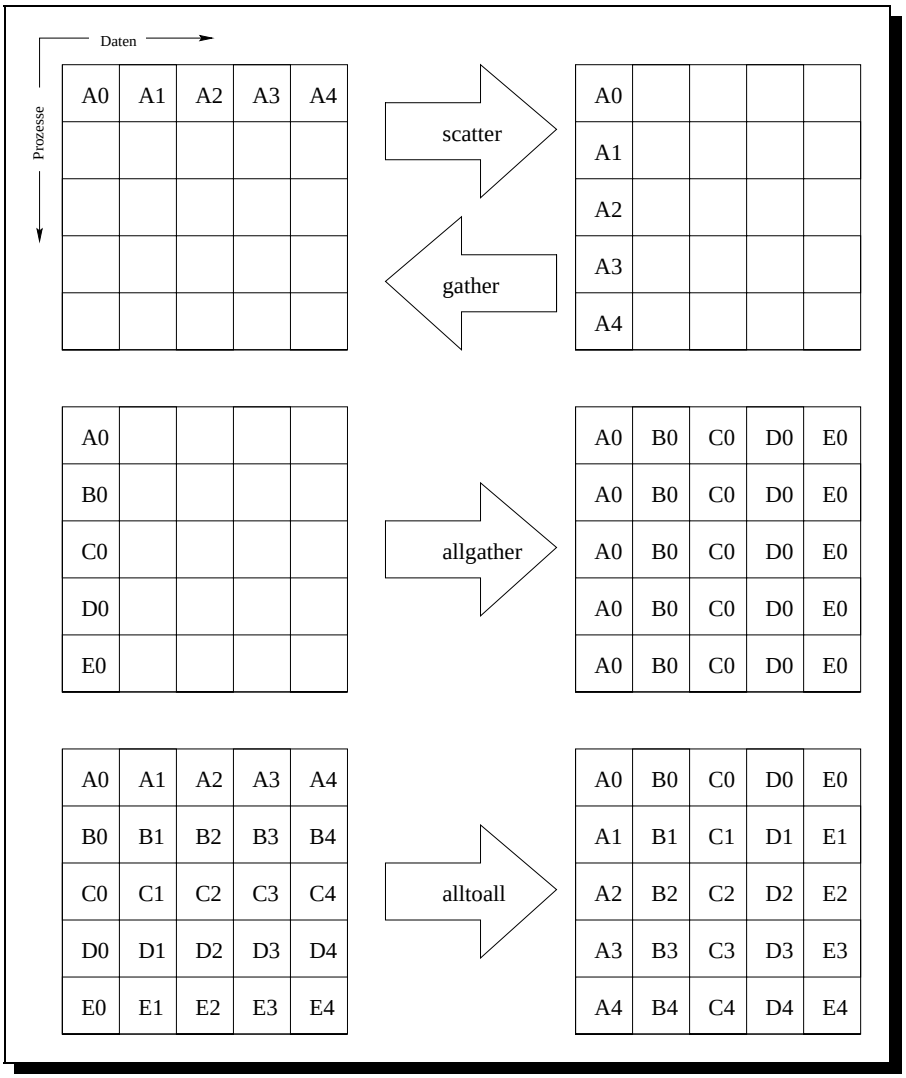

Abbildung 2.4.: MPI Kollektive Kommunikation

2.5.3. MPI - Ausblick

Der Funktionsumfang von MPI ist sehr groß, und kann im vollen Umfang in seiner Spezifikation [17] nachgelesen werden. Mit der Implementierung von MPI-2 als freie Software, bietet MPI die besten Voraussetzungen sich zu dem Standard für Distributed Memory Systeme zu entwickeln. Die Tatsache, daß MPI auch auf Shared Memory Maschinen lauffähig ist, wird dies vielleicht noch beschleunigen.

3. Evolutionäre Algorithmen

3.1. Sequentielle Evolutionäre Algorithmen

Evolutionäre Algorithmen (EA) verkörpern ein interdisziplinäres Forschungsgebiet der Künstlichen Intelligenz, Numerischen Optimierung, Ingenieurwissenschaften und der Biologie. EA ermöglichen es, modellfreie[1] Optimierungs- und Suchverfahren zu entwickeln.

3.1.1. Der biologische Hintergrund[2]

Die Evolutionären Algorithmen wurden den Prinzipien der biologischen Evolution nachempfunden. Es ist daher sinnvoll, die grundlegenden Mechanismen der Evolution am Beispiel der Evolutionstheorie, dem natürlichen Vorbild der EA zu erklären, um die Zusammenhänge, Hintergründe und Verfahrensweisen zu verdeutlichen.

C. DARWIN(1809-1882) hatte 1852 mit seinem Werk "DIE ENTSTEHUNG DER ARTEN .." die noch heute gültigen Grundprinzipien der Evolution bekannt gemacht. Das berühmte Zitat "THE SURVIVAL OF THE FITTEST" beschreibt das Grundprinzip der Natur, in der, über einen langen Zeitraum, nur die am besten angepaßten Individuen überleben werden. Darwin hatte gezeigt, daß die Arten sich im Laufe der Zeit der Umwelt anpassen, und die bis dahin angenommene Unveränderlichkeit der Arten widerlegt.

Grundlegende Elemente der Evolution sind die zufällige Veränderung von Erbinformation (Mutation), die Kombination von bestehenden Eigenschaften (Crossing-Over), und die höhere Überlebens- und Vermehrungschancen von besseren Individuen gegenüber schlechteren (Selektion). Ermöglicht werden Mutation und Crossing auf Zellebene. Die Erbinformation aller höheren Organismen ist auf molekularer Ebene in der DNS (Desoxyribonukleinsäure), einem doppelhelixförmigen Makromolekül, kodiert. Durch vier verschiedene Nukleotidbasen wird die Kodierung der Erbinformation, der Genotyp eines Organismus, realisiert.

[1] Zur Optimierung ist nur eine (Fitneß)-Funktion nötig, um zwischen guten und schlechten Ergebnissen zu unterscheiden, aber kein modellspezifisches Wissen.

[2] Hier kann leider nur eine stark vereinfachte Zusammenfassung widergegeben werden. Die molekulargenetischen Vorgänge sind derart komplex, daß sie den Rahmen dieser Arbeit sprengen würden.

Einen kompletten Doppelhelixstrang bezeichnet man als Chromosom. Auf ihm befinden sich funktionale Einheiten, die bestimmte Eigenschaften des Organismus repräsentieren (z.B. Haar- oder Augenfarbe). Diese Einheiten, Gene genannt, beginnen mit einer Startsequenz (Promotor), und hören mit einer Terminatorsequenz auf. Gene bestehen aus verschieden vielen Basen-Triplets [3], welche jeweils eine Aminosäure repräsentieren. Die Aminosäuren sind die Bausteine für Proteine, die wiederum maßgeblichen Einfluß auf den Stoffwechsel eines Organismus, und somit auf seine Entwicklung und sein Verhalten haben. Es gibt $4^3 = 64$ Kodierungen, aber nur 20 verschiedene Aminosäuren. Dies bedeutet, verschiedene Säuren werden mehrfach repräsentiert. Einige Triplets kodieren Zusatzinformationen, wie Start- und Stoppunkte von Genen.

Eine Funktion der DNS ist die *Protein Biosynthese*. Das Enzym DNS-Polymerase spaltet die DNS zwischen Promotor- und Terminatorsequenz auf. Daraufhin wird entlang eines Genes die Kodesequenz kopiert, indem die RNS (Ribonukleinsäure) als genaue Abschrift des DNS-Abschnitts erzeugt wird. Dieser Vorgang des "Umschreibens" wird *Transscription* genannt. Die RNS wird zu den Ribosomen, einer speziellen Zellstruktur, transportiert. Dort wird aus der Triplettsequenz das kodierte Protein aus Aminosäuren zusammengesetzt. Die erzeugten Proteine werden in den Stoffwechsel involviert, und nehmen direkt Einfluß auf den *Phenotyp*, das Erscheinungsbild des Organismus. Der Phenotyp identischer Genotypen muß nicht zwangsläufig gleich sein, nehmen äußere Faktoren doch maßgeblich Einfluß auf die Entwicklung von Organismen.

Die zweite Funktion der DNS ist die Erbinformationsweitergabe bei der Zellteilung. Bei der normalen Zellteilung (*Mitose*) wird die DNS verdoppelt, und die Zelle teilt sich in zwei identische Tochterzellen. Sich sexuell vermehrende Organismen produzieren bei der *Meiose* aus einer diploiden[4] Zelle vier haploide Zellen. Dazu wird auch hier die DNS verdoppelt, und dann auf vier Ei- bzw. Samenzellen aufgeteilt, welche nur einen einfachen Chromosomensatz besitzen. Bei der Vermehrung verschmelzen dann Ei und Samenzelle zu einer diploiden Zelle. Vorher werden aber die Gene der mütterlichen und väterlichen DNS gemischt und zwei neue Chromosomen gebildet. Dieser Vorgang heißt *Crossing-Over*. Crossing bewirkt ein Mischen von Erbinformationen und somit von phenotypischen Eigenschaften.

Mutation ist ein zufälliges Verändern der DNS. Dies kann passieren, indem bei der Duplikation der DNS Fehler auftreten; sei es zufällig, oder durch äußere Einflüsse, wie Chemikalien oder radioaktive Strahlung. Auch kann die DNS selbst durch Strahlung oder Chemikalien (Mutagene) beschädigt werden. Tritt Mutation bei der Mitose auf, so besitzen alle Zellen, die von der mutierten Zelle abstammen, die veränderte Erbinformation. Ein Beispiel dafür ist die Krankheit Krebs. Findet eine Mutation hingegen bei der Meiose statt, so sind ausschließlich die Kindorganismen von der Mutation betroffen. Der größte Teil von Mutationen hat aber keine Auswirkungen auf ein Lebewesen. Dafür sorgt die Redundanz der Erbinformation, sowie diverse Reparaturmechanismen.

[3] Einheit von drei Nukleotidbasen

[4] Zellen, in dem jedes Chromosom doppelt vorhanden ist

Die Selektion ist der komplexeste der genetischen Operatoren. Sie wirkt indirekt, und bewirkt, daß besser an die Umwelt angepaßte Individuen höhere Überlebens- und Vermehrungschancen haben, d.h. Gene (und somit Eigenschaften), die einem Individuum Vorteile gegenüber anderen verschaffen, haben größere Chancen sich zu vermehren und in zukünftige Generationen zu dominieren. Somit findet langsam eine Anpassung einer Population an die Umwelt statt. Dieser Prozeß ist rekursiv, so verändert die Anpassung an Umweltbedingungen die Umwelt selbst.[5]

3.1.2. Funktionsweise von Evolutionären Algorithmen

Evolutionäre Algorithmen abstrahieren die grundlegenden evolutionstheoretischen Prinzipien von Replikation, Variation und Selektion, und übertragen diese auf Such- und Optimierungsverfahren. Angesichts einer Vielzahl von verschiedenen EA Varianten und Verfahren ist es schwer, einen allgemeinen Algorithmus vorzustellen. Darum werden im Folgenden die Grundprinzipien von EA und die prinzipielle Funktionsweise an Hand eines abstrakten Basis-Algorithmus erklärt. Wo es sinnvoll ist, wird auf die konkrete Umsetzung bei der Modellpartitionierung verwiesen.

Der Begriff ”Evolutionärer Algorithmus” soll eine Vereinigung der Begriffe ”Genetischer Algorithmus” und ”Evolutionäre Strategien” symbolisieren, die für zwei Schulen stehen, die unabhängig von einander die Prinzipien der EA entdeckt haben und sich lange Zeit ignorierten. Die Genetischen Algorithmen wurden in den 60er Jahren von dem Amerikaner JOHN HOLLAND entwickelt, zur gleichen Zeit wie INGO RECHENBERG, ein deutscher Ingenieur, der seine Entwicklung Evolutionäre Strategien nannte. Beide Schulen unterscheiden sich in einigen Details und teilweise in der Terminologie. In dieser Arbeit wird von dem Wissen beider Schulen Gebrauch gemacht, ohne näher darauf einzugehen.

Bezüglich eines konkreten Problems, ist die wichtigste Voraussetzung für eine erfolgreiche Optimierung ein Kriterium, welches gefundene Lösungen hinsichtlich ihrer Güte bewertet, und somit ermöglicht, zwischen besseren und schlechteren Lösungen zu unterscheiden. Mit diesem Kriterium und einer geeignet gewählten Kodierung des Problems kann eine Fitneßfunktion realisiert werden, die den problemspezifischen Kern des EA bildet. Alle weiteren Komponenten eines EA sind problemunabhängig, können aber mit modellspezifischen Wissen ergänzt werden, sofern vorhanden.

Kodierung

Unter ungünstigen Umständen kann es sehr schwer sein, eine geeignete Kodierung für ein spezielles Problem zu finden. Doch es ist die Kodierung, die sehr stark über Erfolg oder Mißerfolg

[5] Wird zum Beispiel eine Raubtierpopulation zu erfolgreich, kann es passieren, daß alle potentiellen Beutetiere ausgerottet werden, und die Raubtierpopulation selbst verhungern muß, oder sich an die neuen Umweltbedingungen anpaßt.

eines EA entscheidet. Daher sollte dies sehr sorgfältig geschehen.

Im Kontext der Modellpartitionierung hingegen ist es recht einfach, eine sinnvolle Kodierung zu finden. Die Aufgabe der Modellpartitionierung ist die Verteilung von Supercones auf die Knoten eines Parallelrechners. Jedes Supercone wird durch eine natürliche Zahl dargestellt und repräsentiert ein Teilstück der Logik, die im Logiksimulator simuliert werden soll. Es liegt also nahe, zur Kodierung ein Array von Integern zu verwenden, dessen Länge gleich der Anzahl der Supercones ist. Hierbei wird jedes Feldelement als Gen, und das Array als Chromosom bezeichnet. Jedes Gen repräsentiert ein Supercone, während der Inhalt der Knotennummer entspricht, auf dem dieses Supercone evaluiert werden soll. Jedes Chromosom kodiert eine Partition, und entspricht einem Individuum. Formal soll ein Individuum mit $\vec{p}_i$ bezeichnet werden, wobei der Index i auf das i-te Individuum einer Population $\mathfrak{P}$ verweist. Da es möglich ist, daß identische Individuen mehrmals in einer Population vorhanden sind, sind die hier betrachteten Mengen Multisets. Die benutzten Mengenoperationen sind dann Operationen über Multisets.

Fitneßfunktion

Um zwischen guten und weniger guten Lösungen zu unterscheiden, wird eine Fitneßfunktion $\mathfrak{F}$ benötigt:

$$\mathfrak{F} : I \to \mathbb{R} \tag{3.1}$$

$$\vec{p}_i \in \mathfrak{P} \subset I \quad \forall i \tag{3.2}$$

wobei I ein beliebiger topologischer Raum sein kann.

Im Falle der Modellpartitionierung ist

$$I = \mathbb{N}^n$$

und

$$\mathfrak{F} : \mathbb{N}^n \to \mathbb{N} \tag{3.3}$$

entspricht der Zeit für einen Simulationszyklus T_i aus Gleichung 2.5. n ist die Zahl der Supercones.

Replikation

Ein Evolutionärer Algorithmus arbeitet über Populationen von Partitionen, welche durch die Zahl der Individuen (μ) und der erzeugten Nachkommen (λ) bestimmt werden. Das Verhältnis

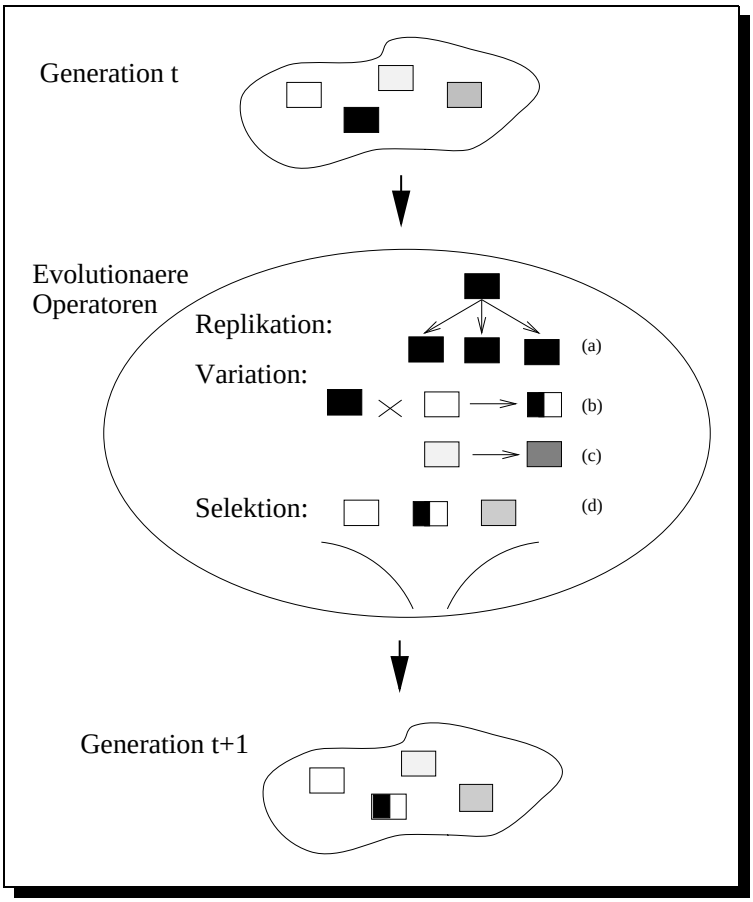

Abbildung 3.1.: EA-Basiszyklus: jede Generation durchläuft die Schritte der Replikation *(a)*, Variation mit Crossing-Over*(b)* und Mutation *(c)*, sowie der Selektion *(d)*.

$\frac{\lambda}{\mu}$ wird als Selektionsdruck bezeichnet und spielt eine große Rolle für den Erfolg eines EA (s. Selektion).

Mit jedem neuen Generationszyklus werden nach einem speziellen Auswahlschema Individuen selektiert und dupliziert. Die Duplikate bilden eine Population von Kindindividuen, die gezielt verändert werden.

Selektionsschemata für die Replikation können z.B. nach folgenden Prinzipien arbeiten:

- Auswahl der besten Individuen
- Auswahl der schlechtesten Individuen
- Auswahlwahrscheinlichkeit proportional zur Fitneß (Roulette Wheel).

Variation

Die Veränderung der Kinder geschieht in der Absicht, die Lösung, die ein Individuum repräsentiert, zu variieren und zu verbessern.

Eine Methode ist die Mutation. Hierbei wird der Inhalt eines zufällig ausgewählten Gens verändert. Die Wahrscheinlichkeit, mit der ein Gen verändert wird, bezeichnet man als Mutationsrate. Die zweite wichtige Methode zur Variation ist das Crossing Over. Die Besonderheit

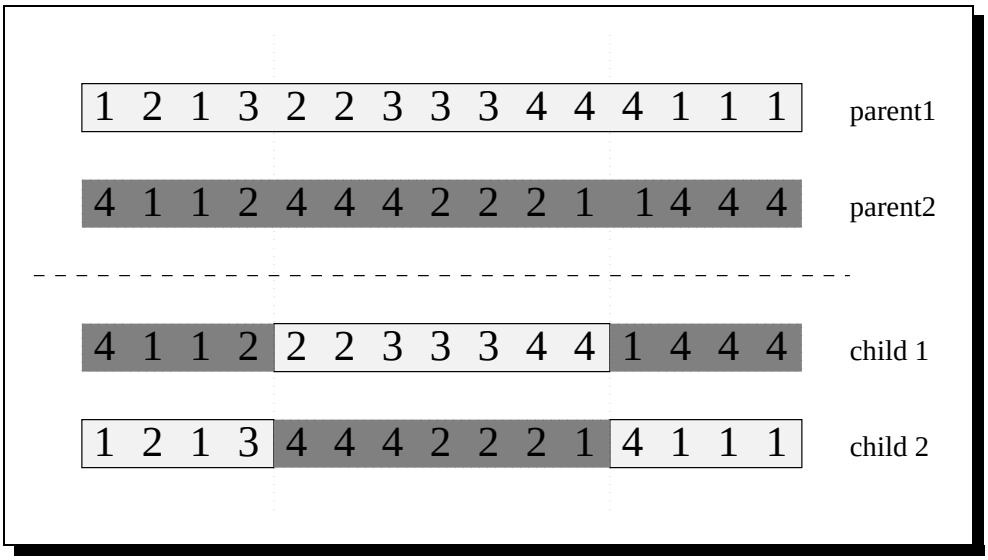

Abbildung 3.2.: Crossing-Over: Die Chromosomen der beiden Elternindividuen werden an zufällig bestimmten Crosspoints aufgetrennt. Die Chromosomen der Kinder setzen sich aus den Bruchstücken der Elternchromosomen zusammen.

ist hierbei, daß Crossing die Chromosomen von zwei Eltern mischt (siehe Abbildung 3.2). Es wird daher zusammen mit der Replikation durchgeführt.

