Forschungszentrum Karlsruhe
Technik und Umwelt

Wissenschaftliche Berichte
FZKA 6045

Entwurf und Realisierung einer flexiblen verteilten Kommunikation für einen parallelen evolutionären Algorithmus

Patricia Biegler[*], **Martina Gorges-Schleuter**

Institut für Angewandte Informatik

[*] Diplomarbeit an der Fakultat für Informatik der Universität Karlsruhe

Forschungszentrum Karlsruhe GmbH, Karlsruhe

1998

Entwurf und Realisierung einer flexiblen verteilten Kommunikation für einen parallelen evolutionären Algorithmus

Zusammenfassung

Dieser FZK-Bericht hat die Parallelisierung der in der Designoptimierungsumgebung SIMOT (SIMulation and Optimization Tool Environment) integrierten evolutionären Werkzeuge zum Inhalt. Diese Optimierungswerkzeuge GAMA (Genetischer Algorithmus zur Modelladaption) und GADO (Genetischer Algorithmus zur Designoptimierung) haben sehr lange Laufzeiten, da für die Bewertung der Güte von Modell- bzw. Designvorschlägen rechenzeitintensive Simulationswerkzeuge wie FEM-Simulator, Schaltkreis-Simulator oder Optiksimulator eingesetzt werden.

Die prinzipiell parallele Natur evolutionärer Algorithmen einerseits und die Verfügbarkeit von Rechenleistung im Rechnernetz andererseits sind die Voraussetzung für den Entwurf und die Realisierung einer Parallelversion der evolutionären Optimierungswerkzeuge. Die Arbeit gibt eine kurze Einführung in die Evolutionären Algorithmen und beschreibt das Parallelisierungspotential. Die Realisierung selber basiert auf PVM (Parallel Virtual Machine) einer sehr verbreiteten und portablen public domain Plattform, die es ermöglicht, eine heterogene Menge von Workstations und Supercomputern wie eine einzige leistungsfähige Maschine zu sehen.

Design and Implementation of a Flexible Distributed Communication for a Parallel Evolutionary Algorithm

Abstract

This report focuses on the parallelization of the evolutionary tools being integrated in the design optimization tool environment SIMOT (SIMulation and Optimization Tool Environment). These optimization tools called GAMA (Genetic Algorithm for Model Adaptation) und GADO (Genetic Algorithm for Design Optimization) have very long run times due the need of time consuming simulators like FEM-simulator, network simulator and optic simulator. The specific simulator calculates the quality of a given model or design which is then used for guiding the evolutionary search process.

The as a matter of principle parallel nature of evolutionary algorithms on one hand and the existance of computing power in our network of workstations on the other hand are the pre-conditions for the design and implementation of a parallel version of the evolutionary optimization tools. This thesis gives a short introduction into evolutionary algorithms and describes the various possibilities for a parallelization. The realization of the parallel concept is based on PVM (Parallel Virtual Machine) a well accepted and portable public domain software allowing a heterogeneous collection of workstations and supercomputers to function as a single high-performance parallel machine.

Inhaltsverzeichnis

1 Einleitung

Optimierungsprobleme stellen sich bei vielen technischen, mathematischen oder ökonomischen Systemen. Bekannte Verfahren zur Lösung dieser Probleme versagen aber oft bei Aufgaben mit vielen Parametern oder Funktionen mit vielen lokalen Optima. Die Evolution in der Natur stellt eine Optimierung von Lebewesen dar, die erstaunlich gute Ergebnisse erzielt. Im Grunde liegt es nahe, das erfolgreiche Prinzip der Evolution auch zur Verbesserung technischer Systeme heranzuziehen.

Die Evolutionsprozesse in der Natur laufen nicht sequentiell ab, sondern parallel. Es gibt immer mehrere Individuen in einer Population und in der Regel fast immer mehrere Populationen zur gleichen Zeit. Es stellt sich damit die Frage, ob evolutionäre Algorithmen ebenfalls als parallele Prozesse simuliert werden können. Vorteile sind zum einen die naturgetreuere Nachbildung, zum anderen aber auch die Beschleunigung der Simulation durch parallele Berechnungen.

Rechnernetze sind zwar keine Parallelrechner, dafür sind sie aber überall im Einsatz. Vielfach stehen eine Anzahl miteinander verbundener Rechner zur Verfügung, die nur selten ausgelastet sind. Angesichts dieser kostengünstigen vorhandenen Kapazitäten ist es interessant herauszufinden, welchen Einfluß die verteilte Abarbeitung eines evolutionären Algorithmus sowohl auf die Güte der gefundenen Lösungen, als auch auf die Laufzeit hat. In diesem Dokument wird dies anhand des evolutionären Algorithmus GLEAM untersucht.

Das *GLEAM*-Verfahren (Genetic Learning Algorithms and Methods), für das eine Anwendungs- und Experimentalumgebung (GLEAM/AE) existiert, ist die Basis des am Institut für Angewandte Informatik des Forschungszentrums Karlsruhe (FZK) entwickelten *SIMOT-Systems* (Simulation and Optimization Tool Environment). Damit können unterschiedliche Optimierungs- und Designaufgaben bearbeitet werden. SIMOT besteht im wesentlichen aus dem FEM-Simulator *ANSYS*, dem Analog-Simulator *ELDO*, sowie den Optimierungswerkzeugen *GADO* (Genetischer Algorithmus zur Design-Optimierung) und *GAMA* (Genetischer Algorithmus zur Modelladaption). Die Optimierungswerkzeuge GADO und GAMA basieren auf dem *GLEAM*-Verfahren.

Programmierumgebungen, die ein verteiltes Rechnen ermöglichen, sind u.a. ISIS [Bir90], Linda [NC89], das verteilte Betriebssystem DCE [Sch93], PVM [VSSM94]. Diese Programmierumgebungen sind sehr unterschiedlich ausgelegt: Sie beruhen meist auf dem nachrichtengekoppelten Paradigma (PVM) oder verwenden entfernte Prozeduraufrufe (DCE) zum verteilten Rechnen; sie sind entweder relativ einfache Programmierumgebungen (PVM) oder komplexe verteilte Betriebssysteme für heterogene Rechnernetze (DCE), die eine gemeinsame Dateiverwaltung und Softwareschutz gewährleisten.

Als Werkzeug für die verteilte Programmierung von GLEAM wurde *PVM* (Parallel Virtual Machine) ausgewählt, da PVM inzwischen eine der verbreitesten und portabelsten Plattformen für Verteilung geworden ist. Dazu haben sicherlich verschiedene Systemhersteller beigetragen, die es für ihre Hardware optimiert haben. Der Bekanntheitsgrad von PVM ist aber auch darauf zurückzuführen, daß das Softwaresystem kostenlos und im Quellcode zu bekommen ist.

Die Arbeit ist folgendermaßen gegliedert: Im 2. Abschnitt werden Grundbegriffe evolutionärer Algorithmen erklärt. Anschließend werden die Parallelisierungsmöglichkeiten dieser Algorithmen beschrieben. Das GLEAM-Verfahren, die Anwendungs- und Experimentalumgebung GLEAM/AE und einige Anwendungsbeispiele werden im Abschnitt 4 vorgestellt. Einen Überblick über die wesentlichen Merkmale von PVM verschafft Abschnitt 5. Die Konzepte zur verteilten Abarbeitung von GLEAM und der Entwurf der verteilten Kommunikation werden in den nächsten beiden Abschnitten beschrieben. Auf einige implementierungstechnische Aspekte wird in Abschnitt 8 eingegangen. Abschnitt 9 stellt dann die sequentielle und verteilte Version von GLEAM gegenüber, gefolgt von einer kurzen Bewertung dieser Arbeit.

2 Grundbegriffe evolutionärer Algorithmen

Die biologische Evolution kann als ein ständig fortschreitender Anpassungs- und Optimierungsprozeß in einer sich wandelnden Umwelt verstanden werden. Charles Darwin (1809–1882) identifizierte *Mutation* (Zufall) und *Selektion* (Notwendigkeit) als die entscheidenden Mechanismen der Evolution.

Die Selektion bestimmt welche Individuen in der nächsten Generation zur Reproduktion zugelassen werden.

Mutationen liefern als zufällige Veränderungen des Erbgutes Varianten der jeweiligen Grundform, die im Selektionsprozeß auf ihre Tauglichkeit hin überprüft werden. Lebewesen mit günstigen Eigenschaften sind demnach mit höherer Wahrscheinlichkeit in der Lage zu überleben und die Erbanlagen an Nachkommen weiterzugeben.

Ein anderer Evolutionsfaktor ist die *Rekombination*. Als Rekombination bezeichnet man die Neukombination von Erbfaktoren, wie sie bei Organismen mit geschlechtlicher Fortpflanzung auftritt. Während Genmutationen das Ausgangsmaterial der Evolution bilden, führt Rekombination zur Durchmischung des genetischen Materials einer Art. Dadurch können sich vorteilhafte Eigenschaften in einem Lebewesen vereinigen, das dann einen Selektionsvorteil besitzt und somit seine Erbanlagen bevorzugt weitergeben kann.

Evolutionäre Algorithmen (EA) abstrahieren die grundlegenden evolutionstheoretischen Prinzipien wie Mutation, Selektion und Rekombination. Sie bilden breit anwendbare Such- und Optimierungsverfahren mit überwiegend heuristischem Charakter. Der Einsatz evolutionärer Heuristiken ist aber nur dann sinnvoll, wenn es keine spezialisierten Verfahren gibt oder die traditionellen Verfahren Schwierigkeiten bekommen, weil die Zielfunktion nichtlinear, multimodal oder diskontinuierlich ist. Evolutionäre Verfahren überwinden diese Probleme dadurch, daß sie keine Annahmen über die Struktur des Problems benötigen. Statt von einer aktuell besten Lösung weiter zu suchen, begibt sich eine Menge von Individuen auf die Suche, die Informationen durch Vererbung austauschen und weitergeben können.

Der Basiszyklus eines evolutionären Algorithmus ist in Abb. 1 dargestellt. Es lassen sich verschiedene EA-Formen unterscheiden, doch stimmt die Grundphilosophie überein. Die wichtigsten beiden sind die Genetischen Algorithmen und die Evolutionsstrategien.

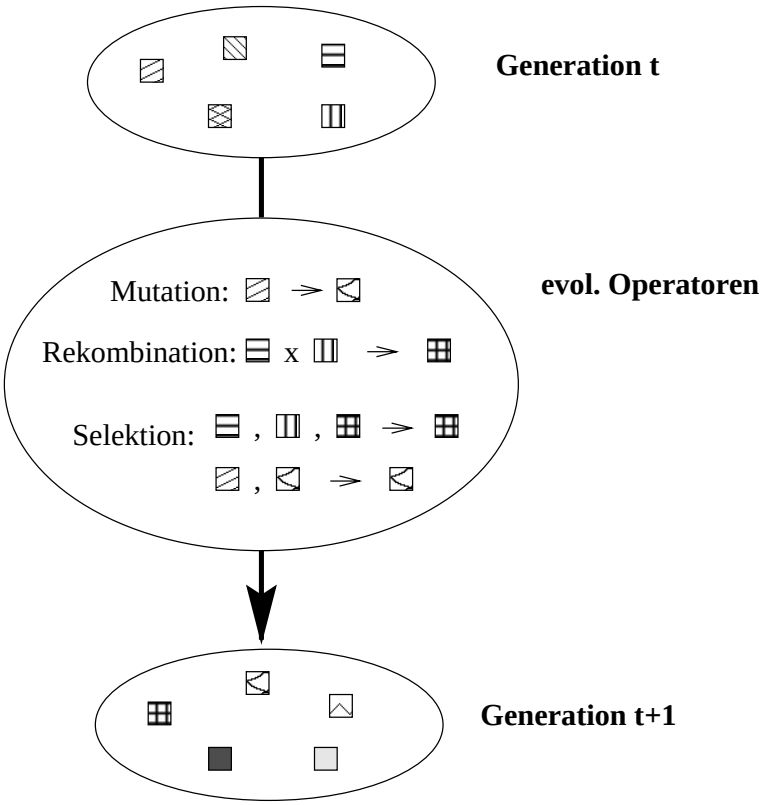

Abbildung 1: Der Basiszyklus eines EA

Die evolutionären Algorithmen unterscheiden sich in folgenden Punkten von konventionellen Optimierungsverfahren:

- Sie verwenden evolutionäre Operatoren, die an geeignet dargestellten Lösungen (Strukturen) ansetzen.
- Sie arbeiten i.a. mit einer *Population* von Lösungsalternativen, so daß der Lösungsraum von verschiedenen Punkten aus parallel durchsucht wird.
- EA enthalten bewußt stochastische Elemente. Daraus entsteht jedoch keine zufällige Suche, sondern eine intelligente Durchmusterung des Suchraums. Die Optimierung schreitet in der Regel zügig voran, weil sich der Suchprozeß auf solche Regionen konzentriert, die erfolgsversprechend sind.

EA sind darüber hinaus durch eine biologisch inspirierte Terminologie gekennzeichnet. Die wichtigsten Begriffe sind:

- *Individuum*: Struktur, die die in geeigneter Weise repräsentierten Elemente einer Lösung enthält.
- *Population* von Individuen: Menge von Strukturen (Lösungsalternativen).

- *Fitneß*: Lösungsqualität hinsichtlich der relevanten Zielkriterien.
- *Generation*: Verfahrensiteration.
- *Eltern*: Individuen, die für die Reproduktion ausgewählt wurden.
- *Kinder, Nachkommen*: Lösungen, die aus den Eltern erzeugt wurden.

Die EA-Hauptformen entstanden nahezu gleichzeitig in den frühen sechziger Jahren. John Holland [Hol75], Begründer der Genetischen Algorithmen (GA), befaßte sich mit dem Studium adaptiver Systeme. Adaption selbst faßte er als Optimierungsprozeß auf, der in der biologischen Evolution vorbildlich gelöst war. Für Rechenberg [Rec73] und Schwefel [Sch81] als Begründer der Evolutionsstrategien (ES) stand dagegen die algorithmische Betrachtung von Anfang an im Vordergrund. Das Grundkonzept war die Optimierung durch Nachahmung evolutionärer Prinzipen in vereinfachter Form. In den folgenden Unterabschnitten werden die beiden EA-Hauptformen näher beschrieben.

2.1 Genetische Algorithmen

GA's wurden ursprünglich nicht als Optimierungsalgorithmen, sondern als verallgemeinerte Modelle adaptiver Systeme geschaffen. Sie imitieren evolutionäre Prozesse unter der besonderen Betonung genetischer Mechanismen.

Der traditionelle Basisalgorithmus (nach Holland) operiert mit einer Population von n Individuen (Bitstrings). Auf ihnen sind jeweils komplette Lösungsvorschläge für die betrachtete Problemstellung *binär* codiert. Komplexere Datenstrukturen wie reelle Zahlen, Listen, Bäume müssen durch entsprechende Abbildungen auf Bitstrings abgebildet werden. Jeder problembezogenen Entscheidungsvariablen entspricht ein Stringsegment. GA arbeiten also nicht direkt mit den Variablen eines Entscheidungsproblems, sondern mit einer Codierung derselben.

Durch die Benutzung einer binären Repräsentation ist die Nachbildung der genetischen Operatoren Mutation bzw. Rekombination (Crossover) einfach. Die Mutation wird durch das zufällige Invertieren eines Bits realisiert. Für die Rekombination wird gleichverteilt-zufällig ein Crossover-Punkt ermittelt, der für beide Strings identisch ist. Indem Teilstücke zwischen den Strings ausgetauscht werden, entstehen zwei rekombinierte Nachkommen (s. Abb. 2).

Folgende Schritte werden im Basis-GA durchlaufen:

1. *Initialisierung:* Es werden zufällig n Individuen für die Startpopulation P_0 generiert.
2. *Iteration:* Solange kein Abbruchkriterium erfüllt ist, erzeuge die Folgegenerationen P_t $(t = 1, 2, ...)$ folgendermaßen:
   (a) *Bestimmung der Lösungsgüte jedes Individuums:* Es werden die Fitneßwerte aller Individuen anhand der anwendungsspezifischen Fitneßfunktion f berechnet. Anschließend wird noch die durchschnittliche Fitneß f^d der Generation P_t ermittelt.

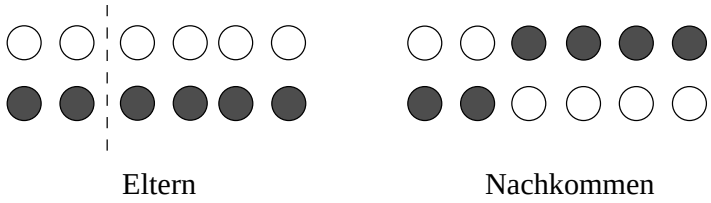

Abbildung 2: Das Ein-Punkt Crossover

(b) *Variation und Selektion:* Für alle k $(k = 1, .., n)$ wird ein Individuum A_k aus P_t fitneßproportional ausgewählt, d. h. daß jedes Individuum mit der Wahrscheinlichkeit $f(A_k)/f^d$ selektiert wird. Dann wird der genetische Operator bestimmt, der anzuwenden ist: Mutation mit Wahrscheinlichkeit P_M, Crossover mit Wahrscheinlichkeit P_C. Wird der Crossover-Operator angewendet, so muß noch ein zweites Individuum fitneßproportional selektiert werden. Eines der beiden Nachkommen wird dann zufällig ausgewählt und der Generation P_{t+1} hinzugefügt.

Als Abbruchkriterien kommen in Betracht die Erreichung einer vorgegebenen Lösungsgüte oder einer vorgegebenen Generationszahl, die Erschöpfung eines Zeit- bzw. Kostenbudgets oder die Stagnation der Lösungsgüte über einen festgelegten Zeitraum.

Eine besondere Eigenschaft des GA ist, daß durch die fitneßproportionale Selektion gute Individuen mit höherer Wahrscheinlichkeit ihre "Gene" (Lösungselemente) an Nachkommen weitergeben. Crossover ist hierbei der dominierende Operator ($P_C > 0,6$), während die Mutation als Hintergrundoperator (meistens $P_M < 0,01$) dem endgültigen Verlust von Bitwerten entgegenwirken soll.

2.2 Evolutionsstrategien

Evolutionsstrategien sind an der TU Berlin entstanden und dadurch eine in Deutschland besonders populäre EA-Variante. Ausgangspunkt ist eine Population von μ Individuen. Jedes Individuum A enthält, als Vektor stetiger Größen, Werte für alle m Entscheidungsvariablen x_m des betrachteten Optimierungsproblems. Hinzu kommen noch n $(n < m)$ sogenannte Mutationsschrittweiten σ_w $(w = 1..n)$, die Bestandteil eines jeden Individuums sind $(A = x_1 x_2 \cdots x_m \sigma_1 \sigma_2 \cdots \sigma_n)$. Sie stellen Strategieparameter des ES-Verfahrens dar. Im Verlauf der Optimierung werden so parallel zur Suche nach guten Werten für die Entscheidungsvariablen auch gute Einstellungen der Mutationsschrittweiten erzielt.

Es werden zwei verschiedene Rekombinationsoperatoren benutzt. Bei der *diskreten Rekombination* besteht die Struktur eines Nachkommen K aus einer zufälligen Mischung der Eltern-Komponenten $(x_j(K) = x_j(A)$ *oder* $x_j(B)$, wobei A, B Eltern sind). Dagegen wird bei der *intermediären Rekombination* eine Nachkommen-Komponente aus den Komponenten der Eltern durch Mittelwertbidung berechnet $(x_j(K) = (x_j(A) + x_j(B))/2)$.

Im Gegensatz zu den genetischen Algorithmen ist bei den Evolutionsstrategien die Mutation kein Hintergrundoperator. Mutationen werden so vorgenommen, daß kleine Änderungen wahrscheinlicher sind als große. Dies wird auf dem Rechner über die Addition normalverteilter[1] Zufallszahlen mit Erwartungswert 0 realisiert. Bei der Mutation der Strategieparameter wird multiplikativ eine logarithmische Normalverteilung verwendet, so daß eine Halbierung bzw. Verdoppelung einer Schrittweite gleich wahrscheinlich vorkommt.

Das Ablaufschema einer Standard-ES sieht folgendermaßen aus:

1. *Initialisierung und Bewertung der Startpopulation:* Es werden zufällig μ Individuen generiert. Die Startwerte der Strategieparameter sollten dabei tendentiell groß gewählt werden, um vorzeitige Konvergenz zu verhindern. Anschließend wird die Fitneß aller Individuen der Startpopulation berechnet.

2. *Iteration:* Solange keine Abbruchbedingung erfüllt ist, werden Folgegenerationen P_t erzeugt:

   (a) *Variation und Bewertung:* Aus den μ Individuen werden λ ($\lambda > \mu$) Nachkommen generiert. Dafür werden die Eltern eines Nachkommen aus der Generation P_t zufällig ausgewählt, d.h. daß, im Gegensatz zu den GA's, jedes Individuum die gleiche Wahrscheinlichkeit besitzt, seine Lösungselemente an Nachkommen weiterzugeben. Ein Nachkomme entsteht durch Rekombination. Dann wird der Mutationsoperator auf die Strategieparameter des Kindes angewendet. Die Entscheidungsvariablen des Nachkommen werden unter Verwendung der so erhaltenen neuen Mutationsschrittweiten berechnet ($x_j^{'}(K) = x_j(K) + N(0, \sigma_j^{'}(K))$). Anschliessend wird die Fitneß des Kindes ermittelt.

   (b) *Selektion:* Es werden zwei Selektionsarten unterschieden: die $(\mu + \lambda)$-Selektion und die $(\mu,\ \lambda)$-Selektion. Bei der $(\mu,\ \lambda)$-Selektion werden aus den λ Nachkommen die μ besten Individuen ausgewählt. Die Lebensdauer eines Individuums ist somit auf eine Generation beschränkt. Bei der $(\mu + \lambda)$-Selektion konkurrieren die Eltern und die Kinder um das Überleben. Die nächste Generation P_{t+1} setzt sich aus den μ besten Individuen der Vereinigungsmenge von Eltern und Kinder ($\mu + \lambda$ Individuen) zusammen. Das beste Individuum überlebt so immer (elitäre Strategie).

Als Abbruchkriterien können die bei den GA's angegebenen Bedingungen genutzt werden. Das Verfahren sollte aber spätestens dann abgebrochen werden, wenn die Fitneß des besten und schlechtesten Individuums fast gleich ist, da in diesem Fall der Suchprozeß wahrscheinlich in einem lokalen Optimum stehengeblieben ist.

Beide beschriebenen Algorithmen gehen davon aus, daß die Fitneß aller Individuen zu jedem Zeitpunkt bekannt ist. Um einen solchen Algorithmus effizient zu parallelisieren muß man aber ohne globales Wissen auskommen. Mit dieser Thematik befaßt sich der nächste Abschnitt.

[1] Normalverteilung: $N(\epsilon, \sigma)$, mit ϵ als Erwartungswert und σ als Standardabweichung

3 Parallelisierungspotentiale evolutionärer Algorithmen

Die biologische Evolution ist ein massiv paralleler Prozeß. Es gibt immer mehrere Individuen in einer Population und meistens mehrere Populationen einer Art zur gleichen Zeit. Zu einem bestimmten Zeitpunkt existieren somit mehrere Individuen die geboren werden, sich verändern oder sterben. In einem sequentiellen Algorithmus kann der Evolutionsprozeß zur gleichen Zeit nur für ein Individuum durchgeführt werden. Wie die biologische Evolution mit Hilfe paralleler Prozesse auf dem Rechner simuliert werden kann, steht im Mittelpunkt der nächsten beiden Abschnitte.

Der erste Schritt, der bei einer Parallelisierung durchgeführt werden muß, ist die Aufteilung der Individuen einer Population auf die verfügbaren parallelen Prozesse. Es stellt sich dann die Frage, welche der nachfolgenden wichtigsten Schritte eines Standard-EA parallelisierbar sind:

1. Erzeugung der Startpopulation, Berechnung der Fitneß der Individuen;
2. Berechnung der Abbruchbedingungen; solange diese nicht erfüllt sind:
   (a) Auswahl der zu rekombinierenden Individuen;
   (b) Anwendung genetischer Operatoren (Rekombination, Mutation);
   (c) Fitneßberechnung der Nachkommen;
   (d) Selektion;

Die Erzeugung der Startpopulation läßt sich parallelisieren, wenn jedes Individuum unabhängig von den anderen Individuen der Population, z. B. durch Auswürfeln, bestimmt wird. Wenn allerdings Vorwissen (z. B. aus anderen Evolutionsläufen) in die Initialisierung einfließt, so ist dies nicht ohne weiteres möglich. Die Berechnung der Fitneß eines konkreten Individuums stellt kein Problem dar, wenn es unabhängig von allen anderen auf seine Tauglichkeit und Güte geprüft werden kann. Dies ist bei den gängigen evolutionären Algorithmen der Fall. Die parallele Fitneßberechnung bedeutet in vielen Anwendungen eine enorme Geschwindigkeitssteigerung, denn oft wird mehr als 80% der Gesamtrechenzeit für die Ermittlung der Fitneß der Individuen verbraucht.

Ob die Überprüfung der Abbruchbedingungen parallelisierbar ist, hängt von den Abbruchkriterien ab. Wenn es sich um die Anzahl der Generationen oder die verbrauchte Rechenzeit handelt, so sind die Bedingungen leicht zu evaluieren und müssen nicht parallelisiert werden. Gewöhnlich wird ein evolutionärer Algorithmus dann abgebrochen, wenn das beste Individuum eine bestimmte Güte erreicht hat oder keine wesentlichen Veränderungen stattfinden. Die Überprüfung dieser Bedingungen erfordert globales Wissen und läßt sich somit nur zentral durchführen. Fazit: Die Abbruch-Prüfung ist eine zentrale Steuerungsfunktion und ist in der Regel nicht parallelisierbar.

Die Auswahl der zu rekombinierenden Individuen ist dann nicht oder nur ungenügend parallelisierbar, wenn die Selektion der Individuen proportional zur Fitneß aller Individuen

vorgenommen werden soll oder von der Fitneß der Gesamtpopulation abhängt. Dies ist aber bei fast allen evolutionären Algorithmen der Fall.

Die Durchführung genetischer Operationen ist völlig unabhängig von den unbeteiligten Individuen und kann deshalb parallel in der gesamten Population erfolgen. Dies kann gegenüber einem seriellen Algorithmus, bei dem Paare von Individuen nacheinander rekombiniert werden, eine wesentliche Verkürzung der Rechenzeit bewirken. Diese Rechenzeiteinsparung fällt um so stärker aus, je komplizierter und rechenintensiver die genetischen Operationen sind.

Nach der Rekombination der Individuen entsteht bei einer fixen Populationsgröße ein Überschuß an Individuen. Aus der gesamten Population müssen die tauglichsten nach einem Selektionskriterium ausgewählt werden, um die Populationsgröße konstant zu halten. Wenn dieses Selektionskriterium die Kenntnis der Fitneß aller Individuen erfordert, so stellt die Selektion, wie die Berechnung der Abbruchkriterien auch, einen zentralen Steuerungsmechanismus dar, der praktisch nicht parallelisiert werden kann. Eine gewisse Parallelisierung ist aber dann zu erreichen, wenn man die Population in Subpopulationen aufteilt und nur lokal selektiert.

Die Standard-EAs, wie auch die in Abschnitt 2 beschriebenen Algorithmen, zeichnen sich dadurch aus, daß ein Nachkommen erst dann selektiert wird, wenn alle Nachkommen einer Generation berechnet wurden. Geht man nun davon aus, daß die Erzeugung von Nachkommen parallel abläuft, so stellt die Selektion einen Synchronisationschritt dar. Wie diese Synchronisation umgangen werden kann, wird im nächsten Unterabschnitt beschrieben.

