Johann Wolfgang Goethe-Universität
Frankfurt am Main

GUM: Eine portable parallele Implementation von Haskell

Seminar
Parallele und verteilte Ausführung funktionaler Programme

Fachbereich 20 - Informatik

Michaela Pfeifer
Wintersemester 1996/97

Inhaltsverzeichnis

1 Einleitung....2

2 Die Arbeitsweise von GUM 3

2.1 Thread Verwaltung....3
2.1.1 Funken (Sparks)....4
2.1.2 Synchronisation....4

2.2 Lastverteilung 5

2.3 Speicherverwaltung 6
2.3.1 Lokale Adressen....6
2.3.2 Globale Adressen 6
2.3.3 Garbage Collection....7
2.3.4 Verteilte Daten....9

2.4 Packing/Unpacking Graph 10

3 Vorläufige Ergebnisse 11

3.1 Leistungsfähigkeit bei Single-Prozessoren 11

3.2 Leistungsfähigkeit bei Multiprozessoren 12

4 Fazit 13

5 Literatur 14

1 Einleitung

GUM (Graph Reduction for a Unified Machine Model) ist eine portable, parallele Implementation der nicht-strikten funktionalen Programmiersprache Haskell. GUM ist eines der ersten Systeme, die öffentlich verfügbar waren. GUM ist sowohl für Shared-Memory (Sun SPARCserver Multiprozessoren) als auch für Distributed-Memory Architekturen erhältlich. Die hohe Wartezeit bei dem Senden von Nachrichten bei einer Distributed-Memory Maschine wird dadurch verbessert, indem Nachrichten asynchron gesendet werden.

GUM besitzt folgende charakteristische Merkmale:

- ***GUM ist portabel,*** sie ist nachrichtenbasierend und benutzt die PVM (Parallele Virtuelle Maschine). Die PVM ist eine Kommunikationsstruktur, die auf fast jedem Multiprozessor vorhanden ist, sowohl auf Shared-Memory- als auch auf Distributed-Memory-Maschinen. Die grundlegende Architektur hierbei ist eine Sammlung von Prozessor-Elementen (kurz: PEs), die durch eine Art Netzwerk miteinander verbunden sind. Auf dieses wird mithilfe der PVM zugegriffen. Die PVM nutzt ihren eigenen Zusatzcode aus, jedoch existieren Abkürzungen, die die Kommunikation zwischen homogenen Maschinen auf einem einzelnen Netzwerk erleichtern.

- ***GUM kann signifikante Beschleunigungen liefern***, entsprechend der besten Compiler-Technologie. Dabei ist es unnötig, zu erwähnen, daß sich diese Forderung an parallele Programme richtet. Die Beschleunigung wird dadurch erreicht, in dem der beste erhältliche sequentielle Haskell Compiler verwendet wird, der Glasgow Haskell Compiler (GHC). In der Tat ist GUM nur ein neues Laufzeitsystem für den GHC. Die sequentiellen Teile eines Programms haben die gleiche Laufzeit, wie wenn sie mit dem GHC für eine sequentielle Maschine compiliert werden würden, abgesehen von einem kleinen konstanten Faktor.

- ***GUM unterstützt unabhängige lokale Garbage Collection mithilfe eines einzelnen globalen virtuellen Speichers***. Jeder PE besitzt einen lokalen Speicher, der einen Teil des globalen virtuellen Speichers realisiert. Ein zweischichtiges Adressierungsschema unterscheidet zwischen lokalen Adressen innerhalb des lokalen PE-Speichers und globalen Adressen, die auf die lokalen Speicher zeigen. Die globalen Adressen werden hierbei so verwaltet, daß jedes PE die Möglichkeit zu einer Garbage Collection innerhalb seines lokalen Speichers besitzt. Diese muß nicht synchron mit den anderen PEs stattfinden (siehe Kapitel 2.3.3).

- ***Threadverteilung wird lazy ausgeführt, während die Datenverteilung nicht lazy ausgeführt wird.*** Threads werden nie zu anderen PEs exportiert, um dadurch ein Gleichgewicht zu erhalten. Sie werden nur dann bewegt, wenn ein Prozessor keine Threads abarbeiten muß und somit im Leerlauf ist (Kapitel 2.2). Man vermeidet so unnötige und kostenintensive Threadverschiebung.