Diese beiden genetischen Operatoren zur Variation, sind typisch für nahezu alle EA. Bei speziellen Anwendungen können sich aber zusätzlich noch problemspezifische Operatoren als sinnvoll erweisen, bzw. Mutation oder Crossing ersetzen.

Selektion

Wurden Replikation und Variation durchgeführt, so ist es die Aufgabe der Selektion die Individuen auszuwählen, die die nächste Generation bilden. Dazu müssen alle Individuen durch die Fitneßfunktion bewertet werden. Bei der Modellpartitionierung ist das Optimierungsziel eine Minimierung der Simulationszeit, es werden also die Individuen mit den kleinsten Fitneßwerten bevorzugt. Die wichtigsten Selektionsmethoden sollen im Folgenden kurz vorgestellt werden.

Roulette Wheel ist ein fitneßproportionales Auswahlverfahren. Die Grundidee ist, daß die Wahrscheinlichkeit, mit der ein Individuum selektiert wird, proportional zur Fitneß ansteigt. Dies hat zur Folge, daß auch schlechte Individuen eine Chance haben, in die nächste Generation übernommen zu werden. Die Wahrscheinlichkeit $P_s(\vec{p}_i)$, daß ein Individuum $\vec{p}_i$ selektiert wird, läßt sich wie folgt berechnen:

$$P_s(\vec{p}_i) = \frac{\mathfrak{F}(\vec{p}_i)}{\sum_{j=1}^{\mu} \mathfrak{F}(\vec{p}_j)} \quad . \tag{3.4}$$

Die $(\mu+\lambda)$-**Strategie** ist ein Selektionsverfahren, bei dem die μ besten Individuen aus Eltern- und Kindpopulation in die nächste Generation übernommen werden. Sehr gute Individuen haben die Chance, unbegrenzt zu existieren, schlechtere hingegen werden sofort eliminiert.

Die (μ,λ)-**Strategie** wählt die besten μ Individuen aus den λ Kindern aus. Jedes Individuum existiert nur eine Generation. Es ist durchaus möglich, daß eine Generation schlechter ist, als die vorherige.

Kombinierte Methoden erweisen sich als sinnvoll, um Nachteile der einzelnen Selektionsmethoden zu kompensieren. Bei Roulette Wheel und (μ,λ) ist es möglich, daß sich die Kindgeneration gegenüber der Elterngeneration verschlechtert. Die $(\mu+\lambda)$-Strategie hat das Problem, daß eine gute Elterngeneration jede ihrer Kinder überlebt, und somit die Population zu frühzeitiger Konvergenz gezwungen wird. Darum werden in der Praxis diese Methoden oft kombiniert. Roulette Wheel kann um eine Elitefunktion erweitert werden, die eine festgelegte Zahl von Eltern als *Elite* übernimmt. (μ,λ) läßt sich mit $(\mu+\lambda)$ zu $(\mu*\lambda)_\nu$ kombinieren. Hier werden ν Individuen nach der $(\mu+\lambda)$-Strategie, der Rest nach der (μ,λ)-Strategie kopiert.

Das Programm pga kann mit verschiedenen Selektionsstrategien arbeiten. Als günstig hat sich die $(\mu*\lambda)_\nu$-Strategie erwiesen.

Evolutionärer Basis Algorithmus

Mit den vorher beschriebenen genetischen Operatoren und Methoden läßt sich ein abstrakter Basis Algorithmus beschreiben (siehe Algorithmus 1). In der Praxis werden die benutzten Algorithmen problembedingt mehr oder weniger von diesem Algorithmus abweichen oder zusätzliche Komponenten aufweisen.

Nach der Initialisierung wird für die μ Individuen $\vec{p}_i$ die Fitneß $\mathfrak{F}(\vec{p}_i)$ berechnet. Anschließend werden durch Crossing-Over von Elternpaaren λ Kinder $\vec{c}_i$ erzeugt. Diese werden zufällig mutiert und durch die Fitneßfunktion bewertet. Aus der Menge aller Kinder und gegebenenfalls aller Eltern werden entsprechend dem Selektionsverfahren μ Individuen ausgewählt, welche die neue Generation bilden. Dieses Verfahren wird so lange wiederholt, bis ein Abbruchkriterium erfüllt ist.

Mit jeder Generation besteht die Möglichkeit, daß sich einzelne Individuen verbessern. Da es aber nicht ausgeschlossen ist, daß sich Individuen verschlechtern, besteht für die Popu-

Algorithmus 1 Basis EA

Vorbedingung: $0 < \mu \leq \lambda$
  Initialisiere Startgeneration $\mathfrak{P}(0)$
  t=0
  while *abort* $\neq$ *TRUE* **do**
    for $i = 1$ to μ **do**
      berechne Fitneß $\mathfrak{F}(\vec{p}_i)$
    end for
    for $i = 1$ to λ step 2 **do**
      wähle 2 Individuen $\vec{p}_x$ und $\vec{p}_y$ aus $\mathfrak{P}(t)$
      erzeuge Nachkommen $\vec{c}_i$ und $\vec{c}_{i+1}$ durch Crossing von $\vec{p}_x$ und $\vec{p}_y$
      Mutiere $\vec{c}_i$ und $\vec{c}_{i+1}$
      berechne Fitneß $\mathfrak{F}(\vec{c}_i)$ und $\mathfrak{F}(\vec{c}_{i+1})$
    end for
    Selektiere μ Individuen aus $\{\vec{c}_1, \ldots, \vec{c}_\lambda\} \cup \mathfrak{P}(t)$ und bilde neue Generation $\mathfrak{P}(t+1)$
    $t = t + 1$
    if Abbruchkriterium erreicht **then**
      abort = *TRUE*
    end if
  end while

lation die Chance, lokale Optima zu überwinden. Die Möglichkeit der Verschlechterung ist essentiell, denn ein lokales Optimum ist gerade dadurch definiert, daß in einem bestimmten Umkreis keine bessere Lösung existiert und erst ab einem bestimmten Abstand die Lösungen besser werden. Wären Verschlechterungen von Individuen ausgeschlossen, dann konvergiert eine Population frühzeitig im ersten erreichten lokalen Optimum.

3.1.3. Formale Darstellung von Evolutionären Algorithmen

Die in Abschnitt 3.1.2 beschriebenen Operatoren und Algorithmen sind, um ein leichtes Erfassen der zu Grunde liegenden Prinzipien zu ermöglichen, sehr einfach und allgemein formuliert. Im folgenden Abschnitt soll eine formale Beschreibung evolutionärer Algorithmen eingeführt werden, um eine exakte Formulierung dieser und ihrer Bestandteile zu ermöglichen. Ausgehend von [1] kann ein Evolutionärer Algorithmus wie folgt definiert werden.

Definition 1 (Evolutionärer Algorithmus) *Ein Evolutionärer Algorithmus (EA) ist definiert als ein 8-Tupel*

$$EA = (I, \mathfrak{F}, \Omega, \Psi, s, \iota, \mu, \lambda) \quad . \tag{3.5}$$

Dabei ist I der topologische Raum der Individuen. $\mathfrak{F}$ *ist eine Fitneßfunktion* $\mathfrak{F} : I \to \mathbb{R}$.

$$\Omega = \{\omega_{\Theta_1}, ..., \omega_{\Theta_z} \mid \omega_{\Theta_i} : I^\lambda \to I^\lambda\} \cup \{\omega_{\Theta_0} : I^\mu \to I^\lambda\} \tag{3.6}$$

ist eine Menge von genetischen Operatoren ω_{Θ_i}*, die jeweils durch eine Menge von Parametern* $\Theta_i \subset \mathbb{R}^n$ *spezifiziert werden.*

$$s_{\Theta_s} = I^{\mu+\lambda} \to I^\mu \tag{3.7}$$

bezeichnet einen Selektionsoperator, welcher die Anzahl der Individuen von λ *oder* $\mu+\lambda$ *auf* μ *ändert, wobei* $\mu, \lambda \in \mathbb{N} \wedge \mu \leq \lambda$. μ *ist die Zahl der Elternindividuen und* λ *die des erzeugten Nachwuchses.* $\iota : I^\mu \to \{true, false\}$ *ist ein Abbruchkriterium für den EA.* Ψ *ist eine Übergangsfunktion* $\Psi : I^\mu \to I^\mu$*, die eine Population* $\mathfrak{P}$ *in eine Folgepopulation überführt, indem die genetischen Operatoren und die Selektion ausgeführt werden:*

$$\begin{aligned} \Psi &= s \circ \omega_{\Theta_1} \circ ... \circ \omega_{\Theta_z} \circ \omega_{\Theta_0} \\ \Psi(\mathfrak{P}) &= s_{\Theta_s}(Q \cup \omega_{\Theta_1}(...(\omega_{\Theta_z}(\omega_{\Theta_0}(\mathfrak{P}))))) \quad . \end{aligned} \tag{3.8}$$

Wobei $Q \in \{\emptyset, \mathfrak{P}\}$.

Ein Individuum $\vec{p}$ ist demnach ein Vektor des Raumes I: $\vec{p} \in I$. Die Menge aller erzeugten Kinder sei im Folgenden mit $\mathfrak{C}$ bezeichnet. Es gilt:

$$\mathfrak{C} = \omega_{\Theta_1}(...(\omega_{\Theta_z}(\omega_{\Theta_0}(\mathfrak{P})))) \quad . \tag{3.9}$$

Definition 2 (Populationsfolge) *Gegeben sei ein Evolutionärer Algorithmus mit der Übergangsfunktion* $\Psi : I^\mu \to I^\mu$ *und einer initialen Population* $\mathfrak{P}(0) \in I^\mu$*. Die Folge* $\mathfrak{P}(0), \mathfrak{P}(1), \mathfrak{P}(2), ...$ *wird Populationsfolge oder Evolution von* $\mathfrak{P}(0)$ *genannt:* $\Leftrightarrow$

$$\forall t \geq 0 : \mathfrak{P}(t+1) = \Psi(\mathfrak{P}(t)) \quad . \tag{3.10}$$

Crossing Over ist ein sexueller Operator, der wie folgt dargestellt werden kann:

$$\begin{aligned} & r_{\Theta_r} \in \Omega \\ & r_{\Theta_r} : I^\mu \to I^\lambda \quad \wedge \\ & \exists \hat{r}_{\Theta_r} : I^2 \to I \\ \mathfrak{P}' = r_{\Theta_r}(\mathfrak{P}) &= \{\hat{r}_{\Theta_r}(\vec{p}_{i_1}, \vec{p}_{i_2}), ..., \hat{r}_{\Theta_r}(\vec{p}_{i_{2\lambda-1}}, \vec{p}_{i_{2\lambda}})\} \quad , \\ & \qquad \textit{mit } i_1, ...i_{2\lambda} \textit{ zufällig und } \vec{p}_j \in \mathfrak{P} \; . \end{aligned} \tag{3.11}$$

Für die Mutation gilt analog:

$$\begin{aligned} & m_{\Theta_m} \in \Omega \\ & m_{\Theta_m} : I^\lambda \to I^\lambda \quad \wedge \\ & \exists \hat{m}_{\Theta_m} : I \to I \\ \mathfrak{P}'' = m_{\Theta_m}(\mathfrak{P}') &= \{\hat{m}_{\Theta_m}(\vec{p}'_{i_1}), ..., \hat{m}_{\Theta_m}(\vec{p}'_{i_\lambda})\} \quad , \\ & \qquad \textit{mit } i_1, ...i_\lambda \textit{ zufällig und } \vec{p}'_j \in \mathfrak{P}' \; . \end{aligned} \tag{3.12}$$

Algorithmus 2 Formaler Basis EA

$t = 0$
initialisiere $\mathfrak{P}(0) = \{\vec{p}_1(0), ...\vec{p}_\mu(0)\} \in I^\mu$
evaluiere $\mathfrak{P}(0) : \{\mathfrak{F}(\vec{p}_1(0)), ..., \mathfrak{F}(\vec{p}_\mu(0))\}$
while $\iota(\mathfrak{P}(t)) \neq true$ **do**
  $\mathfrak{P}'(t) = r(\mathfrak{P}(t))$
  $\mathfrak{C} = \mathfrak{P}''(t) = m(\mathfrak{P}'(t))$
  evaluiere $\mathfrak{C} : \{\mathfrak{F}(\vec{c}_1), ..., \mathfrak{F}(\vec{c}_\lambda)\}$
  selektiere $\mathfrak{P}(t+1) = s(\mathfrak{C} \cup \mathfrak{P})$
  $t = t + 1$
end while

Mit den bisher eingeführten Symbolen läßt sich der Algorithmus 1 von Seite 21 formalisieren (siehe Algorithmus 2).

3.2. Parallele Evolutionäre Algorithmen

3.2.1. Das Inselmodell

Eine Möglichkeit Evolutionäre Algorithmen zu parallelisieren, beruht auf dem Inselmodell[6]. Dabei teilt man eine große Population in mehrere kleinere Subpopulationen, die auf verschiedene Prozessoren (Inseln) verteilt werden, um dort einen sequentiellen Evolutionären Algorithmus zu durchlaufen. Zwischen diesen Inseln können jedoch Individuen von einer Population zu einer anderen migrieren. Über die Zahl und die Häufigkeit der migrierenden Individuen kann maßgeblich Einfluß auf das Verhalten des Parallelen Evolutionären Algorithmus (PEA) genommen werden. Weitere wichtige Parameter für das Verhalten sind die Struktur, die angibt, welche Populationen miteinander kommunizieren und die Art der Selektion und Integration migrierender Individuen.

Wie in 5.4 belegt, sprechen folgende Vorteile für PEA mit dem Inselmodell:

- inhärente Parallelität auf Populationsbasis ⇒
  - einfache Implementierung
  - effiziente Parallelisierung (Speedup ≈ Knotenanzahl)
  - sehr gute Skalierbarkeit

[6] Das Inselmodell geht auf das gleichnamige biologische Modell der Populationsgenetik zurück, wo davon ausgegangen wird, daß mehrere Subpopulationen mit eingeschränkter Kommunikation untereinander realistischer sind, als eine sehr große Population mit vielen Individuen.

- schnellere Optimierung als EA
- bessere Optimierung als EA
- stabilere Optimierung als EA.

PEA können somit auf jedem Knoten eine oder mehrere Populationen evaluieren. Beschränkt man die Kommunikation zwischen den Knoten, so können diese weitestgehend unabhängig voneinander arbeiten und der Kommunikationsaufwand kann im Vergleich zur Gesamtrechenzeit sehr klein gehalten werden. Achtet man bei der Implementierung PEA darauf, daß sich unterschiedlich schnelle Knoten nicht gegenseitig behindern, ist es möglich, einen Zeitgewinn proportional der eingesetzten Rechenknoten zu erreichen (Speedup $\approx$ Knotenanzahl).

Der zweite große Vorteil ist, daß die verschiedenen Subpopulationen verschiedene Teilbereiche des Suchraumes absuchen können. Somit sind qualitativ bessere Lösungen möglich und können in kürzerer Zeit gefunden werden, als es mit einer einzigen großen Population möglich ist.

Die praktische Umsetzung PEA entspricht einer parallelen Ausführung von sequentiellen Evolutionären Algorithmen, die an einer bestimmten Stelle Individuen miteinander austauschen. Algorithmus 3 verdeutlicht das Grundprinzip.

Algorithmus 3 Paralleler EA

FÜHRE AUF N KNOTEN PARALLEL AUS:
Initialisiere Startgeneration
while $abort \neq TRUE$ **do**
  berechne Fitneß $\mathfrak{F}(\vec{p}_i)$
  erzeuge Kinder durch Crossing und Mutation
  Selektiere k Migranten $\vec{m}_j$
  Verteile $\vec{m}_j$ an andere Populationen
  Empfange k Migranten $\vec{m}'_j$ von anderen Populationen
  Selektiere μ Individuen aus der Menge der Kinder, der Eltern und der Migranten entsprechend ihrer Fitneß und bilde neue Generation
  if Abbruchkriterium erreicht **then**
    $abort = TRUE$
  end if
end while
ENDE DES PARALLELEN PROGRAMMES

Von der geschickten Einstellung der Migrationsrate, der Selektions- Integrationsmechanismen und der Migrationsstruktur hängt ein Großteil des Erfolges der PEA ab. Ist die Migrationsrate zu stark, so entspricht das Gesamtverhalten der PEA ungefähr dem eines EA mit einer einzigen großen Population. Die Gefahr frühzeitiger Konvergenz ist hier besonders groß. Wie in Abschnitt 5.3.2 noch näher gezeigt werden wird, hat es sich für die Modellpartitionierung

als sinnvoll erwiesen, die Kommunikationsparameter so zu wählen, daß die einzelnen Subpopulationen weitestgehend unabhängig voneinander den Raum absuchen, aber dennoch von Verbesserungen der anderen Populationen profitieren.

3.2.2. Formale Darstellung von Parallelen Evolutionären Algorithmen

Die in Abschnitt 3.1.3 und [1] eingeführte formale Darstellung läßt sich in Anlehnung an das *CP-Modell* [8] auf Parallele Evolutionäre Algorithmen erweitern.

Definition 3 *$\widehat{EA}$ sei ein EA nach Definition 1 mit folgender Erweiterung:*

$$\begin{aligned} &\xi, \rho \in \Omega \\ &\xi : I^\lambda \to I^\lambda \times I^k \\ &\rho : I^\lambda \times I^k \to I^\lambda \mid k \in \mathbb{N} \quad . \end{aligned} \tag{3.13}$$

Die in Definition 3 eingeführten Operatoren ξ und ρ werden im Folgenden die Migranten aus einer Population erzeugen, bzw. diese in die Population integrieren. Die Zahl der migrierenden Individuen wird durch k festgelegt.