3.1 Generationsmodelle

Der Zeitpunkt der Selektion eines Nachkommen wird durch das Generationsmodell bestimmt. Bei dem *diskreten Generationsmodell*, das auch die Standard-EAs benutzen, existieren zu jedem Zeitpunkt zwei Populationen: Die Eltern-Generation $P(t)$ und die Nachkommen-Generation $P(t+1)$ (s. Abb. 3). Nachdem der Selektionsprozeß stattgefunden hat, wird die Nachkommen-Generation zur Eltern-Generation der nächsten Iteration.

Fällt man die Entscheidung, ob ein Individuum überlebensfähig ist oder nicht, gleich nach dessen Erzeugung, erhält man das *kontinuierliche Generationsmodell*. Beim kontinuierlichen Generationsmodell existiert nur eine Population: Eltern und Nachkommen gehören der gleichen Population an (s. Abb. 4 a).

Die Selektion eines Nachkommen simuliert den Geburtsprozeß und die Ersetzung eines Individuums den Sterbeprozeß, der beim kontinuierlichen Generationsmodell gleich nach der Erzeugung eines überlebensfähigen Nachkommen eintritt. Die Populationsgröße ist konstant. Wenn ein überlebensfähiges Kind erzeugt wurde, muß ein Individuum ersetzt werden. Reproduktion ist in diesem Fall individuenbezogen statt generationsbezogen. Streng genommen kann man auch nicht mehr von einer Population zum Zeitpunkt t sprechen, da jedes Individuum eigentlich einer anderen Generation angehört. Dieses Modell erweist sich als besonders nützlich in Zusammenhang mit der Parallelisierung evolutionärer Algorithmen: Die strenge Kopplung zwischen Individuum und zugehöriger Generation wird

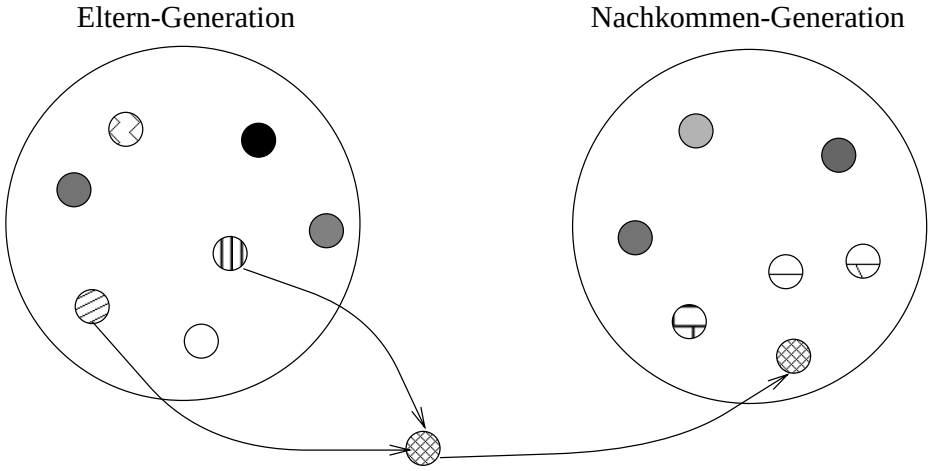

Abbildung 3: Das diskrete Generationsmodell

aufgehoben.

Entscheidet man sich für das kontinuierliche Generationsmodell, so bleibt zu bestimmen, welches Individuum der Population bei der Selektion ersetzt wird.

Eine gute Lösung, die die genetische Vielfalt der Population nicht vermindert, ist eines der Elternteile durch das Nachkommen zu ersetzen (s. Abb. 4 b). Das Nachkommen nimmt so den Platz eines ihm ähnlichen Individuums ein.

Auch wenn man sich für das kontinuierliche Generationsmodell entscheidet, ist nicht unbedingt eine Parallelisierung des Selektionsprozesses möglich. Dies ist der Fall, wenn für die Evaluierung des Selektionskriteriums für die Reproduktion globales Wissen (die Fitneß aller Individuen der Population) benötigt wird. Wie bereits erwähnt, kann man eine gewisse Parallelisierung erreichen, wenn man die Population in Subpopulationen aufteilt. Die Möglichkeiten der Aufteilung einer Population in Subpopulationen werden im nächsten Unterabschnitt besprochen.

3.2 Strukturierung der Population

Die moderne Evolutionstheorie nennt als wesentliche Faktoren der Evolution die Replikation (Rekombination), die Mutation, die Selektion und die Isolation. Aber nur die ersten drei Faktoren sind in jedem evolutionären Algorithmus implementiert.

Auch in den in Abschnitt 2 vorgestellten genetischen Algorithmus bzw. Evolutionsstrategien ist die gegenseitige Beeinflussung aller Individuen gleich. Typischerweise herrscht in einer reell existierenden Population keine Panmixie (unbegrenzter Genfluß), sondern sie zerfällt in mehr oder weniger isolierte Subpopulationen (Demes). Diese Unterteilung führt zu einer Strukturierung der Population.

Es existieren verschiedene Modelle zur Simulation simultan existierender Teilpopulatio-

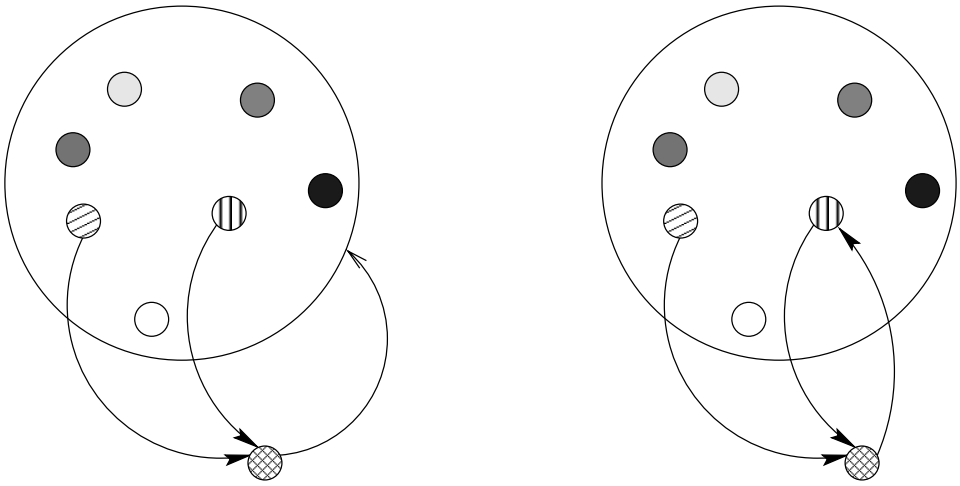

Abbildung 4: Das kontinuierliche Generationsmodell

nen [ES94]. Prinzipiell sind diese aber Variationen einer der beiden Grundmodelle: Migrationsmodell und Nachbarschaftsmodell [GS94b].

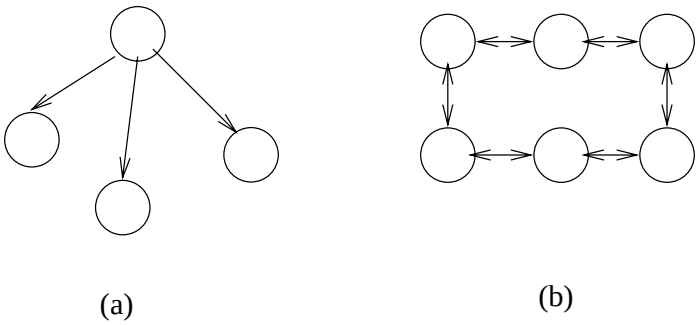

Abbildung 5: Das Inselmodell (a) und das Netzwerkmodell (b)

Migrationsmodelle zeichnen sich dadurch aus, daß ihre Teilpopulationen fest voneinander abgegrenzt sind. Mit Migration wird der Austausch von Individuen zwischen den Teilpopulationen bezeichnet. Wenn jede Teilpopulation von jeder anderen entsprechend einer bestimmten Migrationsrate Individuen erhalten kann, spricht man vom Inselmodell (s. Abb. 5 a). Je nach den lokalen Verhältnissen innerhalb der einzelnen Subpopulationen werden hier Verbindungen zu anderen Populationen aufgebaut.

Reell existierende Populationen sind oft in Kolonien organisiert und Individuen können nur zwischen benachbarten Kolonien ausgetauscht werden. Alle Kolonien bilden zusammen eine geografische Struktur (Ring, Torus, etc). Die Modelle, die diese Besonderheit berücksichtigen, heißen Netzwerkmodelle (stepping-stone-models) (s. Abb. 5 b). Hier sind die Teilpopulationen über feststehende Informationskanäle miteinander verbunden. Der

Vorteil der Migrationsmodelle ist, daß sie mathematisch beschreibbar sind.

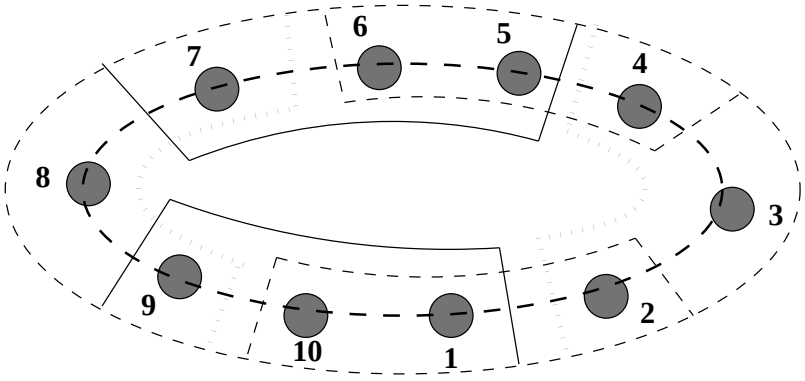

Abbildung 6: Ein ringförmiges Nachbarschaftsmodell

Das Nachbarschaftsmodell ist eine Abwandlung des Netzwerkmodells, bei dem die Teilpopulationen allerdings nicht mehr fest voneinander abgegrenzt sind. Die Individuen werden räumlich fix angeordnet. Nach der räumlichen Anordnung der Population wird jedem Individuum nach einem festen Schema eine Nachbarschaft (von Individuen) zugeordnet. Ein Individuum zusammen mit seinen Nachbarn bildet eine Subpopulation. Subpopulationen tauschen nun Informationen über die Individuen in den Nachbarschaftsgebieten aus, die *nicht disjunkt* sind. Sie überlappen sich und gewährleisten so eine indirekte Verbindung aller Subpopulationen untereinander. Durch den Grad der Überlappung der Nachbarschaften kann man beeinflussen, wie wahrscheinlich der Austausch von Genen zwischen zwei beliebigen Populationen wird. Eine andere Möglichkeit, die Intensität des Informationsflusses zu variieren, ist die Dimension der räumlichen Struktur. Werden die Individuen auf einer eindimensionalen Struktur (z. B. Ring) angeordnet, so überlappen sich die Subpopulationen nur in zwei Richtungen. Bei einer zweidimensionalen Struktur (z. B. Torus) können es aber bereits bis zu acht Richtungen sein. In Abb. 6 ist ein ringförmiges Nachbarschaftsmodell mit einer Nachbarschaftsgröße von 2 dargestellt. Der Übersichtlichkeit halber wurden nur die Nachbarschaften der Individuen 1, 3, 5, 6, 8 und 10 eingezeichnet.

Wenn man die Population in Subpopulationen aufteilt aber trotzdem ein globales Selektionskriterium, d. h. ein Selektionskriterium, das alle Subpopulationen einbezieht, benutzt, so wird dadurch nicht viel zur Parallelisierung des Algorithmus beigetragen. Dies liegt daran, daß die Subpopulationen synchron arbeiten müßten.

3.3 Lokale Selektionskriterien

Das Selektionskriterium legt fest, welches der Nachkommen lebensfähig ist. Wählt man ein lokales Selektionskriterium, so bezieht sich der Selektionsprozeß nur auf eine Subpopulation und benötigt somit kein globales Wissen über die gesamte Population. Dies bringt zwei Vorteile mit sich. Zum einen können sich Mutanten besser als bei Panmixie erhalten, was einer vorzeitigen Konvergenz in suboptimale Lösungen entgegenwirkt. Zum anderen

trägt dies zur Parallelisierbarkeit des Algorithmus bei, da keine globale Synchronisation nach jeder Generation wie bei einem globalen Selektionsmechanismus benötigt wird.

Bei einem lokalen Selektionskriterium wird die Fitneß eines Nachkommen nur *lokal*, d.h. mit der der Individuen aus der Nachbarschaft der Eltern, verglichen. Es können mehrere Überlebensstrategien [GS91] definiert werden, die sich in der Intensität des Selektionsdrucks unterscheiden:

- *accept-all:* Akzeptiere jeden Nachkommen.
- *accept-all-ES:* Akzeptiere jeden Nachkommen, solange dadurch nicht das lokal beste Individuum ersetzt werden soll.
- *local-least:* Akzeptiere den Nachkommen nur, wenn er besser als der lokal Schlechteste ist.
- *local-least-ES:* Akzeptiere den Nachkommen nur, wenn er besser als der lokal Schlechteste ist und dadurch nicht das lokal beste Individuum ersetzt werden soll.
- *worse:* Akzeptiere den Nachkommen nur, wenn er besser oder höchstens 1% schlechter als der lokal Schlechteste ist.
- *worse-ES:* Akzeptiere den Nachkommen nur, wenn er besser oder höchstens 1% schlechter als der lokal Schlechteste ist und dadurch nicht das lokal beste Individuum ersetzt werden soll.
- *better-parent:* Akzeptiere den Nachkommen nur, wenn er besser als das zu ersetzende Elternteil ist.

Die mit *ES* gekennzeichneten Varianten und die Strategie *better-parent* sind elitäre Strategien, da damit der lokal Beste und damit auch der global Beste immer erhalten bleibt. Diese Art der Selektion ist ähnlich der der Evolutionsstrategien, da, außer bei *accept-all*, die besseren Individuen überleben. Aber im Gegensatz zu den Evolutionsstrategien und ähnlich zu den genetischen Algorithmen hängt die Anzahl der Nachkommen eines Individuums von dessen Fitneß ab.

Durch die Einführung von Subpopulationen in Verbindung mit lokalen Selektionskriterien kann einer der wichtigsten Schritte eines evolutionären Algorithmus, der Sequentialität erfordert, parallelisiert werden.

4 Das GLEAM-Verfahren

Nachdem wesentliche Elemente der Evolutionstheorie vorgestellt und einige Fachbegriffe eingeführt wurden, wird nun näher auf das GLEAM-Verfahren eingegangen.

GLEAM (Genetic Learning Algorithms and Methods), dessen Konzept von Prof. Dr. C. Blume (FH Köln, Abt. Gummersbach) [Blu90] entwickelt wurde, verbindet wichtige

Aspekte der Evolutionsstrategien von Prof. Dr. Rechenberg [Rec73] und der Genetischen Algorithmen von Prof. Dr. Holland [Hol75] mit Methoden der klassischen Datenverarbeitung.

GLEAM unterscheidet sich von den vorgestellten evolutionären Algorithmen vor allem durch eine geänderte Repräsentation eines Individuums und dazu passende genetische Operatoren. Das GLEAM-Konzept wurde von Dr. M. Gorges-Schleuter [GS94b] durch eine Strukturierung der Population und ein kontinuierliches Generationsmodell erweitert. In den folgenden Unterabschnitten werden nun diese Besonderheiten erläutert und anschließend wird der Algorithmus vorgestellt. Zum Schluß werden kurz die Anwendungs- und Experimentalumgebung GLEAM/AE und zwei Anwendungsbeispiele beschrieben.

4.1 Repräsentation und genetische Operatoren

Die Repräsentation eines Individuums in GLEAM wird *Kette* genannt und ist eine listenähnliche hierarchische Datenstruktur. Der Aufbau der Datenstruktur hängt von der zu bearbeitenden Anwendung ab. Ein Kettenelement enthält entweder spezifische Parameter oder einen Verweis auf eine andere Kette. Eine Kette kann *Unterkette* einer anderen Kette sein. (s. Abb. 7). Es werden zwei Arten von Unterketten unterschieden. *Lokale Unterketten* werden als Teil einer anderen Kette angesehen. Eine Kette kann aber auch nur eine *Referenz* auf eine andere Kette enthalten, die ihrerseits ein anderes Individuum repräsentiert.

Mehrere Kettenelemente bilden zusammen ein *Segment*. Segmente dienen dazu, mehrere Kettenelemente gegenüber genetischen Operatoren als Ganzes erscheinen zu lassen.

Das GLEAM-Verfahren stellt verschiedene Varianten elementarer genetischer Operatoren (Mutation, Rekombination) zur Verfügung. Damit können komplexe, problemspezifische genetische *Operationen* definiert werden, als die Anwendung

1. eines einzelnen Mutationsoperators auf ein Kind.
2. mehrerer Mutationsoperatoren auf ein Kind, wobei jeder Operator wahrscheinlichkeitsbehaftet sein kann.
3. eines einzelnen Rekombinationsoperators zur Erzeugung von zwei Nachkommen.
4. eines einzelnen Rekombinationsoperators zur Erzeugung von zwei Nachkommen, wobei auf ein Kind zusätzlich ein oder mehrere jeweils wahrscheinlichkeitsbehaftete Mutationsoperatoren angewandt werden.

Pro Paarung können mehrere solcher Operationen benutzt werden.

Mutationsoperatoren sind Segmentmutatoren, Kettenelementmutatoren und Parametermutatoren. Durch Segmentmutatoren werden nicht die Kettenelemente selbst, sondern deren Anordnung oder Segmentzugehörigkeit verändert. Segmentmutatoren sind die Segmentteilung, Inversion (Umkehrung der Kettenreihenfolge eines Segments), Verschmel-

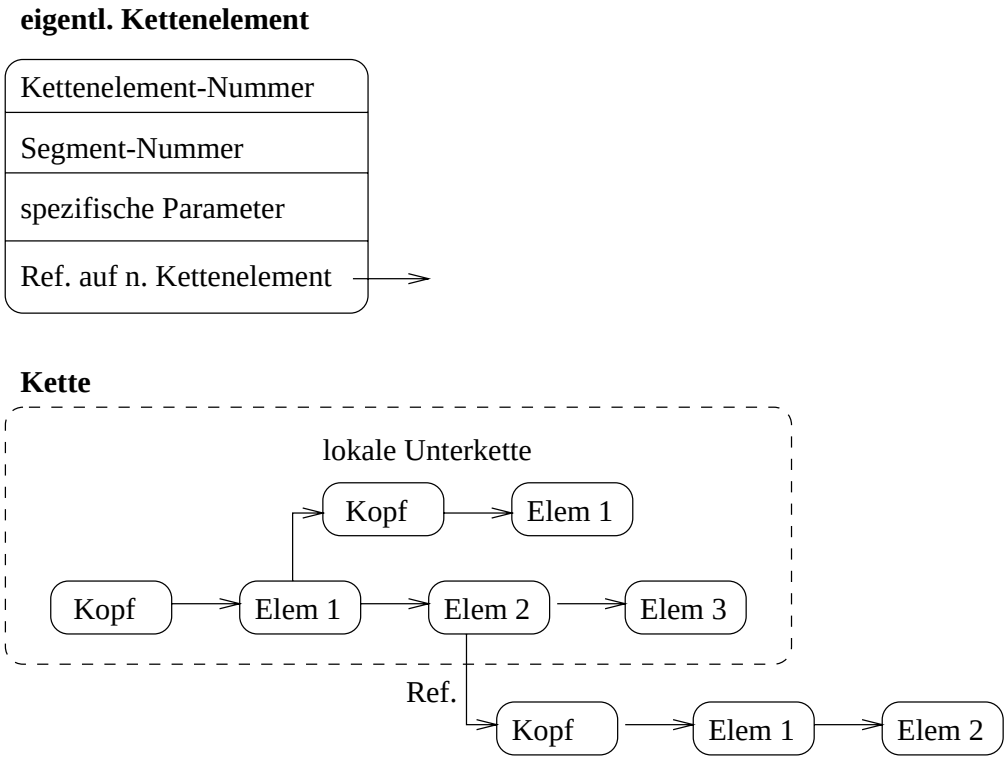

Abbildung 7: Der Aufbau einer GLEAM-Kette

zung zweier benachbarter Segmente und das Löschen, Austauschen oder Verdoppeln eines zufällig ausgewählten Segments.

Parametermutatoren verändern bzw. bestimmen einen neuen Wert für einen oder mehrere Parameter einer ausgewürfelten Anzahl von Kettenelementen. Wenn für den Wertebereich eines Parameters ein Sortierkriterium existiert, so kann durch die Einteilung dieses Bereichs in Klassen sichergestellt werden, daß, wie bei den Evolutionsstrategien auch, kleine genetische Änderungen häufiger vorkommen als große.

Kettenelementmutatoren fügen einer Kette ein neues Kettenelement hinzu, löschen, verdoppeln, verschieben oder tauschen ein zufällig bestimmtes Kettenelement aus.

Als Rekombinationsoperatoren stehen sowohl Ein-Punkt-Crossover als auch Mehrpunkt-Crossover zur Verfügung, wobei auf die Wahrung der Segmentgrenzen geachtet wird (s. Abb. 8). Die daraus resultierenden Nachkommen werden bewertet und der Beste gilt dann als Nachkommen der Paarung.

4.2 Populationsstruktur und Generationsmodell

Für das GLEAM-Verfahren wurde zwecks Strukturierung das Nachbarschaftsmodell (s. Abschnitt 3.2) gewählt, weil es viele Freiräume bietet. Setzt man die Nachbarschaftsgröße der Populationsgröße gleich, erhält man Panmixie. Ist die Nachbarschaftsgröße im Ver-

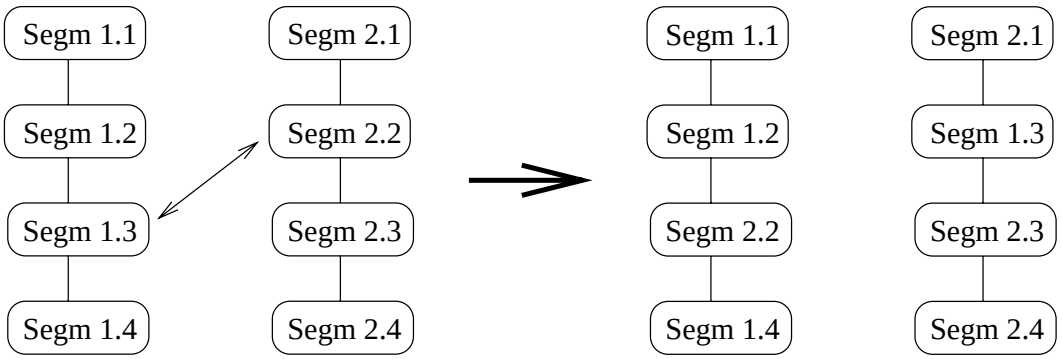

Abbildung 8: Beispiel einer Rekombination

gleich zur Populationsgröße sehr klein, erhält man ein großes Maß an Isolation innerhalb der Subpopulationen. Aber wie auch immer die Nachbarschaftsgröße gewählt wird, haben die besten Individuen immer die Chance, ihre Gene durch die gesamte Population zu propagieren, da das Nachbarschaftsmodell ein Diffusionsmodell ist.

Neben der Einführung einer Strukturierung der Population ist die Benutzung eines kontinuierlichen Generationsmodells, wie in Abbildung 4 b dargestellt, ein weiterer wesentlicher Unterschied zu den klassischen genetischen Algorithmen und Evolutionsstrategien. Gleich nach der Erzeugung eines Nachkommen wird also bestimmt, ob er lebensfähig ist. Ist dies der Fall, so wird eines der Elternteile ersetzt. Es bleibt noch zu bestimmen, welches Elternteil nun ersetzt wird.

Bei der Reproduktion spielt im GLEAM-Verfahren immer ein Elternteil die 'aktive' Rolle. Es sucht sich seinen Partner aus der Nachbarschaft mittels der linearen rangbasierten Selektion aus. Im Gegensatz zu fitneßproportionalen Selektionsverfahren, bei denen die Güte die Auswahl direkt beeinflußt, wird hier die Auswahl durch den Rang des Individuums in der Subpopulation bestimmt. Man hat dabei eine lineare Wahrscheinlichkeitsverteilung, so daß bessere Individuen öfters selektiert werden als schlechte. Außerdem ist die Rekombination zweier Individuen in GLEAM nur dann erlaubt, wenn der Hammingabstand zwischen ihnen nicht zu klein ist, damit sich ihre genetische Information genügend unterscheidet.

Die deterministische Ersetzungsstrategie für das kontinuierliche Generationsmodell in GLEAM lautet: Das Kind ersetzt das 'aktive' Elternteil und somit ein Individuum, dem es ähnlich ist. Die Art der Partnerwahl zusammen mit der Ersetzungsstrategie reduziert die Wahrscheinlichkeit des frühzeitigen Verlusts an genetischer Vielfalt und wirkt dadurch der vorzeitigen Konvergenz auf suboptimale Lösungen entgegen.

4.3 Der Algorithmus

Durch die Einführung des Nachbarschaftsmodells und des kontinuierlichen Generationsmodells sieht der GLEAM-Algorithmus [GS94b] folgendermaßen aus:

1. *Initialisierung:* Es werden gemäß der Initialisierungsstrategie n Individuen für die Startpopulation P generiert und die Nachbarschaften eines jeden Individuums festgelegt.

2. *Iteration:* Solange kein Abbruchkriterium erfüllt ist, werden für jedes Individuum i der Population P folgende Schritte durchlaufen:

   (a) *Partnerwahl:* Ein Partner wird aus der Nachbarschaft des Individuums i durch rangbasierte Selektion ermittelt.

   (b) *Variation und Selektion:* Durch die Anwendung genetischer Operationen (s. Abschnitt 4.1) entstehen Nachkommen. Wenn das beste Kind überlebensfähig ist, so ersetzt es das Individuum i. Ob ein Kind selektiert wird oder nicht, wird durch das Selektionskriterium festgelegt.

Als Abbruchkriterien wurden implementiert: die Erreichung der Lösungsgüte oder einer vorgegebenen Generationszahl, die Überschreitung eines Zeitlimits, die Stagnation der Lösungsgüte oder die Generierung von nicht lebensfähigen Nachkommen über eine festgelegte Anzahl an Generationen.

Als Selektionskriterien wurden die in Abschnitt 3.2 definierten Kriterien übernommen, wobei immer das aktive Elternteil ersetzt wird.

Durch die Initialisierungsstrategie wird festgelegt, wie die Ketten entstehen, die zur Initialisierung der Startpopulation benutzt werden:

- *NEU:* Alle Ketten werden neu ausgewürfelt bis *s_par*[2] Individuen eine hinreichende Fitneß haben.

- *BEST:* Die Individuen werden mit den besten bisher gespeicherten Ketten initialisiert. Falls zuwenig Ketten vorhanden sind, um alle Individuen der Population zu initialisieren, wird der Rest gemäß der Strategie *NEU* aufgefüllt.

- *MIX:* Es werden Ketten mit der Fitneß besser als *s_par* aus den bisher gespeicherten ausgewählt. Alle Ketten müssen verschieden sein. Falls zuwenig Ketten vorhanden sind, wird der Rest wieder gemäß der Strategie *NEU* erzeugt.

- *BEST_NEU:* Die Population wird mit den *s_par* besten bisher gespeicherten Ketten initialisiert. Sind mehr Individuen vorhanden, so wird der Rest gemäß *NEU* aufgefüllt.

- *GEN:* Für die Initialisierung der Anfangspopulation werden *s_par* applikationsspezifisch vorgenerierte Ketten benutzt. Existieren nicht genügend solcher Ketten, so wird der Rest gemäß *NEU* erzeugt.

- *GEN_BEST:* Die Individuen werden, wie auch bei *GEN*, mit applikationsspezifisch vorgenerierten Ketten initialisiert. Reichen diese Ketten nicht für die ganze Population aus, so wird der Rest gemäß *BEST* aufgefüllt.

[2] *s_par* ist ein Strategieparameter, der vom Benutzer anzugeben ist.

- *FROM_FILE:* Ketten werden aus einer Datei eingelesen. Diese Strategie ist vor allem als Aufsetzpunkt nach dem Abbruch einer Evolution gedacht.