- ***Alle Meldungen laufen asynchron***. Die Idee hierbei ist folgende: Hat ein Prozessor einmal eine Nachricht verschickt, so kann er alles darüber vergessen und weitere Nachrichten und Threads abarbeiten, ohne auf Antwort zu warten (Kapitel 2.3.4). Dabei ist zu bemerken, wenn ein Prozessor Daten von einem anderen holen möchte, so sendet er eine Nachricht, deren Antwort willkürlich verzögert werden kann. Zum Beispiel können Daten am anderen Ende noch unter Auswertung stehen. Wenn die Antwort dann schließlich ankommt, so kann diese als unabhängiger Arbeitspunkt betrachtet werden.

2 Die Arbeitsweise von GUM

Die erste Aktion eines GUM-Programms besteht darin, den PVM-Manager-Task zu erstellen, der den Beginn und das Ende des Programms kontrolliert. Der Manager stellt die nötige Anzahl an logischen PEs als PVM-Tasks zur Verfügung, welche die PVM den verfügbaren Prozessoren zuteilt. Jeder PE-Task initialisiert sich dann selbst: die Verarbeitung der Laufzeit-Argumente, Zuteilung des lokalen Speichers usw. Hat sich der PE-Task einmal initialisiert und wurde er von der Identität der anderen Tasks informiert, so wird einer der PE-Tasks als der Main-PE ernannt. Dieses Main-PE beginnt dann mit der Ausführung des Main-Threads des Haskell-Programms.

Das Programm endet, wenn entweder der Main-Thread abgeschlossen oder wenn ein Fehler aufgetreten ist. In jedem Fall wird eine FINISH-Nachricht an den Manager-Task gesendet, der im Gegenzug eine FINISH-Nachricht an alle PE-Tasks weitergibt. Der Manager-Task wartet auf die Antwort jedes PE-Tasks, bevor er das Programm beendet.

Während der Ausführung vollzieht jeder PE folgende Schleife solange, bis er eine FINISH Nachricht erhält:

Main Scheduler:

1. Führe wenn notwendig eine lokale Garbage Collection aus.
2. Bearbeite jede ankommende Nachricht von anderen PEs, wenn möglich sende Antwortnachrichten.
3. Sind lauffähige Threads vorhanden, so starte einen.
4. Ansonsten schaue nach Arbeit.

2.1 Thread Verwaltung

Ein Thread ist ein virtueller Prozessor. Er wird dargestellt durch ein (dem Speicher zugeteiltem) Thread-Zustandsobjekt (TSO). Ein TSO zeigt auf das (dem Speicher zugeteiltem) Stack-Objekt (SO) des Threads. Wird der Stack des Threads größer, so werden weitere Stack-Objekte allokiert und an die oberen angehängt.

Jedes PE hat ein Pool von lauffähigen Threads - oder besser gesagt TSOs -, die auch sein lauffähiger Pool genannt wird. Dieser wurde bereits in Schritt 3 des Main Scheduler hinzugezogen. Hat ein Thread einmal mit der Ausführung eines PE begonnen, so kann er nicht zu einem anderen PE bewegt werden. Dieser Aspekt zeigt sich empfindlich bei Programmen mit beschränktem Parallelismus, wenn ungünstige Umstände während der Ausführung auftreten, in denen ein PE mit mehreren lauffähigen (jedoch unbeweglichen) Threads zu Ende kommt, während andere jedoch frei sind. Aus dieser Tatsache ergibt sich ein zukünftiger Gedanke, wo lauffähige Threads beweglich sein sollen.

Wurde ein Thread zur Ausführung ausgewählt, dann wird er solange laufen, bis entweder der Platz oder die Threadblöcke (durch einen anderen Thread oder durch Zugriff von entfernten Daten) ausgeschöpft sind oder der Thread beendet wurde.

2.1.1 Funken (Sparks)

Parallelismus kann in einem Haskellprogramm nur explizit mithilfe des *par*-Kombinators dargestellt bzw. ausgeführt werden. Zur Zeit müssen diese Kombinatoren noch durch den Programmierer selbst hinzugefügt werden, jedoch soll diese Aufgabe in naher Zukunft durch den Compiler erfüllt werden. Der *par*-Kombinator implementiert eine Form der parallelen Zusammensetzung. Wird der Ausdruck *x 'par' e* ausgewertet, so hat dies folgende Bedeutung: Das Speicherobjekt, auf das sich die Variable *x* bezieht, wird '*gezündet'* und danach wird *e* ausgewertet. Eine bevorzugte (dies bedeutet jedoch nicht, die einzige) Möglichkeit den par-Kombinator zu benutzen, ist demnach folgende:

let x = f a b in x 'par' e,

wobei e sich auf x bezieht. In diesem Fall wird die *Closure*, der durch den Aufruf f a b dargestellt wird, durch den *let*-Operator zugewiesen und danach durch den *par*-Kombinator gezündet. Diese kann somit parallel mit *e* ausgewertet werden.