Definition 4 (Paralleler Evolutionärer Algorithmus) *Ein Paralleler Evolutionärer Algorithmus (PEA) ist definiert als ein Tupel:*

$$PEA = (\widehat{EA}^\kappa, K) \quad . \tag{3.14}$$

Dabei ist $\widehat{EA}^\kappa$ die Menge von κ Evolutionären Algorithmen, entsprechend der Definition 3. K ist ein globaler Genetischer Operator:

$$K : (I^\lambda \times I^k)^\kappa \to (I^\lambda \times I^k)^\kappa \quad , \tag{3.15}$$

der über allen Algorithmen synchron arbeitet, die Bewegung der Individuen realisiert und Bestandteil der Übergangsfunktion Ψ ist.

$$\begin{aligned} \Psi_1 &= s_1 \circ \rho_1 \circ \quad K \circ \xi_1 \circ \omega_{(1,\Theta_1)} \circ \ldots \circ \omega_{(1,\Theta_z)} \circ \omega_{(1,\Theta_0)} \\ &\ldots \qquad K \quad \ldots \\ &\ldots \qquad K \quad \ldots \\ \Psi_\kappa &= s_\kappa \circ \rho_\kappa \circ \quad K \circ \xi_\kappa \circ \omega_{(\kappa,\Theta_1)} \circ \ldots \circ \omega_{(\kappa,\Theta_z)} \circ \omega_{(\kappa,\Theta_0)} \end{aligned} \tag{3.16}$$

Die Übergangsfunktion Ψ kann wie folgt vereinfacht werden:

$$\begin{aligned} Sei\, K &= (\mathfrak{k}_1, ..., \mathfrak{k}_\kappa)^\dagger \mid \mathfrak{k}_j : I^\lambda \times I^k \to I^\lambda \times I^k \\ \Psi(\mathfrak{P}_i) &= s_{i,\Theta_s}(\mathfrak{P}_i \cup \rho_i(\mathfrak{k}_i(\xi_i(\mathfrak{C}_i)))) \mid 1 \leq i \leq \kappa \quad . \end{aligned} \tag{3.17}$$

Ausgeschrieben läßt sich der Übergang von $\mathfrak{P}_i(t)$ nach $\mathfrak{P}_i(t+1)$ wie folgt formulieren:

$$\begin{aligned} \mathfrak{C}_i &= m_i(r_i(\mathfrak{P}_i(t))) \quad mit: \\ (\mathfrak{C}_i, C_i^k) &= \xi_i(\mathfrak{C}_i) && C_i^k \in I^k \wedge C_i^k \subset \mathfrak{C}_i \\ (\mathfrak{C}_i, \hat{C}_i^k) &= \mathfrak{k}_i((\mathfrak{C}_i, C_i^k)) && \hat{C}_i^k \in I^k \\ \hat{\mathfrak{C}}_i &= \rho_i((\mathfrak{C}_i, \hat{C}_i^k)) && k \in \mathbb{N} \mid 0 < k \leq \lambda \\ \mathfrak{P}_i(t+1) &= s_i(\mathfrak{P}_i \cup \hat{\mathfrak{C}}_i) \end{aligned} \tag{3.18}$$

(3.19)

Der Austausch von k Individuen pro Generation, erfolgt mittels der Operatoren $\xi, \mathfrak{k}, \rho$. ξ selektiert aus der Menge der Kinder $\mathfrak{C}_i$ die Menge der Migranten C_i^k. $\mathfrak{k}$ verteilt die Migranten aller Populationen. Aus Sicht eines einzelnen EA erzeugt $\mathfrak{k}$ $\hat{C}_i^k$ aus C_i^k, kann also bedingt als sequentieller Operator aufgefaßt werden. ρ vereint die Menge $\mathfrak{C}_i$ und $\hat{C}_i^k$ wieder, so das mit dem sequentiellen Evolutionären Algorithmus fortgefahren werden kann. Die Migration der Individuen kann über beliebigen Topologien erfolgen. So ist es möglich, daß jede Population mit allen anderen kommuniziert, oder aber eine Population nur mit bestimmten Populationen Individuen austauscht (siehe auch Abschnitt 4.2.1 und Abbildung A.2).

[†] Genaugenommen ist diese Annahme falsch, da sich K als Globaler (paralleler) Operator nicht in lokale (sequentielle) Operatoren zerlegen läßt. Eine korrekte Schreibweise würde außer der Exaktheit keine Vorteile bieten, im Gegenteil, das Modell würde wesentlich komplexer und unübersichtlicher.
Aus Sicht eines EA arbeitet $\mathfrak{k}$ wie ein sequentieller genetischer Operator, der einen Teil der Population verwirft und neue Individuen erzeugt.

4. Effiziente Implementierung von Evolutionären Algorithmen

Zur Umsetzung Evolutionärer Algorithmen für die Modellpartitionierung wurde das Programm pga entwickelt. In diesem Kapitel soll auf einige interessante Teilprobleme bei der Implementierung von pga eingegangen werden.

Der Modellpartitionierungsprozeß zur Vorbereitung der parallelen Logiksimulation ist ein zusätzlicher Zeitfaktor, welcher bei der sequentiellen Logiksimulation vollständig wegfällt. Da Partitionierung und Logiksimulation industriell eingesetzt werden, ist es besonders wichtig die Partitionierungszeiten gering zu halten. Für pga wurden einige spezielle Methoden entwickelt, um die Laufzeit zu verringern.

Die Ausgangsbasis für die effiziente Implementierung bildet das Pointerkonzept der Programmiersprache C. C ist eine Programmiersprache, die dem Entwickler einen sehr guten Kompromiß zwischen Programmierkomfort (und damit geringer Entwicklungszeit) und Anwendungsperformance bietet. Das Pointerkonzept von C ermöglicht es, mit Referenzen von Daten zu arbeiten, statt die Daten im Hauptspeicher zu kopieren. Dies spart bei sehr großen Datenmengen extrem viel Rechenzeit. Viele Algorithmen lassen sich hinsichtlich Rechenzeit oder Speicherbedarf optimieren. Angesichts der großzügigen Speicherressourcen moderner Computer wurde bei pga auf Kosten des Speichers ausschließlich auf Rechenzeit optimiert.

4.1. Optimierung von sequentiellen Evolutionären Algorithmen

4.1.1. Mutation

Die Mutation wird durch die Mutationsrate p_{mut} spezifiziert, die angibt, mit welcher Wahrscheinlichkeit ein Gen verändert wird. Eine mögliche Umsetzung wird in Algorithmus 4 beschrieben. Negativ wirken sich die zwei Berechnungen der Zufallszahlen pro Zyklus in Kom-

Algorithmus 4 Standard Mutationsalgorithmus

Vorbedingung: $0 \geq p_{mut} \geq 1$
  n_{gen}- Anzahl der Gene pro Individuum
  for all $\vec{p}_i(j) : i \leq \lambda \wedge j \leq n_{gen}$ **do**
    $rnd = random(MAX)$ {produziert eine Zufallszahl zwischen 0 und MAX}
    if $(p_{mut} * MAX) \geq rnd$ **then**
      $rnd = random()$
      mutiere $\vec{p}_i(j)$ in Abhängigkeit von *rnd*
    end if
  end for

bination mit der hohen Zyklenzahl aus. Die Zeitkomplexität C_t dieses Algorithmus beträgt:

$$C_t = 2\lambda * n_{gen} * RND \quad , \tag{4.1}$$

wobei *RND* die Zeit für die Bestimmung einer Zufallszahl ist. Für jedes Gen eines jeden Individuums müssen zwei Zufallszahlen ermittelt werden. Die Bestimmung von Zufallszahlen ist im Vergleich zu anderen Operationen relativ aufwendig. Pro Generation werden bei diesem Algorithmus sehr viele erzeugt, so daß besonders bei hohen Individuenanzahlen oder Chromosomenlängen viel Rechenzeit verbraucht wird.

Unter der Annahme, daß bei einer großen Zahl von Individuen oder Genen, die Zahl der durchschnittlich mutierten Gene konstant ist, läßt sich ein günstigerer Algorithmus implementieren. Anhand der Mutationsrate und der Gesamtzahl von Genen, läßt sich die Zahl der durchzuführenden Mutationen vorher berechnen. Anschließend werden dann zufällig Gene ermittelt, die mutiert werden. Die Einschränkung, daß in jeder Generation die gleiche Zahl von Genen mutiert wird, hat in der Praxis keine negativen Auswirkungen.

Algorithmus 5 Optimierter Mutationsalgorithmus

Vorbedingung: $0 \geq p_{mut} \geq 1$
  n_{gen}- Anzahl der Gene pro Individuum
  $mut_{total} = p_{mut} * \lambda * n_{gen}$
  for $i = 1$ *to* mut_{total} **do**
    $idv = random()$ {bestimme zufällig Individuum}
    $gen = random()$ {bestimme zufällig Gen}
    $rnd = random()$
    mutiere $\vec{p_{idv}}(gen)$ in Abhängigkeit von *rnd*
  end for

Die Komplexität für den optimierten Algorithmus beträgt:

$$C_t^{opt} = 3\lambda * p_{mut} * n_{gen} * RND \quad . \tag{4.2}$$

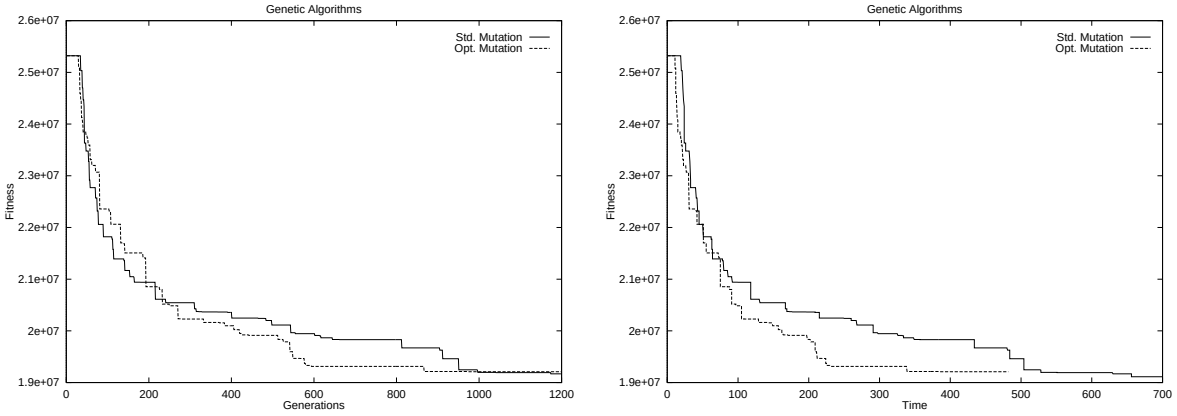

Abbildung 4.1.: Fitneßkurve in Abhängigkeit der Generationen (links) und der Zeit (rechts). Es ist deutlich zu erkennen, daß es keine wesentlichen qualitativen Unterschiede zwischen beiden Verfahren gibt, der optimierte Algorithmus aber schneller ist.

$$\frac{C_t^{opt}}{C_t} = \frac{3}{2p_{mut}} \tag{4.3}$$

Da $p_{mut} \ll 1$ kann die Rechenzeit extrem[1] verringert werden. Abbildung 4.1 zeigt, daß mit dem optimierten Mutationsalgorithmus bis zu 30% Rechenzeit gespart werden kann.

4.1.2. Individuenverwaltung

Ein sehr großer Teil der Rechenzeit wird von der Berechnung der Fitneß verbraucht. Im Falle von pga waren dies bis zu 90%. Wenn sich die Fitneßfunktion nicht beschleunigen oder vereinfachen läßt, so kann aber dafür gesorgt werden, daß jedes Individuum nur einmal evaluiert wird. Die Standard-EA evaluieren meist jedes Eltern- und Kindindividuum in jeder Generation neu, obwohl für die Eltern die Fitneß bereits einmal berechnet wurde. Die Verwaltung der Individuen erfordert komplexe Speicherstrukturen und Sicherheit im Umgang mit Pointern. Das nachfolgend vorgestellte Konzept verbessert die Laufzeit auf drei verschiedenen Ebenen.

- Jedes Individuum wird nur einmal evaluiert.
- Das Kopieren von Speicherbereichen wird auf das Nötigste beschränkt.
- Die Zeitkomplexität bei der Verwaltung der Individuen kann gesenkt werden.

[1] Bei einer Mutationsrate von $p_{mut} = \frac{1}{3000}$, ist die optimierte Methode ca. 2000 mal schneller. Bei einem normalen pga -Lauf liegt die Zeitersparnis im Minutenbereich.

Algorithmus 6 Individuenverwaltung

Vorbedingung: $0 < \mu \leq \lambda$

Sei $0 \leq \nu \leq \lambda$[†]
Sei $\mathfrak{P}$ Menge aller Eltern,
$\mathfrak{C}$ Menge aller Kinder und
$\mathfrak{T}$ Menge aller selektierten Individuen

$t = 0$
$el = \nu$
initialisiere $\mathfrak{P}(0) = \{\vec{p}_1(0), ...\vec{p}_\mu(0)\} \in I^\mu$
evaluiere $\mathfrak{P}(0) : \{\mathfrak{F}(\vec{p}_1(0)), ..., \mathfrak{F}(\vec{p}_\mu(0))\}$
sortiere $\mathfrak{P}(0)$ *nach Fitneß*
while $\iota(\mathfrak{P}(t)) \neq true$ **do**
  $el = \nu$
  $\mathfrak{C} = m(r(\mathfrak{P}(t)))$
  evaluiere $\mathfrak{C} : \{\mathfrak{F}(\vec{c}_1), ..., \mathfrak{F}(\vec{c}_\lambda)\}$
  sortiere $\mathfrak{C}$ *nach Fitneß*
  $p = 1$
  $c = 1$
  for $k = 1$ to μ **do**
    if $el \geq 0$ **then**
      $el = el - 1$
      if $\mathfrak{F}(\vec{p}_p) < \mathfrak{F}(\vec{c}_c)$ **then**
        $\vec{t}_k \Longleftrightarrow \vec{p}_p$ {vertausche Eltern- mit Temp-Individuum}
        $\mathfrak{F}(\vec{t}_k) \Longleftrightarrow \mathfrak{F}(\vec{p}_p)$ {vertausche Fitneß}
        $p = p + 1$
      end if
    else {Kind ist besser als Elternindividuum oder Elternindividuum ist keine Elite mehr ($el < 0$)}
      $\vec{t}_k \Longleftrightarrow \vec{c}_c$ {vertausche Kind mit Temp-Individuum}
      $\mathfrak{F}(\vec{t}_k) \Longleftrightarrow \mathfrak{F}(\vec{c}_c)$ {vertausche Fitneß}
      $c = c + 1$
    end if
  end for
  $\mathfrak{P} \Longleftrightarrow \mathfrak{T}$ {vertausche Menge $\mathfrak{P}$ mit $\mathfrak{T}$}
  $\mathfrak{F}(\mathfrak{P}) \Longleftrightarrow \mathfrak{F}(\mathfrak{T})$ {vertausche Fitneß der Menge $\mathfrak{P}$ mit Fitneß von $\mathfrak{T}$}
  $t = t + 1$
end while

[†] Zahl der als Elite übernommenen Eltern (siehe $(\mu * \lambda)_\nu$-Strategie in Abschnitt 3.1.2)

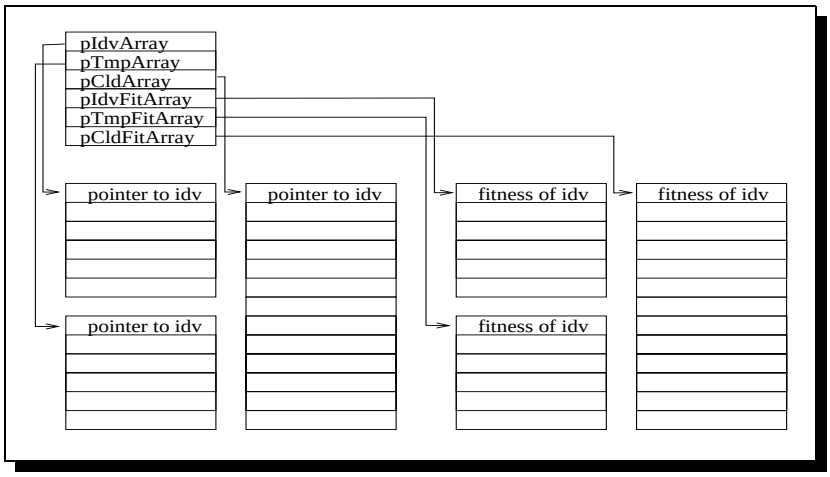

Abbildung 4.2.: Individuenverwaltung: Memorystruktur
Über 6 Pointer werden 6 verschiedene Felder verwaltet. `pIdvArray`, `pCldArray`, `pTmpArray` verweisen jeweils auf ein Feld von Pointern, die wiederum ein Individuum referenzieren. `pFitArray`, `pFitArray`, `pFitArray` weisen auf die zugehörigen Fitneßwerte.

Der Algorithmus 6 geht von der in Abbildung 4.2 dargestellten Speicherstruktur aus. Eine Besonderheit dieses Algorithmusses ist, daß alle Individuen und Kinder nach ihrer Fitneß sortiert werden. Dieser Mehraufwand bringt bei der Selektion entscheidende Vorteile. Um die nächste Generation zu erstellen, werden entsprechend ihrer Fitneß und der Elitewerte die Pointer der Eltern- und Kindindividuen in ein temporäres Array kopiert. Wichtig ist, daß nur die Pointer vertauscht werden und keine Speicherinhalte verschoben werden. Weil Eltern und Kinder jeweils in sortierter Form vorliegen, kann das Durchsuchen der Arrays sehr einfach und effektiv gehalten werden. So lange Eltern als Elite in die nächste Generation übernommen werden dürfen, werden Eltern und Kinder miteinander verglichen und der Bessere wird übernommen. Unabhängig davon, ob Eltern- oder Kindindividuum übernommen wurde, wird der Wert, der angibt, wie viele Individuen noch als Elite übernommen werden dürfen dekrementiert. Darf keine Elite mehr übernommen werden, wird die neue Generation mit den besten, der noch vorhandenen Kinder aufgefüllt. Gleichzeitig mit dem Aufbau des temporären Individuenarrays wird analog eine temporäre Fitneßtabelle generiert. Zum Schluß werden nur die Pointer, die auf die temporären Arrays zeigen mit den Pointern für die Individuen- und Fitneßtabelle vertauscht, so daß das temporäre Individuenarray zu dem Individuenarray wird und umgekehrt. Das gleiche geschieht mit den Fitneßtabellen. Somit wurden nur Pointer vertauscht und kein einziger Speicherbereich physikalisch verschoben. Beim Aufbau des temporären Individuenarrays müssen die Pointer unbedingt vertauscht werden, um die Integrität des allozierten Speichers zu gewährleisten, ein einfaches Kopieren in die Temporärtabelle reicht nicht aus. Bei dem Vertauschen werden die alten Eltern- und Kindtabellen zerstört. Da aber die Kindtabelle bei der Replikation komplett überschrieben wird und die Elterntabelle zur Temporärtabelle und somit in der nächsten Generation neu erstellt wird, ist dies nicht von Relevanz.

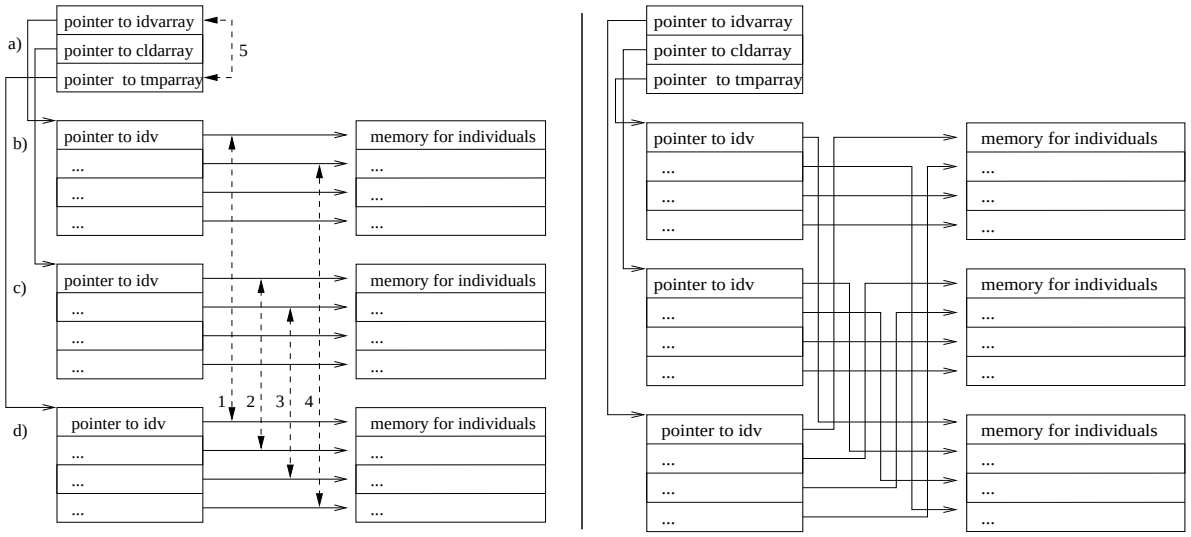

Abbildung 4.3.: Umsetzung der Selektion auf Speicherebene. *a)* verwaltet die Pointer auf die Pointertabellen. Die gestrichelten Pfeile mit nebenstehender Nummer zeigen die Reihenfolge für eine mögliche Vertauschung von Pointern. Beim Ergebnis der Vertauschung (rechts) ist wichtig, daß das Array für die Individuen *b)* mit dem Temporärfeld *d)* vertauscht worden ist und alle selektierten Pointer auf andere Speicherbereiche zeigen. Nur die Tabelle *c)*, welche die Pointer auf die Kinder verwaltet bleibt gleich.