Alle Initialisierungsstrategien außer *NEU* und *GEN* sind dazu gedacht, Wissen aus vorherigen Evolutionsläufen in die zu startende Evolution einfließen zu lassen.

Aus diesem Abschnitt wird deutlich, daß verschiedene Parameter vom Benutzer einstellbar sind. Dies geschieht über die Anwendungs- und Experimentalumgebung *GLEAM/AE*.

4.4 GLEAM/AE

GLEAM/AE ist eine Umgebung für unterschiedliche Anwendungen, die sich im günstigsten Fall nur durch den Simulator zur Berechnung der Fitneß eines Lösungsvorschlages unterscheiden (s. Abb 9). Außerdem ist das System für andere Benchmark-Anwendungen, wie sie zur Weiterentwicklung des GLEAM-Verfahrens benötigt werden, erweiterbar.

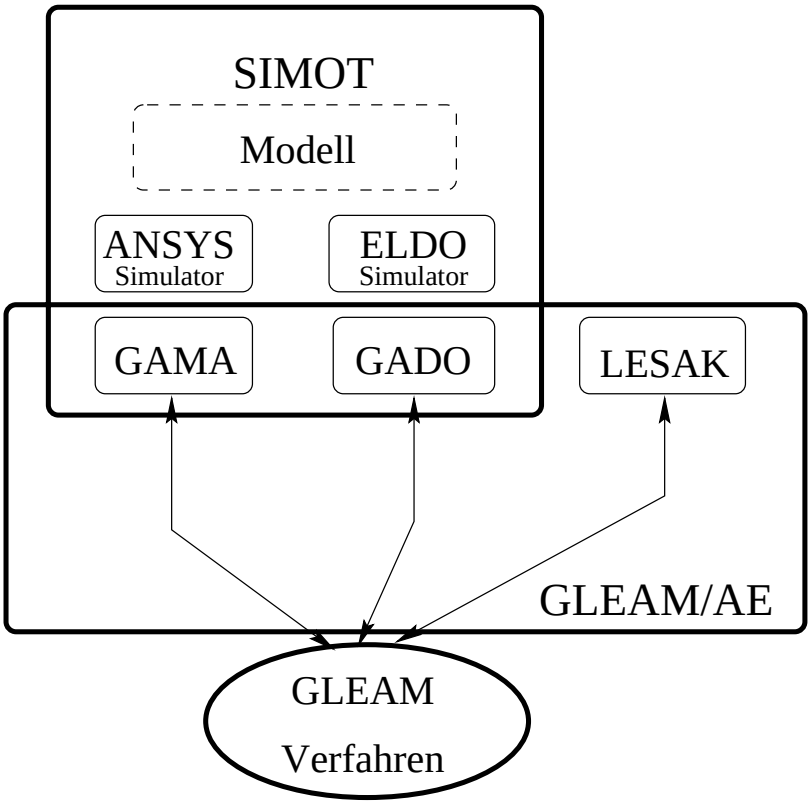

Abbildung 9: Ein Überblick über GLEAM/AE

Mit GLEAM/AE kann der Anwender mittels einer Benutzeroberfläche eine bestimmte Anwendung auswählen und Steuerungsmaßnahmen ergreifen. Es stehen ihm hierfür Menüpunkte verschiedener Funktionalität zur Verfügung. Es können Parameter, wie Evolutionsparameter, Bewertungskriterien oder Systemparameter, eingestellt werden. Bewertungsfunktionen werden auf Wunsch grafisch angezeigt. Ein Ketteneditor ermöglicht es

dem Anwender sich Ketten anzuschauen und zu manipulieren. Außerdem können Ketten und Parametereinstellungen in Dateien gesichert werden, um später auf den gleichen Zustand aufzusetzen. Es stehen auch eine Reihe von Funktionen zur Steuerung des Simulators zur Verfügung. So kann dieser über ein Menü neu gestartet werden, eine Statusanfrage erfolgen oder eine bestimmte Kette zur Bewertung an den Simulator geschickt werden. Es ist auch möglich, mehrere Evolutionsläufe im Batch-Betrieb zu starten. Dabei kann die Evolution vom Benutzer jederzeit unterbrochen werden, um zum Beispiel die Abbruchkriterien zu ändern.

Als nächstes werden zwei Anwendungsbeispiele des GLEAM-Verfahrens (LESAK und GADO), die in GLEAM/AE eingebettet sind, vorgestellt.

4.5 Anwendungen von GLEAM

Der Einsatz des GLEAM-Verfahrens ist, wie bei anderen evolutionären Algorithmen auch, dann sinnvoll, wenn es zu einer Problemstellung kein anderes Lösungsverfahren gibt, das in einer vernünftigen Zeit ein akzeptables Resultat liefert. Solche Aufgaben werden in der Praxis meist mit Hilfe der Erfahrung und Intuition eines Spezialisten gelöst. Beispiele für komplexe Aufgabenstellungen sind unter anderen die Produktionsplanung, die kollisionsfreie Bahnplanung für Industrieroboter und die Designoptimierung.

4.5.1 Kollisionsfreie Bahnplanung für Roboter

LESAK (Lernendes System für Aktionen) (siehe [WJ92]), eine erste Implementierung des GLEAM-Verfahrens, sollte vor allem der Verifizierung und Untersuchung der Methode selbst dienen und war weniger zur Bearbeitung bisher schlecht oder gar nicht gelöster Aufgaben gedacht. Als Anwendung wurde die Bewegungsplanung eines Industrieroboters auf Achsebene gewählt. Dies hatte zwei Gründe:

1. Erstens ist die exakte Lösung der Aufgabe, den Greifer auf einer geraden Bahn zu einem Ziel unter Kollisionsvermeidung mit Hindernissen (und sich selbst) zu bewegen, für 5 und 6-achsige Roboter bekannt und damit ist die von LESAK erzeugte Lösung bewertbar. Sie kann somit als Benchmark dienen, bei der das globale Optimum bekannt ist.

2. Zweitens ist die Bewegungsplanung auf Achsebene hinreichend kompliziert, man denke nur daran, wie schwierig es ist, eine Marionette vernünftig zu bewegen (wobei die Kinematik einer Marionette einfach im Vergleich zu einem 6-achsigen Roboter ist).

Da für bis zu 6-achsige Roboter mathematische Lösungen für die Bewegungsplanung existieren, wurde das Testsystem so ausgelegt, daß es Roboter mit bis zu 16 rotatorischen Achsen bearbeiten kann. Diese Erweiterung der Aufgabe wäre mit keiner konventionellen Robotersteuerung lösbar, mit Hilfe von GLEAM konnten jedoch gute Ergebnisse erzielt werden. Damit ist der Tauglichkeitsnachweis von GLEAM zur Lösung komplexer

Planungsaufgaben exemplarisch erbracht. Es zeigt sich auch (eine hinreichend genaue Modellierung und Simulation vorausgesetzt), daß Ergebnisse erzielt werden können, die der exakten Lösung weit näher kommen, als es für eine ingenieurmäßige Lösung einer Planungs- oder Optimierungsaufgabe notwendig wäre.

4.5.2 Designoptimierung

Der Prozeß des Entwurfs technischer Systeme ist derzeit dadurch gekennzeichnet, daß ausgehend von einem (oder mehreren) Erstentwürfen verschiedene Varianten und eventuell sogar Neudesigns ausprobiert werden bis eine Lösung vorliegt, die als akzeptabel eingestuft wird. Wie gut sie tatsächlich ist, weiß niemand, da das optimale Design unbekannt ist. Der Designprozeß ist also eher eine "trial-and-error-Suche" denn eine systematische Untersuchung.

Es stellt sich nun die Frage, wie man den Prozeß der Entwurfsverbesserung systematisieren und (teil-)automatisieren kann. Bei der Veränderung eines Entwurfs werden bestimmte Parameter variiert. Auch größere Veränderungen lassen sich bis zu einem gewissen Grad als Parametervariationen darstellen. Innerhalb einer so vom Designer festgelegten Bandbreite können nun Entwurfsvarianten auf Vektoren in einem Parameterraum zurückgeführt werden, die innerhalb bestimmter Intervalle veränderbar sind. Mit Hilfe des GLEAM-Verfahrens, daß auch für hochdimensionale Probleme geeignet ist, kann dieser Parameterraum exploriert werden. Dazu ist eine Bewertung vorgeschlagener Designvarianten notwendig, die in der Regel durch eine Simulation des Entwurfs erfolgt. Abbildung 10 verdeutlicht das Zusammenwirken von *Genetischer Maschine*, *Simulation* und *Bewertung*.

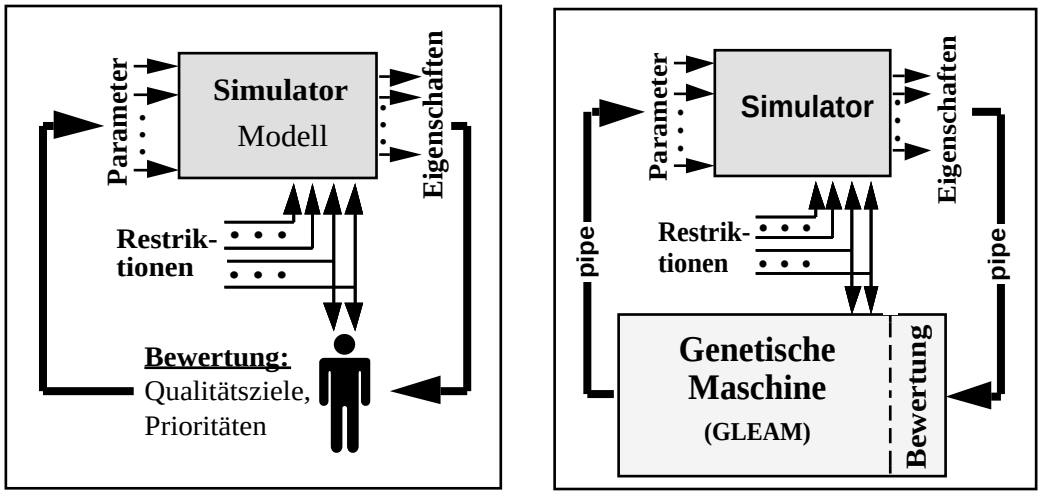

Abbildung 10: Konventioneller Design-Prozeß vs. Design-Optimierungswerkzeug

Das Ziel ist dabei nicht, den Designer zu ersetzen, sondern ihm ein mächtiges Werkzeug in die Hand zu geben. Auch mit *GADO* (Genetischer Algorithmus zur Design-Optimierung) verbleiben die wesentlichen kreativen Aufgaben beim Menschen: Erstellung von Erstentwürfen, präzise Festlegung der Entwurfsziele, korrigierender Eingriff in den teilauto-

matisierten Entwurfsprozeß. GADO ist in SIMOT integriert (s. Abb. 9). Mit Hilfe von GADO wird zur Zeit das Design einer Mikropumpe optimiert. Die Simulationszeit zur Bestimmung der Fitneß eines Individuums beträgt drei bis vier Minuten. Für eine Population von 60 Individuen benötigte der Algorithmus für 57 Generationen 23 Tage (siehe [GSJM+96]). Um diesen Prozeß zu beschleunigen, sollen nun diese Berechnungen (Simulationen) parallel durchgeführt werden. Bevor auf die Konzepte der verteilten Abarbeitung von GLEAM eingegangen wird, wird zuerst das Werkzeug, PVM, vorgestellt.

5 PVM – ein Überblick

Das Softwaresystem PVM (Parallel Virtual Machine) ist im Forschungsbereich entstanden. Es wurde an der Emory University in Zusammenarbeit mit dem Oak Ridge National Laboratory für heterogene Rechnernetze entwickelt. Die Implementierung ist im Quellcode verfügbar und wurde von verschiedenen Herstellern für ihre Hardware optimiert.

PVM konzentriert sich in erster Linie auf die breitestmögliche Nutzbarmachung vernetzter *heterogener* Rechnerressourcen. Das Netzwerk ist hierbei in der Regel ein Standard-LAN und leichte Portierbarkeit und Interoperabilität spielen die herausragende Rolle.

Hintergründe dieser Überlegungen sind sicherlich die Vorteile, die ein verteiltes System mit sich bringt:

- Vorhandene Rechner können durch verteiltes Rechnen besser ausgelastet werden. Eine Anschaffung von Multiprozessorsystemen ist somit meistens nicht mehr notwendig.
- Durch die Verwendung aller Speicher der benutzten Rechner kann lokaler Platzmangel vermieden werden.
- Bestimmte Aufgaben können auf Spezialrechnern (z.B. Vektorrechner, DB-Server) ausgeführt werden.
- Verteilte Systeme können fehlertolerant programmiert werden. Wenn ein Rechner ausfällt, muß nicht gleich das gesamte System zusammenbrechen.

Nachteile verteilter Systeme sind die erschwerte Gewährleistung der Sicherheit und die teilweise immer noch relativ langsame Nachrichtenübermittlung über das lokale Netz.

Mit PVM hat man ein komplettes Softwaresystem zur Verfügung, das es Programmierern erlaubt, viele Prozesse in mehreren vernetzten Rechnern unterschiedlicher Hersteller und unterschiedlicher Prozessor-Architektur gemeinsam so zu nutzen, als hätten sie einen einzigen einheitlichen Parallelrechner zur Verfügung. Der so gebildete 'virtuelle' Parallelrechner stellt sich als ein System mit verteiltem Speicher dar, in dem die Daten als Nachrichten explizit transportiert werden müssen. Diese Architektur ist der kleinste gemeinsame Nenner, der auf allen in Betracht gezogenen realen Systemen ausreichend effizient implementiert werden kann. Die Palette der Systeme, auf denen PVM benutzt werden

kann, reicht von verschiedenen Workstations im Netzwerk über Vektorrechner, dedizierte Cluster und speichergekoppelte Multiprozessorsysteme bis hin zu einem bunten Gemisch dieser Typen.

Die folgenden Prinzipien sind dem Design von PVM zugrunde gelegt:

- Der Anwender soll eine virtuelle Maschine zur Ausführung seines PVM-Programms aus der Menge der vorhandenen Systeme individuell konfigurieren können. Die virtuelle Maschine darf sogar während des Programmablaufs dynamisch verkleinert oder vergrößert werden.
- PVM-Programme bestehen aus einzelnen Tasks, die in der Regel als Unix-Prozesse implementiert sind und die logisch nicht die Zahl oder Art der vorhandenen Prozessoren widerspiegeln müssen. Voraussetzung ist natürlich, daß Multitasking auf den einzelnen Prozessoren unterstützt wird.
- Die Tasks eines PVM-Programms können verschiedene Aufgaben erledigen und man kann auch bestimmen auf welchem Rechner oder Rechnertyp bestimmte Tasks laufen sollen.
- Kommunikation zwischen den Tasks erfolgt durch das explizite Senden und Empfangen von Datenpaketen. Die Größe dieser Nachrichten ist nur durch den verfügbaren Speicher begrenzt. PVM nutzt nach Möglichkeit lokale Kommunikationsmechanismen von Multiprozessormaschinen.
- PVM unterstützt heterogene Cluster und erledigt die Konvertierung von Datentypen, wo eine solche erforderlich ist.
- PVM stellt Mechanismen zur Verfügung, die es ermöglichen, den Ausfall von Teilen der virtuellen Maschine zu bemerken und darauf zu reagieren. Dadurch kann man in PVM eine recht weitgehende Fehlertoleranz implementieren.
- Zur Installation von PVM sind keine Superuser-Rechte erforderlich, es genügt ein gewöhnliches Login auf jedem der benutzten Systeme.

Grundsätzlich besteht PVM aus zwei Teilen: der erste ist ein Dämon-Prozeß, der auf jedem der beteiligten Systeme laufen muß und der für das Verwalten und Koordinieren der Tasks eines PVM-Programms verantwortlich ist. Der andere Bestandteil sind Bibliotheken mit den PVM-Routinen, die die Anwender in ihren Programmen benutzen können. Die Routinen dienen im wesentlichen dem dynamischen Konfigurieren der virtuellen Maschine, der Prozeßsteuerung und der Kommunikation. PVM unterstützt nichtblockierendes Senden, blockierendes und nichtblockierendes Empfangen und das Verschicken einer Nachricht an eine Auswahl von Tasks (Broadcast). Gegenwärtig unterstützt das PVM-Paket die Sprachen C/C++ und Fortran.

Das Programmiermodell von PVM legt nur fest, daß eine Anwendung aus einer Menge von Tasks besteht. Jede Task erledigt dabei einen Teil des Rechenaufkommens der Anwendung. PVM unterstützt sowohl Funktionsparallelität (Programmparallelität) als auch Datenparallelität, ja sogar eine Mischung der beiden (s. Abb. 11). Funktionsparallelität liegt vor,

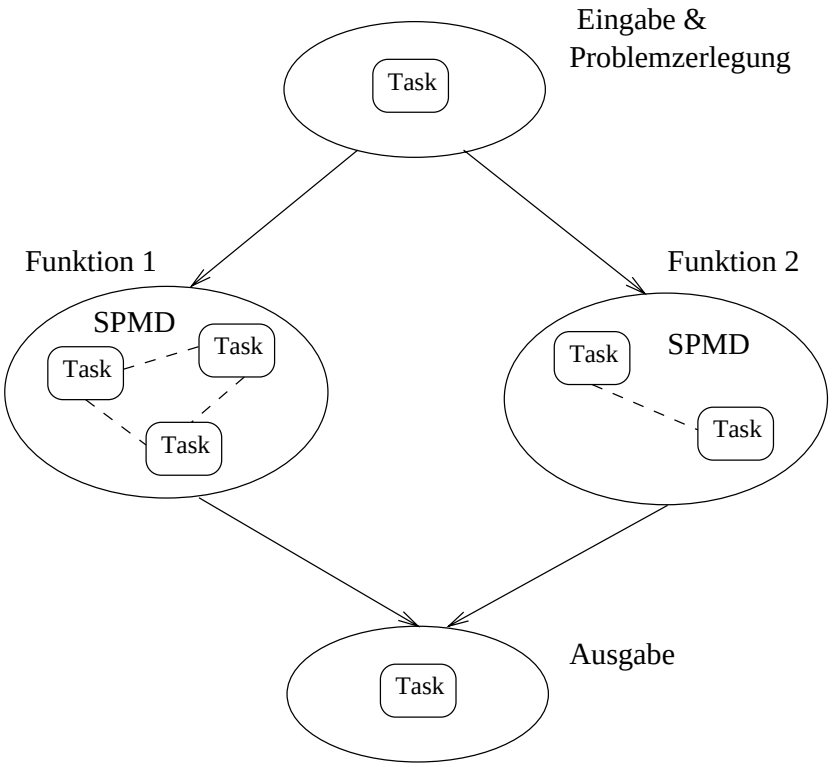

Abbildung 11: Das Programmiermodell von PVM

wenn jeder Task eine andere Aufgabe (z.B. Eingabe, Problemzerlegung, Problemlösung, Ausgabe) löst. Bei der Datenparallelität, auch als SPMD-Modell (single program multiple data) bekannt, führen alle Tasks das gleiche Programm aus, bearbeiten allerdings jeweils nur einen Teil der Daten. Zwischen den Tasks einer Anwendung kann, obwohl nicht immer erforderlich, Datenaustausch oder gegenseitige Synchronisation stattfinden.

Nachdem nun sowohl das GLEAM-Verfahren als auch das zu benutzende Werkzeug vorgestellt wurden, beschäftigen sich die nächsten beiden Abschnitte mit dem Entwurf der verteilten Version des GLEAM-Verfahrens.

6 Konzepte zur verteilten Abarbeitung von GLEAM

Bei dem Entwurf der verteilten Version des GLEAM-Verfahrens war als Rahmenbedingung zu beachten, daß sie auch in die Anwendungs- und Experimentalumgebung GLEAM/AE eingebettet wird. Dies hat zur Folge, daß die parallel laufenden Prozesse so konzipiert werden müssen, daß sie zusätzlich zur Durchführung der Evolution z. B. auch auf die durch den Benutzer von GLEAM/AE angeforderte Unterbrechung eines Evolutionslaufes oder auf Änderungen der Zielwerte reagieren können.

Um dies zu realisieren bietet sich als Programmiermodell das *Master-Slave*-Modell an.

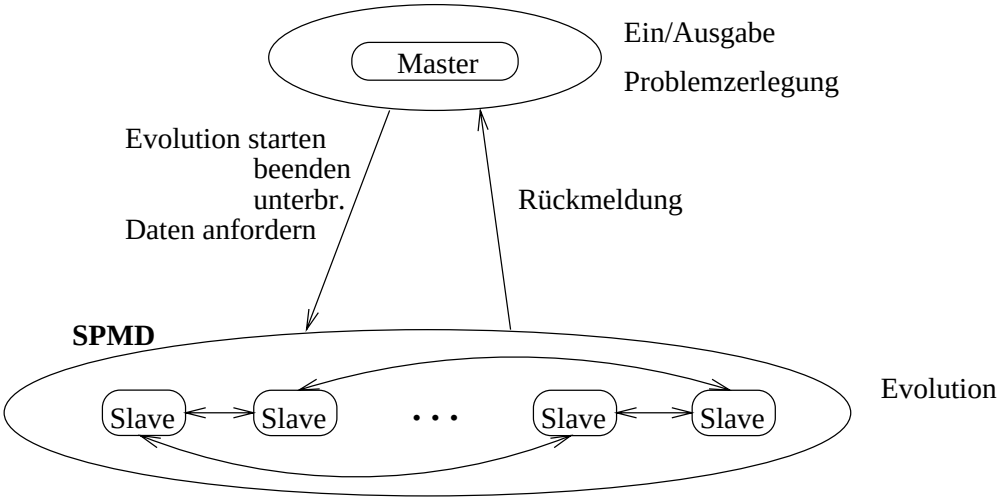

Abbildung 12: Das für GLEAM gewählte Programmiermodell

Ein Prozeß, der Master, übernimmt die nicht parallelisierbaren Aufgaben des sequentiellen Programms wie Benutzerdialog (Ein- und Ausgabe), Fehlerbehandlung, Problemzerlegung, etc. Dieser Prozeß ist der einzige, der die Wünsche des Anwenders entgegennimmt. Er hat somit die zusätzliche Aufgabe, die restlichen Prozesse, Slaves genannt, zu steuern. Die Slaves leisten dann die tatsächliche Arbeit, d. h. sie führen die Evolution der Individuen durch. Der Master einerseits und die Gesamtheit der Slaves andererseits erledigen verschiedene Aufgaben, es liegt also Funktionsparallelität vor. Die Slaves hingegen führen alle das gleiche Programm aus; sie bearbeiten jeweils die ihnen vom Master zugeteilten Individuen. Zwischen den Slaves herrscht somit Datenparallelität (SPMD). Das Master-Slave-Programmiermodell ist in Abbildung 12 grafisch dargestellt.

Kommunikation zwischen Master und Slaves ist unbedingt notwendig, denn ansonsten wäre keine Steuerung der Slaves durch den Master möglich. Auch könnten die Slaves das Fortschreiten der Evolution dem Master, und somit dem Benutzer, nicht mitteilen. Es stellt sich nun die Frage, ob Informationsaustausch zwischen den Slaves notwendig ist. Diese Frage wird im nächsten Unterabschnitt beantwortet.

6.1 Aufteilung der Population

Wie bereits erwähnt, ist eine der Aufgaben des Masters die Slaves mit zu verarbeitenden Daten, d. h. Individuen, zu versorgen. Er muß die Gesamtpopulation auf die vorhandenen Slave-Prozesse so aufteilen, daß diese möglichst gleich ausgelastet sind.

Dem GLEAM-Verfahren liegt das Nachbarschaftsmodell zugrunde und die Nachbarschaften der einzelnen Individuen überlappen sich (s. Abb. 6). Für die Randindividuen der Teilpopulation eines Slaves bedeutet dies, daß sich immer ein Teil der Nachbarn auf einem anderen Slave befinden werden. So zum Beispiel ist bei einer Ringpopulation mit einer Populationsgröße von 10 Individuen, einer Nachbarschaftsgröße von 2 und zwei Slaves je-

weils ein Nachbar der Randindividuen 1, 5, 6 und 10 einem anderen Slave-Prozeß zugeteilt (s. Abb. 13). Dadurch wird Informationsaustausch auch zwischen den Slaves notwendig.

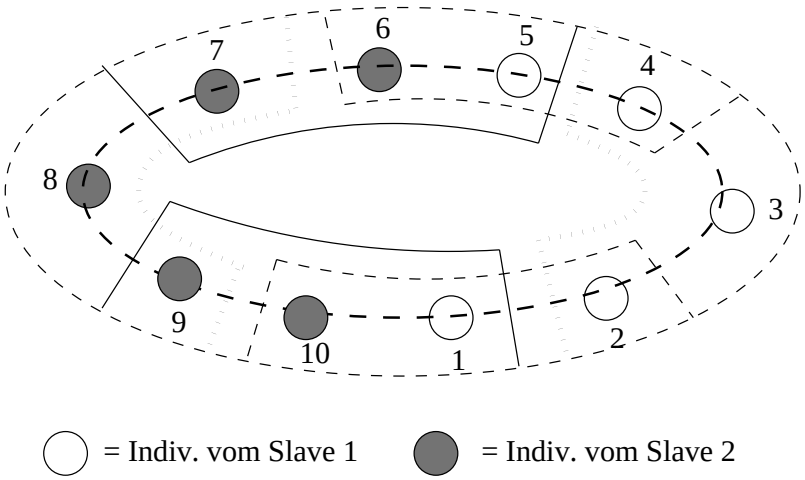

Abbildung 13: Verteilung der Individuen einer Population auf Slave-Prozesse

Bezüglich des Zeitpunktes der Entscheidung, wieviel Individuen welchem Slave überlassen werden, können zwei Strategien unterschieden werden: *statisches* und *dynamisches* Scheduling. Bei dem statischen Scheduling teilt der Master jedem Slave eine feste, meistens in etwa gleiche, Anzahl an Individuen zu. Bei dem dynamischen Scheduling liegt die Kontrolle beim Master. Er überläßt jedem Slave-Prozeß eine bestimmte Anzahl von Individuen und versorgt diesen, sobald er seine Arbeit erledigt hat, mit neuen Individuen. Dadurch bearbeiten Prozesse, die auf einem leistungsfähigeren oder nicht so ausgelasteten Rechner laufen, mehr Individuen. Das dynamische Scheduling hat also den Vorteil, daß man damit flexibel auf eine sich verändernde Rechenlast des Systems reagieren kann. Das dynamische Scheduling ist aber leider nicht für Probleme geeignet, bei denen die Slave-Prozesse von Zeit zu Zeit untereinander Informationen austauschen. In diesem Fall müßte nämlich jeder Slave, bevor er mit einem anderen kommuniziert, die Information vom Master einholen, wo sich die Individuen, die ihn interessieren, gerade befinden. Dies würde zu einem höheren Kommunikationsaufwand und vor allem zu einem Engpaß beim Master führen.

Für das GLEAM-Verfahren wurde das statische Scheduling gewählt, damit Informationsaustausch direkt, d. h. ohne Einbeziehung des Masters, stattfinden kann. Um die Anzahl der Individuen zu bestimmen, die einem Slave-Prozeß zugeteilt werden, geht der Master folgendermaßen vor: Als erstes wird $n/nproz$ bestimmt, wobei n die Größe der Population und $nproz$ die Anzahl der verfügbaren Slaves ist. Wenn bei der Division von n durch $nproz$ ein Rest r ($r \neq 0$) übrigbleibt, so werden die ersten r Slave-Prozesse jeweils ein Individuum mehr als die restlichen $nproz - r$ bearbeiten müssen.

Nachdem der Master die Populationsgröße der einzelnen Teilpopulationen bestimmt hat, kann der eigentliche Evolutionsprozeß beginnen. Dafür muß er die Slaves mit den für die Evolution notwendigen Parameter wie Populationsgröße, Nachbarschaftsgröße, Initialisie-

rungs- und Überlebensstrategie, etc versorgen. Danach muß die Population initialisiert werden.