Eine Closure zu zünden, ist in der Regel eine relativ kostengünstige Operation, die nur aus dem Hinzufügen eines Zeigers auf die Closure zu dem lauffähigen Pool des PE besteht. Ein Funke ist ein Hinweis darauf, daß eine Closure nützlicherweise parallel ausgeführt werden könnte.

2.1.2 Synchronisation

Es ist offensichtlich wünschenswert, daß zwei Threads nicht die gleiche Closure simultan ausführen, damit die Arbeit nicht verdoppelt werden muß. Diese Synchronisation wird folgendermaßen erreicht:

1. Erhält ein Thread eine Closure (d.h. er startet, diesen auszuwerten), dann überschreibt er die Closure mit einem schwarzen Loch[1] (*black hole)*.
2. Nimmt ein Thread schwarzes Loch auf, so speichert er dessen Status in seinem TSO. Danach hängt er seinen TSO an die Thread-Queue an, die durch die *schwarzen Löcher* gesperrt wurden und gibt den Thread an den Scheduler weiter.
3. Hat ein Thread die Ausführung einer Closure beendet, so überschreibt er den letzteren mit seinem Wert (die Update-Operation). Hat er dieses ausgeführt, so bewegt er jeden TSO in der Queue zu seinem lauffähigen Pool.

Es ist zu bemerken, daß die Kosten für die Synchronisation nur dann anfallen, wenn zwei Threads tatsächlich zusammenstoßen. Dies ist besonders dann der Fall, wenn ein Thread einen Unterausdruck zündet und diesen Unterausdruck selbst auswertet, bevor dieser Funke wieder in einen Thread umgewandelt und ausgeführt worden ist. In diesem Zusammenhang fallen keine Synchronisationskosten an. Tatsächlich wird der vermeintliche Unterthread dynamisch zurück an den Vater-Thread gehängt und der Funke wird somit zu einem Waisen (Orphan), das bedeutet, der Funke gehört zu keinem Thread.

[1]Tatsächlich werden Closures nur dann mit *black hole's* überschrieben, wenn sich der Inhalt eines Treads ändert. Der Vorteil dieser *lazy* Arbeitsweise ist, daß dadurch einige Closures aufgenommen und wieder upgedatet wurden, ohne mit einem *black hole* überschrieben worden zu sein.

2.2 Lastverteilung

Wenn (und wirklich nur wenn) ein PE nichts anderes zu tun hat, so versucht er, einen Funken von seinem lauffähigen Pool zu holen, falls ein solcher vorhanden ist. Der Funke kann ein besitzloser Funke sein, da die Closure, auf die er sich bezieht, bereits ausgewertet ist oder sich unter der Auswertung durch einen anderen Thread befindet. Ist dies der Fall, so legt der PE diesen Funken einfach wieder zurück und versucht sein Glück mit dem nächsten Funken. Der PE führt dies nach dem FIFO (First-In First-Out) - Prinzip durch. Findet ein PE einen sinnvollen Funken, so überträgt er diesen in einen Thread, in dem dem Thread ein neuer TSO zugewiesen wird. Danach beginnt er mit dessen Ausführung.

Sind keine lokalen Funken vorhanden, so sucht der PE nach Arbeit von anderen PEs. Dies geschieht, indem er eine FISH-Nachricht losschickt, die von PE zu PE schwimmt und dort nach vorhandener Arbeit sucht. Da am Anfang ja nur das Main-PE beschäftigt ist - er erhält den ersten lauffähigen Thread -, bedeutet dies, daß alle anderen PEs nach Arbeit suchen, sobald mit der Ausführung begonnen wurde.

Wurde eine FISH-Nachricht erstellt, so wird diese zufällig zu einem anderen PE gesendet. Hat der Empfänger der FISH-Nachricht keine verfügbaren Funken, so erhöht er das ‘Alter’ der FISH-Nachricht und sendet diese an einen anderen PE weiter. Dies geschieht wie zuvor auf zufällige Weise. Das ‘Alter’ einer FISH-Nachricht beschränkt sich auf die Anzahl PEs, die eine Nachricht besucht: Ist das Alter der FISH-Nachricht gleich der Anzahl vorhandener PEs, so schickt der letzte besuchte PE die erfolglose Nachricht zu dem entsendenden PE zurück. Nach Empfang der eigenen, erfolglosen FISH-Nachricht wartet der entsendende PE einige Zeit, bevor er eine andere FISH-Nachricht abschickt. Der Grund für die Verzögerung ist folgender: Man möchte vermeiden, daß die Maschine mit FISH-Nachrichten überschüttet wird, obwohl dort nur einige beschäftige PEs sind. Ein PE darf immer nur eine einzige FISH-Nachricht ausstehen haben.