Die Zeitkomplexität für diesen Algorithmus setzt sich aus der Zeit für das Sortieren[2] und die Zeit für das Auswählen der geeigneten Individuen zusammen. Im Folgenden sei M die Komplexität für das Vertauschen zweier Pointer und C die für einen Vergleich. ν ist die Zahl, der als Elite kopierten Eltern. Die Komplexität für Algorithmus 6 läßt sich wie folgt abschätzen:

$$\begin{aligned} C_t &= O(sort) + O(selekt) \\ &= \lambda \log(\lambda)(C+M)) + \nu C + \mu M \quad . \end{aligned} \tag{4.4}$$

Es gilt $M \approx C$.

Sei $\lambda = 3\mu = 6\nu$.

$$\begin{aligned} C_t &= \lambda \log(\lambda) * 2M + \frac{\lambda}{6}M + \frac{\lambda}{3}M \\ &= M(2\lambda \log(\lambda) + \frac{\lambda}{2}) \end{aligned} \tag{4.5}$$

$$C_t = O(\lambda \log(\lambda)) \tag{4.6}$$

[2] Der Aufwand für Quicksort beträgt $O(n \log(n))$

Dabei ist M (move) die Zeit, um zwei Variablen zu vertauschen, und C (compare) die, für den Vergleich von zwei Variablen.

Verzichtet man auf das Sortieren, so erhöht sich der Aufwand bei der Selektion. Um alle μ Individuen zu selektieren, müssen ν mal alle Kinder und Eltern miteinander verglichen werden und $\mu - \nu$ mal alle Kinder. Hinzu kommen μ Verschiebungen und μ Vergleiche, die notwendig sind um festzustellen, ob ein bestimmtes Individuum bereits selektiert worden ist. Die Komplexität beträgt:

$$C_t^{alt} = \nu(\mu+\lambda) * C + (\mu-\nu)\lambda * C + \mu M + \mu C \quad .$$

Es gilt $M \approx C \quad .$

Sei $\lambda = 3\mu = 6\nu \quad .$

$$C_t^{alt} = M(\frac{2}{9}\lambda^2 + \frac{5}{6}\lambda) \tag{4.7}$$

$$C_t^{alt} = O(\lambda^2) \tag{4.8}$$

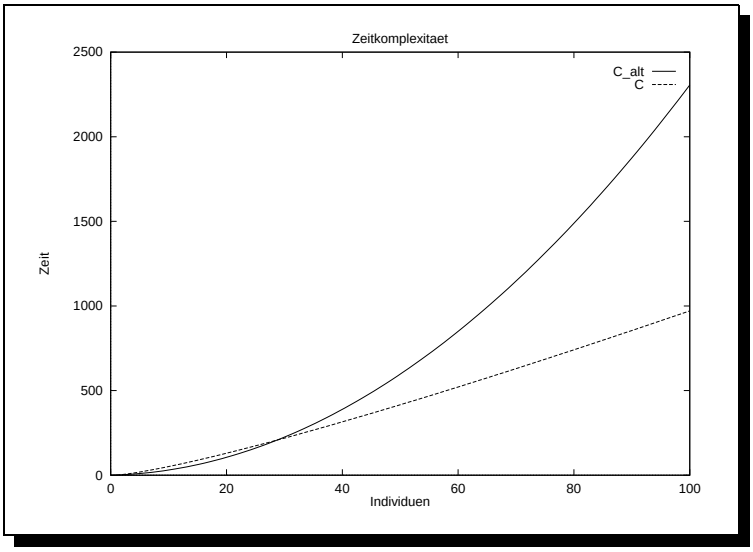

Abbildung 4.4.: In der graphischen Darstellung der Gleichungen 4.5 und 4.7 kann man den Vorteil der optimierten Individuenverwaltung (mit Vorsortierung) (C) gegenüber der normalen (C_alt) sofort erkennen.

Es ist leicht zu erkennen, daß C_t besser als C_t^{alt} ist. Abbildung 4.4 zeigt, daß ab ca. 30 Kinder der Algorithmus 6 schneller ist, als der Algorithmus ohne Sortieren, bei 100 Kindern ist er schon mehr als doppelt so schnell.

4.2. Implementierung von Parallelen Evolutionären Algorithmen

4.2.1. Das Kommunikationskonzept von pga

Die Effizienz eines parallelen Programmes hängt stark von einem sorgfältigen Entwurf des Kommunikationskonzeptes ab. Das Kommunikationskonzept von pga wurde unter zwei Gesichtspunkten entworfen:

1. möglichst flexible Konfiguration zur Laufzeit,
2. robustes Verhalten bei Ausführung auf unterschiedlich schnellen und stark belasteten Rechnernetzen.

Punkt 1 wird durch die Tatsache motiviert, daß vor der Implementierung nicht bekannt war, welche Kommunikationsstrukturen und -parameter zu akzeptablen Ergebnissen führen werden. Um nicht ständig Änderungen im Quelltext vorzunehmen und anderen Nutzern die Möglichkeit zum Experimentieren zu geben, wurde versucht die Konfiguration über Parameter beim Programmaufruf zu ermöglichen. Die Motivation für Punkt 2 liegt in der Tatsache begründet, daß pga vor allem auf Clustern mit unterschiedlich schnellen oder verschieden stark belasteten Knoten zum Einsatz kommen wird. Um zu verhindern, daß langsame Knoten die Gesamtlaufzeit verschlechtern, wurde das Konzept der “Lazy Parallelisation” entworfen.

Wichtige Parameter für das Verhalten von PEA sind:

Kommunikationsstruktur bestimmt die virtuelle Topologie der Populationen und legt damit explizit fest, welche Populationen miteinander kommunizieren.

Kommunikationsdynamik beschreibt die Veränderung des Kommunikationsverhaltens im Laufe der Evolution.

Migrationsrate bestimmt, wie oft Individuen ausgetauscht werden.

Migrationsstärke gibt an, wie viele Individuen migrieren.

Variable Einstellungsmöglichkeiten für Migrationsrate, Migrationsstärke und Kommunikationsdynamik sind einfach zu implementieren. Anspruchsvoll hingegen ist die Programmierung für variable Kommunikationsstrukturen. Abbildung A.2 zeigt einige von pga unterstützte Topologien. Jede Kommunikationsstruktur läßt sich als Graph darstellen, der in eine $\kappa \times \kappa$ Matrix abgebildet werden kann, wobei κ die Zahl der Knoten ist. Sei $A = \begin{pmatrix} a_{0,0} & \dots & a_{0,\kappa} \\ \dots & \dots & \dots \\ a_{\kappa,0} & \dots & a_{\kappa,\kappa} \end{pmatrix}$, dann gibt $a_{i,j}$ die Zahl der Individuen an, die von Population i zur Population j migrieren werden (Migrationsstärke).

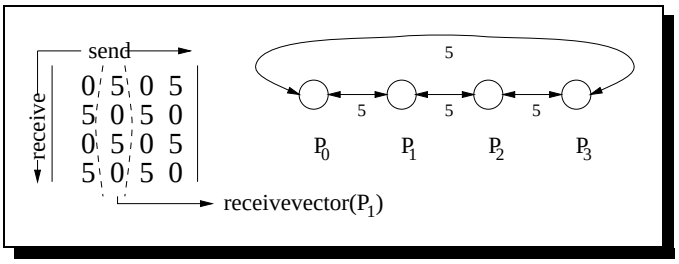

Abbildung 4.5.: Kommunikationsmatrix für den abgebildeten Graphen. Aus den Spalten können die Receivevektoren und aus den Zeilen die Sendvektoren für jede einzelne Population gewonnen werden.

Aus dieser globalen Kommunikationsmatrix kann jede Population die zugehörigen Send- und Receivevektoren ablesen und somit sofort entscheiden, wieviel Individuen zu welcher Population geschickt werden müssen und wieviel Individuen man empfangen muß.

Synchrone Kommunikation auf unterschiedlich schnellen Rechnern hat den Nachteil, daß der schnellere Prozeß an jeder Kommunikationsbarriere auf den langsameren warten muß. Betrachtet man zum Beispiel ein Cluster von 8 gleichschnellen Knoten, bei denen 1 Knoten durch einen Fremdprozeß zusätzlich belastet ist, so ist dieser belastete Knoten nur noch halb so schnell wie die anderen. Diese müssen dann bei gleicher Last jeweils 50% der Rechenzeit auf den langsamsten Knoten warten, was wiederum bedeutet, daß nahezu 50% der Gesamtrechenzeit mit Warten verloren gehen. Das hier vorgestellte Prinzip der Lazy Parallelisation geht von der Annahme aus, daß es für einen PEA zu keiner oder nur zu einer geringfügigen Verschlechterung führt, wenn einige Kommunikationen nicht ausgeführt werden, weil der Zielknoten noch nicht zur Kommunikation bereit ist.

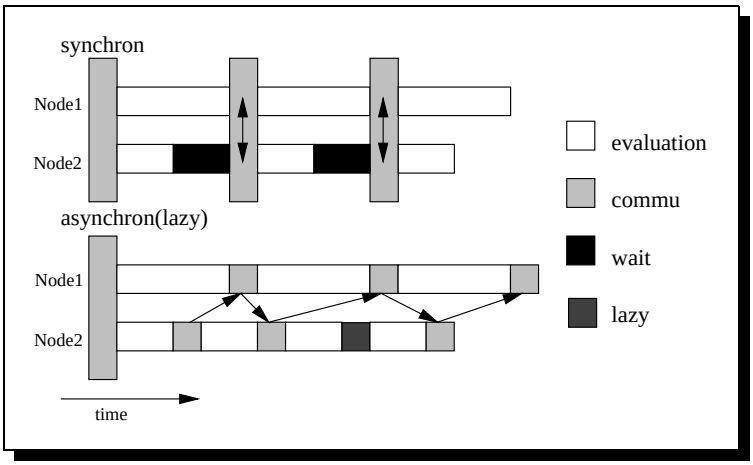

Abbildung 4.6.: Kommunikationsarten: synchron im Vergleich zu lazy

MPI stellt mit den Nonblocking Send- und Receivefunktionen einen Mechanismus zur Verfügung, der es ermöglicht, eine Kommunikation einzuleiten, aber im Programm fortzufah-

Algorithmus 7 Lazy Communication

Vorbedingung: $myid \wedge comm\_to \neq myid$
  $success = nonbl\_receive\_from(comm\_to, \&Idvbuffer)$
  if $success = TRUE$ **then**
    $integrate\_idv(\&Idvbuffer)$
  end if
  if $last\_comm\_to(comm\_to) = TRUE$ **then**
    $nbl\_send\_to(comm\_to, \&Sendbuffer)$
  end if

ren und nicht auf den Abschluß der Kommunikation warten zu müssen. Mit diesen Methoden läßt sich der Algorithmus für die Lazy Parallelisation formulieren (siehe Algorithmus 7). Der Befehl zum Empfangen von Individuen, fragt nur ab, ob diese schon empfangen wurden, wartet aber nicht auf den Erfolg, sondern gibt die Kontrolle an das Programm zurück. Bevor man Individuen versendet, fragt man den Erfolg der letzten Kommunikation ab. Ist dieser positiv, so können neue Individuen verschickt werden, wieder ohne auf den Erfolg der Kommunikation zu warten. Ist die letzte Kommunikation hingegen noch nicht abgeschlossen, so verwirft man die für die Migration vorgesehenen Individuen und fährt im Programm ohne Kommunikation fort.

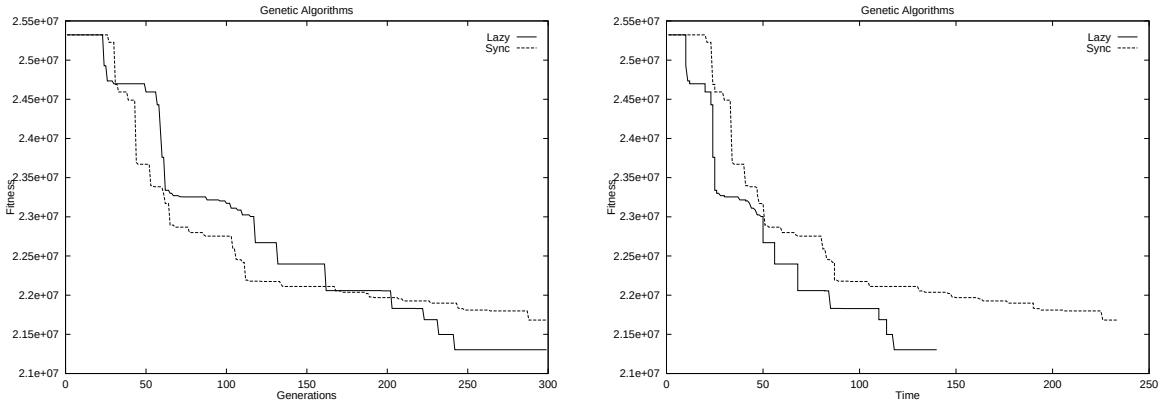

Abbildung 4.7.: Fitneßkurven von Synchroner und Lazy Parallelisierung auf einem belasteten System. Bei gleicher Qualität ist Lazy Parallelisation schneller.

Abbildung 4.6 stellt Synchrone und Lazy Parallelisierung auf 2 Knoten gegenüber. Einem Knoten werden erst wieder Individuen zugeschickt, wenn die vorherige Kommunikation erfolgreich abgeschlossen wurde. Experimente haben bewiesen, daß es zu keiner Verschlechterung im Verhalten des PEA kommt, wenn asynchrone Kommunikationsmethoden eingesetzt wurden (siehe Abb. 4.7).

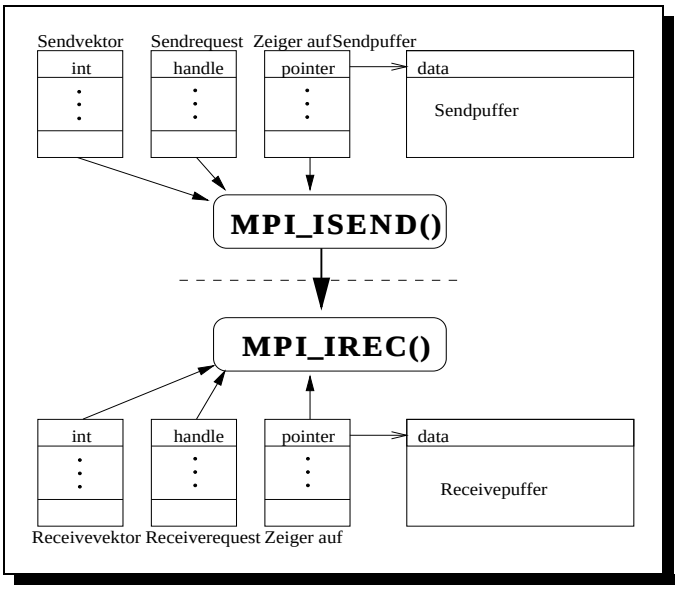

Abbildung 4.8.: Kommunikationstabellen

In der praktischen Umsetzung muß pga 6 Tabellen verwalten (Abb.4.8). Zum einen die Pointertabellen für den Send- und Receivepuffer, die Tabellen für die Send- und Receiverequests, über die der Erfolg der letzten Kommunikation abgefragt werden kann, und die Send/Receivevektoren. Algorithmus 4.2.1 zeigt die praktische Umsetzung des Lazy- Send in C-Code.

Algorithmus 8 C-Funktion: pga_send_idv()

```
int pga_send_idv(){
 int i,flag=0,size;
 for(i=0;i<popNr;i++){
  if(sendVec[i]!=0)
   if(((genertion % rateVec[i])==0)||(sendFlagV[i]!=0)){
    if(sendFlagV[i]==0){       /**keine anderes Send zu diesem Partner*/
     size=pga_build_send_buf(i);/**erzeuge Sendpuffer*/
     MPI_Isend(ppsendBuf[i],size,MPI_INT,i,TAG_IDV,\
       MPI_COMM_WORLD,&psendReq[i]);/**rufe Nonblocking Sendprozess auf*/
     sendFlagV[i]=1;}    /**Markiere Send zu diesem Partner*/
    if(sendFlagV[i]!=0){ /**anderes Send zu diesem Knoten*/
     MPI_Test(&psendReq[i], &flag, &status);
     if (flag != 0) sendFlagV[i]=0;/**Wenn Kommu ok, Loesche Markierung*/
    }}}
 return(0);}
```

4.3. Problemspezifische genetische Operatoren

Neben den allgemeinen genetischen Operatoren Mutation und Crossing-Over ist es oft sinnvoll, auch Operatoren einzusetzen, die mit einem speziellen Modellwissen arbeiten. Problemspezifische Operatoren bieten die Möglichkeit, eventuell vorhandenes Wissen über den Suchraum in die EA zu integrieren. Allerdings sollte man in jedem Fall überprüfen, ob das vorhandene Wissen nicht ausreicht, einen deterministischen Optimierungsalgorithmus zu realisieren, da diese meist bessere Resultate erbringen, als EA. Im Folgenden werden einige Operatoren vorgestellt, die für die Modellpartitionierung zur parallelen Logiksimulation entworfen oder angepaßt wurden.

4.3.1. Inversion und Flipping

Genaugenommen zählt Inversion zu den Standardoperatoren Genetischer Algorithmen. Es wird aber in diesem Abschnitt vorgestellt, da der Nutzen sehr stark problemabhängig ist. Flipping von Chromosomenstücken kann sinnvoll sein, weil es bei der betrachteten Modellpartitionierung nur wichtig ist, daß bestimmte Supercones zusammen auf einem Knoten evaluiert werden, aber nicht auf welchem. Sowohl Inversion, als auch Flipping wählen zufällig ein Chromosomenteilstück, zufälliger Länge und verändern dies. Während Inversion das gewählte Teilstück umdreht und wieder einfügt, verändert Flipping den Inhalt der einzelnen Gene (siehe Abb. 4.9). Wenn g_{min}, g_{max} der kleinste bzw. größte Wert ist, den ein Gen annehmen kann, so berechnet sich der neue Inhalt eines Genes durch $g_{neu}^{flp}(i) = g_{min} + g_{max} - g_{alt}(i)$.