6.2 Initialisierung der Population

Wie in Abschnitt 4.3 beschrieben, wurden für GLEAM die Initialisierungsstrategien NEU, BEST, MIX, BEST_NEU, GEN, GEN_BEST und FROM_FILE definiert.

Bei der Initialisierungsstrategie NEU und GEN können alle Slaves die Initialisierung parallel ausführen, da kein globales Wissen nötig ist. Auch die Initialisierungsstrategie FROM_FILE benötigt keine Informationen vom Master. Jeder Slave liest seine Daten aus einer Datei aus.

Die sonstigen Initialisierungsstrategien (BEST, BEST_NEU, MIX, GEN_BEST) hingegen benötigen globales Wissen, d. h. daß nun auch der Master zum Einsatz kommt. Er wählt Ketten entsprechend der Strategie aus den gespeicherten Ketten aus und verteilt sie zufällig auf die Slaves. Wenn es aber nicht genug gespeicherte Ketten gab, um alle Individuen zu initialisieren, erzeugt er keine neuen. Die Generierung neuer Ketten überläßt er den Slaves, um den Kommunikationsaufwand als auch den sequentiellen Anteil des Programms so gering wie möglich zu halten.

Die Initialisierungsphase ist kritisch: Tritt ein Fehler bei der Initialisierung einer der Slaves auf, so ist es nicht sinnvoll, die Durchführung der Evolution zu starten. Dies liegt daran, daß wegen des Nachbarschaftsmodells jeder Slave zumindest ein Individuum x besitzt, das Nachbar eines Individuums y ist, das sich auf einem anderen Slave befindet. Da das Individuum y seinen Partner durch rangbasierte Selektion ermittelt, muß er die Fitneß aller seiner Nachbarn kennen. Außerdem wird durch den Ausfall eines Slaves die Populationsstruktur zerstört und der Informationsfluß zwischen den Subpopulationen behindert. Wie Fehler, die in der Initialisierungsphase auftreten, behandelt werden, wird in Abschnitt 6.5 beschrieben.

Nachdem alle Teilpopulationen erfolgreich initialisiert wurden, kann der Evolutionsprozeß beginnen.

6.3 Der Evolutionsprozeß

Innerhalb einer Teilpopulation wird die Evolution der Individuen sequentiell durchgeführt. Der Evolutionsprozeß besteht aus der Partnerwahl, der Erzeugung von Nachkommen durch die Anwendung genetischer Operationen und der Selektion von Nachkommen.

Die Partnerwahl erfolgt durch rangbasierte Selektion (s. Abschnitt 4.3). Dies bedeutet, daß ein Individuum die Fitneß aller seiner Nachbarn kennen muß. Wenn einige seiner Nachbarn nicht in seiner Teilpopulation enthalten sind, so muß er diese Information von den entsprechenden Slaves anfordern. Wird einer dieser nicht lokalen Individuen als Partner gewählt, so muß zusätzlich das Individuum zwecks Rekombination angefordert werden. Diese einfache Art und Weise der Anforderung von Individuen bei Bedarf hat aber einen

gravierenden Nachteil: Tritt in einem der Slave-Prozesse während der Evolution ein Fehler auf, so schlägt die Evolution auch in allen anderen Slaves, die Informationen von diesem brauchen, fehl.

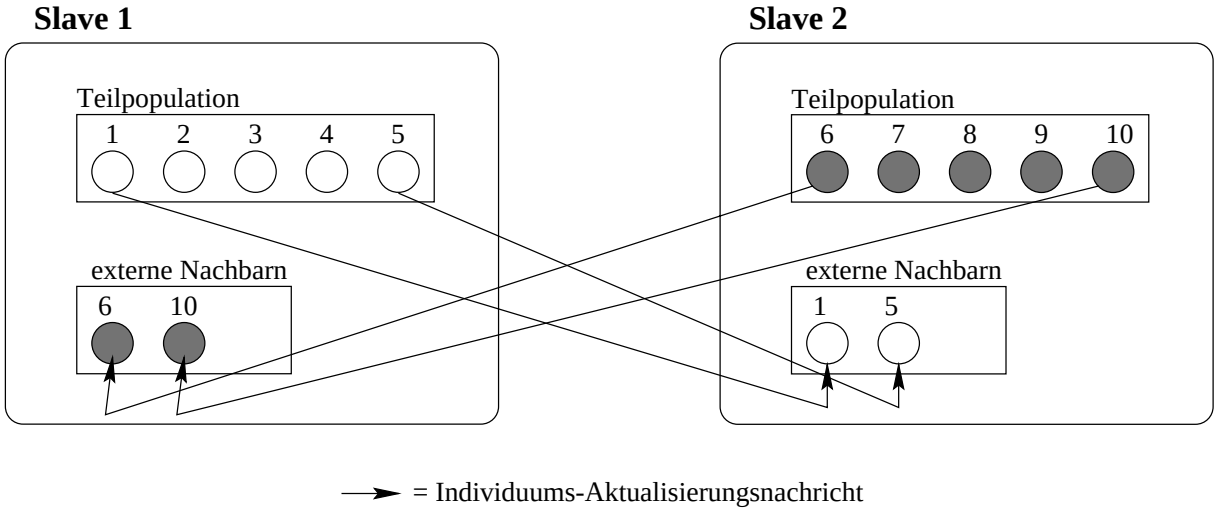

Abbildung 14: Die Individuen einer Teilpopulation und ihre externen Nachbarn für das in Abb. 13 vorgestellte Beispiel

Abhilfe für das Problem der engen Kopplung zwischen den Slave-Prozessen schafft folgende Vorgehensweise: Nachdem die Initialisierung abgeschlossen ist, schickt jeder Slave als erstes alle Individuen, die als Nachbarn in anderen Slaves gebraucht werden, zu. Dadurch kennt jeder Slave die komplette Nachbarschaft seiner Individuen, also auch die 'externen' Nachbarn (s. Abb. 14). Danach wird die Evolution gestartet. Ändert sich nun ein Individuum eines Slaves, so schickt dieser dieses Individuum an die Slaves, die es als Nachbar kennen (Individuums-Aktualisierung). So wird der Informationsfluß zwischen den Teilpopulationen gesichert. Fällt aber ein Slave aus, so wirkt sich das auf die anderen Slaves nur durch eine Stagnation der betroffenen Nachbarn aus, denn sie erhalten keine Aktualisierungsnachrichten mehr.

Wie auch in Abschnitt 3 beschrieben, benötigt die Anwendung genetischer Operationen kein globales Wissen. Auch die Selektion der Nachkommen kommt im GLEAM-Verfahren ohne weitere globalen Informationen aus: Ihre Fitneß wird nur lokal, d. h. nur mit der Fitneß der Individuen aus der Nachbarschaft des aktiven Elternteils, verglichen. Die Information über die Fitneß der Nachbarschaft eines Individuums ist sowieso vorhanden, da sie ja schon für die Partnerwahl benötigt wird. Wird nun eines der Nachkommen selektiert, so ersetzt es immer das aktive Elternteil und somit ein lokal vorhandenes Individuum.

Der Evolutionsprozeß wird solange ausgeführt, bis eines der Abbruchkriterien erfüllt ist.

6.4 Die Abbruchkriterien

Als Abbruchkriterien wurden für das GLEAM-Verfahren definiert:

- Erreichung einer vorgegebenen Fitneß für ein Individuum der gesamten Population;
- Die Überschreitung eines Zeitlimits;
- Die Überschreitung einer Maximalanzahl von Generationen;
- Die Stagnation der Lösungsgüte über eine bestimmte Anzahl von Generationen;
- Die Erzeugung von nicht lebensfähigen Nachkommen über eine bestimmte Anzahl von Generationen.

Es stellt sich nun die Frage, wie diese Abbruchkriterien in der verteilten Version des GLEAM-Verfahrens realisiert werden. Die erste Bedingung, das Erreichen einer vorgegebenen Fitneß f für ein Individuum, kann von den Slaves überprüft werden. Wird in einer Teilpopulation ein Individuum mit der Fitneß f generiert, so bricht der entsprechende Slave die Evolution ab und teilt dies dem Master mit. Der Master muß dann dafür sorgen, daß auch die übrigen Slaves anhalten.

Die Prüfung auf Überschreitung eines Zeitlimits oder einer Maximalanzahl an Generationen könnte prinzipiell sowohl von den Slaves als auch vom Master durchgeführt werden. Überläßt man diese Aufgabe den Slaves, so werden sie fast immer zu unterschiedlichen Zeitpunkten aufhören, da die Erfüllung dieser beiden Bedingungen in hohem Maße von der Leistungsfähigkeit des Rechners, auf dem die Slaves laufen, abhängt. Der Master müßte also warten, bis der langsamste der Slaves abbricht.

Die Durchsetzung der Einhaltung eines Zeitlimits ist vom Master leicht zu realisieren. Außerdem wird somit insgesamt weniger Rechenzeit verbraucht, weil nur ein Prozeß sich um die Berechnung dieser Bedingung kümmern muß. Komplizierter fällt allerdings die Überprüfung der Überschreitung einer Maximalanzahl an Generationen aus. Dafür benötigt er Daten von den Slaves, die den Fortschritt der Evolution in den Teilpopulationen dokumentieren. Diese Daten erhält der Master sowieso, weil dies die Einbettung in GLEAM/AE[3] erfordert. Bei dem Entwurf des Masters hat man nun den Entscheidungsspielraum die Evolution dann abzubrechen, wenn einer der Slaves, alle Slave im Mittel oder alle Slaves die vorgegebene Maximalanzahl an Generationen erreicht haben. Bricht man die Evolution bereits ab, wenn nur einer der Slave-Prozesse eine bestimmte Anzahl an Generationen berechnet hat, so ist dies, wenn man die Heterogenität der meisten Rechnernetze betrachtet, zu früh. Wartet man auf alle Slaves, so hat man gegenüber der Berechnung der Abbruchbedingung durch die Slave-Prozesse keinen Vorteil. Die beste Lösung ist der Abbruch der Evolution, wenn der Mittelwert über die Anzahl der berechneten Generationen aller Slave-Prozesse die vorgegebene Anzahl erreicht. Die Abbruchbedingung lautet also:

$$\frac{\sum(Anzahl\ berechneter\ Generationen\ pro\ Slave)}{Anzahl\ Slaves} \geq Maximalanzahl\ an\ Generationen.$$

Man hat dabei den Vorteil, daß die Slave-Prozesse gleichmäßig ausgelastet werden: Prozesse, die schneller vorankommen, berechnen mehr Generationen als die die dies nicht können.

[3] Bestimmte Daten, wie Güte des besten Individuums der Population, Generationsnummer, etc werden dem Benutzer während des Evolutionslaufes angezeigt.

Die letzten zwei Bedingungen, die Stagnation der Lösungsgüte bzw. die Erzeugung von nicht lebensfähigen Nachkommen über eine bestimmte Anzahl an Generationen, sind eigentlich dazu gedacht, die Evolution dann abzubrechen, wenn kein Fortschritt mehr zu erwarten ist. Dies bedeutet, daß in der verteilten Version des GLEAM-Verfahrens diese Bedingungen in *allen* Teilpopulationen erfüllt sein müssen, da sonst noch ein Fortschritt erzielt werden kann. Diese Bedingungen können somit nur vom Master überprüft werden, wobei wiederum Daten von den Slaves erforderlich sind.

In den bisherigen Unterabschnitten wurde bereits deutlich, daß verschiedene Fehler auftreten können. Welche Möglichkeiten es gibt auf diese Fehler zu reagieren wird nun im nächsten Unterabschnitt erläutert.

6.5 Fehlerbehandlung

Die Ursachen des Auftretens eines Fehlers sind vielfältig: Es können Programmierfehler sein, die einen Prozeß zum Absturz bringen oder aber auch Speichermangel, der das Fortführen eines Programms unmöglich macht. In einer verteilten Umgebung kommen noch weitere Fehlerursachen hinzu, wie z. B. Kommunikationsfehler.

Arbeiten nun mehrere Prozesse zur Lösung eines Problems zusammen, so muß auch definiert werden, wie ein Prozeß auf Kommunikationsfehler oder auf den Absturz eines kooperierenden Prozesses reagieren soll.

Kommunikationsfehler können zum einen die Verfälschung, zum anderen aber auch das Verlorengehen einer Nachricht sein. Weil es vorkommen kann, daß Nachrichten gar nicht ankommen, sollte ein Prozeß nicht unendlich lange auf eine Nachricht warten. Erhält ein Prozeß keine oder eine verfälschte Nachricht, so kann er als erstes diese nochmal vom sendenden Prozeß anfordern. Erhält er die Nachricht nach einer bestimmten Zeit immer noch nicht, so bleibt festzustellen, wie wichtig diese Nachricht für ihn ist. Handelt es sich um wichtige Daten, ohne die der Prozeß nicht weiterarbeiten kann, so muß auch er beendet werden.

Beim Master-Slave-Programmiermodell spielt ein Prozeß, der Master, eine besondere Rolle: Er koordinert die restlichen Prozesse. Deswegen sollten beim Absturz des Masters auch alle Slave-Prozesse beendet werden. Stürzt hingegen ein Slave-Prozeß ab, so hat dies nicht unbedingt eine fatale Wirkung auf andere Prozesse, es sei denn er besaß Daten, die das Fortfahren der restlichen Prozesse unmöglich macht.

Es ist die Aufgabe des Masters, auf Abstürze der Slaves zu reagieren, da der Master eine koordinierende Rolle spielt. Eine Reaktionsmöglichkeit wäre sicherlich, einen neuen Slave-Prozeß zu starten. Diese Methode birgt das Risiko, daß dieser Prozeß erneut abstürzt, vor allem weil er die gleichen Eingabedaten erhält. Für das GLEAM-Verfahren würde außerdem wegen der statisch festgelegten Aufteilung der Population ein nicht unerheblicher Verwaltungsaufwand entstehen: Allen Slaves, deren externe Nachbarn diesem neuen Prozeß zugeteilt wurden, müssen benachrichtigt werden. Außerdem darf keine Kommunikation mit diesem Prozeß stattfinden, bis seine Initialisierungsphase nicht abgeschlossen ist. Der Master muß allerdings sicherstellen, daß Prozesse, die auf wichtige Nachrichten

von ihm warten, nicht terminieren. Noch komplizierter ist der Fall, bei dem der Absturz eines Prozesses durch den Ausfall eines Rechners verursacht wird. Dann muß der Master zuerst einen entsprechenden Rechner finden. Für den Fall, daß die Anwendung einen externen Simulator benutzt, bräuchte man 'Reserve'-Rechner, da oft nur ein Simulator pro Rechner gestartet werden kann. Stürzen mehrere Prozesse ab, gelangt man mit dieser Methode in eine Sackgasse. Wegen des großen Verwaltungsaufwands und der notwendigen Berücksichtigung von Sonderfällen wurde diese Methode nicht implementiert.

Eine andere Möglichkeit Abstürze von Slave-Prozessen zu behandeln besteht darin, diese wie andere schwerwiegende Fehler der Slaves, z. B. Fehler des Simulators, zu behandeln. Der Master unterscheidet bei diesen Fehlern, ob sie während der Initialisierungsphase oder während des Evolutionsprozeßes stattfinden.

In der Initialisierungsphase werden die Individuen der einzelnen Slaves generiert. Tritt ein Fehler auf, so wird dies vom Master registriert. Er startet dann die Initialisierungsphase erneut, wobei er die Gesamtpopulation nur auf die Prozesse verteilt, deren erste Initialisierung problemlos verlaufen ist. Schlägt die Initialisierung auch ein zweites Mal fehl, so wird die Evolution abgebrochen. Die Wahrscheinlichkeit, daß die Initialisierung auch beim zweiten Mal nicht erfolgreich beendet wird, ist allerdings sehr gering.

Als nächstes findet zwischen den Slaves Kommunikation statt, um ihre externen Nachbarn zu initialisieren. Stürzt in dieser Phase ein Prozeß ab oder ist er über einen längeren Zeitraum nicht erreichbar, so schlägt die Evolution auch in den Prozessen fehl, deren externe Nachbarn sich auf dem betroffenen Prozeß befinden. Eine andere Möglichkeit wäre sicherlich auch die Initialisierung der betroffenen externen Nachbarn mit Individuen von anderen Slaves durchzuführen. Dies würde jedoch die Populationsstruktur zerstören.

Während der tatsächlichen Evolution hat der Absturz eines Prozesses die Verringerung der Population um die Anzahl der Individuen, die von dem Prozeß bearbeitet wurden, zur Folge. Er behindert aber nicht die Evolution der restlichen Teilpopulationen. Die Gesamtpopulation verhält sich so, als ob in einer Teilpopulation keine Veränderung stattfindet. Dies wirkt sich wie eine Art Informationsbarriere aus.

Diese Methode der Fehlerbehandlung benötigt den korrigierenden Eingriff des Masters nur in der Initialisierungsphase. Als kritisch ist eigentlich nur die Initialisierung der externen Nachbarn zu bewerten. Der Absturz eines Slave-Prozesses kann hier, wenn die externen Nachbarn eines Prozesses von mehreren Slaves kommen, den Absturz weiterer Slave-Prozesse nach sich ziehen. Diese Phase ist aber, gemessen an der Dauer der Evolution, sehr kurz. Die Evolutionsphase hingegen, die auch Tage dauern kann, ist fehlertolerant: Ein schwerwiegender Fehler in einem Slave-Prozeß behindert die restlichen Slaves nicht. Diese Methode der Fehlerbehandlung wurde implementiert, weil sie ein hohes Maß an Fehlertoleranz bietet aber nur einen geringen Verwaltungsaufwand und keine Berücksichtigung von Sonderfällen benötigt.

Es stellt sich die Frage, ob man der Informationsbarriere, die durch den Absturz eines Slaves zustande kommt, entgegenwirken kann. Dies ist möglich, indem die konstant bleibende Teilpopulation überbrückt wird: Die externen Nachbarn der Slaves, die von der konstanten Teilpopulation stammen, werden durch Individuen anderer Teilpopulationen ersetzt. Diese Vorgehensweise hat allerdings zwei gravierende Nachteile. Zum einen geht

genetische Information verloren, da die Individuen der konstant bleibenden Teilpopulation nicht weiter betrachtet werden. Andererseits wird dadurch die Populationsstruktur zerstört, außer wenn es sich um eine lineare Struktur (z. B. Ring) handelt. Hinzu kommt der Verwaltungsaufwand des Masters, der bestimmen muß, welche Teilpopulationen nun als benachbart gelten. Außerdem ist die erneute Initialisierung externer Nachbarn notwendig. Da das Verfahren zum Aufheben der Informationsbarriere eine Reihe von Nachteilen mit sich bringt, wurde es auch nicht implementiert.

Zusammengefaßt sieht die Fehlerbehandlung für die in einer verteilten Umgebung zusätzlich vorkommenden Fehler folgendermaßen aus:

- Wird eine Verfälschung von Nachrichten durch Kommunikationsfehler erkannt, so erfolgt eine erneute Anforderung der Daten, wenn diese wichtig sind (z. B. Evolutionsparameter, Initialisierung externer Nachbarn). Eine erneute Anforderung entfällt jedoch bei verfälschten Individuums-Aktualisierungsnachrichten, da diese, wenn sie selten vorkommen, für die Weiterführung der Evolution keine determinierende Rolle spielen.
- Das Verlorengehen von Nachrichten und das daraus resultierende lange Warten von Prozessen wird unterbunden durch das Setzen von Zeitschranken.
- Beim Absturz des Master-Prozesses werden auch die Slave-Prozesse beendet.
- Das Auftreten von Fehlern (Abstürze eingeschlossen) bei der Initialisierung der Slaves führt dazu, daß der Master eine erneute Initialisierung startet. Allerdings verteilt er die Population nur auf diejenigen Slaves, die die Initialisierungsphase zuvor erfolgreich beendet haben.
- Abstürze von Slave-Prozessen während der kurzen Phase des Austauschs von externen Nachbarn sind kritisch: Der Absturz eines Prozesses kann das Beenden weiterer Prozesse nach sich ziehen, die Informationen vom abgestürzten Prozeß benötigen.
- Fehler, die während der Durchführung der Evolution in den Slaves auftreten, benötigen nicht den korrigierenden Eingriff des Masters. Die restlichen Slaves werden durch den Fehler eines Slaves in der Fortführung ihres Evolutionsprozesses nicht behindert. Erst wenn in *allen* Slaves die Evolution fehlgeschlagen hat, gilt der Evolutionsprozeß auch für den Master als abgebrochen.

Nachdem nun die wesentlichen Merkmale beschrieben wurden, die für die verteilte Version des GLEAM-Verfahrens wichtig sind, bleibt noch festzuhalten, wie sie in die Anwendungs- und Experimentalumgebung GLEAM/AE eingebettet wird.

6.6 Einbettung in GLEAM/AE

Durch die Einbettung in GLEAM/AE müssen sowohl der Master als auch die Slaves zusätzliche Aufgaben bewältigen. Der Master muß zum einen Unterbrechungswünsche des Benutzers an die Slaves weitergeben, zum anderen Menüpunkte bereitstellen, über

die Parameter eingestellt und Informationen abgefragt werden können. Die Slaves müssen während der Evolution den Master periodisch mit Informationen über deren Fortschritt versorgen, damit dieser die Informationen dem Benutzer zur Verfügung stellen kann.

Wenn der Benutzer die Evolution unterbricht, könnte man als erstes meinen, daß die Slave-Prozesse, die im Hintergrund laufen, nicht auch angehalten werden müssen. Der Benutzer erzielt durch die Unterbrechung den gewünschten Effekt, weitere Menüpunkte aufrufen zu können, während die Evolution weiterläuft. Wenn die Evolution weiterläuft, so verschicken die Slaves Informationen an den Master. Werden diese vom Master nicht entgegengenommen, während der Benutzer andere Menüpunkte bedient, so kann es zu einem Nachrichtenstau beim Master kommen, der möglicherweise die Speichergrenzen sprengt. Aber auch wenn jeder der über 50 verfügbaren Menüpunkte erweitert wird, so daß Nachrichten entgegengenommen werden können, ist das Problem nicht vollständig gelöst: Während der Master aktiv auf eine Eingabe des Benutzers wartet, können keine Nachrichten empfangen werden. Durch die Erweiterung aller Menüpunkte würde das Programm aber kaum noch wartbar sein.

Eine andere Möglichkeit wäre, die Evolution weiterlaufen zu lassen und den Slaves nur mitzuteilen, daß sie das Senden von periodischen Nachrichten unterlassen. Diese Methode hat den Nachteil, daß die Slaves weiterrechnen ohne das die Abbruchbedingungen überprüft werden können. Außerdem kann der Benutzer während einer Unterbrechung Parameter ändern, die auch den Slaves mitgeteilt werden müssen. Solche Änderungen dürfen den Slaves nur mitgeteilt werden, wenn sie sich in einem konsistenten Zustand befinden.

Um diese Nachteile nicht in Kauf nehmen zu müssen, führt ein Unterbrechungswunsch des Benutzers auch zur Unterbrechung der Evolution der Slaves. Die Slaves halten dann in einem konsistenten Zustand und warten auf eine Nachricht vom Master, um mit der Evolution fortzufahren. Diese Nachricht enthält dann auch alle Parameter, die während der Unterbrechung geändert wurden.

Für die verteilte Version des GLEAM-Verfahrens werden zusätzliche Eingabedaten benötigt:

- Die Namen der Rechner, die für die Evolution benutzt werden sollen.
- Die Anzahl der Slave-Prozesse, die auf einem Rechner gestartet werden sollen.
- Die Ausführlichkeit der Nachrichten, die die Slaves periodisch dem Master schicken.
- Die Anzeigehäufigkeit des Fortschritts der Evolution.

Es müssen also Menüpunkte vorgesehen werden, mit Hilfe derer einerseits Parameter eingestellt werden können, andererseits aber auch Informationen über den Status der Slaves oder den Fortschritt der Evolution in den jeweiligen Teilpopulationen abgefragt werden können.

In Abschnitt 6 wurden Konzepte vorgestellt, die eine verteilte Abarbeitung von GLEAM ermöglichen. Der wichtigste Aspekt, durch den sich die verteilte Version von GLEAM von der sequentiellen Version unterscheidet, ist die Kommunikation zwischen den Prozessen. Darauf wird nun im nächsten Abschnitt eingegangen.

7 Entwurf der verteilten Kommunikation

Während der Abarbeitung eines sequentiellen Programms nimmt ein Prozeß in Abhängigkeit von der Programmsteuerung und dem Inhalt seiner Datenstrukturen verschiedene Zustände ein. Handelt es sich um einen Prozeß, der mit anderen Prozessen Nachrichten austauscht, so können Zustandsübergänge auch durch das Eintreffen von Nachrichten ausgelöst werden. In diesem Abschnitt stehen diese Zustandsübergänge im Mittelpunkt. Die hier vorgestellten Zustandsübergangsdiagramme enthalten der Übersichtlichkeit halber auch nicht alle möglichen Zustände. Fehlerzustände werden nur dann betrachtet, wenn die Fehlerbehandlung nicht trivial ist.

Im vorhergehenden Abschnitt wurde immer davon ausgegangen, daß die benötigte Anzahl von Slave-Prozessen vorhanden ist. Im nächsten Unterabschnitt wird nun besprochen, was der Master beim Starten der Slaves berücksichtigen muß.

7.1 Das Starten der Slaves

Die erste Frage, die sich in Zusammenhang mit dem Starten der Slave-Prozesse stellt, ist, ob die Lebensdauer eines Slaves auf die Dauer eines Evolutionslaufs begrenzt werden soll. Wenn man bedenkt, daß das Erzeugen und Terminieren von Prozessen einen erheblichen Aufwand für das Betriebssystem verursacht, kann man diese Frage mit nein beantworten. Es ist also sinnvoller, die Slave-Prozesse einmal zu starten und sie solange für Evolutionsläufe zu benutzen, bis der Master-Prozeß beendet wird oder der Benutzer eine andere Rechner-Konfiguration benutzen möchte (s. Abb. 15). Ein Slave-Prozeß verhält sich also eigentlich wie ein Server, der einen Evolutionsauftrag entgegennimmt, die Evolution durchführt und dann wieder auf einen weiteren Auftrag wartet.

Die zweite Frage, die aufkommt, ist, ob das Erzeugen der Slave-Prozesse automatisch beim Starten des Master-Prozesses oder explizit durch Anforderung des Benutzers geschieht. Da GLEAM/AE auch benutzt wird, um nur Simulationen durchzuführen, würde das automatische Starten der Slave-Prozesse in diesem Fall eine unnötige Belastung des Systems darstellen. Deswegen bleibt der Zeitpunkt des Startens von Slave-Prozessen dem Anwender überlassen.

Für die in Abschnitt 6.5 vorgestellte Fehlerbehandlung ist es notwendig, daß der Master-Prozeß den Absturz eines der durch ihn gestarteten Slave-Prozesse bemerkt. Dies ist in PVM mit Hilfe der Funktion *pvm_notify* leicht zu implementieren: Der PVM-Dämon-Prozeß schickt dem aufrufenden Prozeß eine Nachricht, wenn einer der als Parameter angegebenen PVM-Prozesse terminiert. Diese Nachricht wird mit MSG_TEXIT bezeichnet. Stürzt ein Rechner ab oder wird er aus der PVM-Konfiguration entfernt, so sendet der PVM-Dämon die Nachricht MSG_HOSTD. Auf das Eintreffen dieser Nachrichten reagiert der Master wie auch auf die MSG_FAIL-Nachricht (siehe unten). In den nachfolgenden Abbildungen sind sie deshalb auch nicht eingezeichnet.