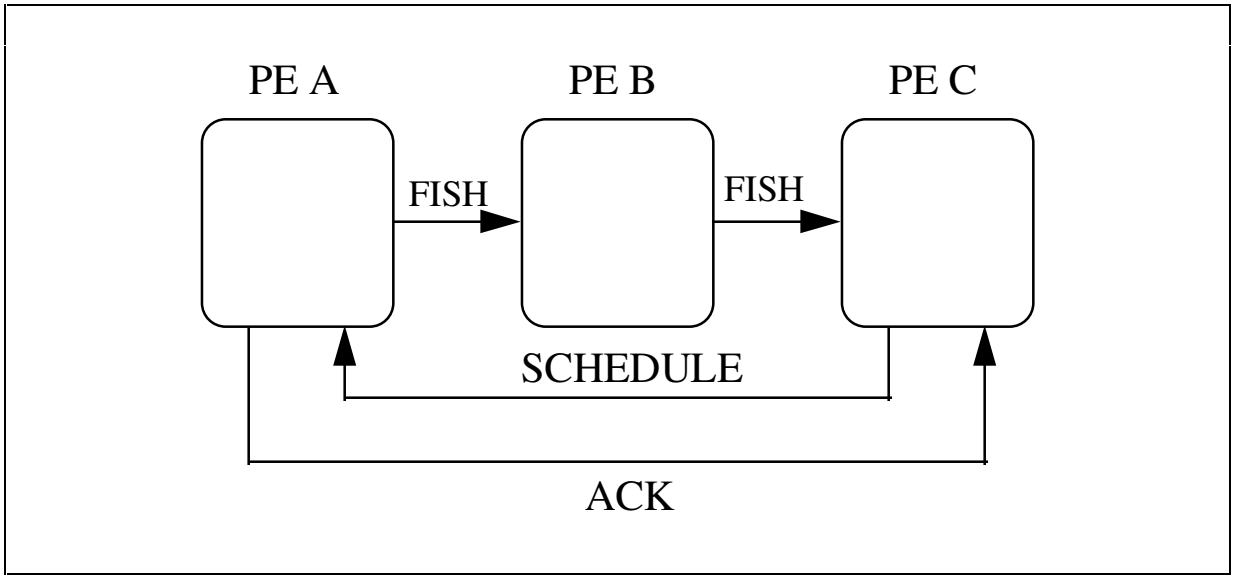

Abbildung 2-1: FISH/SCHEDULE/ACK - Abfolge

Wenn das PE, das eine FISH-Nachricht erhält, einen brauchbaren Funken (d.h. einen Funken, der sich noch nicht unter Auswertung befindet) besitzt, so sendet er eine SCHEDULE-Nachricht an den PE, der die FISH-Nachricht verschickt hat. Die SCHEDULE-

Nachricht enthält die gezündete Closure gepackt mit dem Graph (wie in Kapitel 2.4 beschrieben wird). Der PE, der die Nachricht ausgesendet hat, entpackt den Graph und fügt die neu erhaltene Closure in sein lokales lauffähiges Pool. Danach sendet er eine ACK-Nachricht an das PE, das die Closure verschickt hat, zurück, um den neuen Standort der Closure mitzuteilen. Dabei ist zu bemerken, daß das PE, das die FISH-Nachricht ausgesendet hat, sich möglicherweise nicht mehr im Status bereit befindet, da eine andere ankommende Nachricht einen Thread entblockiert haben kann, bevor die SCHEDULE-Nachricht ankam. Abbildung 2-1 zeigt die Abfolge von Nachrichten, die durch eine FISH-Nachricht ausgelöst wird.

2.3 Speicherverwaltung

GUM besitzt eine wichtige Grundfunktion: die Verwaltung des gemeinsamen virtuellen Speichers.

2.3.1 Lokale Adressen

Da GUM auf dem Glasgow Haskell Compiler aufbaut, wird ein großer Teil der Ausführung in gleicher Weise ausgeführt wie auf einen Uniprozessor. Dies bedeutet im allgemeinen:

- Jeder PE hat seinen eigenen lokalen Heap.
- Neue Objekte des Heaps werden von einem zusammenhängenden freien Speicherblock dem lokalen Heap zugeteilt.
- Die Adressen der Heapobjekte, die durch compilierten Kode verändert werden können, sind einfache Ein-Wort-Zeiger innerhalb des lokalen Heap. Diese nennen wir *lokale Adressen*.
- Jeder PE besitzt die Möglichkeit, lokale Garbage Collection auszuführen und zwar ist dies unabhängig von allen anderen PEs. Diese entscheidende Eigenschaft ermöglicht jedem PE, kostengünstig seinen eigenen ‘Abfall’ zu beseitigen, der bei der normalen Ausführung produziert wurde.