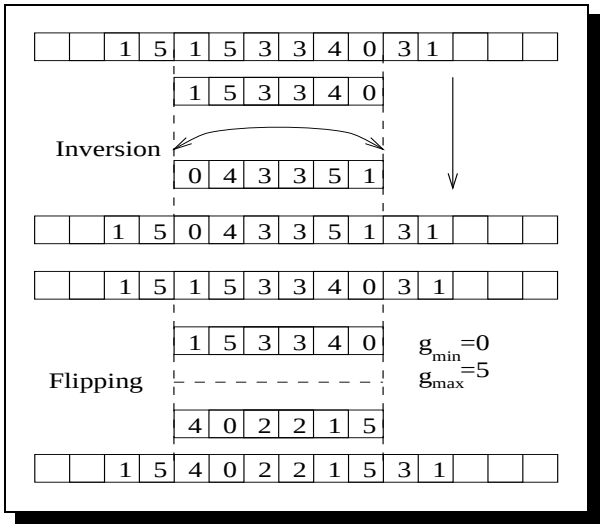

Abbildung 4.9.: Funktionsweise von Inversion und Flipping

In einem frühen Entwicklungsstadium von pga hatte sich herausgestellt, daß Inversion und Flipping einen Teil der Mutation ersetzen können. Mit der Fertigstellung von pga und der

darin implementierten Operatoren kommt Inversion und Flipping praktisch keine Bedeutung mehr zu. Zwar verschlechtern beide das Endergebnis nicht, aber bei optimaler Mutation sind sie überflüssig.

4.3.2. Alpha-Move

Alpha-Move ist ein deterministischer Operator, welcher auf zufällig ausgewählte Individuen angewandt wird und im Rahmen dieser Diplomarbeit entwickelt wurde. Grundlage für Alpha-Move ist die Tatsache, daß die einzelnen Blockzeiten eines Individuums stark variieren können. Bei Partitionen mit größerer Blockanzahlen ist es häufig, daß sich Blockzeiten um den Faktor 10 unterscheiden. Der Grundgedanke von Alpha-Move ist, dem langsamsten Block explizit Rechenlast zu entziehen und diese dem bisher schnellsten Block zuzuteilen. Sei t_e^{max}, t_e^{min} die Evaluierungszeit des langsamsten, bzw. schnellsten Blocks. Unter der Annahme, die Simulationszeit eines Blockes (Blockzeit) besteht ausschließlich aus der durch die Last verursachte Evaluierungszeit, müßte die Last l_e^{mv} verschoben werden, die der Hälfte der Differenz der Blockzeiten:

$$l_e^{mv} \hat{=} t_e^{mv} = \frac{t_e^{max} - t_e^{min}}{2} \tag{4.9}$$

entspricht. Dabei soll Last vorerst als eine Menge von abstrakten Objekten betrachtet werden, welche einen Evaluierungsaufwand und somit Evaluierungszeit verursachen. Nach dieser Verschiebung haben unter unserer Annahme beide beteiligten Blöcke dieselbe Simulationszeit.

Allerdings setzt sich die Blockzeit aus Evaluierungszeit und Kommunikationszeit zusammen. Letztere kann sich bei geringer Modifikation der Blockzusammensetzung drastisch verändern. Somit ist es wahrscheinlich, daß nach einer Lastverschiebung der bisher schnellste Block zum langsamsten Block wird, weil sich zusätzlich zur Evaluierungszeit die Kommunikationszeit erhöht. Die Änderung der Kommunikationszeit läßt sich nicht abschätzen, sondern nur mit einer Neuberechnung der Fitneßfunktion bestimmen. Dies würde die Ausführungszeit von Alpha-Move aber zu stark erhöhen. Daher wurde der Ansatz gewählt, nicht die Hälfte der Blockzeitdifferenz zu verschieben, sondern einen durch $0 < \alpha \leq 1$ gewichteten Teil davon. Somit muß die Last verschoben werden, die der mit α gewichteten Zeit $\hat{t}_e^{mv}$ entspricht:

$$l_e^{mv} \hat{=} \hat{t}_e^{mv} = \alpha t_e^{mv} = \alpha \frac{t_e^{max} - t_e^{min}}{2} \,. \tag{4.10}$$

Setzt man voraus, daß die Last auf einem Block homogen verteilt ist, und somit jede Lasteinheit die gleiche Rechenzeit verursacht, so gilt:

$$\frac{t_e^{mv}}{l_e^{mv}} = \frac{t_e^{max}}{l_e^{max}} \,. \tag{4.11}$$

Aus 4.10 und 4.11 folgt:

$$l_e^{mv} = \alpha \frac{\frac{t_e^{max} - t_e^{min}}{2}}{t_e^{max}} l_e^{max} \quad . \tag{4.12}$$

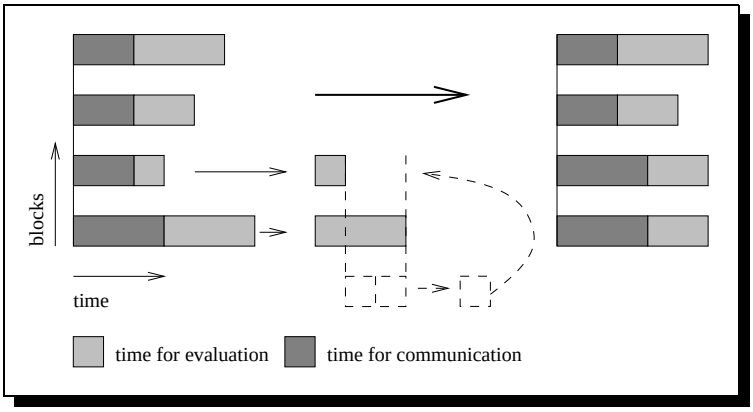

Abbildung 4.10.: Alpha-Move

Obwohl verschiedene Supercones unterschiedliche Evaluierungszeiten verursachen, wird unter dem Aspekt einer schnellen Laufzeit l_e^{max} gleich der Anzahl, der auf dem langsamsten Block evaluierten Supercones gesetzt. Somit entspricht l_e^{mv} der Zahl der Supercones, die verschoben werden müssen.

Eine genaue Untersuchung der Auswirkungen von Alpha-Move hat das Ergebnis gebracht, daß Individuen, die durch Alpha-Move verbessert wurden, kaum Potential für weitere Verbesserungen während der Evolution mehr hatten und von nicht modifizierten Individuen verdrängt wurden. Dies ist vor allem darauf zurückzuführen, daß sich bei einer Verschiebung die Kommunikationslast der betroffenen Blöcke stark ändert, und weil verschiedene Supercones unterschiedliche Rechenlast erzeugen.

4.3.3. Komplexe Fitneßfunktion

Da bei einer parallelen Simulation der langsamste Block die Gesamtlaufzeit bestimmt, berücksichtigt die herkömmliche Fitneßfunktion ausschließlich das Maximum, aller Blockzeiten eines Individuums. Der Nachteil dieser Funktion ist, daß die anderen Blockzeiten völlig wegfallen. Sind die Blockzeiten der anderen Individuen ähnlich groß wie das Maximum, so ist abzusehen, daß dieses Individuum sich nicht mehr viel verbessern wird. Abbildung 4.11 zeigt zwei Individuen mit der gleichen Fitneß, wobei aber die linke Partition deutlich mehr Potential bei einer weiteren Evolution besitzt, als die rechte.

Sei t_i die Blockzeit des Blockes i, so kann die herkömmliche Fitneßfunktion geschrieben werden, als :

$$f_1(\vec{p}) = max_i(t_i(\vec{p})) \; . \tag{4.13}$$

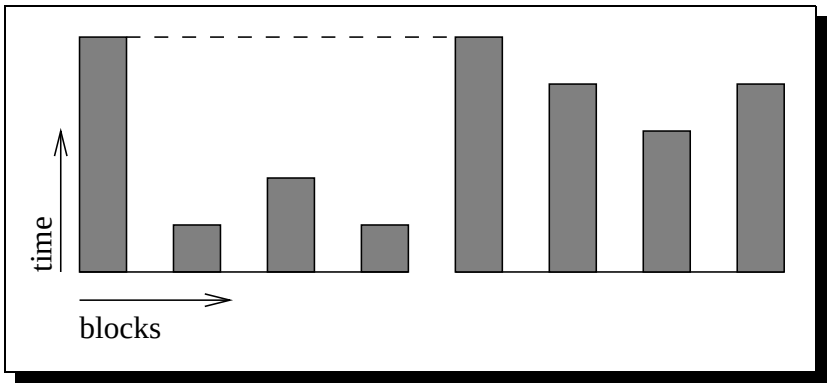

Abbildung 4.11.: Blockzeiten zweier Individuen

Eine Funktion, die die Blockzeiten aller Blöcke berücksichtigt, ist:

$$f_2(\vec{p}) = \sqrt{\sum_{i=0}^{n} t_i^2(\vec{p})}\,. \tag{4.14}$$

Erstaunlich ist, daß EA mit f_2 als Fitneßfunktion gleichwertige Ergebnisse erreichen, wie mit f_1. Kombiniert man beide Funktionen zu einer komplexen Fitneßfunktion, so ist es mit dieser möglich, EA zu realisieren, die nicht nur bessere Ergebnisse produzieren, sondern diese auch viel schneller erreichen. Sei g_{max} die Zahl der maximalen und g die der aktuellen Generationen. Es hat sich folgende Fitneßfunktion als günstig erwiesen:

$$f_3(\vec{p}) = \frac{g_{max}}{10\,g} f_2(\vec{p}) + f_1(\vec{p})\,. \tag{4.15}$$

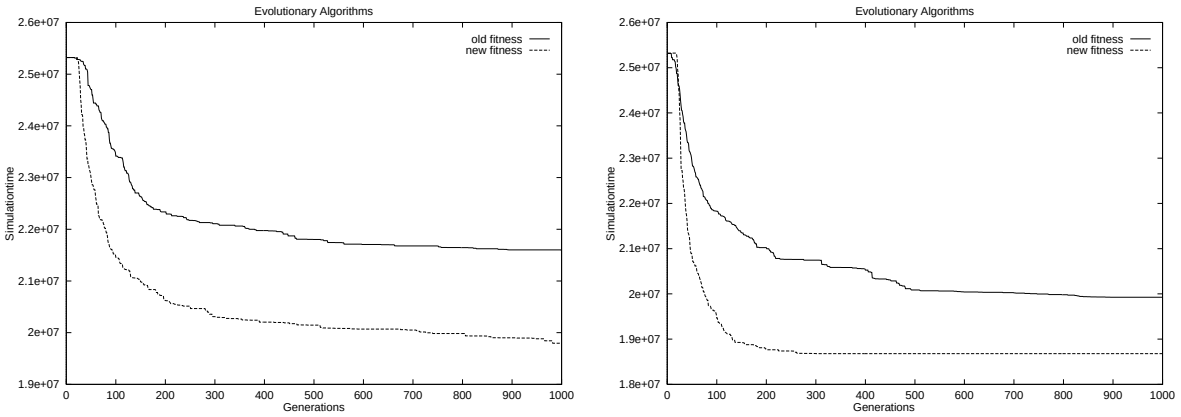

Abbildung 4.12.: Vergleich zwischen f_1 und f_3 für sequentielle EA (links) und den parallelen EA(rechts).

Abbildung 4.12 zeigt einen Vergleich zwischen f_1 und f_3. Bei f_3 ist deutlich der schnelle Abfall der Fitneß zu sehen. Dies hat zur Folge, daß bei Läufen von PEA mit f_3 nach 300 bis 500 Generationen kaum eine Verbesserung mehr möglich ist. Somit können PEA-Läufe früher beendet werden.

Die Funktion f_3 hat sehr interessante Auswirkungen auf die Funktionsweise anderer Operatoren. Prinzipiell kann festgestellt werden, daß die komplexe Fitneßfunktion die Evolutionären Algorithmen von konkreten Parametereinstellungen unabhängiger macht und einige Operatoren ihren Einfluß auf die EA ganz verlieren[3].

So hat zum Beispiel die Zahl der als Elite übernommenen Individuen keine Auswirkungen mehr auf die Evolution (siehe Abbildung 4.13). Erweist sich bei EA mit f_1 noch ein Elitewert von 20% als sinnvoll, so spielt er bei EA mit f_3 keine Rolle mehr. Dies liegt darin begründet, daß f_2 durch den Faktor $\frac{g_{max}}{10g}$ mit den fortlaufenden Generationen abgekühlt wird. Somit gewinnt zum einen f_1 immer mehr an Einfluß, zum anderen wird dadurch das gleiche Individuum zu einem späteren Zeitpunkt besser bewertet, als am Anfang der Evolution. Dieses Abklingen der Fitneßfunktion führt implizit ein Alter der Generationen ein, wobei jüngere Individuen bevorzugt werden. Untersuchungen haben gezeigt, daß Individuen höchstens 4-5 Generationen überleben. Weiter konnte untermauert werden, daß es die Kombination des Generationenalters und der komplexen Fitneßfunktion ist, welche den Qualitätssprung ermöglicht. Trennt man beide Faktoren[4], so findet zwar immer noch eine Leistungssteigerung statt, doch ist diese bei weitem nicht so groß.

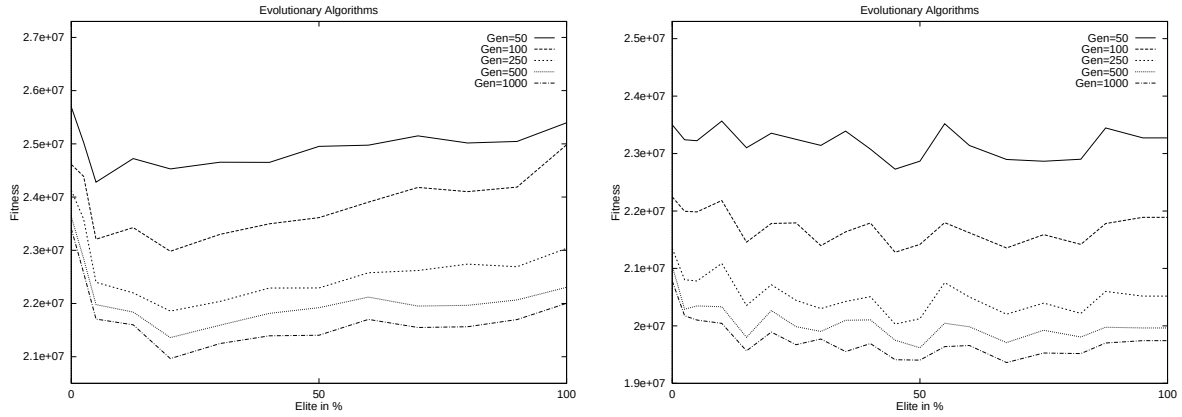

Abbildung 4.13.: Die Fitneß in Abhängigkeit der Elite. Links mit f_1 und rechts mit f_3 als Fitneßfunktion.

[3] Beispiele dafür sind u.a. Inversion,Flipping und Alpha-Move

[4] Das Generationenalter kann ausgeschaltet werden, indem man in jeder Generation die Fitneß aller Individuen neu berechnet. Andersrum läßt sich das Generationenalter unabhängig von f_3 einführen, indem man f_1 mit der Zeit abkühlt.

5. Experimentelle Ergebnisse

Parallel zur Entwicklung von pga wurde der Einfluß verschiedener Parameter auf das Verhalten von EA untersucht. Dazu mußten sehr umfangreiche Tests durchgeführt werden. In diesem Kapitel sollen die Testszenarien und die Ergebnisse dieser Tests vorgestellt werden.

Das Problem bei der Untersuchung der PEA ist, daß es außer dem Fitneßverlauf kein praktikables Kriterium über Güte und Verlauf Evolutionärer Algorithmen gibt. Alle bekannten theoretischen Abschätzungen über Parametereinstellungen von Genetischen Operatoren, das Verhalten von Populationen im Suchraum und dergleichen beruhen entweder auf starken Vereinfachungen, sind abstrakter Natur oder setzen Wissen über den Suchraum voraus. Da im Falle der Modellpartitionierung kein konkretes Wissen über den Suchraum vorlag, sind diese Kriterien und Abschätzungen hier wertlos. Die Untersuchung der PEA für die Modellpartitionierung konnte demnach nur empirisch erfolgen.

5.1. Testszenarien

Das Ergebnis Evolutionärer Algorithmen ist stark vom Zufall abhängig. Wird der Zufallsgenerator mit unterschiedlichen Werten initialisiert, ist es möglich, daß die Ergebnisse sehr stark voneinander abweichen.

Um die Zufallskomponente weitestgehend auszuschalten, wurden bei den sequentiellen EA jeweils 16 Programme mit denselben Parametern ausgeführt, die sich nur in der Initialisierung der Zufallsgeneratoren unterschieden. Mit Hilfe von PERL-Scripten wurde der Durchschnitt aller Läufe berechnet, sowie diverse andere Transformationen durchgeführt. Die statistische Mittelung der Meßergebnisse ermöglicht es, konkrete Aussagen über die Auswirkungen von Parametern zu treffen. Man muß allerdings beachten, daß pro Parametereinstellung aus Zeitgründen[1] nur 16 verschiedene Versuche durchgeführt werden konnten und die Ergebnisse somit noch immer statistisch verrauscht sind. Trotzdem ist es möglich, für konkrete Parameter optimale Werte zu ermitteln. Als günstig hat sich die Darstellung der Fitneß in Abhängigkeit eines Parameters nach einer bestimmten Zahl von Generationen gezeigt (siehe Abbildung 5.1 rechts.). Bei Parallelen Evolutionären Algorithmen hat sich gezeigt, daß die Parallelität ei-

[1] Zur Ermittlung der in diesem Kapitel aufgeführten Daten wurden ca. 8000 Testläufe á 10 min durchgeführt. Zeitweise rechneten 112 SP2-Knoten an diesen Problemen.

ne weitgehende Unabhängigkeit gegenüber der Initialisierung des Zufallsgenerators bewirkt. Aus diesem Grund wurde bei parallelen Läufen meistens nur ein Lauf durchgeführt. Alle elementaren Parameter, wie die Anzahl von Individuen, Mutationsrate, Crosspoints, wurden bei sequentiellen EA untersucht, um deren Auswirkung auf EA besser isolieren zu können. Wie später noch gezeigt wird, treten bei PEA Synergieeffekte auf, die die Ergebnisse von PEA verbessern, aber auch robuster gegenüber der konkreten Parameterwahl werden lassen. Falls nicht anders angegeben beziehen sich die angegebenen Werte auf das Monet-Prozessormodell mit 250 Supercones, Vorpartitionierung mit STEP und Startpartitionierung mit MOCC [11]. Dieses Modell beschreibt einen kompletten IBM 390 Prozessor und entspricht am ehesten den relevanten Einsatzgebieten für TEXSIM/MVLSIM. Alle Fitneßwerte sind vorhergesagte Simulationszeiten pro Takt in μs oder ns.

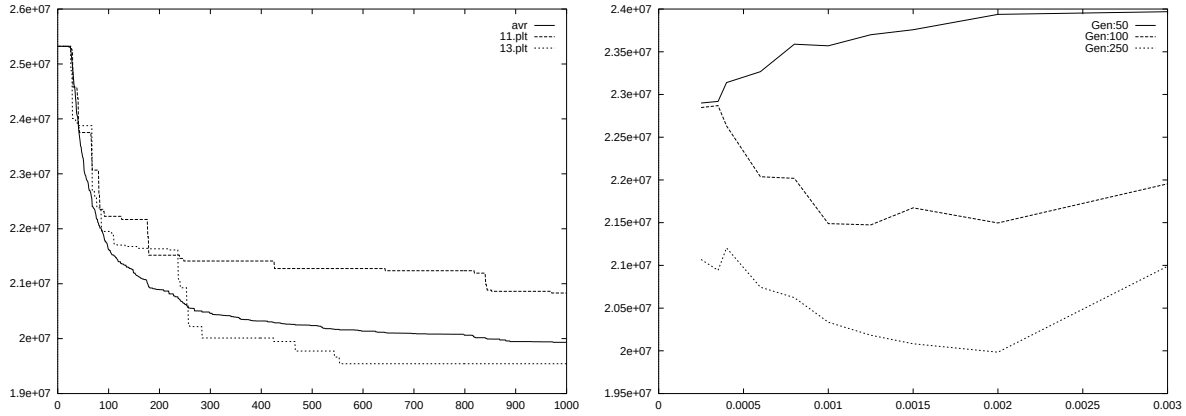

Abbildung 5.1.: Beispieldiagramme:
Links: Durchschnittliche Fitneß im Vergleich zu 2 von 16 tatsächlichen Fitneßverläufen
Rechts: Durchschnittliche Fitneß in Abhängigkeit der Mutationsrate nach 50, 100 und 250 Generationen.