Die Slave-Prozesse müssen ihrerseits auch in der Lage sein, den Absturz oder das Terminieren des Master-Prozesses zu bemerken. Sie terminieren, wenn sie die Nachricht

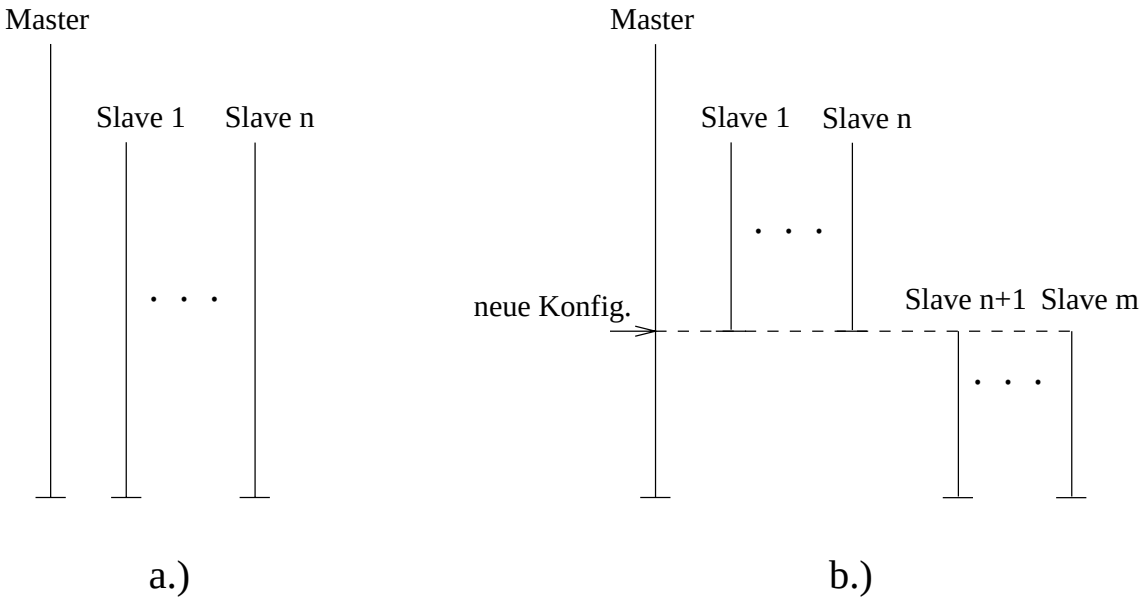

Abbildung 15: Lebensdauer der Slave-Prozesse (a) ohne bzw. (b) mit Änderung der Rechner-Konfiguration durch den Benutzer

MSG_TEXIT erhalten, die die Beendigung des Master-Prozesses dokumentiert.

Beim Starten der Slaves muß der Master auch eine Verwaltungsstruktur anlegen, die für jeden Slave Informationen wie dessen Prozeßidentifikator, Status (Zustand), etc speichert. Diese Daten werden sowohl für die Koordination der Slaves als auch für die Versorgung des Anwenders mit Informationen benötigt.

Eine beim Master eintreffende Nachricht enthält, wie jede andere PVM-Nachricht auch, den Prozeßidentifikator des sendenden Prozesses. Handelt es sich um eine Nachricht von einem Slave, die z. B. die Änderung des Slave-Status erfordert, so müßte der Master in der Verwaltungsstruktur nach dem entsprechenden Prozeßidentifikator suchen, um die Änderung durchführen zu können. Um dem Master diese Suche zu ersparen, erhält jeder Slave eine Nummer. Die Nachrichten der Slaves an den Master enthalten immer diese Nummer, die dann dem Master einen direkten Zugriff auf die Verwaltungsstruktur erlaubt.

Die Master-Slave-Interaktion beim Starten der Slaves ist in Abbildung 16 zusammengefaßt. Die Identifizierungsnummer wird den Slaves kurz nach deren Erzeugung mittels der Nachricht MSG_INIT mitgeteilt. Nachdem der Slave seine Nummer erhalten hat, geht er in den Grundzustand SLAV_IDLE über. In diesem Zustand befindet sich der Slave solange, bis die Durchführung einer Evolution vom Master angestoßen wird oder der Slave terminieren muß.

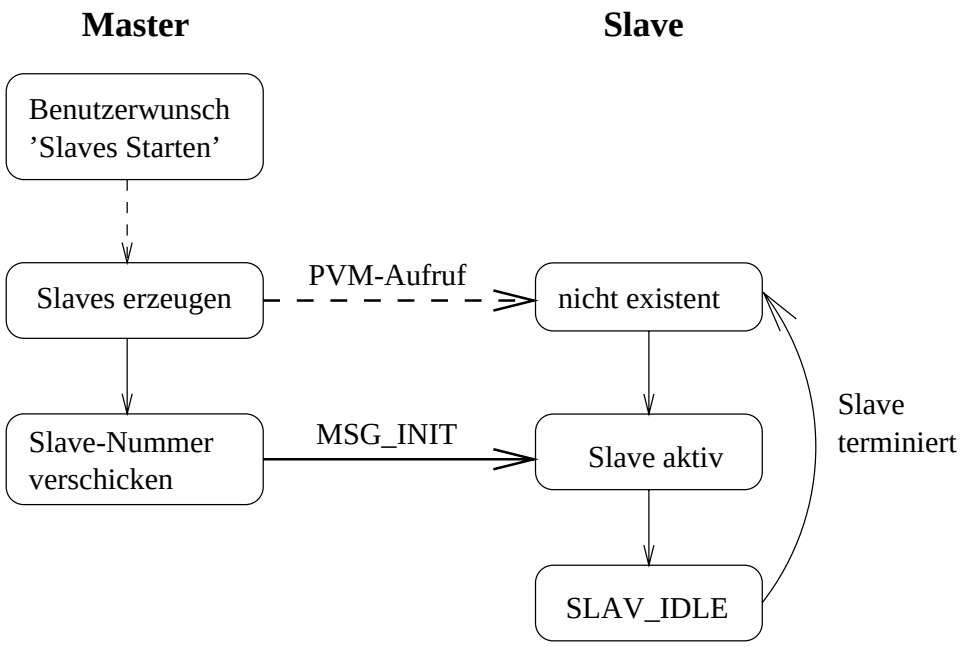

Abbildung 16: Starten eines Slave-Prozesses

7.2 Ablauf einer Evolution

Ein Evolutionslauf wird in GLEAM/AE durch die Spezifikation einer Jobliste definiert. Eine Jobliste besteht aus mehreren Jobs, die hintereinander ausgeführt werden. Ein Job spezifiziert, wie oft eine Evolution mit gleichen Evolutionsparametern gestartet werden soll. Das Definieren von Joblisten erleichtert es dem Benutzer Statistikauswertungen, die zur Bestimmung der Güte von heuristischen Verfahren unbedingt notwendig sind, durchzuführen.

Die Überwachung der Abarbeitung von Joblisten bzw. Jobs ist Aufgabe des Masters. Der Slave beschränkt sich darauf, einen Teil *einer Population eines Jobs* zu bearbeiten.

Um eine Evolution durchführen zu können, benötigen die Slaves Eingabedaten vom Master. Dies sind zum einen Evolutionsparameter, die für alle Slaves gleich sind, zum anderen aber auch spezifische Daten, die sich von Slave zu Slave unterscheiden. Evolutionsparameter sind die Nachbarschaftsgröße, die Initialisierungsstrategie, das Selektionskriterium, die maximal zu erreichende Lösungsgüte, etc. Spezifische Daten sind die Subpopulationsgröße, die Anzahl externer Individuen und alle individuenspezifischen Daten. Die individuenspezifischen Daten umfassen die Nachbarschaft eines Individuums und die Information über die Slaves, auf denen das Individuum ein externer Nachbar ist.

Der erste Schritt, den der Master ausführen muß, um eine Evolution für eine Population zu starten, ist die Berechnung aller Daten, die von den Slaves benötigt werden. Im wesentlichen sind das die Größe der Subpopulationen, die Nachbarschaftsmatrix und die Datenstruktur, die festlegt auf welchen Slaves ein Individuum ein externer Nachbar ist.

Die Nachbarschaftsmatrix umfaßt die Nachbarn eines jeden Individuums. Dadurch, daß eine Nachbarschaftsmatrix benutzt wird, können beliebige geographische Strukturen definiert werden: Auch Strukturen, die nicht durch einfache mathematische Ausdrücke (wie z. B. Ring und Torus) beschrieben werden können, können durch Einlesen der Matrix definiert werden. Die Kommunikation zwischen Master und Slave sieht für jede geographische Struktur gleich aus: Einem Slave wird der Teil der Matrix übermittelt, die die Nachbarschaft der Individuen seiner Teilpopulation enthält.

Ein Slave verläßt den Grundzustand erst nachdem er alle notwendigen Daten erhalten hat. Die Übermittlung der Daten geschieht durch das Senden der beiden Nachrichten MSG_INIT_PARS und MSG_SPEC_INIT: MSG_INIT_PARS enthält hierbei die Evolutionsparameter, während MSG_SPEC_INIT die slave-spezifischen Daten umfaßt. Für das Verschicken der Nachricht MSG_INIT_PARS benutzt der Master den Broadcast-Mechanismus von PVM (s. Abschnitt 5).

Nachdem der Slave alle Daten erhalten und überprüft hat, kann er mit der Initialisierung der Population beginnen. Bei den Initialisierungsstrategien NEU und GEN benötigt er hierfür auch keine Daten vom Master. Bei den sonstigen Initialisierungsstrategien muß der Slave warten, bis er die Nachricht MSG_KETTEN_INIT erhält. Weil der Master den Zustand 'Initialisierungsketten senden' nicht immer einnimmt, ist er in Abbildung 17 punktiert dargestellt.

Das Verschicken von Ketten ist vor allem dann schwierig, wenn eine Kette Verweise auf andere Unterketten enthält. Da Master und Slave getrennte Adreßräume besitzen, müssen auch alle referenzierten Ketten zum Slave gesendet werden. Dies gilt natürlich auch für die Kommunikation zwischen den Slaves.

Verläuft die Initialisierung der Population eines Slaves fehlerfrei, so teilt er dies dem Master mittels der Nachricht MSG_INIT_OK mit. Der Master wartet, bis er von allen Slaves eine Nachricht erhalten hat. Melden alle Slaves eine fehlerfreie Initialisierung, so ist für den Master die Initialisierungsphase beendet.

Um den Evolutionsprozeß durchführen zu können, benötigen die Slaves außer einer fehlerfreien Initialisierung ihrer Teilpopulation auch ihre externen Nachbarn. Als nächstes findet also der Austausch externer Nachbarn zwischen den Slaves statt. Jeder Slave verschickt zuerst alle Individuen, die externe Nachbarn auf anderen Slaves sind, mittels der Nachricht MSG_N_UPDATE. Anschließend wartet er darauf, daß die Nachrichten mit seinen externen Nachbarn eintreffen. Tritt ein Fehler bei der Übertragung auf, so fordert er das Individuum erneut an. Dazu verschickt er an den entsprechenden Slave die Nachricht MSG_REQ_UPDATE (s. Abb. 18). Trifft ein Individuum in einer bestimmten Zeitspanne nicht ein, so geht der Slave in den Zustand N_FAIL über. Hat er aber alle externen Nachbarn erhalten, so kann er mit der Evolution beginnen.

Zum Zeitpunkt des Austauschs von externen Nachbarn ist noch nicht sichergestellt, daß die Initialisierungsphase von allen Slaves erfolgreich beendet wird. Schlägt die Initialisierung fehl, so wurden Daten unnötig verschickt, da die Initialisierung vom Master neu gestartet wird (siehe Abb. 19). Die Möglichkeit, den Austausch externer Nachbarn erst nach dem Erhalten der Nachricht MSG_EVO_BEG (s. Abb. 17) durchzuführen, wurde nicht implementiert: Nach dem Synchronisationsschritt würden alle Slaves gleichzeitig

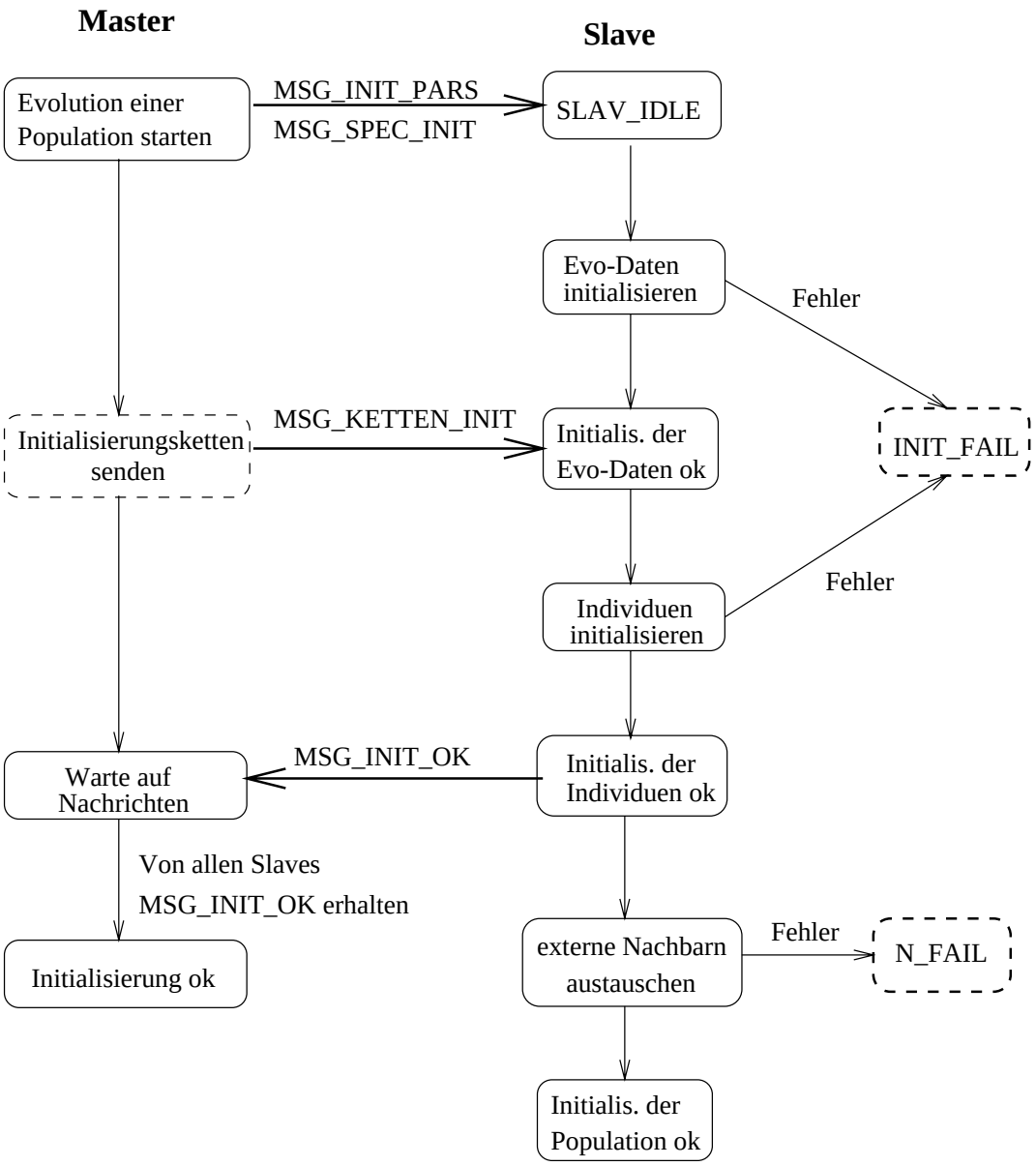

Abbildung 17: Fehlerfreie Initialisierungsphase

senden, was zu einer hohen Netzlast und somit zu Fehlern führen kann. Diese Spitzenlast umgeht man, wenn die Slaves gleich nach der Initialisierung Nachbarn austauschen, da es wenig wahrscheinlich ist, daß alle Slaves gleich schnell ihre Individuen initialisieren.

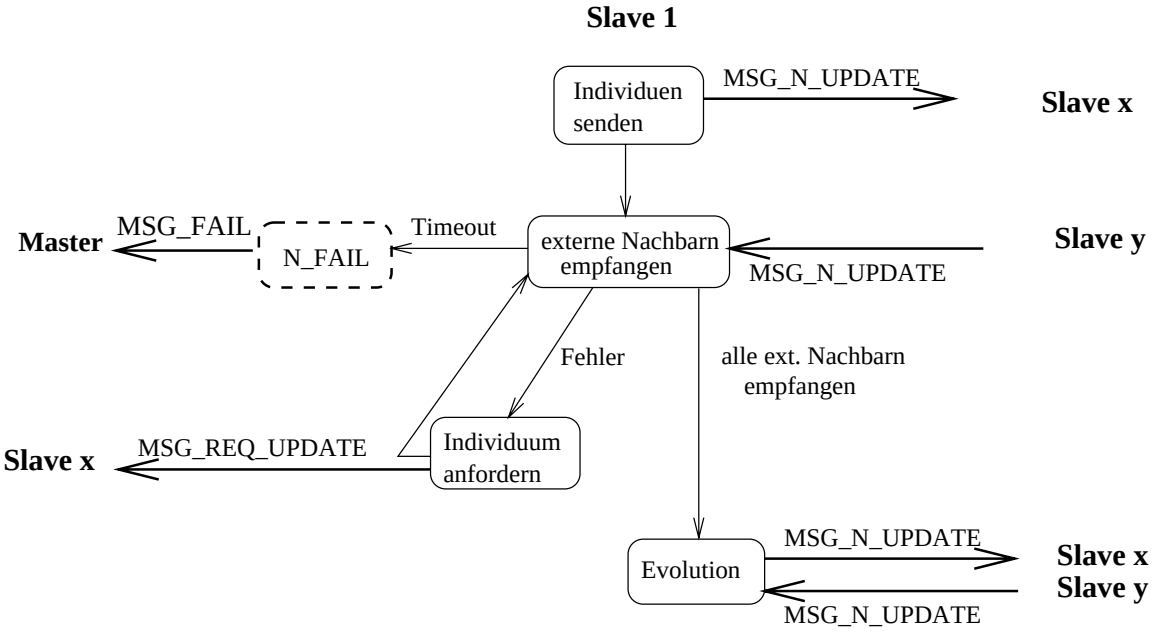

Abbildung 18: Nachrichtenaustausch zwischen Slaves

Auch während der Evolution findet Nachrichtenaustausch zwischen den Slaves statt: Jedes mal, wenn durch den Evolutionsprozeß ein Individuum durch ein neu generiertes ersetzt wird, wird es mittels der Nachricht MSG_N_UPDATE an alle Slaves verschickt, die es als externen Nachbarn speichern.

Die Interaktion zwischen Master und Slave bei einer fehlerfreien Initialisierung ist in Abbildung 17 zusammengefaßt. Leider kann es auch vorkommen, daß während der Initialisierung Fehler auftreten. Tritt der Fehler beim Master auf, so wird die Evolution abgebrochen (Zustand STOP), da dann auch die Slaves sicherlich keine korrekte Initialisierung durchführen können. Die Evolution wird auch abgebrochen, wenn keiner der Slaves die Nachricht MSG_INIT_OK an den Master schickt. In diesem Fall muß der Anwender anhand der Fehlermeldungen und der Stati der Slaves überprüfen, weshalb eine Evolution nicht durchführbar ist. Die Nachricht MSG_FAIL enthält neben der Nummer des betroffenen Slaves immer auch Informationen über die Ursache des Fehlers.

Interessant für die Fehlerbehandlung wird es allerdings erst, wenn nur einige der Slaves terminieren oder einen Fehler über die Nachricht MSG_FAIL melden. Der Master versucht den Fehler zu beheben, indem er den Evolutionsprozeß nur noch auf den Slaves startet, deren Initialisierung problemlos verlaufen ist. Dazu muß er diese Slaves veranlassen, in ihren Grundzustand zurückzukehren: Der Master schickt ihnen deswegen die Nachricht MSG_SLAV_RESET. Auch die Slaves, die gerade externe Nachbarn austauschen oder sich im Zustand N_FAIL befinden, werden veranlaßt in den Zustand SLAV_IDLE zurückzukehren. Danach wird die Initialisierung erneut gestartet. Treten auch beim zweiten Versuch

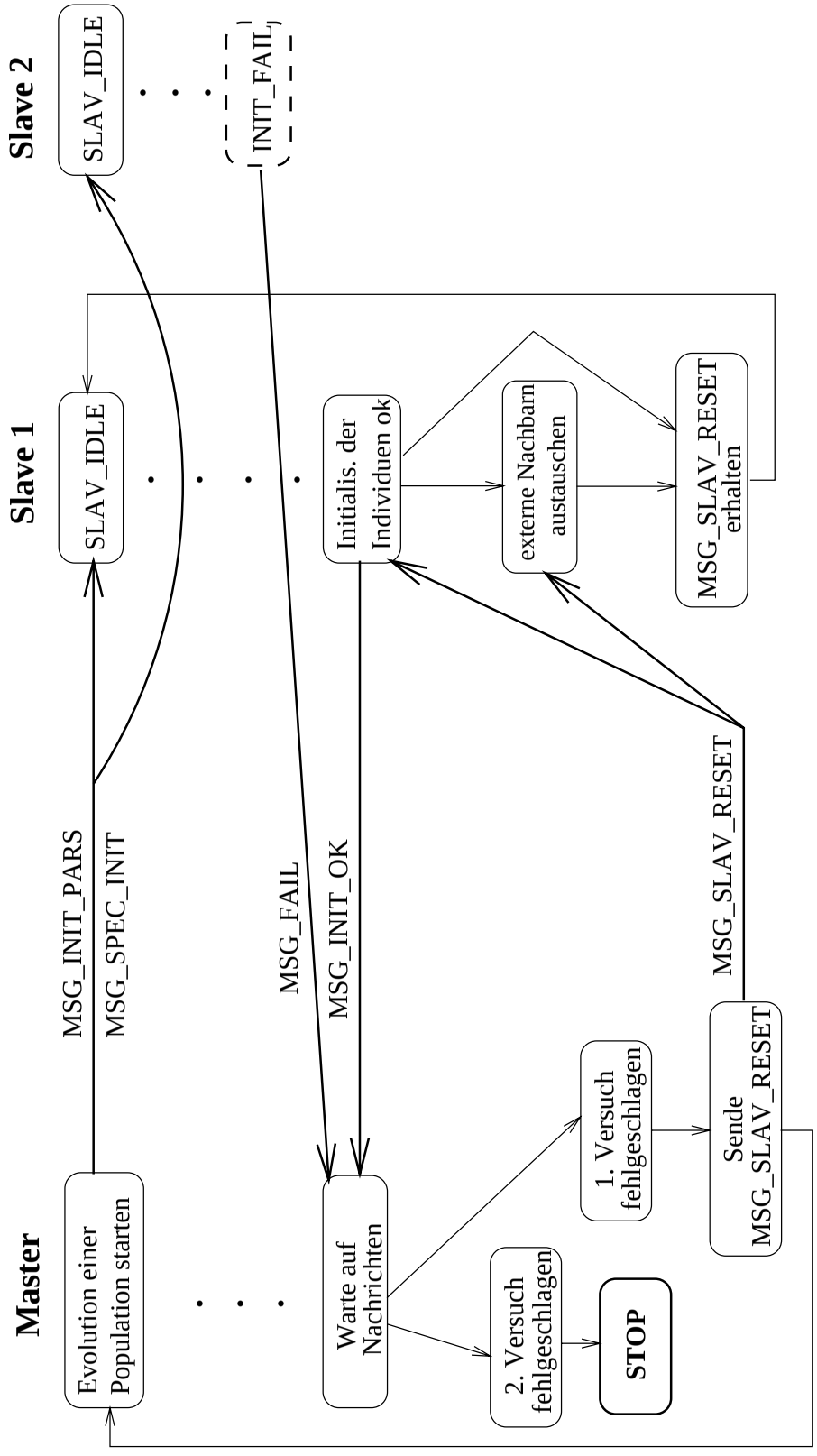

Abbildung 19: Initialisierungsphase mit Fehlern

Fehler auf, so wird der Evolutionsprozeß abgebrochen. Die Slaves, die sich im Zustand INIT_FAIL befinden, verlassen diesen nur, wenn der Master-Prozeß terminiert oder ein neuer Evolutionsprozeß vom Benutzer gestartet wird.

Wenn die Initialisierungsphase erfolgreich beendet wurde, startet der Master den Evolutionsprozeß, indem er den Slaves die Nachricht MSG_EVO_BEG sendet (s. Abb. 20). Während die tatsächliche Arbeit nun von den Slaves erledigt wird, muß der Master nur auf Nachrichten der Slaves warten und die Abbruchbedingungen überwachen.

Damit der Master die Abbruchbedingungen prüfen kann und damit auch der Benutzer mit Informationen über den Fortschritt der Evolution versorgt werden kann, schickt jeder Slave periodisch Daten (Nachricht MSG_GEN_ERG) zum Master. Der Benutzer kann über die Benutzeroberfläche einstellen, wie umfangreich diese Daten sein sollen. Es wurden drei Arten von Ergebnisnachrichten definiert:

- MSG_MIN_ERG enthält nur die Daten, die der Master für die Berechnung der Abbruchbedingungen benötigt.
- MSG_NORMAL_ERG beinhaltet im Gegensatz zu MSG_MIN_ERG nicht nur die Güte des besten Individuums, sondern die Güte aller Individuen einer Teilpopulation.
- MSG_MAX_ERG umfaßt zusätzlich zu den Daten aus MSG_NORMAL_ERG auch noch Statistikdaten, wie z. B. die Anzahl der Aufrufe einer bestimmten genetischen Operation.

Der Master berechnet jedesmal nachdem er eine Ergebnisnachricht erhalten hat, die Abbruchbedingungen. Sind diese nicht erfüllt, so läßt er die Slaves weiterrechnen und wartet weiter auf Nachrichten. Gilt jedoch eine der Abbruchbedingungen, so muß er die Slaves veranlassen den Evolutionsprozeß zu beenden. Dazu sendet er die Nachricht MSG_STOP. Die Slaves beenden nach dem Erhalten dieser Nachricht die Evolution und senden das Endergebnis an den Master. Die Nachricht MSG_END_ERG enthält die Daten, die auch die Nachricht MSG_MAX_ERG umfaßt. Nachdem von allen Slaves die Nachricht MSG_END_ERG oder MSG_FAIL eingetroffen ist, gilt auch für den Master die Evolution als beendet.

Das einzige Abbruchkriterium, das auch von den Slaves überprüft wird, ist das Erreichen der vorgegebenen Zielnote in seiner Teilpopulation. Erreicht eines der Individuum die Lösungsgüte, so kann er die Evolution schon vor dem Erhalt der Nachricht MSG_STOP beenden. Die Interaktion zwischen Master und Slave während des Ablaufs einer Evolution ist in Abbildung 20 dargestellt. Wie auch daraus ersichtlich ist, führt das Empfangen der Nachricht MSG_FAIL, die von den Slaves im Zustand EVO_FAIL oder N_FAIL gesendet wird, beim Master zu keinem Fehler. Erst wenn alle Slaves die Nachricht MSG_FAIL gesendet haben, d. h. wenn in keinem der Slaves eine Evolution stattfindet, nimmt der Master den Zustand STOP ein (in Abb. 20 nicht eingezeichnet).