Manchmal muß das System zur Laufzeit ein Speicherobjekt von dem lokalen Speicher eines PE’s zu dem eines anderen bewegen. Dies geschieht zum Beispiel dann, wenn PE C in Abbildung 2-1, der genügend Funken besitzt, eine FISH-Nachricht erhält. Dieser sendet einen seiner gezündeten Closures zu PE A, der der Sender der FISH-Nachricht war. Wird die Closure auf diese Weise bewegt, so wird der ursprüngliche Closure mit einem *FetchMe Objekt* überschrieben, welches die globale Adresse der neuen Kopie von A enthält. Was ist der Grund, die Closure zu überschreiben ? Es würde ein Fehler sein, diesen einfach nur zu kopieren, denn sonst würden sowohl PE A als auch C diesen getrennt ausführen.

2.3.2 Globale Adressen

Zunächst kann man zwar denken, daß eine globale Adresse (GA) aus dem Bezeichner des beteiligten PE bestehen sollte, zusammen mit der lokalen Adresse eines Objekts auf das PE. Solch ein Schema hätte jedoch zur Folge, daß die PEs keine komprimierende Garbage Collection in ihrem lokalen Heap ausführen könnten, da diese die lokale Adresse der meisten

Objekte verändern würde. Da eine Garbage Collection jedoch ein fester Bestandteil der effizienten Compilertechnologie von GUM ist, wird diese Einschränkung verworfen.

Aus diesem Grund erhält jedes global sichtbare Objekt einen festen *lokalen Bezeichner* (dieser ist normalerweise eine natürliche Zahl). Eine globale Adresse besteht aus einem Paar, dieses ist zusammengesetzt aus dem PE-Bezeichner und dem lokalen Bezeichner. Jedes PE besitzt eine Globale Indirektionstabelle (GIT). Diese GIT teilt die lokalen Bezeichner den lokalen Adressen des entsprechenden Speicher-Objekts zu. Die GIT wird als die Ausgangspunkt für die lokale Garbage Collection betrachtet und soll die neuen Lokalisationen der lokalen Heap-Objekte im Anschluß an eine Garbage Collection wiedergeben. Man kann somit sagen, ein PE besitzt ein global sichtbares Objekt (dies bedeutet, es besitzt eine globale Adresse), wenn die globale Adresse des Objekts die des PE-Bezeichner beinhaltet.

Ein Heap-Objekt wird *globalisiert* (das heißt, es erhält eine globale Adresse), in dem ein ungebrauchter lokaler Bezeichner allokiert und die GIT vergrößert wird, um den lokalen Bezeichner zu den Objektadressen neu einzutragen. Natürlich ist es auch möglich, daß das Objekt bereits eine globale Adresse besitzt. In diesem Falle wird die Abbildung von lokalen zu globalen Adressen gewartet (gesondert in jedem PE) und zwar die *LA → GA Tabelle*. Diese wird bei jeder *Globalisierung* überprüft. Natürlich muß die LA → GA Tabelle nach jeder Garbage Collection wieder neu aufgebaut werden, da sich die lokalen Adressen der Objekte geändert haben könnten.

Ein PE sollte auch die Kopien von global sichtbaren Heap-Objekten enthalten, über die andere PE's verfügen. Zum Beispiel: PE A sollte eine Kopie der Liste besitzen, die er von PE B erhalten hat. Angenommen die Wurzel der Liste hat die GA (B,34). Dann sollte PE A wissen, daß die Wurzel seiner Liste von Kopien auch die GA (B,34) hat, falls er diese noch einmal benötigen sollte. Ist dies der Fall, dann kann er einfach die Kopie benutzen, die er schon besitzt, anstatt sich die Liste noch einmal holen zu müssen.

Wir erreichen die gemeinsame Benutzung, indem wir (in jedem PE) die Abbildung von globalen Adressen zu lokalen Adressen, die GA → LA Tabelle warten. Wenn PE A die Liste zum ersten Mal abruft, so gibt er die Abbildung von (B,34) zur abgerufenen Kopie in seine GA → LA Tabelle ein. Wenn er (B,34) wieder benötigt, durchsucht er zuerst die GA → LA Tabelle und sieht, daß diese schon eine lokale Kopie besitzt.

Um dies noch einmal zusammenzufassen: Jeder PE muß die folgenden drei Tabellen warten. In der Praxis sind diese Tabellen in einer einzelnen Datenstruktur vereinigt.