5.2. Parameter bei sequentiellen Evolutionären Algorithmen

5.2.1. Verhältnis von Individuen und Nachwuchs

Mit der Wahl der Anzahl von Individuen und Kindern, wird maßgeblich Einfluß auf das Verhalten eines Evolutionären Algorithmus genommen. Die Zahl der Kinder bestimmt die Explorationsrate des Algorithmus. Je größer diese Zahl, desto größer ist die Wahrscheinlichkeit,

daß sich die Population verbessert. Weil aber jedes neu erzeugte Individuum durch die Fitneßfunktion bewertet werden muß, verursacht eine hohe Zahl von Nachwuchs eine längere Rechenzeit.

Die Zahl der Individuen bestimmt das Potential von möglichen Lösungen, die in die nächste Generation übernommen werden, oder anders ausgedrückt, wieviele Individuen verworfen werden müssen. Das Verhältnis von Kindern zu Eltern wird als Selektionsdruck bezeichnet. Ist dieser zu gering, findet keine Evolution statt. Ist der Selektionsdruck zu hoch, leidet die Konvergenzgeschwindigkeit. Dieser Zusammenhang wird in Abbildung 5.2 graphisch dargestellt. Für kurze Läufe empfiehlt sich ein Verhältnis von 3:1 -5:1. Bei längeren Läufen ist dieses Verhältnis nicht ganz so kritisch und kann von 2:1 bis 10:1 reichen.

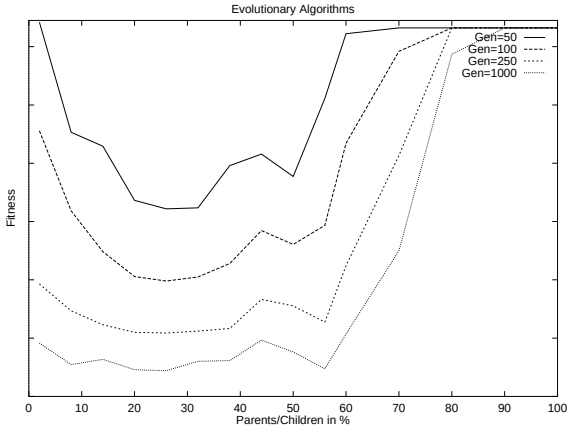

Abbildung 5.2.: Die durchschnittliche Fitneß in Abhängigkeit des inversen Selektionsdruckes in %.

Abbildung 5.3 verdeutlicht den Zusammenhang zwischen Zahl der evaluierten Kinder und der Rechenzeit. Während das Verhältnis Fitneß zu Generationen klar für hohe Kinder- und Individuenzahlen spricht, zeigt die Darstellung in Abhängigkeit der realen Rechenzeit, daß es bei kürzeren Läufen sinnvoller ist, mehr Generationen mit weniger Kindern zu evaluieren. Für Läufe, die unabhängig der Laufzeit nach der besten Lösung suchen, ist eine hohe Zahl von Individuen und Kindern geeigneter.

5.2.2. Mutationsrate

Unter dem Begriff Mutation sind mehr oder weniger drastische Änderungen einzelner Gene zusammengefaßt. Das Programm pga unterstützt zwei Arten der Mutation.

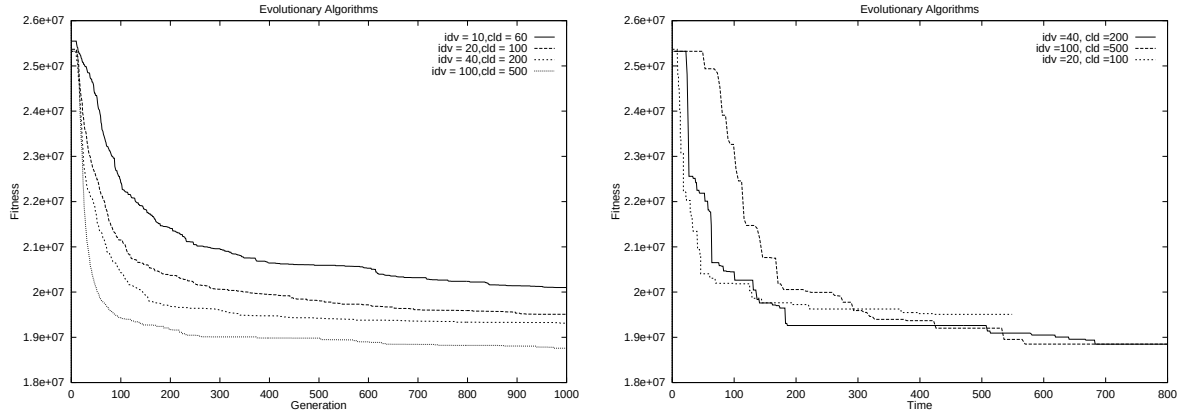

Abbildung 5.3.: Die Fitneß in Abhängigkeit der Generationen (links) und der Zeit (rechts).

1. Zufällige Veränderung des Genes m: $\vec{p}_i[m] = rnd(maxGen)$
2. Veränderung des Genes durch Inkrementierung/Dekrementierung: $\vec{p}_i[m] = \vec{p}_i[m] \pm 1$

In beiden Fällen wird das zu mutierende Gen zufällig ermittelt. Weil es bei pga nur wichtig ist, welche Supercones gemeinsam auf einen Knoten evaluiert werden, aber nicht auf welchem, ist der konkrete Wert eines Genes nicht relevant, sondern nur, wie viele und welche Gene den gleichen Wert haben. Somit gibt es keine Nachbarschaftsbeziehung zwischen den Werten einzelner Gene. Daher hat sich die erste Mutationsmethode als günstiger erwiesen.

Proto_Supercones	Kinder	optimale Mutationsrate
Monet_250	20	0.0027
Monet_250	40	0.0025
Monet_250	100	0.0023
Monet_250	500	0.0025
Monet_500	100	0.0018
Picasso_250	40	0.002
Picasso_250	100	0.0022

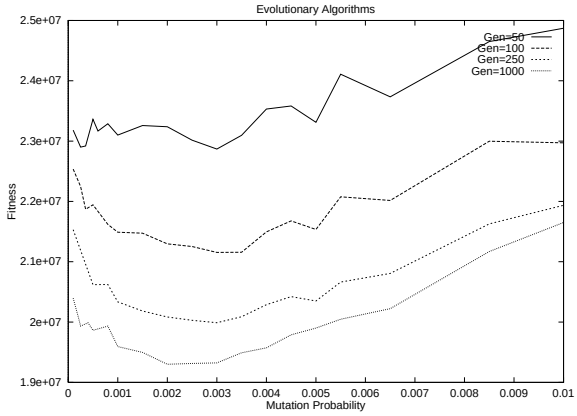

Abbildung 5.4.: Die durchschnittliche Fitneß in Abhängigkeit der Mutationsrate. Links die optimalen Mutationsraten für verschiedene Protos für 500 Generationen.

Ein Maß für die Stärke der Änderungen, die pro Generation durchgeführt werden, ist die Mutationsrate. Ist sie zu groß, werden die Änderungen zu stark und die Wahrscheinlichkeit auf eine Verbesserung sinkt rapide. Ist die Mutationsrate zu klein, sinkt die Konvergenzgeschwindigkeit des EA. In beiden Fällen werden die Ergebnisse des EA hinter den Möglichkeiten zurückbleiben. Abbildung 5.4 zeigt die optimalen Mutationsraten[2]. Dem rechten Bild ist zu entnehmen, daß bei wenig Generationen (50) die Mutation eine untergeordnete Rolle spielt. Ab 100 Generationen kristallisiert sich ein Optimum um 0.0025 heraus.

5.2.3. Crossing-Over

Mit der Anzahl der Crosspoints kann festgelegt werden, wie oft zwei Individuen rekombiniert werden. Experimente haben gezeigt, daß 1-Point-Crossing-Over die schlechtesten Ergebnisse erzeugt. Ein klares Optimum existiert, wenn das Crossing-Over nicht benutzt wird (0 Crosspoints). In diesem Fall findet keine Durchmischung der Population statt. Jedes Individuum erkundet allein den Suchraum. Das Ausschalten des Crossing-Over erweist sich aber bei PEA als ungünstig (siehe Abschnitt5.3.1). Wie in Abbildung 5.5 zu sehen, können bei längeren Läufen gute Resultate auch mit 4-7 Crosspoints erzielt werden.

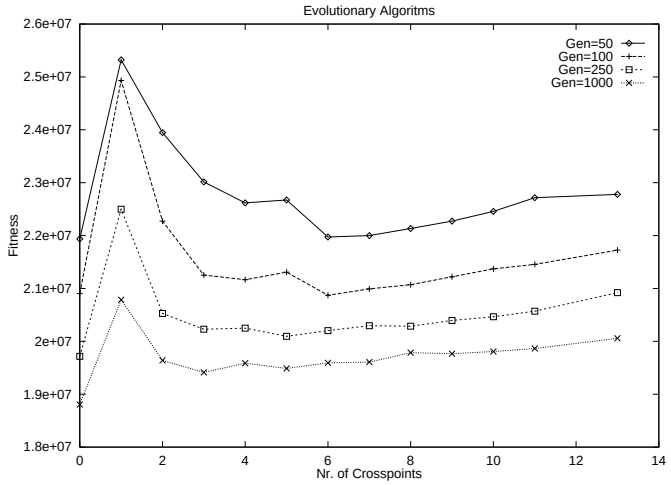

Abbildung 5.5.: Die durchschnittliche Fitneß in Abhängigkeit der Crosspoints.

[2]Die Werte wurden empirisch aus je 16×15 Testläufen ermittelt.

5.3. Parameter bei Parallelen Evolutionären Algorithmen

5.3.1. Unterschiede zu sequentiellen Evolutionären Algorithmen

Prinzipiell können die optimalen Parameter von den sequentiellen EA übernommen werden. Eine Ausnahme bildet das Crossing-Over. Ist es bei den EA noch günstig, das Crossing ganz auszuschalten, empfiehlt sich bei pga eine Zahl der Crosspoints von 2-8. Je weniger Generationen evaluiert werden sollen, desto größeren Einfluß hat das Crossing (siehe Abbildung 5.6 links).

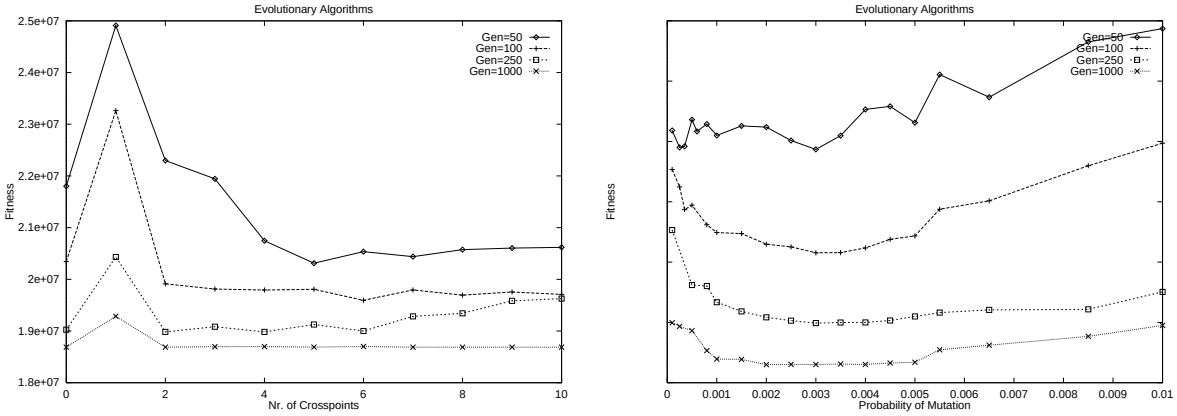

Abbildung 5.6.: pga : Durchschnittliche Fitneß:
Links: in Abhängigkeit der Crosspoints
Rechts: in Abhängigkeit der Mutationsrate

Markant für pga ist die Robustheit gegenüber Parametereinstellungen. In Abbildung 5.6 kann man sehen, daß der Bereich, innerhalb dessen optimale Ergebnisse erzielt werden, mit fortschreiten der Generationen immer größer wird. Mutationsraten von 0.001-0.005 erzeugen nach 1000 Generationen Ergebnisse nahezu gleicher Güte. Somit ist bei pga die Wahl der optimalen Parameter bei weitem nicht so kritisch zu sehen, wie bei ga .

5.3.2. Kommunikationsstruktur

Die Untersuchungen des Kommunikationsverhaltens haben gezeigt, optimale Ergebnisse werden vor allem dann erreicht, wenn die Kommunikationsparameter so gewählt wurden, daß jede einzelne Population weitestgehend unabhängig arbeiten kann.

Struktur	Populationen	Fitneß nach x Generationen in μs				
		50	100	250	500	1000
Shift	8	20823	19111	18712	18668	18668
Shift	16	20884	19968	19249	18992	18684
dShift	8	20904	19503	18983	18849	18624
dShift	16	20799	19787	19301	18707	18691
Matrix	8	20473	19407	19023	18678	18678
Matrix	16	20304	19458	18776	18678	18678
dMatrix	8	20997	19463	19063	18705	18688
dMatrix	16	20678	19515	19003	18688	18678
Star	8	21763	20160	19538	18833	18755
Star	16	21711	20091	19125	18778	18720
dStar	8	21254	19848	19266	18743	18687
dStar	16	21010	19501	19246	18678	18678
alltoall	8	21520	20744	19902	*19037*	*18980*
alltoall	16	*21862*	*20958*	*20120*	18923	18878

Tabelle 5.1.: Fitneß für verschiedene Kommunikationsstrukturen.

Selbstverständlich muß es möglich sein, daß globale Verbesserungen alle Subpopulationen erreichen. Ist die Migration zwischen den Subpopulationen zu stark, so kann eine suboptimale Lösung das gesamte Suchverhalten beeinflussen und verhindern, daß eine andere Population ein besseres Ergebnis findet.

Wie den Tabellen 5.1 und 5.2 zu entnehmen ist, sollten Migrationsrate und Migrationsstärke eher gering gewählt werden. Dies bedeutet, es ist besser sehr oft (aller 1-5 Generationen) sehr wenige Individuen (1-5) auszutauschen, als seltener, große Individuenzahlen. Als Migrationstopologie empfiehlt sich eine Struktur mit geringem Vernetzungsgrad. Die Sterntopologie und vor allem die Alltoallverbindung erzeugen wesentlich schlechtere Ergebnisse als die anderen Strukturen.

5.4. Zusammenfassung der Experimentellen Ergebnisse

Die Experimente haben die in 3.2.1 getroffenen Behauptungen bestätigt. PEA führen sehr stabil zu sehr guten Optimierungsergebnissen und weisen eine große Toleranz gegenüber suboptimalen Parametereinstellungen auf. Die erreichten Ergebnisse sind durchgängig besser als die von sequentiellen EA und sie werden in wesentlich kürzerer Zeit erreicht (siehe Abbildung 5.7).

Migrations-	Migrationsrate						
stärke	1	5	10	25	50	75	100
50 Generationen							
1	20912	22457	22853	22460	22590	22590	22590
5	**20623**	22089	22502	22289	22590	22590	22590
10	20694	21767	22303	22518	22590	22590	22590
20	21756	22222	22293	22076	22590	22590	22590
250 Generationen							
1	19374	19643	19753	20060	20190	19800	19920
5	**19271**	19575	19821	19994	20057	20000	19924
10	19429	19344	19489	20017	20099	19960	20032
20	19406	19398	19393	19630	19875	19631	19836
500 Generationen							
1	19189	19389	19353	19530	19463	19448	19534
5	19181	19389	19353	19530	19463	19448	19534
10	19291	**18747**	19327	19214	19642	19569	19639
20	19386	18797	19364	19430	19398	19306	19391

Tabelle 5.2.: Fitneß für verschiedene Migrationsraten.

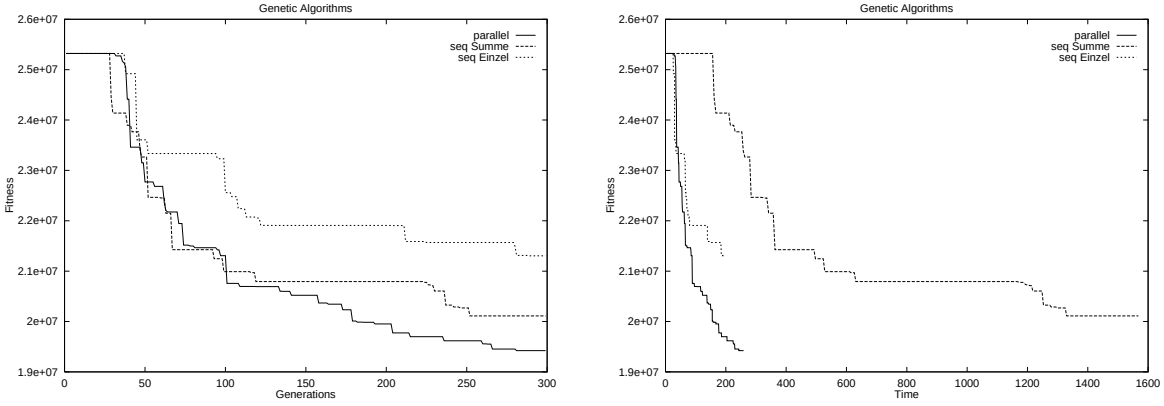

Abbildung 5.7.: Fitneß für PEA mit 8 Subpopulationen á 40 Eltern und 100 Kindern, EA mit 8*40 Eltern und 8*100 Kindern (seq.Summe) und EA mit 40 Eltern und 100 Kindern (seq.Einzel).

Ausblick

Aufbauend auf dieser Arbeit sind 3 weiterführende Arbeiten denkbar. Die Berechnung der Fitneß verbraucht den größten Teil der Rechenzeit. Eine Beschleunigung oder die Untersuchung anderer Möglichkeiten der Fitneßberechnung stellen eine lohnenswerte Aufgabe dar. Dabei wäre zu untersuchen, ob es sinnvoll ist, die Fitneß zu Beginn der Evolution nur sehr grob abzuschätzen, da in diesem Stadium eine genaue Simulationszeitvorhersage nicht nötig ist.

Für spezielle Anwendungen hat es sich als günstig erwiesen, EA mit deterministischen Optimierungsalgorithmen, wie Hill-Climbing oder Iterativen Algorithmen zu kombinieren. Es ist zu untersuchen, ob durch eine Kombination verschiedener Algorithmenklassen oder die Integration von deterministischen Algorithmen als genetische Operatoren in Evolutionäre Algorithmen bessere Modellpartitionen für die parallele Logiksimulation erzeugt werden können.

Der Mehrpopulationsansatz von pga bietet die Möglichkeit über Gruppen von Subpopulationen eine Metaevolution durchzuführen. Damit wäre es zum Beispiel möglich, die optimale Parametereinstellung für verschiedene Prozessormodelle zu ermitteln. Es ist sogar denkbar, mittels Genetischer Programmierung Operatoren zu generieren, und durch die Metaevolution zu optimieren.