Bevor der Evolutionsprozeß für die nächste Population gestartet werden kann, muß der Master die Slaves veranlassen, den Zustand SLAV_IDLE einzunehmen. Deswegen schickt

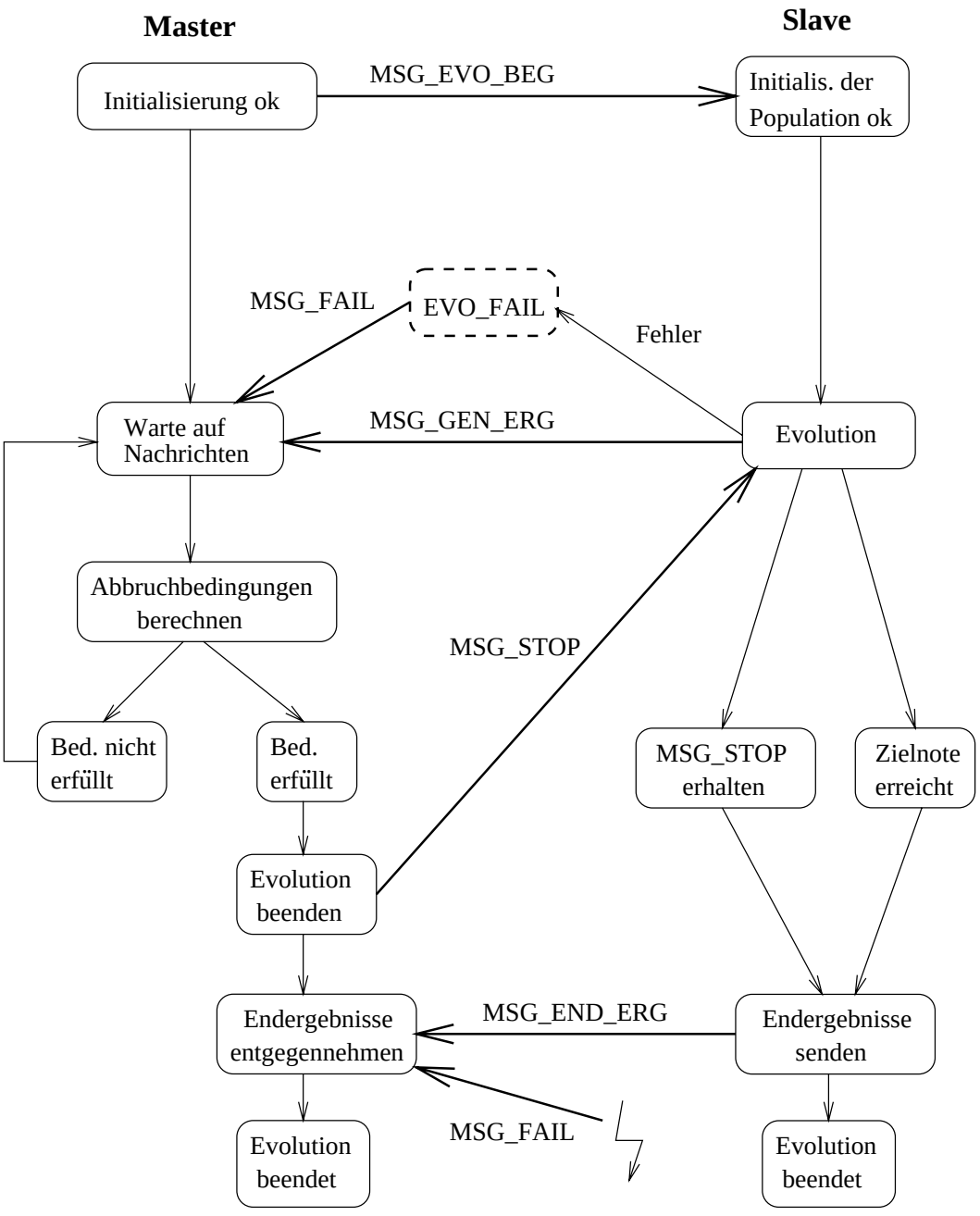

Abbildung 20: Interaktion zwischen Master und Slave während des Evolutionsprozesses

er allen Slave-Prozessen, die nicht terminiert haben, die Nachricht MSG_SLAV_RESET. Nach dem Erhalten dieser Nachricht befinden sich alle Slaves, also auch die, die vorher im Zustand INIT_FAIL, N_FAIL oder EVO_FAIL waren, wieder im Zustand SLAV_IDLE.

In den Abbildungen 17 – 20 wurde der Ablauf einer Evolution dargestellt, der nicht vom Benutzer unterbrochen wird. Im nächsten Unterabschnitt wird nun die Interaktion zwischen Master und Slave nach einer Unterbrechungsanforderung des Benutzers beschrieben.

7.3 Unterbrechung der Evolution

Der Benutzer kann die Unterbrechung einer Evolution durch das Drücken einer Taste jederzeit anfordern. Die Unterbrechung wird dann zum nächstmöglichen Zeitpunkt durchgeführt. In der sequentiellen Version von GLEAM/AE sind diese Zeitpunkte das Beenden der Initialisierung bzw. die Anwendung eines genetischen Operators auf ein Individuum während der Evolution.

In der verteilten Version von GLEAM/AE werden Unterbrechungswünsche des Benutzers immer dann durchgeführt, wenn der Master auf das Eintreffen von Nachrichten wartet (s. Abb. 17 und 20). Liegt eine Unterbrechungsanforderung des Benutzers vor, so sendet er den Slaves die Nachricht MSG_INTERRUPT, damit diese in einem definierten Zustand halten und somit keine periodischen Nachrichten mehr an den Master schicken. Eine Rückmeldung der Slaves ist nicht notwendig, da ja nur sichergestellt werden muß, daß es beim Master nicht zu einem Nachrichtenstau kommt.

Die Slaves können die Nachricht MSG_INTERRUPT entgegennehmen nachdem sie die Initialisierung der Population beendet haben oder, wie im sequentiellen Fall auch, nach jeder Anwendung eines genetischen Operators auf ein Individuum während der Evolution. Sie befinden sich solange im Zustand EVO_INTERRUPT, bis sie die Nachricht MSG_CONTINUE vom Master erhalten. Der Master seinerseits schickt diese Nachricht, wenn der Benutzer die Evolution fortsetzen möchte. Die Nachricht MSG_CONTINUE enthält alle Parameter, die der Benutzer während der Unterbrechung geändert hat.

Auch während die Evolution unterbrochen ist, kann Nachrichtenaustausch zwischen Master und Slaves stattfinden. Dies ist der Fall, wenn der Benutzer Menüpunkte aufruft, die die Kenntnis der Kettenstruktur von Individuen erfordern. In GLEAM/AE sind das zur Zeit der Ketteneditor, mit dem man den Aufbau eines Individuums betrachten kann, und das Speichern von Individuen. Das Speichern von Individuen einer Teilpopulation veranlaßt der Master durch das Schicken der Nachricht MSG_SAVE_REQ an den entsprechenden Slave. Danach wartet er auf die Rückmeldung vom Slave. Dieser teilt mittels der Nachricht MSG_SAVE_RESP dem Master mit, ob das Speichern erfolgreich durchgeführt werden konnte (s. Abb. 22).

Wenn sich der Benutzer ein bestimmtes Individuum anschauen möchte, so muß der Master dieses Individuum zuerst von dem entsprechenden Slave anfordern. Dies geschieht durch das Senden der Nachricht MSG_REQ_KETTE. Der Slave antwortet mit der Nachricht MSG_KETTE, wenn das Individuum erfolgreich verschickt werden konnte, und mit MSG_KETTE_FAIL, wenn ein Fehler bei der Bearbeitung der Anfrage aufgetreten ist.

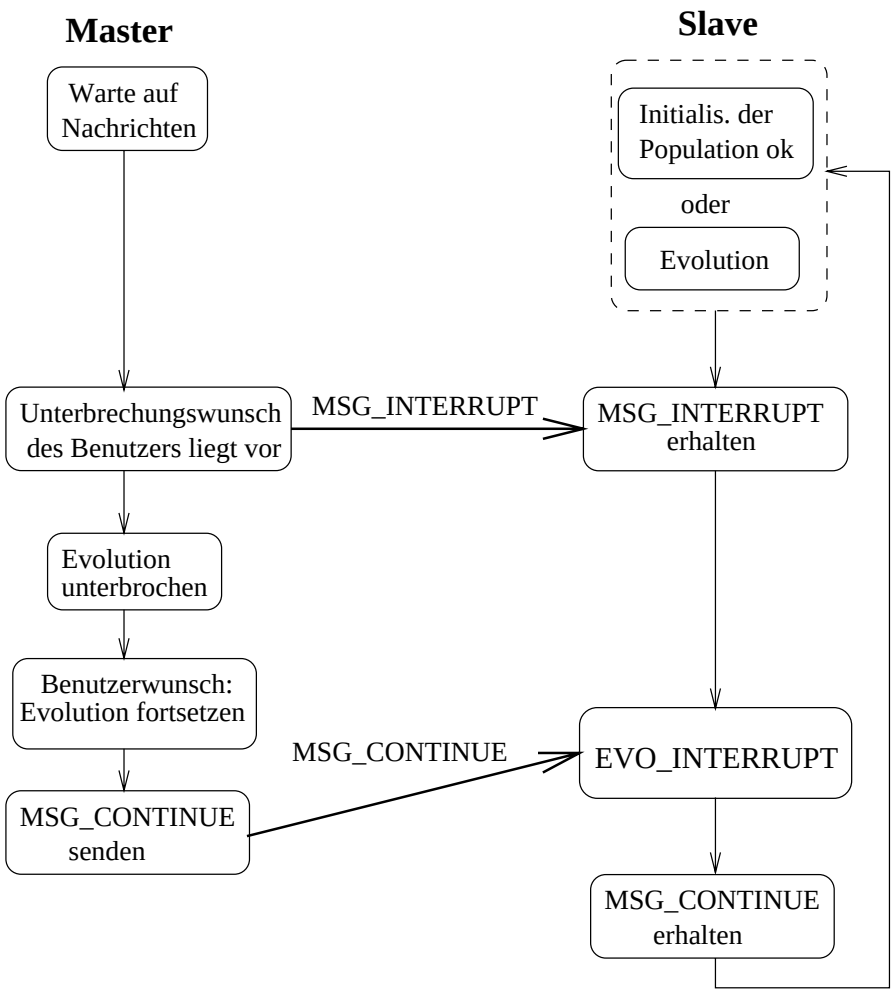

Abbildung 21: Interaktion zwischen Master und Slave zwecks Unterbrechungssteuerung

Dieser benutzergesteuerte Nachrichtenaustausch, der hier für die Unterbrechungsphase einer Evolution beschrieben wurde, kann natürlich auch stattfinden, nachdem die Evolution erfolgreich beendet wurde.

Bevor nun die Ergebnisse der durchgeführten Tests vorgestellt werden, geht der nächste Abschnitt auf die Implemtierung der verteilten Version von GLEAM ein.

8 Implementierungstechnische Aspekte

Die sequentielle Version von GLEAM/AE ist ein in der Programmiersprache C geschriebenes Programmpaket, dessen Entwicklung noch nicht abgeschlossen ist. Deswegen kommen Änderungen des Programmcodes auch relativ häufig vor. Damit die Wartbarkeit der verteilten Version bei solchen Änderungen keinen großen Aufwand verursacht, sollte möglichst viel Code der sequentiellen Version benutzt werden.

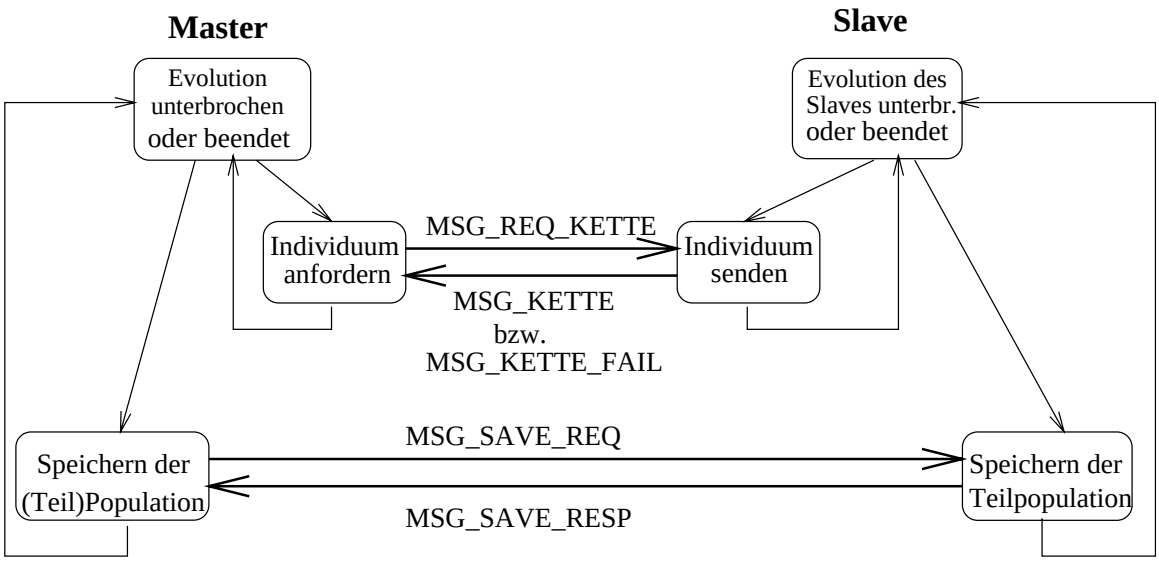

Abbildung 22: Möglicher Nachrichtenaustausch zwischen Master und Slave nach dem Beenden oder Unterbrechen der Evolution

Abbildung 23 zeigt die Verzeichnisstruktur des gesamten Softwaresystems. In den Unterverzeichnissen `c_vers` und `tk_vers` befinden sich die Hauptprogramme für die zeichenorientierte bzw. fensterorientierte Benutzeroberfläche des Systems. Die Unterverzeichnisse von `packages` enthalten die Implementierung verschiedener Dienste. So zum Beispiel stellt `sys` alle systemnahen Dienste zur Verfügung, während `simu` die Anbindung an einen applikationsspezifischen internen oder externen Simulator realisiert.

Für die verteilte Version von GLEAM/AE konnten die Dateien der meisten Unterverzeichnisse unverändert übernommen werden. Lediglich die Fehlerbehandlung (`fbhm`), die Ketten-Ein/Ausgabe (`chio`) und die Benutzeroberfläche mußten erweitert werden. Die größten Änderungen waren in `evo` (Evolution) notwendig, da die Evolution im verteilten Fall nicht mehr von einem einzigen Prozeß erledigt wird. Die Unterverzeichnisse `sl_verw` und `slav` wurden hinzugefügt. Während `slav` das Hauptprogramm der Slaves enthält, stellt `sl_verw` ein Modul zur Verwaltung von Slave-Prozessen zur Verfügung.

Die Ketten-Ein/Ausgabe (`chio`) wurde um das Verschicken von Ketten mittels PVM-Nachrichten erweitert: Da PVM nur Routinen zur Übertragung elementarer Datentypen (`int`, `float`, etc) bereitstellt, muß jede komplexe Datenstruktur, also auch eine Kette, vor dem Senden linearisiert und nach dem Empfang wieder "zusammengestellt" werden.

Im Unterverzeichnis `evo` befindet sich im wesentlichen die Implementierung der in den Abschnitten 6 und 7 vorgestellten Konzepte.

Das Modul zur Verwaltung von Slave-Prozessen enthält Routinen, mit Hilfe derer die Datenstrukturen, die die slave-spezifischen Daten wie Prozeßidentifikator, Status, etc enthalten, manipuliert werden können.

Die Erweiterung von `fbhm` und die der Benutzeroberfläche werden in den beiden nächsten

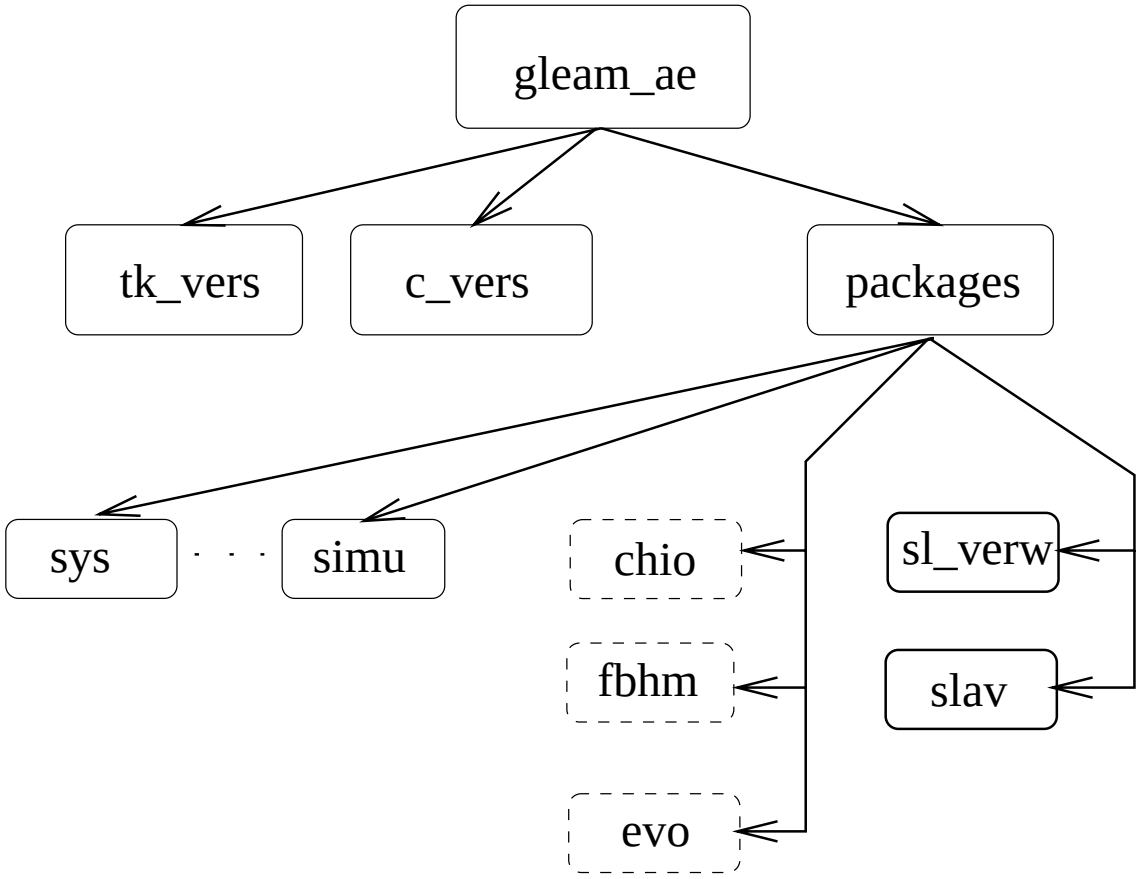

Abbildung 23: Verzeichnisstruktur des Systems

Unterabschnitten beschrieben.

8.1 Anzeigen von Fehlermeldungen

Im Verzeichnis `fbhm` ist die Implementierung der Routinen zur Fehlermeldungsbehandlung untergebracht. Diese Routinen stellen Basisdienste dar, die von den restlichen Funktionen des Programmpakets genutzt werden können. Es existieren drei Klassen von Meldungen mit den entsprechenden Routinen: `fatal` (für schwerwiegende Fehler), `fehler` und `meldung`. Die Parameter eines `fatal`/`fehler`/`meldung`-Aufrufs werden in einem lokal verwalteten Puffer gespeichert, der über eine Ausgaberoutine geleert und als formatierte Zeichenketten dem Anwender angezeigt werden.

Der Master-Prozeß kann die oben genannten Routinen unverändert benutzen. Probleme gibt es jedoch bei den Slave-Prozessen, denn sie laufen im Hintergrund und verfügen über kein Ausgabefenster. Ihre Ausgaben werden zwar von PVM in eine Datei umgeleitet, die aber leider nicht besonders gut lesbar ist, da sie auch sonstige PVM-spezifische Meldungen enthält.

Deswegen wurde die Implementierung der Routinen `fatal`, `fehler` und `meldung` für die Slaves verändert: Der Aufruf einer der Routinen führt nicht mehr zum Ablegen der Parameter in einen Puffer, sondern zum Senden einer Nachricht MSG_FATAL, MSG_FEHLER, bzw. MSG_MLDG an den Master. Diese Nachricht enthält alle Parameter des Aufrufs und zusätzlich die Nummer des Slaves, bei dem die Fehlermeldung aufgetreten ist.

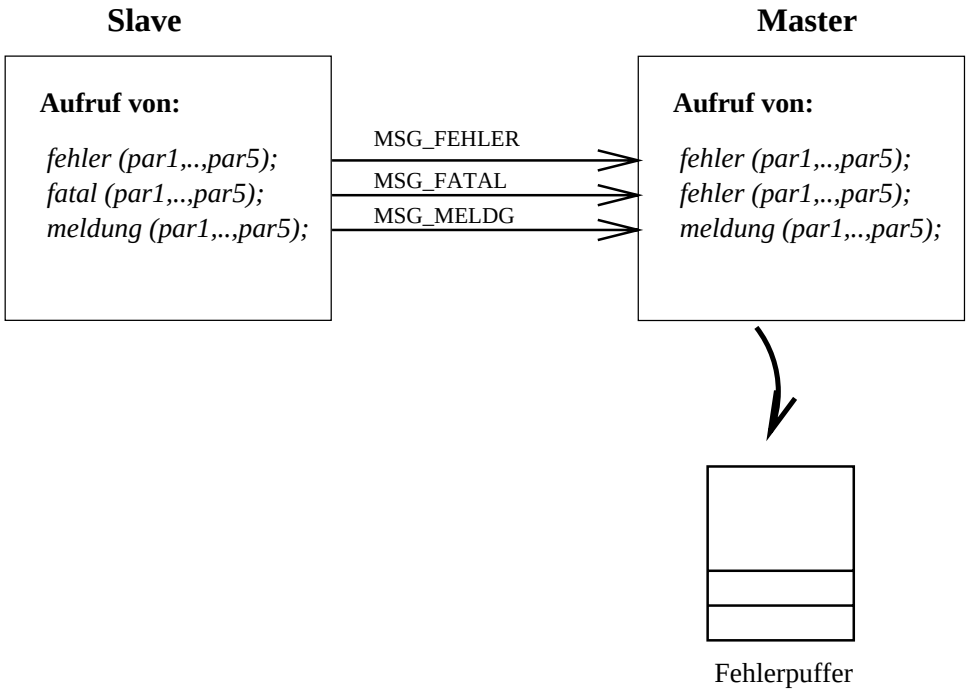

Abbildung 24: Behandlung von Fehlermeldungen

Der Master ruft dann beim Erhalt einer Fehlernachricht die entsprechende Fehlermeldungsroutine mit den in der Nachricht enthaltenen Parameter auf. Eine Ausnahme bildet die Nachricht MSG_FATAL: Da eine fatale Nachricht des Slaves für den Master kein schwerwiegender Fehler ist, wird statt `fatal` die Routine `fehler` aufgerufen (s. Abb. 24). Dadurch wird eine Fehlermeldung des Slaves genau wie eine Fehlermeldung des Masters behandelt und angezeigt. Um feststellen zu können, woher eine Fehlermeldung kommt, enthält jede Fehlermeldung eines Slaves auch dessen Nummer.

8.2 Erweiterung der Benutzeroberfläche

Wie in Abschnitt 6.6 bereits erwähnt, werden für die verteilte Version des GLEAM-Verfahrens zusätzliche Eingabedaten benötigt. Auch muß dem Benutzer die Möglichkeit gegeben werden, Informationen über einzelne Slaves abzufragen.

Die Benutzeroberfläche von GLEAM/AE wurde für die verteilte Version um einen Menüpunkt erweitert, der folgende Untermenüs umfaßt:

- *Manuelle Eingabe der Rechnerkonfiguration:* Hier kann der Benutzer die Namen der

Rechner, die für die Evolution benutzt werden, und die Anzahl der Slave-Prozesse, die pro Rechner gestartet werden sollen, manuell eingeben.

- *Einlesen der Rechnerkonfiguration aus einer Datei:* Alternativ zum obigen Menüpunkt kann die Rechnerkonfiguration auch aus einer Datei eingelesen werden, deren Name der Benutzer angeben muß.

- *Starten der Slave-Prozesse:* Erst wenn der Anwender diesen Menüpunkt aufruft, werden die Slave-Prozesse gestartet. Wurden bereits vorher Slaves gestartet, d. h. daß der Benutzer nun die Rechnerkonfiguration ändern möchte, so werden vor dem Starten der neuen Prozesse die alten beendet.

- *Zusatzparameter einstellen:* Hier kann der Benutzer festlegen, wie ausführlich die periodischen Nachrichten sein sollen. Er kann sich für eine der folgenden Nachrichten entscheiden: MSG_MIN_ERG, MSG_NORMAL_ERG, MSG_MAX_ERG (s. Abschnitt 7.2). Außerdem kann hier die Anzeigehäufigkeit des Evolutionsfortschritts angegeben werden.

- *Zeige den Status der Slaves:* Während die bisher vorgestellen Menüpunkte dazu dienen, das Programm mit Eingabeparameter zu versorgen, sind die nächsten dazu gedacht, den Benutzer mit Informationen zu versorgen. Mit Hilfe dieses Menüs kann sich der Benutzer den Status der Slaves anzeigen lassen. Der Status eines Slaves wechselt im fehlerfreien Fall während eines Evolutionslaufs von SLAV_IDLE über "Initialisierung ok" zu "Evolution läuft" und schließlich "Evolution beendet". Im Fehlerfall liefert der Status Hinweise auf die Ursache des Fehlers: "Prozeß beendet", "Rechner abgestürzt", "Fehler beim Starten des Simulators", "Fehler bei der Initialisierung der Nachbarn", etc.

- *Retten einer (Teil)Population:* Hier kann der Benutzer die gesamte Population oder nur bestimmte Teilpopulationen in Dateien retten. Diese Dateien sind als Aufsetzpunkt für einen späteren Evolutionslauf gedacht (siehe Initialisierungsstrategie FROM_FILE, Abschnitt 6.2).

- *Zeige Evolutionsdaten:* Während die Evolution läuft, wird der Benutzer zwar über den Fortschritt der Evolution informiert, wobei aber nicht der Fortschritt jeder einzelnen Teilpopulation angezeigt wird. Dieser Menüpunkt zeigt für jede Teilpopulation die wichtigsten Daten, wie Note des besten Individuums, Anzahl der berechneten Generationen, etc, an.

Nachdem nun in den letzten Abschnitten die Konzepte der verteilten Version des GLEAM-Verfahrens und auch einige implementierungstechnische Aspekte beschrieben wurden, stellt der nächste Abschnitt die erzielten Testergebnisse vor.

9 Testergebnisse

GLEAM/AE wird am Institut für Angewandte Informatik des Forschungszentrums Karlsruhe zur Zeit vor allem zur Optimierung des Designs einer Mikropumpe eingesetzt (s. Ab-

schnitt 4.5.2). Durch die Benutzung der verteilten Version von GLEAM/AE erhofft man sich eine Verkürzung der langen Laufzeit eines Optimierungsprozesses. Als Testumgebung kommt diese Anwendung jedoch nicht in Frage: Zum einen würden Testläufe viel zu lange dauern, zum anderen verfügt das Institut nur über zwei Lizenzen für das Simulationsprogramm.

Um Tests durchführen zu können, wurde GLEAM/AE um eine Anwendung zur Lösung des Problems des Handelsreisenden (Traveling Salesman Problem) erweitert.

9.1 Testumgebung

Das Problem des Handelsreisenden (TSP) besteht darin, den kürzesten Weg zwischen N vorgegebenen Städten zu finden, unter den Randbedingungen, daß keine Stadt zweimal besucht wird und die Start- und Zielstadt identisch sind. Für jedes Städtepaar (c_i, c_j) ist hierbei die Entfernung $d(c_i, c_j)$ gegeben. Gilt für alle Entfernungen $d(c_i, c_j) = d(c_j, c_i)$, so spricht man vom symmetrischen TSP (STSP), ansonsten vom asymmetrischen TSP (ATSP). Es wurde bewiesen, daß das Traveling Salesman Problem zu der Klasse der *NP*-vollständigen Probleme[CHP82] gehört.

Das TSP ist deswegen als Testumgebung geeignet, weil der Suchraum groß ist und viele suboptimale Lösungen existieren. Für das symmetrische TSP gibt es $(N-1)!/2$ mögliche Wege. Bereits für $N = 100$ müßten deterministische Verfahren die Länge von $4,661 \cdot 10^{155}$ Wegen berechnen!