- Seine GIT teilt jeden zugewiesenen lokalen Bezeichner zu seiner lokalen Adresse ein.
- Seine GA → LA Tabelle teilt einige *fremde* globale Adressen (deren PE Bezeichner nicht lokal sind) zu ihren lokalen Gegenstücken ein. Dabei ist zu beachten, daß jede *fremde* GA zu genau einem LA eingeteilt wird.
- Seine LA → GA Tabelle teilt lokale Adressen ihren zugehörigen globalen Adressen zu (falls vorhanden).

2.3.3 Garbage Collection

Das oben beschriebene Konzept hat offensichtlich folgenden Nachteil: Besitzt ein Objekt einmal einen Eintrag in der GIT, so kann es nicht mehr Garbage collectet werden (da die GIT als Ausgangspunkt für die lokale Garbage Collection genutzt wird). Außerdem kann der lokale Bezeichner nicht wiederbenutzt werden. Aus diesem Grund wird in GUM *gewichtete*

Referenzzählung benutzt, um lokale Bezeichner frei zu bekommen und hieraus die Objekte, die sie identifizieren.

Die GIT und die GA → LA Tabelle werden vergrößert, um sowohl das Gewicht als auch die lokale Adresse festzuhalten. Die Invariante ist für eine gegebene globale Adresse G die Summe aus:

- dem Gewicht von G in der GA → LA Tabelle für alle fremden PEs (d.h. die eine Kopie zu einem anderen PE besitzen), und
- dem Gewicht von G in seiner eigenen GIT, und
- dem Gewicht von G, der gerade eine 'fliegende' Nachricht besitzt (d.h. der gerade eine Kopie an einen anderen PE sendet).

Diese Summe ist gleich *MaxWeight*, dies ist eine feste Konstante. Hierbei ist zu bemerken, daß *MaxWeight* immer den gleichen Wert besitzt und sich nur die einzelnen Gewichte verschieben können. Mit dieser Invarianten kann man nun folgende Regeln zur Adressverarbeitung angeben, die unabhängig für jeden PE gelten:

1. Die Einträge in der GIT eines PE's, die das Gewicht MaxWeight besitzen, können Garbage collectet werden und die lokalen Bezeichner können zur Wiederbenutzung frei gegeben werden. (Der Grund hierfür ist die Invariante. Keine anderen PEs oder Nachrichten besitzen eine Referenz auf diese globale Adresse).
2. Ein PE kann wählen, ob seine lokale Adresse in seiner GA → LA Tabelle als Ausgangspunkt für eine lokale Garbage Collection betrachtet wird oder nicht. Wenn genug Platz zur Verfügung ist, so kann er sie als Ausgangspunkt betrachten, da sich lokale Kopien von globalen Objekten in der Zukunft als sinnvoll erweisen können.

   Ist jedoch nicht genügend Platz zur Verfügung, sollte man sie jedoch nicht als Ausgangspunkt betrachten. Ist eine lokale Garbage Collection abgeschlossen, wird die GA → LA Tabelle überprüft. Jeder Eintrag, dessen lokales Objekt (aus irgendeinem Grund) während einer Garbage Collection als lebend identifiziert wurde, muß nun auf den neuen Standort des Objekts zeigen. Die Einträge und die dazugehörigen Gewichte, deren Objekte als tot identifiziert wurden, werden diese an die besitzenden PEs mit einer FREE-Nachricht zurückschicken. Diese wiederum fügen die Gewichtung in der Nachricht zu seinem GIT-Eintrag hinzu.
3. Sendet ein PE eine GA zu einem anderen PE, so wird das Gewicht halbiert, das in der GIT oder in der GA → LA Tabelle (abhängig davon, ob dieser PE der Besitzer der GA ist oder nicht) gehalten wird. Dies bedeutet, sowohl die Nachricht als auch die GA, die in der Tabelle zurückbleibt, erhalten jeweils die Hälfte des Gewichts. Der empfangende PE fügt die Gewichtung in seiner GIT oder seiner GA → LA Tabelle hinzu.
4. Ist das Gewicht in einer GA, das verschickt werden soll, 1, so kann es nicht weiter halbiert werden. In diesem Fall wird eine neue GA allokiert. Diese neue GA zeigt auf die gleiche lokale Adresse in der GIT. Die neuen und alten GAs sind Aliasse für das gleiche Speicherobjekt. Dies hat leider den Nachteil, da eine gemeinsame Benutzung nicht geschützt wird. Um jede spätere Verfrachtung auf eine GA vor der Allokation einer neuen GA zu schützen, wird eine neue GA identifiziert. Diese neue GA besitzt eine Gewichtung, die weitergegeben werden kann, wie die einer bevorzugten GA. Eine Lookup in der LA → GA Tabelle liefert immer die bevorzugte GA.