Anhang

A. Programmdokumentation

A.1. Installation

Dieser Abschnitt soll eine Einführung in die Installation, Konfiguration und Benutzung von pga geben. Das Programm pga ermöglicht die Optimierung von Modellpartitionen für die parallele Logiksimulation mit Hilfe von parallelen Evolutionären Algorithmen. In pga ist ga enthalten, ein Programm, in dem sequentielle Evolutionäre Algorithmen realisiert sind. Das Programmpaket pga der Version 1.0 liegt als gepacktes tar-Archiv vor. Es wurde unter AIX4.1 sowie LINUX[1] entwickelt und getestet, sollte aber problemlos auf andere UNIX-Systeme portiert werden können, da der Autor bei der Entwicklung strikt den ANSI C und MPI Standard eingehalten hat. Die Installation ist denkbar einfach.

```
cd INSTALLDIR[2]
gunzip pga.tar.gz
tar xvf pga.tar
```

Die Verzeichnisstruktur nach Ausführung dieser Befehle wird in Abbildung A.1 gezeigt. Die

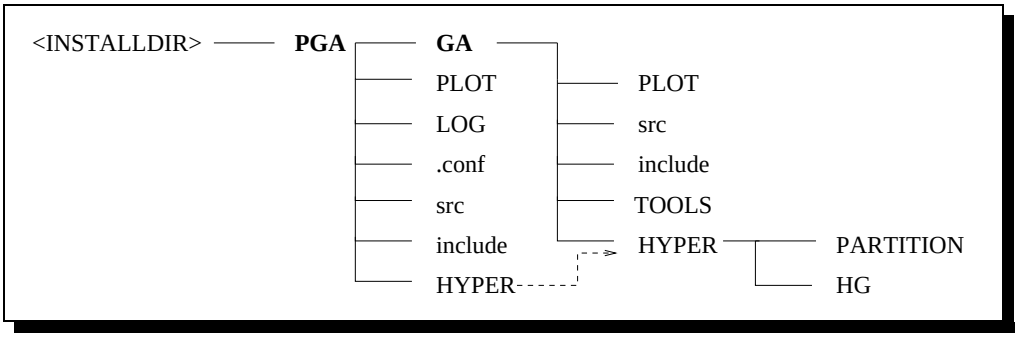

Abbildung A.1.: Verzeichnisstruktur von PGA nach dem Entpacken.

Directories `src` und `include` enthalten den Quellcode und die Includedateien für pga bzw.

[1] Kernel 2.32

[2] `INSTALLDIR` ist ein vom Nutzer gewähltes Verzeichnis, von dem aus pga installiert werden soll.

ga , PLOT die entstehenden Plotfiles (s. A.4). In .conf stehen die Konfigurationsdateien für die einzelnen Rechner bei parallelen Programmläufen. Unter HYPER/PARTITION werden die Startgenerationen gesucht und in HYPER/HG die Hypergraphdatei zur Fitneßberechnung.

Für die Benutzung von parallel ga muß eine MPI Kommunikationsbibliothek (Message Passing Interface) installiert sein, die dem Standard MPI 1.1 oder höher genügt[3]. Auch ist dafür zu sorgen, daß bei jedem parallelen Lauf die ausführbare Datei pga , die jeweilige Konfigurationsdatei und die benötigten Startpartitionen inklusive Hypergraph auf allen benutzten Computern in der selben Version zur Verfügung stehen. Am einfachsten ist dies realisierbar, indem pga auf einem Network File System (NFS, NFS+, AFS ...) installiert wird und so an alle beteiligten Rechner exportiert werden kann.

A.1.1. Die Kompilation des Quellcodes

Um die Programme speziell für das verwendete Computersystem zu optimieren, empfiehlt es sich, diese neu zu compilieren. Dazu adaptiert man in INSTALLDIR/PGA oder in INSTALLDIR/PGA/GA das Makefile, je nachdem, ob man die parallele oder sequentielle Version von pga benutzen möchte. Die Makefiles sind kommentiert, so das es mit geringem Aufwand möglich ist, die entsprechenden Änderungen vorzunehmen. Mit

```
make clean
make
rm *.o
```

werden die Programmdateien neu erstellt und überflüssige Dateien gelöscht. Das Programm läßt sich jetzt mit pga bzw. ga starten, auch wenn dies ohne vorherige Konfiguration der Parameter nicht sehr sinnvoll ist.

A.2. Konfiguration und Parameterwahl

Das Programmkonzept sieht für die Konfiguration von pga drei Möglichkeiten vor.

1. Im Quellcode (PGA/GA/include/defaults.h) ist es möglich, Defaulteinstellungen festzulegen. Zweck dieser Datei ist es, alle Parametervariablen auf einen definierten Wert zu initialisieren. Veränderungen an dieser Datei sollten nur mit größter Vorsicht erfolgen. Nach einer Veränderung muß der komplette Quellcode neu kompiliert werden.

[3] Die Software MPICH ist eine solche Bibliothek. Sie wurde am ARGONNE NATIONAL LABORATORY entwickelt und ist für nahezu alle UNIX-Systeme frei verfügbar. Weitere Informationen und Sourcen unter: ”http://www.mcs.anl.gov/mpi/”. Eine kommerzielle Implementierung des MPI ist das PARALLEL ENVIRONMENT von IBM.

2. Die Konfigurationsdateien `/PGA/GA/.ga` für `ga` und `/PGA/GA/.ga`, `PGA/.conf/.ga_0 .. .ga_n` für `pga` bieten die Möglichkeit, Grundeinstellungen vorzunehmen, die über mehrere Versuchsreihen hinweg gültig sind, wie z.B. Werte für den Hypergraphen, aber auch einzelne Läufe können so konfiguriert werden. Die hier getroffenen Einstellungen aktualisieren die in Punkt 1 festgelegten Werte. Bei `pga` überschreiben die Werte vom `.conf/.ga_n` die der Datei `.ga`. Es ist also sinnvoll, in `.ga` die globalen Einstellungen festzulegen und in den spezifischen Konfigurationsdateien spezielle Einstellungen für jeden Knoten.

3. `ga` und bedingt auch `pga`[4] können mit Kommandozeilenparametern aufgerufen werden, welche wiederum die in 1 und 2 getroffenen Einstellungen überschreiben.

```
# This is default config file for GA
# by Hendrik Schulze 1998
# every parameter needs a separate line
# commando -line parameter or parameter
# overides this parameter

#Max Nr. of generations
* gen=1000
#Nr. of individuen
* Idv=50
#Nr. of children
* Cld=100
```

Tabelle A.1.: Ausschnitt aus einer Konfigurationsdatei

Eine Übersicht über alle möglichen Parameter und deren Definitionsbereiche geben die Tabellen A.3 und A.4 auf den Seiten 56 und 57.

A.2.1. Die Konfigurationsdatei

`ga` und `pga` erwarten in ihrem Hauptverzeichnis die Datei `.ga`. Die darin zugewiesenen Werte werden nacheinander eingelesen und überschreiben die vorher gültigen Einstellungen. Soll `pga` auf n Knoten parallel ausgeführt werden, so existiert auf jedem Knoten eine Instanz von `pga` , die von 0 bis n-1 durchnumeriert sind. Die Reihenfolge wird in der Datei `host.list` festgelegt. Jede Instanz i ($0 \leq i < n$) erwartet eine Datei `.conf/.ga_i` , die analog zu `ga`

[4] nur in Zusammenarbeit mit MAP

Parameter	min	max	Typ	Bemerkung
gen	1	max[a]	int	Anz. der zu berechnenden Generationen
idv	1	max	int	Anzahl der Individuen
cld	idv	max	int	Anzahl des Nachwuchses
mut	0	1	float	Mutationsrate
eli	0	idv	int	Anz. der als Elite übernommenen Eltern in die nächste Generation
crs	0	19	int	Anzahl der Crosspoints
inv	0	1	float	Inversionsrate
flp	0	1	float	Flippingrate
maxinv	0	100	int	Maximale Inversionslänge in %
maxflp	0	100	int	Maximale Flippinglänge in %
hgf	—	—	str	Name der Hypergraphdatei
stp	—	—	str	Nameswurzel der Dateien mit den Startgenerationen
out	—	—	str	Verzeichnisname für PLOT-Dateien
minstp	1	max	int	min. Parameter für Startgenerationsdatei
maxstp	minstp	max	int	max. Parameter für Startgenerationsdatei
plt	0	1	int	1=PLOT-Datei erzeugen
log	0	1	int	1=LOG-Datei erzeugen (nur bei pga)
fit	0	1	int	0: Fitneß = Simulationszeit 1: Fitneß= komplexe Fitneßfunktion
avr	0	1	int	1=Durchschnittsfitneß an
msg	0	4	int	Message Level
crm	1	max	int	Anzahl der Gene
mincrm	-max	max	int	minimaler Wert, den ein Gen annehmen kann
maxcrm	mincrm	max	int	maximaler Wert, den ein Gen annehmen kann
modemut	0	2	int	Art der Mutation, 1: $gen_{neu} = rnd$ 2: $gen_{neu} = gen_{alt} \pm 1$ 0: zufällig Methode 1 oder 2
sed	0	max	int	Init. des Zufallsgenerators
sav	0	max	int	Anzahl der zu speichernden Dateien
ana_dst	0	max	int	Distance vom besten Individuum
alpha	0	1	float	Alpha für Alpha-Move
alpmv	0	1	float	Alpha-Movingrate
coolmut	0	2	float	Abkühlungsrate für Mutation wenn coolmut$> 1 \Rightarrow$ Erhitzung
minmut	0	1	float	Untere Grenze für Mutation bei Abkühlung
maxmut	0	1	float	Obere Grenze für Mutation bei Abkühlung

[a] 2147483647

Tabelle A.3.: Wichtige Parameter.

Parameter	min	max	Typ	Bemerkung
comm_ring	0	1	int	1=Kommunikation ringförmig
comm_intgr	0	1	int	1=migrierende Individuen werden integriert
comm_idvnr	0	max[a]	int	Zahl der migrierenden Individuen
comm_rate	0	max	int	Aller wieviel Generationen werden Individuen ausgetauscht
comm_fitrate	0	max	int	Aller wieviel Generationen werden Fitneßwerte ausgetauscht.
coolcomm	0	2	float	Abkühlungsrate für dynamische Kommunikationsabkühlung
comm_sigma	0	max	float	Sigma für Villmanabkühlung
comm_dynamic	-	-	str	Modus für Kommunikationsdynamik. (static, dynamic, villman)
comm_structure	-	-	str	Name für Kommunikationsstruktur (shift, dshift, star, dstar ,matrix, dmatrix, individual)

[a] 2147483647

Tabelle A.4.: Parameter für pga

eingelesen wird und deren Werte gegebenenfalls überschreibt. Tabelle A.1 zeigt den prinzipiellen Aufbau einer Konfigurationsdatei. Wichtig ist, daß Kommentare mit ”#” beginnen und Parametereinstellungen mit ”*”. Steht derselbe Parameter mehrmals in der Datei, so wird der zuletzt gültige Wert übernommen. Werden falsche Parameter in eingegeben, so versucht pga diese zu korrigieren und gibt eine Warnung aus, oder pga beendet sich mit einer Fehlermeldung.

A.2.2. Programmaufruf mit Parametern

Mit

```
ga -para1=wert1 -para2=wert2 ... -paraN=wertN
```

läßt sich ga mit Kommandozeilenparametern starten. Bei der Vielzahl der möglichen Parameter kann leicht der Überblick verloren gehen, weshalb empfohlen wird, die Konfigurationsdatei sinnvoll zu adaptieren und nur die davon abweichenden Eingabeparameter in der Kommandozeile zu übergeben. Wird pga in der parallelen Laufzeitumgebung PARALLELMAP gestartet (siehe A.5), so kann, mittels der darin implementierten Mechanismen, pga mit Parametern gestartet werden, die von der jeweiligen Konfigurationsdatei abweichen. Ansonsten verzichtet der Autor darauf, pga mittels Parameterübergabe konfigurierbar zu machen, da die Menge der möglichen Einstellungen sich um die Anzahl der benutzten Rechnerknoten vervielfacht und undurchschaubar würde.

Unter AIX[5] wird pga mittels des Aufrufs: "pga" gestartet[6]. Unter LINUX, mit einer Installation von MPICH erfolgt der Aufruf durch: "mpirun -np<knotenzahl> pga".[7]

A.3. Arbeiten mit pga

Im Folgenden wird auf die wichtigsten Arbeitsschritte bei der Benutzung von pga eingegangen. Besonderheiten in Zusammenarbeit mit MAP sind dem Abschnitt A.5 zu entnehmen.

A.3.1. Laden von Hypergraphen

Um die Fitneß korrekt zu berechnen, ist es notwendig den zu den Startpartitionen gehörenden Hypergraphen zu laden. pga erwartet diesen in dem Verzeichnis .../PGA/HYPER/HG bzw. in .../PGA/GA/HYPER/HG, welche per Symlink auch identisch sein dürfen. Bei der Ausführung von pga benötigt jede Instanz den gleichen Hypergraphen. Der Name der Hypergraphdatei wird mit dem Parameter -hgf <Dateiname> übergeben.

A.3.2. Laden von Startpartitionen

Das Programm pga ist in der Lage, eine Serie von Startpartitionen als Anfangspopulation zu laden. Die Namen dieser Dateien müssen folgender Konvention genügen: <Dateiname>*i*.string, wobei *i* eine laufende Nummer ist, in der sich die einzelnen Dateien unterscheiden. Die Dateien werden im dem Verzeichnis .../PGA/HYPER/PARTITION bzw. in .../PGA/GA/HYPER/PARTITION gesucht. Die zugehörigen Übergabeparameter sind -stp <Dateiname>, -minstp <min_Stp> und -maxstp <max_Stp>. min_Stp und max_Stp stehen für die kleinsten und größten Werte, die *i* annehmen kann. Stimmt die Anzahl der zu ladenden Partitionen nicht mit der Anzahl der Individuen überein, so werden die Partitionen zyklisch mehrfach geladen, bzw. einige unterdrückt. Werden keine Startpartitionen angegeben, so werden die Startpartitionen zufällig erzeugt. Da pga aus den geladenen Partitionen die Anzahl der Gene und der Blöcke berechnet, müssen bei einer zufälligen Initialisierung die Parameter für die Gen- und Blockanzahl[8] richtig gesetzt werden.

[5] Nur wenn PARALLEL ENVIRONMENT installiert ist.

[6] Zuvor muß die Systemvariable MP_PROCS auf die Zahl der gewünschten Knoten gesetzt werden und in der Datei .../PGA/host.list alle beteiligten Knoten eingetragen werden.

[7] Auch hier muß vorher die Datei machines.LINUX richtig konfiguriert sein.

[8] -crm, -mincrm, -maxcrm

A.3.3. Speichern

Mit `-sav <`*n*` >` kann man festlegen, wieviel der besten Individuen abgespeichert werden sollen. Die Individuen werden unter `...HYPER/PARTITION/` unter dem Namen `<Dateiname>GA`*i*`_`*Node*`.string` gesichert, wobei ”Dateiname” der Name ist, der beim Laden der Startpartitionen angegeben wurde. *i* ist eine laufende Nummer von 0 bis *n* -1. *Node* enthält die Nummer des jeweiligen Knotens, auch dem die `pga` -Instanz läuft. Im sequentiellen Fall entfällt *Node*. Wurde kein Name angegeben so wird unter `GA`*i*`_`*Node*`.string` gespeichert.

A.3.4. Sinnvolle Parametereinstellungen

Es ist sehr schwer, gute Parameter für Evolutionäre Algorithmen anzugeben, da diese sehr problemabhängig sind und auch bei gleichen Problemstellungen sehr stark in ihrer Güte schwanken. Im Folgenden werden die wichtigsten Parameter vorgestellt. Dazu ist meist ein Intervall angegeben, das den Bereich von sinnvollen Einstellungsmöglichkeiten vorgibt.

Individuen Die Anzahl der Individuen bestimmt im wesentlichen, wieviel Information von Generation zu Generation vererbt werden kann. `-idv (20-1000)`

Kinder Die Zahl der Kinder legt fest, wieviel neue Möglichkeiten pro Generation evaluiert werden können. Je mehr, desto besser. Allerdings hat die Zahl der Kinder fast alleinigen Einfluß auf den Rechenaufwand. Als günstig hat sich das zwei- bis zehnfache der Individuenanzahl herausgestellt. `-cld (40-6000)`

Generationen Hier legt man fest, wieviel Generationen berechnet werden sollen. Dieser Wert hängt sehr stark von der Zahl der Kinder ab und der Rechenzeit, die einem zur Verfügung steht. `-gen (50-5000)`

Fitneß Es gibt die Möglichkeit zwischen zwei verschiedenen Fitneßfunktionen umzuschalten (siehe Abschnitt 4.3.3).
`-fit` 0 : f_1 Fitneß=Simulationszeit.
`-fit` 1 : f_3 Fitneß=Komplexe Fitneßfunktion nach Gleichung 4.15.

Elite Da nicht gewährleistet ist, daß der Nachwuchs einer jeden Generation bessere Ergebnisse als die Elterngeneration hervorbringt, ist es sinnvoll eine gewisse Anzahl von Eltern in die nächste Generation zu übernehmen, wenn diese besser sind, als der Nachwuchs. Mit der Benutzung von f_3 als Fitneßfunktion (`-fit 1`), wird dieser Wert überflüssig (siehe Abschnitt 4.3.3). `-eli (4-idv)`, mit idv - Zahl der Individuen

Crossing Die Zahl der Cross-Points beim Crossing-Over läßt sich mit `-crs (0-10)` festlegen.

Mutation Bestimmt die Wahrscheinlichkeit, mit der jedes einzelne Gen eines Chromosomens verändert (mutiert) wird. Dieser Wert sollte nicht zu hoch liegen. `-mut (0.0002 -0.02)`

OUTPUT Um bei einer größeren Anzahl von Testläufen die Übersicht zu behalten, ist es möglich das Plotverzeichnis vom Defaultwert `./PLOT` auf einen beliebigen Wert zu verändern.
`-out` path : ”path” ist der Pfad oberhalb des pga /ga Basisverzeichnis, also `./PGA/path, ./PGA/GA/path` oder unter der MAP `.../MAP/usr/ga/path`.

A.3.5. Parameter für parallele Berechnungen

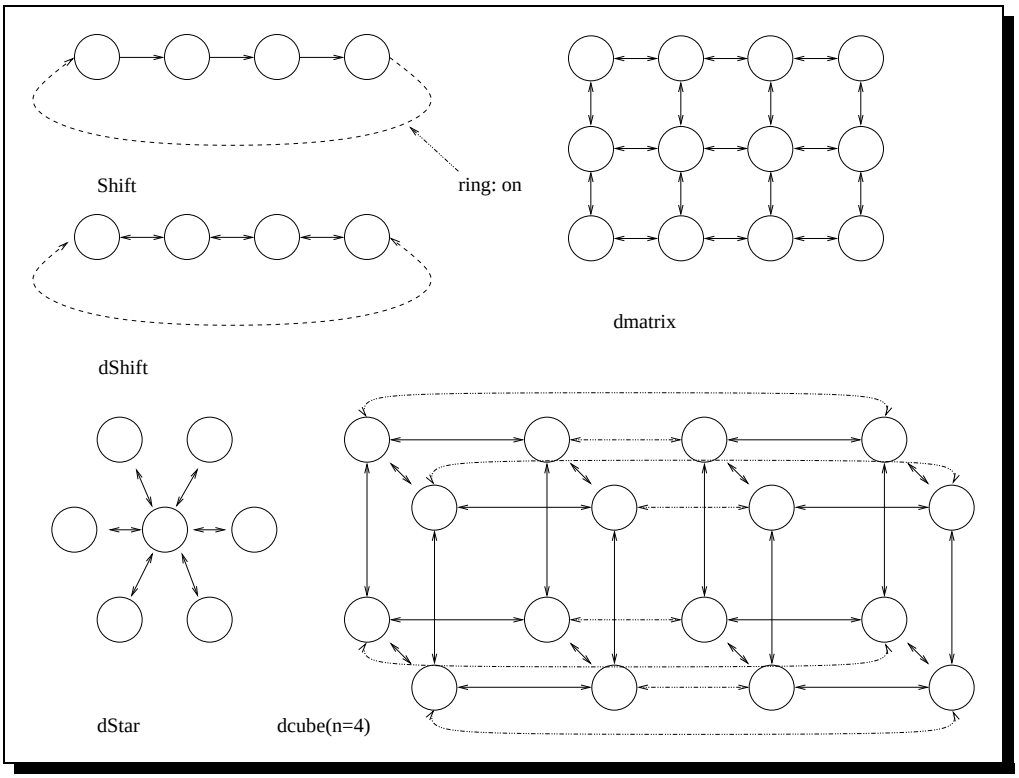

Abbildung A.2.: Beispiel für mögliche Kommunikationsstrukturen.