Der von Dr. M. Gorges-Schleuter entwickelte genetische Algorithmus zur Lösung des Problems des Handelsreisenden[GS91, GS97, GS] zeichnet sich dadurch aus, daß er über eine spezielle Wegdarstellung und darauf zugeschnittene genetische Operatoren verfügt. Spezielle genetische Operatoren werden benötigt, damit sichergestellt wird, daß aus gültigen Rundwegen durch Mutation oder Rekombination wieder ein gültiger Rundweg entsteht. Die Simulation zur Bestimmung der Güte eines Individuums besteht hier aus der Berechnung der Weglänge.

Damit das GLEAM-Verfahren nicht um genetische Operatoren erweitert werden muß, die nur für die Lösung des Problems des Handelsreisenden spezifisch sind, wurde die Anbindung dieser Anwendung an GLEAM/AE folgendermaßen realisiert: Es wurden Konvertierungsroutinen geschrieben, die die Kettenrepräsentation auf die TSP-spezifische Repräsentation abbilden und umgekehrt. Jedesmal wenn eine TSP-spezifische Operation ausgeführt wird, müssen also Konvertierungsroutinen aufgerufen werden. Dies führt dazu, daß die Laufzeit von GLEAM höher ist als die auf das TSP-Problem zugeschnittene Anwendung.

Für die nachfolgenden Tests wurde das SUN-Netz der Abteilung Mikrosysteminformatik des Instituts für Angewandte Informatik des Forschungszentrums Karlsruhe benutzt. Dieses besteht aus Rechnern des Typs UltraSparc, SparcStation10 und SparcStation20. Das Rechnernetz besitzt eine maximale Übertragungsrate von 10 Mbit/s.

Die sequentielle Version von GLEAM wurde immer auf einer SUN UltraSparc ausgeführt. Für die verteilte Version von GLEAM wurden für Testläufe mit 2, 4 und 6 Slave-Prozessen

SUN UltraSparc's benutzt. Bei Testläufen mit mehreren Slaves wurden auch die restlichen SUN-Rechner hinzugenommen. PVM wurde so konfiguriert, daß es Daten im XDR-Format überträgt. Die von PVM durchgeführten Konvertierungen sind notwendig, weil die SUN SparcStation's 32-Bit-Prozessoren besitzen während die SUN UltraSparc's 64-Bit-Prozessoren haben.

Die Einstellungen der Evolutionsparameter, die während der Tests nicht variiert wurden, sind: Initialisierungsstrategie GEN (Vorgenerierung von Individuen), Nachbarschaftsgröße 8, Überlebensstrategie *better-parent* (s. Abschnitt 3.3). Die Partnerwahl erfolgt mittels linearer rangbasierter Selektion mit $max = 1,7$. Somit besitzt auch das lokal schlechteste Individuum eine Chance von $2 - 1,7 = 0,3$, als Partner gewählt zu werden. Die Nachricht MSG_GEN_ERG umfaßt nur die unbedingt notwendigen Daten (MSG_MIN_ERG).

9.2 Visualisierung eines Evolutionslaufs

Für PVM ist ein Visualisierungsprogramm verfügbar, XPVM, das dem Benutzer erlaubt die Aktivitäten seines PVM-Programms graphisch zu verfolgen. Während eine Anwendung läuft, sammelt XPVM Informationen über die Benutzung von Sende- und Empfangroutinen. Dem Anwender werden mehrere Sichten angeboten. In Abbildung 25 sind zwei der Sichten dargestellt. Die *Utilization View* gibt anteilig an, womit die gestarteten Prozesse zu einem bestimmten Zeitpunkt beschäftigt sind: Durchführung von Berechnungen (computing), blockierendes Warten auf Nachrichten (waiting) oder Ausführen von PVM-Operationen (overhead). Mit der *Space-Time View* kann man verfolgen, welche Nachrichten zu welchem Zeitpunkt zwischen den Prozessen verschickt werden. Man erhält auf Wunsch auch Informationen über die Größe der Nachrichten.

In Abbildung 25 ist der Ablauf einer Evolution zur Lösung eines Problems mit 198 Städten für eine Population mit 64 Individuen während einer Generation dargestellt. Es sind 3 Prozesse aktiv: der Master-Prozeß und zwei Slave-Prozesse. Die Kommunikation zwischen den Prozessen ist anhand der *Space-Time View* schön zu beobachten. Gleich nach dem Starten der Slaves, schickt der Master ihnen eine Nachricht (MSG_INIT), die die Nummer der Slaves enthält. Bis der Evolutionsprozeß gestartet wird, warten die beiden Slaves auf einen Auftrag, während der Master Eingaben des Benutzers entgegennimmt. Die dünne vertikale Linie markiert in den beiden Sichten den Start einer Evolution. Es werden die beiden Nachrichten MSG_INIT_PARS (Größe 320 Bytes) und MSG_SPEC_INIT (hier: Größe 1292 Bytes) an die Slaves gesendet. Diese initialisieren dann ihre Teilpopulation.

Was nach der Initialisierung der Teilpopulationen geschieht, ist in Abbildung 26 vergrößert dargestellt.

Zum Zeitpunkt *t1* bestätigt Slave 1 den erfolgreichen Abschluß der Initialisierungsphase (MSG_INIT_OK). Anschließend verschickt er die Individuen, die auf Slave 2 externe Nachbarn sind und wartet auf seine externen Nachbarn. Slave 2 beendet die Initialisierung erst zum Zeitpunkt *t2*, verschickt dann die von Slave 1 benötigten Individuen und nimmt anschließend seine bereits eingetroffenen externen Nachbarn entgegen. Die Nachrichten, die Individuen enthalten, sind die größten Nachrichten. Für ein Problem mit 198 Städten beträgt die Größe einer Nachricht 1660 Bytes, während sie bei 783 Städten 5972 Bytes

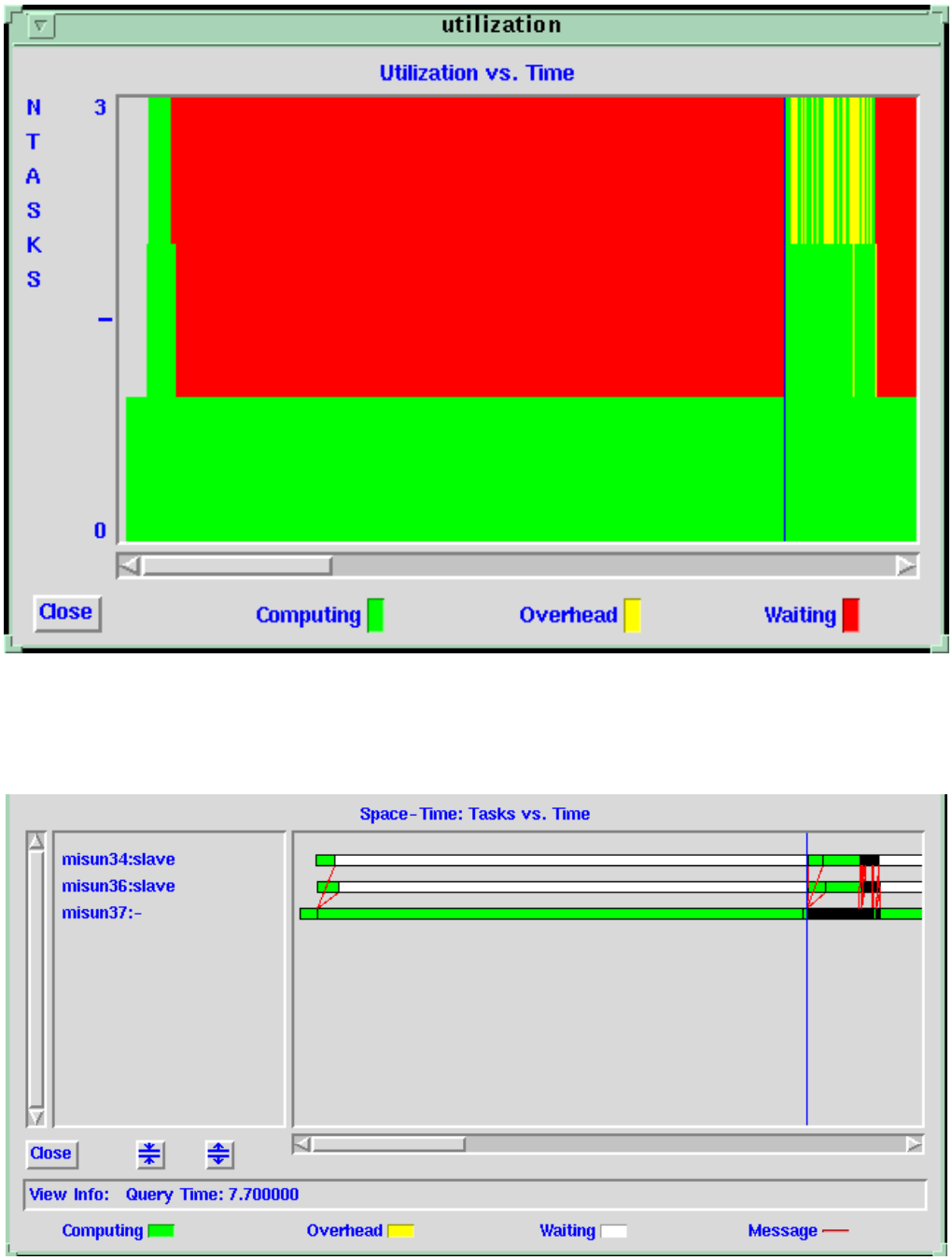

Abbildung 25: Zwei Sichten von XPVM: *Utilization View* (oben) und *Space-Time View* (unten) für einen Evolutionslauf

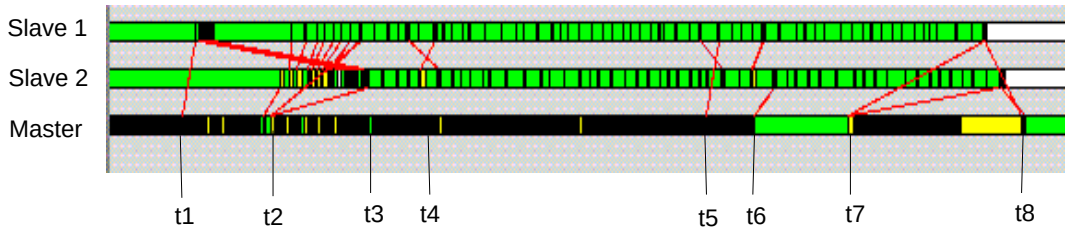

Abbildung 26: XPVM: Ablauf einer Evolution nach der Initialisierungsphase

umfaßt. Das Senden, Übertragen und Empfangen dauert meistens (in Abhängigkeit von der Netzlast) zwischen 1–10 ms. Darin ist allerdings nicht das Ver- und Entpacken von Ketten enthalten.

Nachdem der Master zum Zeitpunkt *t2* auch die zweite MSG_INIT_OK-Nachricht erhalten hat, schickt er gleich die Nachricht MSG_EVO_BEG. Etwa zum Zeitpunkt *t3* wird diese von den Slaves entgegengenommen, nachdem sie die externen Nachbarn ausgetauscht haben. Jeder der Slaves führt nun den Evolutionsprozeß für seine Teilpopulation durch, wobei zu den Zeitpunkten *t4, t5, t6* Aktualisierungsnachrichten externer Nachbarn zwischen den Slaves verschickt werden. Zum Zeitpunkt *t5* hat Slave 1 die erste Generation berechnet, sendet die Ergebnisnachricht MSG_GEN_ERG (Größe 32 Bytes) und geht zur nächsten Generation über. Slave 2 sendet die Nachricht MSG_GEN_ERG erst zum Zeitpunkt *t6*. Der Master bringt die periodische Ergebnis-Anzeige auf den Bildschirm, berechnet die Abbruchbedingungen und stellt fest, daß die vorgegebene Generationsanzahl (hier: 1) erreicht ist. Er sendet die Nachricht MSG_STOP zum Zeitpukt *t7* und wartet auf die Endergebnisnachrichten (MSG_END_ERG). Diese treffen zum Zeitpunkt *t8* ein (Größe 556 Bytes). Gleich nachdem sie die Ergebnisnachricht gesendet haben, warten die Slaves auf einen neuen Auftrag (zu erkennen an der weißen Farbe des Balkens oder des kleinen dunklen Bereichs im rechten Teil der *Utilization View*).

Die Laufzeit betrug laut XPVM vom Senden der MSG_INIT-Nachricht bis zum Empfangen der Endergebnisnachrichten 1,1 s. Wie hoch die Laufzeit der sequentiellen Version von GLEAM in Vergleich zur Laufzeit der verteilten Version mit 2 oder mehreren Slave-Prozessen ist, wird im nächsten Abschnitt vorgestellt.

9.3 Sequentielle vs. verteilte Version von GLEAM

Die Laufzeiten der sequentiellen und verteilten Version von GLEAM wurden anhand von 4 Problemen verglichen, die sich in der Zahl der besuchten Städte unterscheiden: 198, 442, 532 und 783. Für die Lösung der Probleme wurde eine ringförmig angeordnete Population mit 64 Individuen gewählt. Als Abbruchkriterium galt das Erreichen einer Generationsanzahl von 1000. Für jedes Problem wurden 50 Populationen berechnet. Auf einem Rechner wurde immer nur ein Slave gestartet.

Die erzielten Testergebnisse sind in den Tabellen aus Abbildung 27 zusammengefaßt und

in den Abbildungen 28 und 29 grafisch dargestellt.

Ein erster Test sollte zeigen, ob mit der Konfiguration 1-Master/1-Slave Zeit eingespart werden kann. In diesem Fall findet eine Arbeitsteilung zwischen Master und Slave statt: Während der Slave die Evolution durchführt, kümmert sich der Master um die Anzeige von Ergebnissen und die Berechnung der Abbruchbedingungen. Die gemessenen Laufzeiten bei der sequentiellen Version und der verteilten Version waren, wie erwartet, in etwa gleich. Dies ist darauf zurückzuführen, daß das periodische Verschicken von Ergebnisnachrichten einen Aufwand verursacht, der mit dem Aufwand zum Anzeigen von Ergebnissen und Berechnung von Abbruchbedingungen vergleichbar ist.

Das 198-Städte-Problem zeichnet sich dadurch aus, daß es die kleinsten Simulationszeiten der 4 hier vorgestellten Probleme hat: Sie liegen, je nach Rechnertyp, zwischen $0,16$ und $0,48$ ms. Wird die Evolution von 2 oder 4 Slaves durchgeführt, so nimmt die benötigte Laufzeit in etwa linear ab. Auch mit 8 bzw. 16 Slaves kann man noch Laufzeitgewinne verzeichnen, aber bereits 20 Slaves brauchen mehr Zeit als 16. Die Erklärung hierfür ist einfach: Da die Simulationszeiten klein sind, ist die Zeit, die für das Aktualisieren der externen Nachbarn benötigt wird, nicht mehr vernachlässigbar. Wenn sich die Anzahl der Slave-Prozesse verdoppelt, verdoppelt sich in etwa auch die Anzahl der durchzuführenden Aktualisierungen (s. Abb. 27).

Bei den restlichen 3 Problemen ist eine Halbierung der Rechenzeit beim Rechnen mit 2 Slaves festzustellen. Die Laufzeiten für das 532- und 783-Städte-Problem betragen sogar weniger als die Hälfte der Laufzeiten der sequentiellen Version. Dies ist darauf zurückzuführen, daß die Slaves weniger Individuen bearbeiten und dadurch mehr Informationen im Hauptspeicher gehalten werden können. Ein fast lineares Verhalten ist auch bei einer Verdoppelung der Anzahl der Slaves auf 4 zu beobachten. Werden weitere Slaves hinzugenommen, so macht sich auch bei diesen Problemen, deren Simulationszeiten bei $0,4$–$1,2$ ms (442 Städte), $0,5$–$1,4$ ms (532 Städte), $0,8$–$2,2$ ms (733 Städte) liegen, die für die Aktualisierungen der externen Nachbarn benötigte Zeit bemerkbar. Aber im Gegensatz zum 198-Städte-Problem, wo 20 Slaves 1 Sekunde länger brauchten als 16, ist bei den restlichen Problemen eine, wenn auch geringe, Reduzierung der Laufzeit zu beobachten. So wird beim 783-Städte-Problem durch die weiteren 4 hinzugefügten Slaves die Laufzeit um 18 s auf 118 s, bei dem 442-Städte-Problem um 7 s auf 41 s gesenkt.

Falls die Population in viele Teilpopulationen zerlegt wird (viele Slave-Prozesse), ist darauf zu achten, daß der Master-Prozeß nicht auf einem langsamen Rechner gestartet wird. Da der Master-Prozeß eigentlich keine Evolution durchführt und nur für das Anzeigen von Ergebnissen zuständig ist, könnte man meinen, daß die Slave-Prozesse den schnellen Rechnern einer Konfiguration zugeteilt werden sollten. Wenn die Slave-Prozesse nun Ergebnisnachrichten in kurzen Abständen senden, so kann es leicht vorkommen, daß der Master-Prozeß durch die Nachrichtenflut überfordert wird. Wenn er die MSG_STOP-Nachricht an die Slaves schickt, haben diese bereits viel mehr Generationen berechnet als notwendig. Bei Problemen mit kurzen Simulationszeiten und vielen Slave-Prozessen wäre es also vorteilhaft, daß auch der Master-Prozeß auf einem schnellen Rechner gestartet wird.

Festzuhalten ist noch, daß die Anzahl der Aktualisierungen nicht von der Größe des Pro-

198-Städte-Problem:

Anzahl Slaves	Laufzeit (s)	Beste Lösung	Schlechteste Lösung	≤ 15781	Anzahl Aktualis.
0 (seq.)	134,0	15780	15799	94%	0
2	69,7	15780	15785	96%	420
4	37,9	15780	15781	100%	850
6	27,0	15780	15784	98%	1250
8	25,0	15780	15785	94%	1500

442-Städte-Problem:

Anzahl Slaves	Laufzeit (s)	Beste Lösung	Schlechteste Lösung	≤ 50950	Anzahl Aktualis.
0 (seq.)	419,8	50904	51235	50%	0
2	207,1	50927	51033	54%	680
4	113,3	50927	51097	48%	1320
6	75,3	50926	51081	58%	2000
8	65,0	50928	51163	60%	2550

532-Städte-Problem:

Anzahl Slaves	Laufzeit (s)	Beste Lösung	Schlechteste Lösung	≤ 27760	Anzahl Aktualis.
0 (seq.)	617,1	27718	27835	28%	0
2	297,7	27732	27829	36%	930
4	150,5	27728	27837	42%	1900
6	106,0	27727	27853	24%	2800
8	95,0	27718	27844	36%	3450

783-Städte-Problem:

Anzahl Slaves	Laufzeit (s)	Beste Lösung	Schlechteste Lösung	≤ 8900	Anzahl Aktualis.
0 (seq.)	1371,6	8883	9016	6%	0
2	680,0	8866	9024	4%	770
4	364,7	8854	9006	20%	1500
6	244,6	8868	9042	10%	2220
8	214,2	8846	9016	24%	3000

Abbildung 27: Testergebnisse

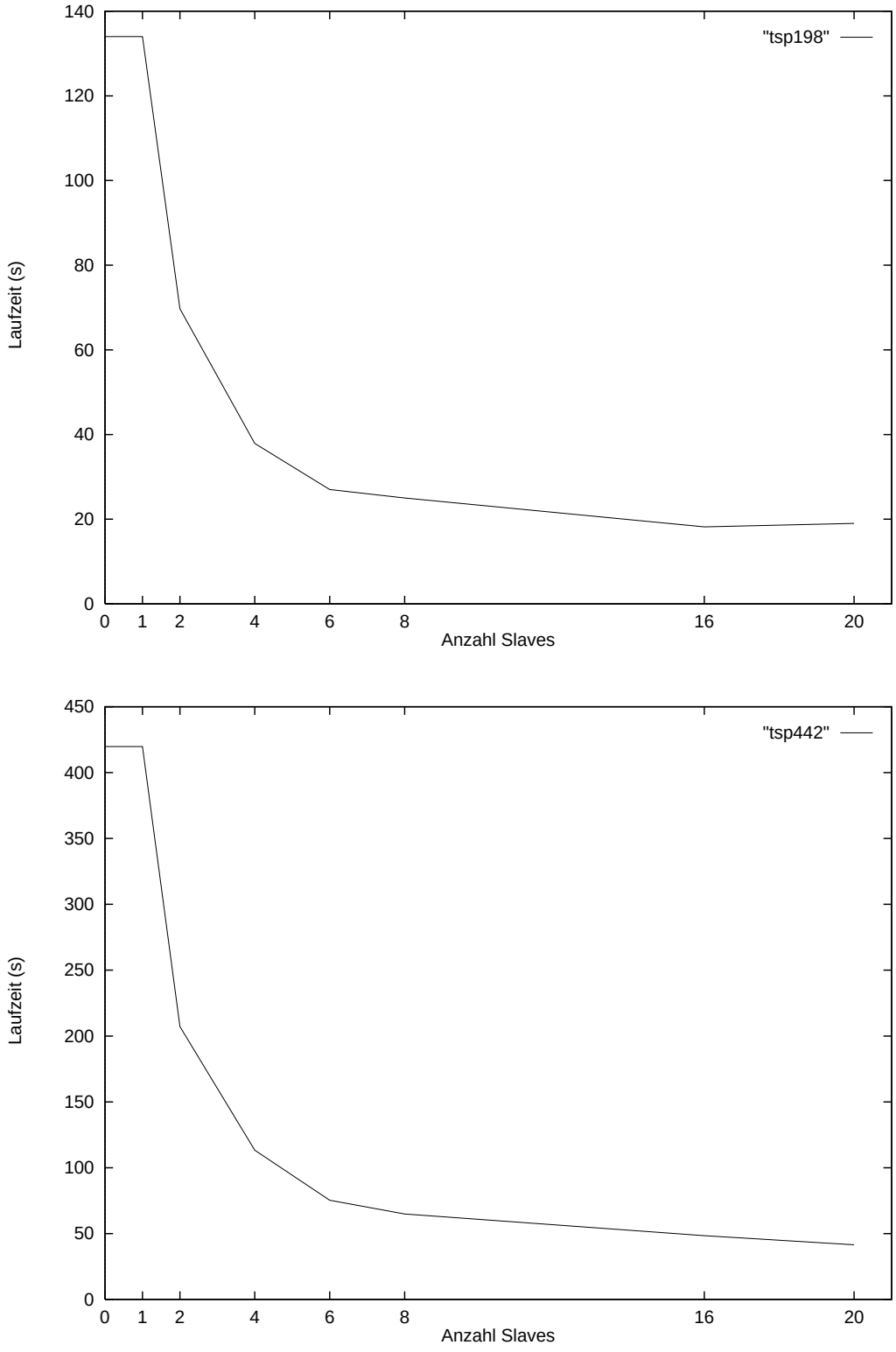

Abbildung 28: Verringerung der Laufzeit durch das Hinzunehmen von Rechnern bei dem 198- bzw. 442-Städte-Problem

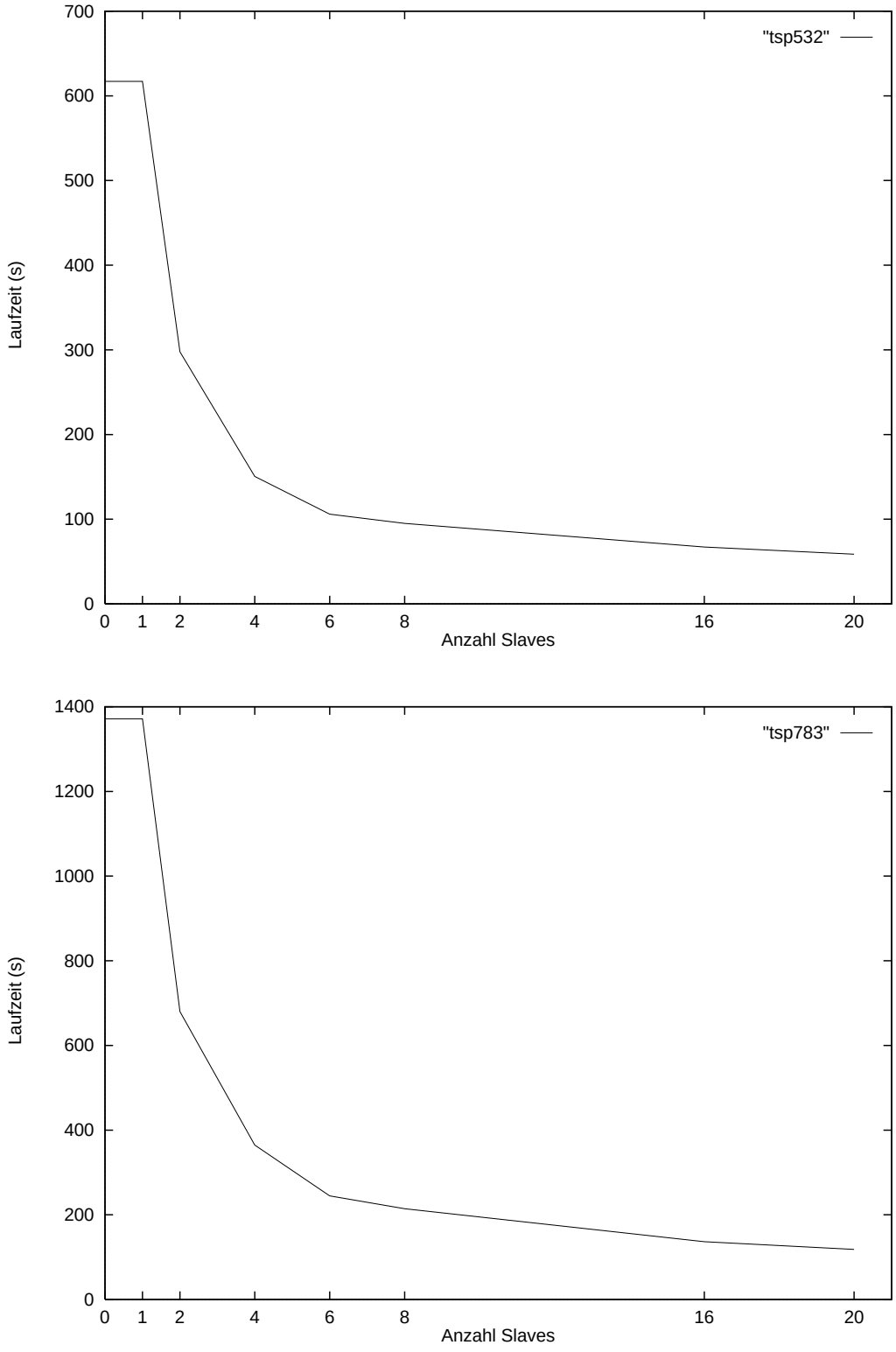

Abbildung 29: Verringerung der Laufzeit durch das Hinzunehmen von Rechnern bei dem 532- bzw. 783-Städte-Problem

blems, sondern dessen Struktur abhängt. So finden beim 532-Städte-Problem, das als besonders schwierig gilt, mehr Aktualisierungen statt als beim 783-Städte-Problem.

Durch den Übergang von der sequentiellen zur verteilten Version des GLEAM-Verfahrens wurde der Teil des Algorithmus, der die Evolution eines Individuums umfaßt, nicht verändert. Und trotzdem wird durch die Verteilung die Suche nach Lösungen beeinflußt. Während im sequentiellen Fall nur ein Zufallszahlengenerator benutzt wird, sind es im verteilten Fall mehrere (einer pro Slave-Prozeß). Einen noch größeren Einfluß hat die Heterogenität des Rechnernetzes: Werden für einen Evolutionslauf Rechner mit unterschiedlicher Schnelligkeit benutzt, so werden in den Teilpopulationen, deren Evolution von schnellen Rechnern durchgeführt wird, viel mehr Generationen berechnet als in den restlichen. Ist die Anfangskonfiguration dieser Teilpopulationen günstig, so werden dadurch, daß viele Generationen berechnet werden, auch gute Lösungen gefunden. Die Teilpopulationen, die sich langsam entwickeln, wirken der Stagnation entgegen: Wenn die sich schnell entwickelnden Teilpopulationen in eine Stagnationsphase geraten, erhalten sie, durch neue Informationen von den langsamen Teilpopulationen, die Chance sich weiterzuentwickeln.