Der einzige Garbage, der in diesem Konzept nicht berücksichtigt wird, sind Zyklen, die über die PEs verbreitet sind. Lokale Garbage Collection mit gewichteter Referenzzählung scheint demnach den meisten Garbage wiederzufinden.

2.3.4 Verteilte Daten

Globale Referenzen werden durch spezielle *Fetch*M*e* Objekte bearbeitet. Trifft ein Thread auf ein FetchMe Objekt, so werden folgende Schritte durchgeführt:

1. Das FetchMe Objekt erhält eine neue lokale GA. Es besitzt bereits eine fremde GA, nämlich die GA des anderen Objekts. Dieser Schritt entwickelt somit einen temporären Alias für diese.
2. Das FetchMe Objekt wird mit dem *Fetching* Objekt überschrieben.
3. Der fordernde Thread wird blockiert. Dies geschieht durch Anfügen seines TSO an die blockierte Queue, die an das Fetching Objekt angehängt wird.
4. Eine FETCH-Nachricht wird an das PE gesendet, das der Besitzer der fremden GA des FetchMe Objekts ist.
5. Das PE springt zu dem Main-Scheduler zurück. D.h. es werden zunächst andere Threads gestartet oder eine Garbage Collection durchgeführt, während auf die Antwort der FETCH-Nachricht gewartet wird. Jeder nachfolgende Thread, der auch das gleiche fremde Objekt anfordert, wird sich ebenso an die Queue anschließen, welche an das Fetching Objekt angefügt ist.

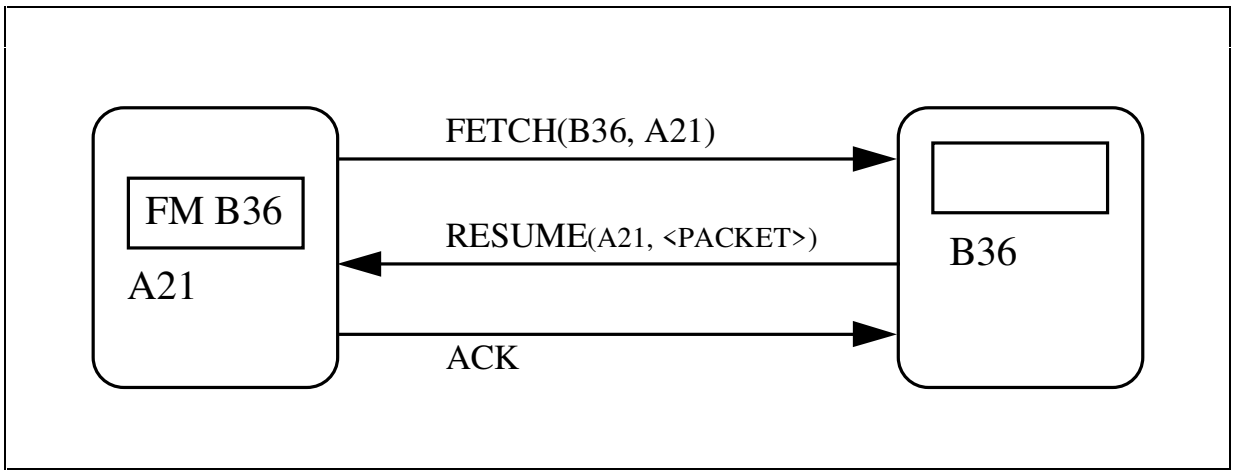

Abbildung 2-2: Fetch/ Resume/ ACK-Abfolge

Nach Empfang einer FETCH-Nachricht wird der Ziel-PE das verwendete Objekt zusammen mit dem Graphen verpacken und diese in einer RESUME-Nachricht an den Urheber schicken. Kommt die RESUME-Nachricht an, entpackt der PE den Graphen und startet den Thread, der durch das Fetching Objekt blockiert war. Danach wird das Fetching Objekt wieder auf den Ausgangspunkt des Graphen zeigen. Wurden diese Schritte abgearbeitet, wird eine ACK-Nachricht an das PE gesendet, das die RESUME-Nachricht verschickt hat (Kapitel 2.4 erklärt den Grund hierfür). Abbildung 2-2 stellt den gesamten Ablauf noch einmal dar.