Neben den in A.3.4 aufgeführten Einstellungen kann man bei einem parallelen Lauf von pga zusätzlich noch Parameter bezüglich der Kommunikation einstellen.

Struktur Gibt an, innerhalb welcher Topologie die einzelnen Knoten von pga miteinander kommunizieren. Siehe Abbildung A.2.
`-comm_structure ( (d)shift | (d)star | (d)matrix |`

`(d)ncube |individual)`
Die mit ”d” beginnenden Einstellungen bedeuten, daß die Knoten in beide Richtungen kommunizieren.

Dynamik Die Veränderung des Kommunikationverhaltens wird als Kommunikationsdynamik bezeichnet. Mögliche Einstellungen sind `static` für konstante Kommunikationsraten, `dynamic` für eine einfache Abkühlung, und `villman` eine von T.Villman [22] entwickelte Abkühlungsstrategie. Parameter: `-comm_dynamic`.

Häufigkeit Mit der Häufigkeit wird festgelegt, wie oft Individuen ausgetauscht werden sollen. `-comm_rate (1-100)`

Anzahl Bei statischen Kommunikationen kann hier festgelegt werden, wieviele Individuen ausgetauscht werden müssen. Bei dynamischen Kommunikationsarten, wird damit der Startwert eingestellt. `-comm_idvnr (1-30)`

Fitneß Mit `-comm_fitrate(1-25)` läßt sich einstellen, wie oft die einzelnen Knoten ihre aktuelle beste Fitneß ausgeben.

Abkühlung Bei einfacher Abkühlung wird mit `-coolcomm(0.9-0.99)` die Abkühlungsrate eingestellt. Mit jeder Generation wird entsprechend der Abkühlungsrate ermittelt, wieviele Individuen ausgetauscht werden müssen. Der Parameter `-minmut` gibt eine untere Schranke an, die nicht unterschritten wird. So kann sichergestellt werden, daß die Zahl der auszutauschenden Individuen nie Null werden kann.

Ringstruktur Mit `-comm_ring 1` kann wird die Kommunikationsstruktur zusätzlich noch ringförmig geschlossen. (Stark abhängig von der gewählten Topologie.)

A.3.6. Analysefunktionen

Die Funktionen zur Analyse und Fehlersuche wurden implementiert, um das korrekte Arbeiten von pga überprüfen zu können, sowie einzelne Informationen über das Verhalten der Evolutionären Algorithmen zu gewinnen. Diese Funktionen sollten bei einem alltäglichen Einsatz von pga nicht relevant sein und werden darum hier nur kurz erwähnt.

Abstand Der durchschnittliche Abstand vom besten Individuum kann mit `-ana_dst N` aller N Generationen berechnet und ausgegeben werden. Dabei wird der Abstand zweier Individuen als die Zahl der Gene, in denen diese sich unterscheiden, definiert. Der Wert wird in der Datei `PLOT/NR_NODE.stat` ausgegeben, wobei ”NR” eine laufende Nummer ist und ”NODE” die Nummer der pga -Instanz.

Plotfiles Mit `-plt 1` werden automatisch Plotfiles (siehe A.4 erzeugt, welche mit GNUPLOT ausgewertet werden können.

Fitneß Die durchschnittliche Fitneß wird mit durch den Parameter `-avr 1` bei jeder Generation berechnet.

Meldungen Mit `-msg (0-4)` kann der Message-Level eingestellt werden, wobei für die normale Arbeit 1-2 sinnvoll ist. 0 gibt keine Meldungen aus, 3 und 4 sind nur zur Fehlersuche sinnvoll.

Blockzeiten Der konkrete Zeitbedarf einzelner Blöcke der besten Partition kann mit `-exa N` aller N Generationen abgespeichert werden. Die zugehörige Datei steht unter `PGA/PLOT`.

A.4. Auswerten von Plotdateien

Ist die Option `-plt 1` gesetzt, speichert `pga` unter `.../PGA/PLOT` bzw. `.../PGA/GA/PLOT` Plotdateien ab. Diese Dateien sind dazu gedacht, Programmläufe durch Diagramme zu visualisieren. Im Dateinamen der Plotdatei ist eine laufende Nummer sowie die Nummer des Knotens, auf dem diese erzeugt wurde codiert (`nummer_knoten.plt`). Die laufende Nummer wird über die Datei `.../PGA/.ga_plot` bestimmt. Um die Nummer auf 1 zu stellen, braucht man nur diese Datei zu löschen. Die Datei der Form

```
####################################
# Genetic Algorithms Plotfile
# Leipzig University / Hendrik Schulze
#GA-Plotfile-Nr.: 1
#Generations: 500
#Nr. of parents: 42
#Nr. of children: 90
#Mutationrate: 0.000350
#Elite: 6
.
#Crosspoints: 3
#Startparameter : ML100M0S_STEP0_250_1_MOCC2.15_
####################################
1 2 44145968 65132300
2 3 42159008 64313400
3 5 41005767 65401700
.
```

Tabelle A.5.: Ausschnitt aus einer Plotdatei

`nummer_all.plt` enthält die Informationen über die besten Fitneßwerte eines ganzen parallelen Laufes, während die anderen sich auf einen speziellen Knoten beziehen. Es ist leicht einzusehen, daß sehr schnell eine große Filemenge entstehen kann. Darum werden in jeder Plotdatei zusätzlich noch die wichtigsten Parameter abgespeichert. (Siehe Tabelle A.5.)

A.4.1. MKPLOT

Ein wichtiges Werkzeug zum Auswerten der Plotdateien ist MKPLOT. Es erstellt automatisch ein Script für GNUPLOT und ermöglicht es, verschiedene Plotfiles in einem Diagramm darzustellen und dieses ggf. als Encapsulated Postscript zu speichern. Am besten erklärt sich MKPLOT durch Beispiele.

1. `mkplot` gibt kurze Hilfe aus.
2. `mkplot -f1_0,2_3,5_all` erzeugt ein GNUPLOT-Script mit dem Defaultnamen mkplt.out, welches die Dateien `1_0.plt`, `2_3.plt` und `5_all.plt` visualisiert.
3. `mkplot -t -eEA1.eps -oea1 -f1,2` generiert das Script `ea1`, welches ein eps-File (`EA1.eps` (Siehe Abbildung A.4.) erzeugt mit `1.plt` und `2.plt`, als Plots. Der Parameter `-t` bewirkt, daß die Fitneß in Abhängigkeit der tatsächlichen Laufzeit gezeichnet wird. Ist er nicht gesetzt, so wird die Zahl der Generationen als X-Achse genommen.

Abbildung A.3.: Beispiele für MKPLOT. Das Ergebnis, des in Punkt 3 gezeigten Beispieles kann man in Abbildung A.4 sehen.

A.4.2. GNUPLOT

GNUPLOT ist ein Kommandozeilen orientiertes Programm zum Zeichnen von mathematischen Funktionen und Visualisieren von Daten. Es unterliegt der GNU PUBLIC LICENCE (GPL), ist also frei mit Quellcode verfügbar. Es ist für alle UNIX Systeme vorhanden und kann Ausgaben für eine Vielzahl von Devices erzeugen[9]. In dieser Arbeit wird vor allem seine Fähigkeit, Scripte abzuarbeiten, benutzt.

[9] u.a. X11,VGA,LaTeX, Encapsulated Postscript, Postscript, CorelDraw, PCL5, sowie Formate für einige Drucker

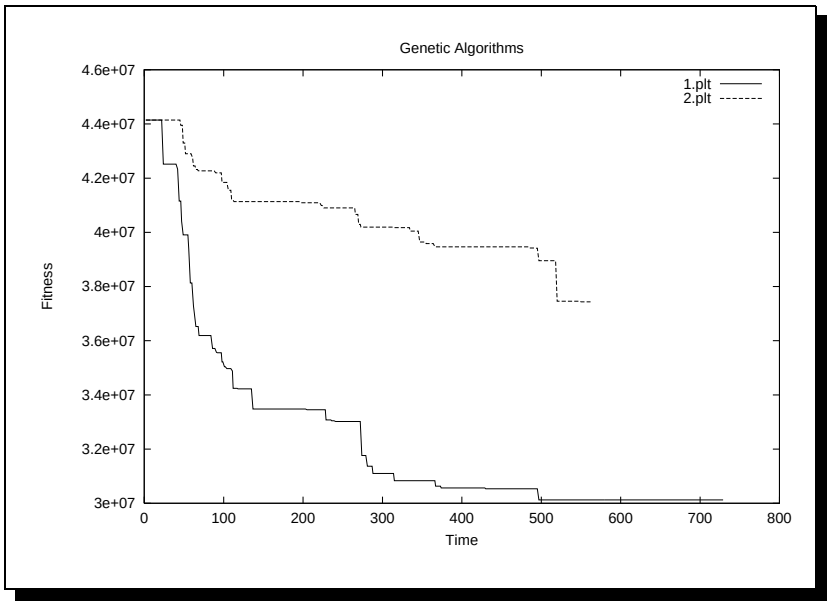

Abbildung A.4.: Beispiel für das in Abbildung A.3 erzeugte Diagramm.

Um ein von MKPLOT erzeugtes Script abzuarbeiten, muß man das Programm mit "`gnuplot`" aufrufen. In der gnuplot-shell kann das auszuführende Script mit "`load <ScriptName>`" geladen werden. Es wird sofort ausgeführt. Soll GNUPLOT nur ein EPS-File erzeugen, so kann das Programm auch direkt mit dem Scriptnamen aufrufen werden. Für das Beispiel aus Abbildung A.3 ist dies "`gnuplot ea1`". Für weiterführende Informationen ist die Onlinehilfe und [5] zu empfehlen.

A.5. pga und MAP

Parallel zu pga wurde von H.Hennings die parallele Laufzeitumgebung MAP (Model Analysing and Partitioning) entwickelt[7]. MAP besteht aus Ebenen, welche durch 3 Programme repräsentiert werden (siehe Abbildung A.5). Der Client ist die graphische Interface. Er wurde in TCL/TK geschrieben. Der Server verwaltet die benutzbaren Knoten und weist freie Ressourcen zu, die von Benutzern über den Client angefordert wurden. Auf allen verfügbaren Knoten wird bei Bedarf der Worker gestartet, linkt dynamisch die angeforderten Programme und ruft diese auf. Die einzelnen Worker können über MPI kommunizieren, während die Client-Server- und Server-Worker Kommunikation über Sockets realisiert wird.

Um pga unter MAP auszuführen, muß es mit der Option "`-D d_MAP`" kompiliert werden. Es ist sicher zu stellen, daß der gesamte Quellcode mit "`make shared`" neu kompiliert wird, da pga sonst nicht lauffähig ist.

Der Aufruf eines Programmes in einem MAP-Script erfolgt durch:

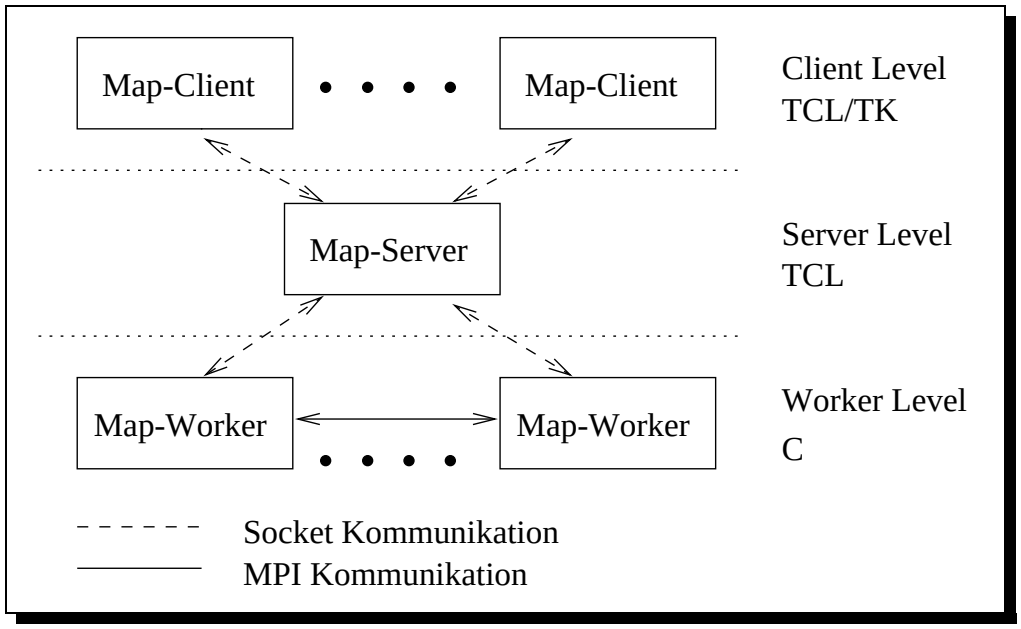

Abbildung A.5.: Die drei MAP -Level. Es kann beliebig viele Clients und Worker geben, aber nur einen Server. Auch können alle Komponenten auf verschiedenen Rechnern laufen.

"`libname_functionname <parameterlist>`". Wurde die Library pga.so erfolgreich erstellt und nach `.../MAP/shared` kopiert, so kann man `pga` durch "`pga_pga <parameter>`" starten. "`pga`" ist in diesem Fall der Funktionsname, der ermöglicht, daß `pga` sich unter MAP richtig initialisiert. Für die Parametereingabe gilt das in A.2.2 für `ga` gesagte (siehe auch Tabelle A.6). Wird nur der Libraryname eingegeben, so sucht der Worker nach einer Funktion mit gleichem Namen, wie die Library. Der Aufruf `pga` ist also äquivalent zu `pga_pga`.

A.5.1. Laden von Startpartitionen und Hypergraphen unter MAP

Mit den Defaulteinstellungen benutzt `pga` die MAP -Mechanismen und übernimmt zuvor erstellte Hypergraphen und Startpartitionen[10]. Sollen eigene Hypergraphen und Startpartitionen geladen werden, so muß `-prt_load 1` gesetzt werden[11]. Weiterhin müssen die Parameter zum Laden von Hypergraphen und Startpartitionen richtig gesetzt sein (siehe A.3.1 und A.3.2).

Wichtig: sollen die MAP internen Mechanismen zum Laden von Startpartitionen genutzt werden und diese nicht extern geladen werden, so dürfen die Parameter `-minstp` und `-maxstp` nirgendwo gesetzt werden!

[10] Es ist sicher zu stellen, daß diese auch erstellt wurden (siehe Tab.A.6)

[11] Am besten in `.../MAP/usr/ga/.ga`, wenn alle Knoten diese Dateien laden sollen.

```
alias PROTO PICMOFP
reserveNodes -number 2
#Script für Knoten 1
queue
load_dadb
load_proto PROTO
base_Step -pre yes -blocks 250 -num 1
hg_hg
base_Step -pre no -cones 250 -blocks 4 -num 10
delete_dadb
#Aufruf von pga mit Matrix Topologie
pga -comm_structure matrix

#Script für Knoten 2
queue
load_dadb
load_proto PROTO
base_Step -pre yes -blocks 250 -num 1
hg_hg
base_Step -pre no -cones 250 -blocks 4 -num 10
delete_dadb
#Aufruf von pga mit Matrix Topologie und Mutation 0.0025
pga -comm_structure matrix -mut 0.0025
```

Tabelle A.6.: Beispiel für ein MAP-Script zum Start von pga auf 2 Knoten.

A.5.2. Besonderheiten von pga unter MAP

Wird pga von MAP aufgerufen, so werden alle von pga benutzten Dateien relativ zum MAP Pfad erwartet. MAP gewährt jedem Programm unter `.../MAP/usr/<name>/` frei verfügbaren Plattenplatz. Für pga bedeutet dies konkret, daß alle wichtigen Dateien unter `...MAP/usr/ga/` abzulegen sind.

Literaturverzeichnis

[1] Thomas Bäck. *Evolutionary Algorithms in Theory an Practice*. Oxford University Press, New York, 1996.

[2] Bruno Codenotti and Mauro Leoncini. *Introduction to Parallel Processing*. Addison-Wesley, 1993.

[3] DFG-Project Model Partitioning Algorithms For Parallel Compiled-Mode Logic Simulation. World Wide Web: http://tech01.informatik.uni-leipzig.de/team/english/.

[4] Denis Döhler. Entwurf und Implementierung eines parallelen Logiksimulators auf Basis von TEXSIM. Master's thesis, Universität Leipzig, Mathematisches Institut, Dez 1996.

[5] Gnuplot homepage. World Wide Web http://www.cs.dartmouth.edu/gnuplot_info.html.

[6] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[7] Hilmar Hennings. Entwurf und Implementierung der Komponente parallelMap zur Modellanalyse und -partitionierung im Kontext von parallelTEXSIM. Master's thesis, Universität Leipzig, Institut für Informatik, Sep 1998.

[8] Klaus Hering. Parallel cycle simulation. Technical Report 13, Institut für Informatik, Universität Leipzig, 1996.

[9] Klaus Hering, Reiner Haupt, and Thomas Villmann. Cone-basierte, hierarchische Modellpartitionierung zur parallelen compilergesteuerten Logiksimulation beim VLSI-Design. Technical Report 13, Institut für Informatik, Universität Leipzig, 1995.

[10] Klaus Hering, Reiner Haupt, and Thomas Villmann. An improved mixture of experts approach for model partitioning in VLSI-Design using Genetic Algorithms. Technical Report 14, Institut für Informatik, Universität Leipzig, 1995.

[11] Klaus Hering, Reiner Haupt, and Thomas Villmann. Hierarchical strategy of model partitioning for VLSI-design, using an improved mixture of experts approach. In *Proceedings of the Conference on Parallel and Distributed Simulation*, pages 106–113, Los Alamos, 1996. IEEE Computer Society Press.

[12] I.N.Bromstein and K.A.Semendjajew. *Taschenbuch der Mathemathik*. Verlag Harri Deutsch, Thun und Frankfurt/Main.

[13] Gerd Meister. A survey on parallel logic simulation. Technical report, Department of Computer Science, Universtity of Saarland, Sep 1993.

[14] Melanie Mitchel. *An Introduction to Genetic Algorithms*. MIT Press Cambridge, Massachusetts, 1996.

[15] Argonne national laboratory. World Wide Web: http://www.mcs.anl.gov/mpi/.

[16] Mpi-Forum. World Wide Web: http://www.mpi-forum.org.

[17] MPI-Forum. Mpi: A message passing standard, Jun 1995. World Wide Web: http://www.mpi-forum.org/docs/mpi-11.ps.

[18] MPI-Forum. Mpi-2: Extentions to the message passing standard, Jul 1997. World Wide Web: http://www.mpi-forum.org/docs/mpi-2.ps.

[19] Volker Nissen. *Evolutionäre Algorithmen*. Deutscher Universitäts Verlag, 1994.

[20] Robert Reilein. Modellpartitionierung zur parallelen Logiksimulation. Master's thesis, Universität Leipzig, Institut für Informatik, Feb 1998.

[21] Wilhelm G. Spruth. *The Design of a Microprocessor*. Springer Verlag, 1989.

[22] Thomas Villmann, Reiner Haupt, Klaus Hering, and Hendrik Schulze. Parallel evolutionary algorithms with SOM-like migration and their application to real-world data sets. Eingereicht für ICANNGA 99, 6 1998.

Erklärung

Ich versichere, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, 10. August 1998