Der Einfluß den die unterschiedliche Schnelligkeit von Rechnern auf die Güte der gefundenen Lösungen hat, ist an der 1-Master/8-Slaves-Konfiguration zu beobachten. Hier wurden 6 in etwa gleich schnelle und 2 langsamere Rechner benutzt. Wenn man die Güte der gefundenen Lösungen aus Abbildung 27 näher betrachtet, so kann man folgende Anmerkungen machen: Für das einfache 198-Städte-Problem haben alle betrachteten Konfigurationen die optimale Lösung gefunden. Über 94% der Evolutionsläufe fanden eine Weglänge, die um maximal 1 größer war als die optimale Weglänge. Die von der sequentiellen Version gefundene Lösung für das 442-Städte-Problem (Weglänge 50904) ist als Glückstreffer zu betrachten. Diese Lösung wurde nur ein einziges Mal gefunden. Die zweitbeste Lösung lag hier bei 50927. Der Anteil der gefundenen Wege, deren Weglänge unter 50950 liegt, bewegt sich zwischen 50% und 60%. Insgesamt sind auch für das 442-Städte-Problem keine wesentlichen Unterschiede zu beobachten.

Interessant wird es erst bei den schwierigeren 532- und 783-Städte-Problem. Bei 783 Städten führt die Heterogenität des Rechnernetzes dazu, daß sowohl die beste Lösung (Weglänge 8846), als auch im Mittel bessere Lösungen gefunden werden. Beim 532-Städte-Problem wird mit 8 unterschiedlich schnellen Slaves als beste Lösung die gleiche Lösung wie im sequentiellen Fall gefunden. Auch hier liefert diese Konfiguration im Mittel bessere Lösungen.

Durch Heterogenität wird die Entwicklung der Teilpopulationen entkoppelt und der Zufall spielt eine größere Rolle. Die Chance, die die großen Generationsunterschiede zwischen den Slaves zur Entwicklung guter Lösungen bietet, birgt aber auf der anderen Seite auch die Gefahr, daß gerade die Teilpopulationen, deren Anfangskonfiguration günstig ist, sich kaum entwickeln können, weil sie einem langsamen Rechner zugeteilt wurden.

9.4 Laufzeitanalysen

Die Simulationszeiten für die im vorherigen Abschnitt vorgestellten Probleme bewegen sich im Millisekunden-Bereich. Wie im vorherigen Abschnitt bereits erwähnt, steigt mit

dem Hinzunehmen von weiteren Slaves das Nachrichtenaufkommen zwischen den Slaves. Dies führt dazu, daß der Aufwand zur Verarbeitung der Nachrichten den Gewinn an Rechenzeit, der durch die zusätzlichen Rechner entsteht, um einiges schmälert.

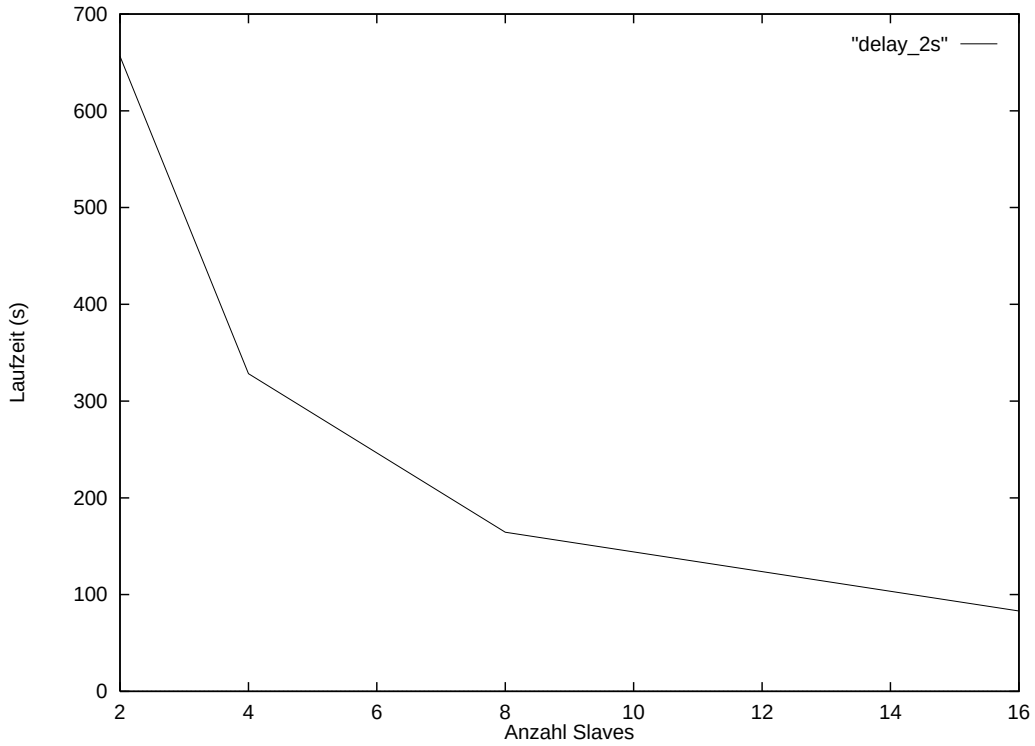

Abbildung 30: Laufzeit in Abhängigkeit von der Anzahl der Slaves bei Simulationszeiten von circa 2 Sekunden

Die Vermutung liegt also nahe, daß bei längeren Simulationszeiten, verglichen am Aufwand der Aktualisierung von Nachbarn, höhere Rechenzeitgewinne möglich sind. Die Simulationszeit wurde für einen ersten Versuch künstlich auf circa zwei Sekunden verlängert. Die Populationsgröße wurde auf 16 reduziert, um auch den Extremfall, daß die Teilpopulation eines Slaves nur aus einem Individuum besteht, in die Tests einzubeziehen: Werden 16 Slaves gestartet, so wird in diesem Fall die maximal mögliche Verteilung erreicht, d. h. daß auch die Anzahl der Aktualisierungen maximal wird. Außerdem wurde, da nur Laufzeitverhalten interessiert, die Maximalanzahl an Generationen auf 20 gesenkt. Die durchschnittlichen Laufzeiten aus 10 Evolutionsläufen, bei denen jeder Slave einem anderen Rechner zugeordnet wurde, sind in Abbildung 30 dargestellt. Durch die Verdoppelung der Anzahl der Slaves von 2 auf 4, 8 und 16 halbierte sich die Rechenzeit von 656 s mit 2 Slaves auf 328 s, 164 s bzw. 82 s. Man erhält also bereits bei einer Simulationszeit von etwa 2 Sekunden ein lineares Verhalten! Die Testergebnisse wurden auch durch einen Test mit einer Simulationszeit von 20 Sekunden bestätigt.

Für die Optimierung des Designs der Mikropumpe (s. Abschnitt 4.5.2), wo die Simulationszeiten drei bis vier Minuten betragen, ist also durch Verteilung eine lineare Leistungssteigerung zu erwarten.

9.5 Einfluß der Populationsstrukturen

Die sequentielle Version des GLEAM-Verfahrens ist auf eine ringförmige Populationsstruktur zugeschnitten. Die verteilte Version unterstützt nun durch die Einführung der Nachbarschaftsmatrix beliebige Populationsstrukturen.

Um den Einfluß der Populationsstrukturen auf die Güte der gefundenen Lösungen zu untersuchen, wurde eine Population mit 64 Individuen auf einem Ring bzw. auf einem 8x8-Torus angeordnet. Bei einer Nachbarschaftsgröße von 8 sind beim Torus zwei Formen der Nachbarschaft möglich: *Lineare* Nachbarschaft, die diejenigen 8 Individuen umfaßt, die in 2 Schritten vom zentralen Individuum erreichbar sind, bzw. *kompakte* Nachbarschaft, die die 8 nächsten Individuen des zentralen Individuums umfaßt (s. Abb. 31). Der kleinste Kreis, der die lineare Nachbarschaft eines Individuums einschließt, hat einen Radius von 2, während für eine kompakte Nachbarschaft ein Radius von $\sqrt{2}$ ausreicht.

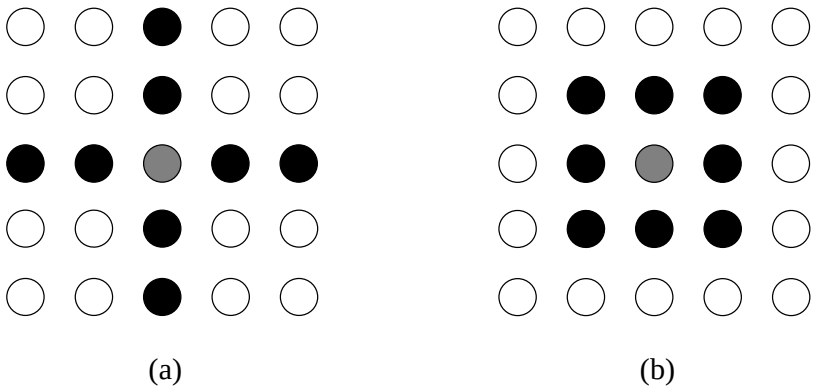

Abbildung 31: Nachbarn des grauen Individuums in der Populationsstrukturen (a) Torus-linear bzw. (b) Torus-kompakt

Der wesentliche Unterschied zwischen einem Ring und einem Torus besteht darin, daß beim Ring Informationen nur in zwei Richtungen (linear) durch die Population weitergereicht werden können, während es beim Torus vier Richtungen bei linearer Nachbarschaft und 8 Richtungen bei kompakter Nachbarschaft sind. Dadurch erlaubt der Torus einen schnelleren Austausch von Informationen, wobei die Form der Nachbarschaft und das Verhältnis zwischen Größe der Nachbarschaft und Populationsgröße eine Rolle spielt[JS96]. Je schneller sich Informationen durch die gesamte Population verbreiten können, desto höher ist der Selektionsdruck. Ein hoher Selektionsdruck führt dazu, daß die Evolution schneller gute Lösungen findet. Der Nachteil ist aber auch, daß der Faktor Isolation nicht mehr zum tragen kommt und es dadurch schwerer wird, ein lokales Optimum zu verlassen. Bei einem langsamen Diffusionsprozeß braucht die Evolution länger, um eine gute Lösung zu finden, ist dafür aber weniger anfällig für Stagnation.

Wie hoch der Selektionsdruck bei den hier zu untersuchenden Strukturen (Ring, Torus-linear und Torus-kompakt mit Nachbarschaftsgröße 8 und Populationsgröße 64) ist, wurde folgendermaßen ermittelt: Die Population wird so initialisiert, daß ein Individuum x viel besser als die anderen ist. Wird nun x von einem Individuum y als Partner gewählt,

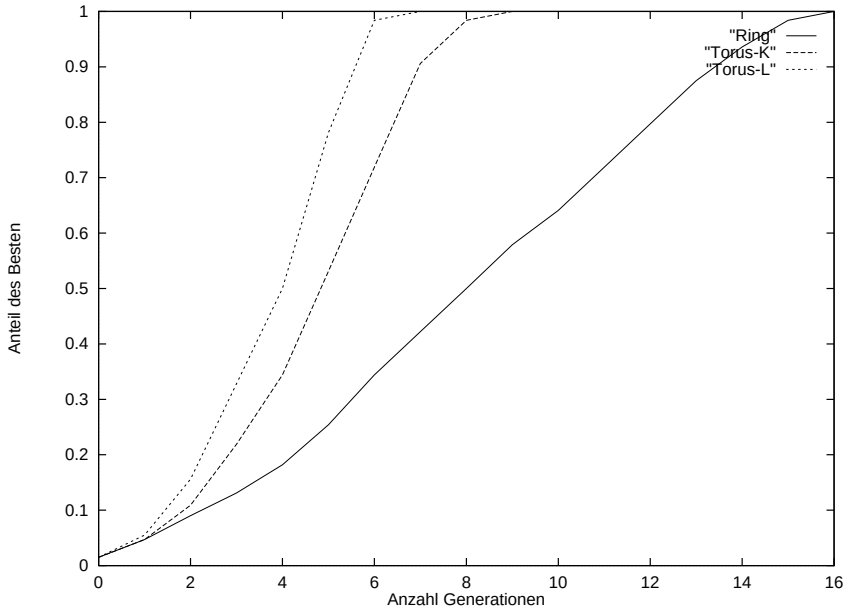

Abbildung 32: Anteil des Besten in der Gesamtpopulation nach einer bestimmten Anzahl von Generationen in Abhängigkeit von der gewählten Populationsstruktur

so wird y durch x ersetzt. Gemessen wird die Anzahl der Generationen, bis die ganze Population nur noch aus Kopien des Individuums x besteht. Für die Selektionsstrategie BETTER_PARENT und linearer rangbasierter Selektion mit $max = 1,7$ sind die erzielten Ergebnisse für die oben angeführten Populationsstrukturen in Abbildung 32 dargestellt. Daraus ist ersichtlich, daß Torus-linear den größten Selektionsdruck besitzt, gefolgt vom Torus-kompakt und der Ring-Struktur.

Populations-struktur	Laufzeit (s)	Beste Lösung	Schlechteste Lösung	Durchschnittl. Weglänge	Anzahl Aktualis.
Ring	299,1	27730	27854	27774	930
Torus-linear	318,4	27734	27971	27772	3540
Torus-kompakt	313,3	27741	27827	27767	1830

Tabelle 1: Testergebnisse des 532-Städte-Problems in Abhängigkeit von der gewählten Populationsstruktur nach 1000 Generationen

Wie die Güte der gefundenen Lösungen bei der Wahl verschiedener Populationsstrukturen ist, wurde an dem 532-Städte-Problem getestet. Es wurden 20 Testläufe in der Konfiguration 1-Master/2-Slaves gestartet, wobei jeweils 1000 Generationen berechnet wurden. Die Ergebnisse sind in Tabelle 1 zusammengefaßt. Die durchschnittlichen Weglängen sind ungefähr gleich, wobei wie zu erwarten war, der Ring von allen Strukturen die beste Lösung findet. Dies liegt daran, daß beim Ring wegen dem sehr hohen Maß an Isolation die Durchmischung der Population besser ist und so eine kontinuierliche Diffusion

stattfindet. Die Torus-Struktur hingegen ermöglicht durch wenig Isolation eine schnelle Diffusion, die beim Torus-linear am schnellsten ist.

In Abbildung 33 ist die Entwicklung der Populationen für den Evolutionslauf dargestellt, in dem die jeweils besten Lösungen gefunden wurden. Daraus ist ersichtlich, daß sich sowohl bei Torus-linear als auch bei Torus-kompakt die Güte des Populationsbesten relativ sprunghaft ändert, während beim Ring eine kontinuierliche Verbesserung zu beobachten ist. Der extrem hohe Selektionsdruck führt beim Torus-linear dazu, daß am Anfang die Güte schnell wächst, daß aber später dieser Selektionsdruck, verstärkt durch die Selektionsstrategie BETTER_PARENT, das Finden von besseren Lösungen behindert. Torus-kompakt, mit seinem nicht so hohen Selektionsdruck findet am Anfang nicht so schnell gute Lösungen wie der Torus-linear, liefert dafür aber später die besseren Lösungen als dieser. Die Ring-Population entwickelt sich im Verhältnis zu den beiden Torus-Strukturen langsamer, ist aber durch die größere Anzahl von verschiedenen Individuen in der Lage, sich stetig weiterzuentwickeln.

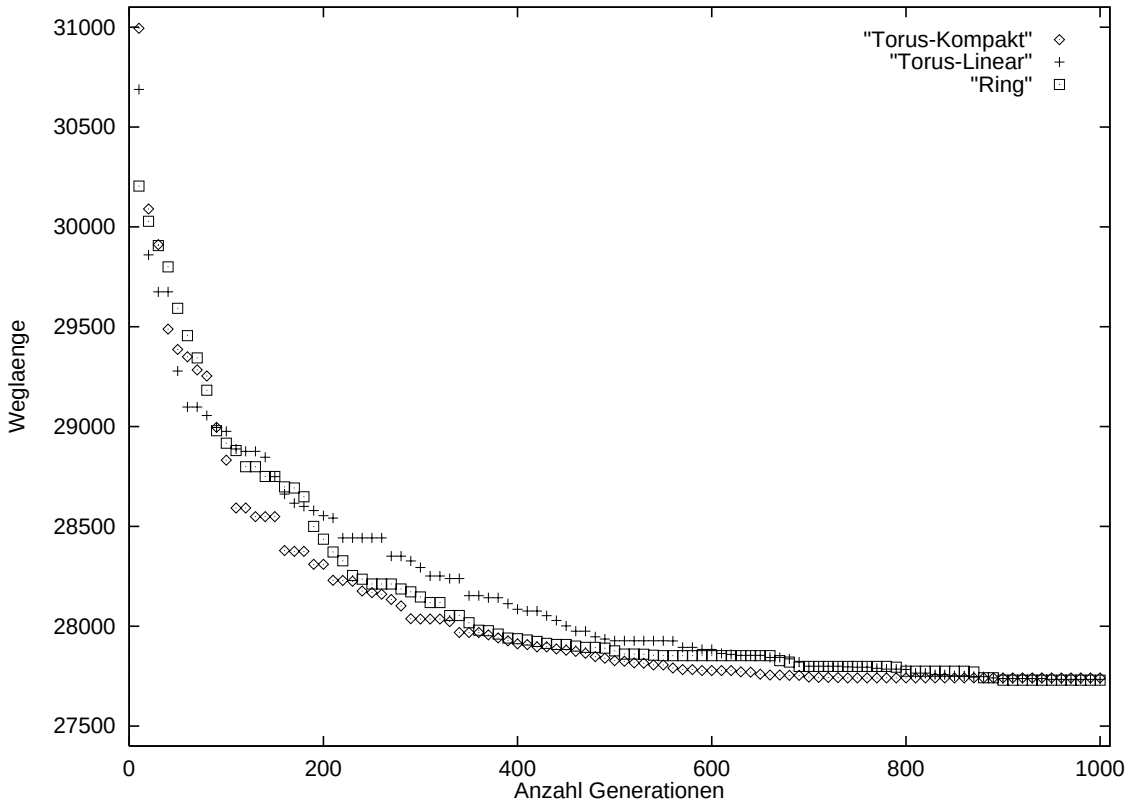

Abbildung 33: Güte des Populationsbesten während der Evolution in Abhängigkeit von der gewählten Populationsstruktur

Dieses Beispiel zeigt, daß der Ring die Möglichkeit hat, die besten Lösungen zu finden. Wenn der Anwender jedoch nur schnell eine gute Lösung haben möchte, so ist der Torus mit kompakter Nachbarschaft die bessere Wahl.

10 Bewertung und Ausblick

Die hier vorgestellten Ergebnisse zeigen, daß die Geschwindigkeit eines 10 Mbit/s schnellen lokalen Netzes bereits ausreicht, um auch Programme mit großem Rechenaufwand zu beschleunigen, die bei verteilter Abarbeitung relativ kommunikationsintensiv sind. Für das hier vorgestellte GLEAM-Verfahren ist sogar eine lineare Leistungssteigerung möglich, falls die Zeit, die für die Berechnungen gebraucht wird, länger ist als die Verarbeitung ankommender Nachrichten.

Ein weiterer Vorteil, den man durch die Benutzung mehrerer Rechner zur Lösung eines Problems erzielt, ist, daß die Populationsgröße variiert werden kann. So können Populationsgrößen, die im sequentiellen Fall leicht zu Speichermangel führen, im verteilten Fall bearbeitet werden, vorausgesetzt eine ausreichend große Anzahl von Rechnern steht zur Verfügung.

Die verteilte Version des GLEAM-Verfahrens besitzt außerdem ein hohes Maß an Fehlertoleranz. Nach der Initialisierungsphase entwickeln sich die Teilpopulationen asynchron weiter, wobei sich ein Fehler in einem der Slave-Prozesse nur durch Stagnation seiner Teilpopulation bemerkbar macht. Die Entkopplung der Teilpopulationen wird noch deutlicher, wenn deren Evolution von unterschiedlich schnellen Rechnern durchgeführt wird. Der Zufall spielt dann eine größere Rolle, was, wie die erzielten Ergebnisse belegen, einen durchaus positiven Einfluß auf die Entwicklung der Gesamtpopulation haben kann.

Durch die Einführung der Nachbarschaftsmatrix können beliebige Populationsstrukturen unterstützt werden. Dadurch wird dem Benutzer die Möglichkeit gegeben, eine seinem Problem angepaßte Struktur zu wählen. Möchte man z. B. schnell eine gute Lösung, so sind dafür eher Strukturen mit einem hohen Selektionsdruck vorteilhaft. Sucht man eine möglichst gute Lösung, wobei die Zeit keine Rolle spielt, so eignen sich dafür Strukturen mit niedrigem Selektionsdruck.

Die verteilte Version des GLEAM-Verfahrens ist erweiterungsfähig. Leicht zu realisieren ist die teilpopulationspezifische Variation der Evolutionsparameter. So können zum Beispiel in den Teilpopulationen verschiedene Selektionsstrategien benutzt werden, um der Stagnation entgegenzuwirken. Eine andere Methode gegen Stagnation anzukämpfen ist die Reinitialisierung von Teilpopulationen, wenn sich die Individuen sehr ähnlich sind. Dabei ist allerdings zu beachten, daß das lokal beste Individuum erhalten bleibt.

Während der Untersuchungen wurde festgestellt, daß bei einer Konfiguration mit Rechnern unterschiedlicher Leistungsfähigkeit große Generationsdifferenzen zwischen den Teilpopulationen auftreten können. Denkbar ist deshalb eine Erweiterung der Abbruchbedingungen, die vor allem in heterogenen Rechnernetzen dafür sorgen soll, daß jede Teilpopulation eine Mindestanzahl von Generationen berechnen muß.

Vielversprechende Ergebnisse für das Problem des Handelsreisenden wurden erzielt, indem das Konzept von Populationshierarchien benutzt wurde[GS]. Dies besteht darin, daß mehrere strukturierte Populationen parallel bearbeitet werden, wobei eine Migration der besten Individuen zwischen den Populationen stattfindet. Es werden dadurch zwei Modelle vereint: das Inselmodell auf der Ebene der Populationen und das Nachbarschaftsmodell

auf der Ebene der Subpopulationen. Interessant ist sicherlich herauszufinden, ob dieses Prinzip der Populationshierarchien auch für andere GLEAM-Anwendungen Vorteile bringt.

In dieser Arbeit wurde die verteilte Version des GLEAM-Verfahrens vorgestellt. Durch den Übergang von der sequentiellen zur parallelen Durchführung der Evolution wurde in erster Reihe eine Laufzeitreduzierung und eine naturgetreuere Abbildung der biologischen Evolution erreicht. Außerdem wurde aber auch festgestellt, daß durch die Benutzung eines heterogenen Rechnernetzes die Qualität der gefundenen Lösungen beeinflußbar ist. Welche Auswirkungen genau die Interaktion zwischen eng kooperierenden Teilpopulationen, die sich in unterschiedlichen Entwicklungsphasen befinden, hat, könnte das Ziel weiterer Untersuchungen sein.

Literatur

[AGS94] R. Manchek, A. Geist, J. Dongarra, V. Sunderam. *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing.* The MIT Press, 1994.

[Bal93] S. Baluja. *Structure and Performance of Fine-grained Parallelism in Genetic Search.* In Proc. of the 5th International Conference on Genetic Algorithms, Urbana-Champaign, July 17–21, 1993, Morgan Kaufmann Publishers, S. 155–162

[Bir90] K. Birman. *The ISIS system manual, version 2.0.* Cornell University, Computer Science Department, 1990.

[Blu90] C. Blume. *GLEAM – A System for Simulated Intuitive Learning.* In Proc. of the 1st Int. Workshop on Parallel Problem Solving from Nature, Dortmund, Okt. 1–3, 1990, LNCS 496, Springer-Verlag, 1991, S. 48–54.

[CHP82] K. Steiglitz, C. H. Papadimitriou. *Combinatorial Optimization.* Prentice Hall, 1982.

[ES94] S. Federsen, E. Schöneburg, F. Heizmann. *Genetische Algorithmen und Evolutionstrategien.* Addison-Wesley, 1994.

[GS] M. Gorges-Schleuter. *On the Power of Evolutionary Optimization at the Example of ATSP and large TSP Problems.* To be published.

[GS91] M. Gorges-Schleuter. *Genetic Algorithms and Population Structures – A Massively Parallel Algorithm –.* Dissertation. Universität Dortmund, Fachbereich Informatik, 1991.

[GS94a] M. Gorges-Schleuter. *Evolutionary TSP Optimization: An Application Case Study.* In Plantamura [PSV94], 1994, S. 287–318.

[GS94b] M. Gorges-Schleuter. *Parallel Evolutionary Algorithms and the Concept of Population Structures.* In Plantamura [PSV94], 1994, S. 261–286.

[GS97] M. Gorges-Schleuter. *Asparagos96 and the Traveling Salesman Problem.* In Proc. IEEE ICEC'97, Indianapolis, April 14–16, 1997.

[GSJM+96] M. Gorges-Schleuter, W. Jakob, S. Meinzer, A. Quinte, W. Süß, H. Eggert. *An Evolutionary Algorithm for Design Optimization of Microsystems.* In Proc. of the 4th Int. Conf. on Parallel Problem Solving from Nature, Berlin, LNCS 1141, Springer-Verlag, 1996, S. 1022–1032.

[Hol75] J. H. Holland. *Adaption in Natural and Artificial Systems.* University of Michigan press, Ann Arbor, 1975.

[JMQ96a] W. Jakob, S. Meinzer, A. Quinte. *Simulator und genetischer Algorithmus verkürzen die Entwicklungsphase.* elektronik industrie 5, 1996.

[JS96] K. De Jong, J. Sarma. *An Analysis of the Effects of Neighborhood Size and Shape on Local Selection Algorithm.* In Proc. of the 4th Int. Conf. on Parallel Problem Solving from Nature, Berlin, LNCS 1141, Springer-Verlag, 1996, S. 236–244.

[(Jon93] K. De Jong. *On the State of Evolutionary Computation*, In Proc. of the 5th International Conference on Genetic Algorithms, Urbana-Champaign, July 17-21, 1993, Morgan Kaufmann Publishers, S. 618–623.

[MG94] A. Samarin, M. Goossens, F. Mittelbach. *The LaTeX Companion.* Addison Wesley, 1994.

[NC89] D. Gelernter, N. Carriero. *Linda in Context.* Communications of the ACM, Band 32, 1989.

[Nis94] V. Nissen. *Evolutionäre Algorithmen.* Deutscher Universitäts-Verlag, 1994.

[PSV94] V. L. Plantamura, B. Soucek, G. Visaggio (Ed.). *Frontier Decision Support Concepts.* John Wiley & Sons, 1994.

[Rec73] I. Rechenberg. *Evolutionsstrategie — Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann–Verlag, 1973.

[Sch81] H. P. Schwefel. *Numerical Optimization of Computer Models.* Wiley, 1981.

[Sch93] A. Schill. *DCE - Das OSF Distributed Computing Environment.* Springer-Verlag, 1993.

[Ung94] T. Ungerer. *Parallelrechner und Parallelprogrammierung.* Universität Karlsruhe, Fakultät für Informatik, 1994.

[VSSM94] J. Dongarra, V. S. Sunderam, A. Geist, R. Manchek. *The PVM Concurrent Computing System: Evolution, Experiences, and Trends.* Parallel Computing, Band 20, 1994.

[WJ92] C. Blume, W. Jakob, M. Gorges-Schleuter. *Application of Genetic Algorithms to Task Planning and Learning.* In Proc. of the 2nd Int. Conf. on Parallel Problem Solving from Nature, Brüssel, Sept. 28–30, 1992, Elsevier Science Publishers, S. 291–300.