2.4 Packing/Unpacking Graph

Wird ein Objekt angefordert, so wird auch der dazugehörige Graph mit in das gleiche Paket gepackt. Das Paket beinhaltet das Objekt, das die Anzahl von expliziten FETCH-Nachrichten reduzieren soll, die verschickt werden sollen. Das Ziel hierbei ist, daß die Durchlaufzeit durch Netzwerke mit hohen Wartezeiten ansteigen soll, in dem weniger, größere Nachrichten gesendet werden. Das Packen eines willkürlichen Graphen ist ein nicht triviales Problem. Dieses wird in [1] weiter erläutert.

Packen erfolgt Objekt für Objekt, wobei der Graph nach der Breadth-First-Methode durchzogen wird. Wurde jedes Objekt gepackt, so erhält es wenn nötig eine globale Adresse. Außerdem wird seine Lokalisation in dem Paket in einer Tabelle aufgezeichnet, damit keine gemeinsame Nutzung und keine Kreisläufe vorkommen können. Das Packen wird dann beendet, wenn entweder jeder vorhandene Graph verpackt ist oder das Paket voll ist. Ist das Paket voll, so werden Verweise von gepackten Objekten auf lokale Speicherobjekte als FetchMe Objekte verpackt.

Entpacken vollzieht sich über das gesamte Paket, hierbei wird der Graph in einer Breadth-First Methode wieder aufgebaut. Während jedes Objekt entpackt wird, wird die GA → LA Tabelle auf lokale Kopien des Objekts durchsucht. Sind kein Kopien vorhanden, werden die GA → LA und die LA → GA Tabellen entsprechend upgedatet. Ist jedoch schon eine lokale Kopie des Objekts vorhanden, so muß man aufpassen, damit man das richtige Objekt auswählt. Hierbei wird ein neues Objekt in Normalform einem bereits existierenden FetchMe Objekt vorgezogen. Die Gewicht der neuen GA wird der Gewicht der existierenden Referenz hinzugefügt. Das Duplikat wird mit einer Weiterleitung auf das Objekt überschrieben.

Während Objekte, die Werte in Normalform darstellen, uneingeschränkt kopiert werden, so sichert man ab, daß immer nur eine Kopie einer Closure vorhanden ist, der möglicherweise eine unbegrenzte Menge von Arbeit repräsentieren könnte. Wurde ein Package entpackt, so wird ein ACK-Nachricht versendet, um auf jede neue Lokalisation von jeder Closure in dem Package hinzuweisen.

3 Vorläufige Ergebnisse

Dieser Abschnitt zeigt einige Ergebnisse von Experimenten mit GUM. Diese wurden ausgeführt, um zu zeigen, daß die grundlegenden Mechanismen in GUM korrekt arbeiten. Außerdem wollte man mit diesen Experimenten Auswertungen bzgl. der Leistungsfähigkeit durchführen. Die Ergebnisse zeigen, daß Beschleunigungen unter Benutzung von GUM möglich sind. Sie soll jedoch nicht beweisen, daß reelle Programme durch GUM beschleunigt werden können

3.1 Leistungsfähigkeit bei Single-Prozessoren

Das erste Experiment untersucht die Kosten einer parallelen Auswertung im Vergleich zu einer sequentiellen Auswertung auf einem Single-Prozessor. Da der Zusatzcode auf einem Single-Prozessor sehr klein ist, kann man nicht erwarten, eine gute absolute Beschleunigung zu erhalten, d.h. eine Beschleunigung relativ zu einer guten sequentiellen Implementation. Der Zusatzcode eines Single-Prozessors kann folgendermaßen eingeordnet werden:

- Für jedes Programm existiert ein mehr oder weniger fester Anteil an Zusatzcode unabhängig davon, ob Parallelismus benutzt wird oder nicht. Ein Beispiel für diesen Zusatzcode ist folgendes: GUM muß für jedes neues Objekt testen, ob sich dieses bereits unter Auswertung befindet oder nicht
- Es existiert Zusatzcode, der bei jedem Funken in einem Programm aufgerufen wird.

Wir untersuchen diesen Zusatzcode, indem wir in einem Single-Prozesssor den Divide-and Conquer Faktor benutzen. In der parallelen Form wird der Compiler dazu gezwungen, Code einzufügen, um ein Speicherobjekt für jeden Funken aufzubauen. Wird das Programm jedoch in der üblichen sequentiellen Weise geschrieben, so muß jeder Thread nur wenig Arbeit ausführen, bevor er einen anderen Thread zündet. Der Zusatzcode im Parallelismus ist somit sehr groß.

Der Divide-and-Conquer Faktor, der hier benutzt wird, arbeitet mit einem expliziten Cut-Off Parameter: Ist die Größe des Problems kleiner als der Cut-Off, dann wird das Problem dadurch gelöst, indem reiner sequentieller Code benutzt wird, ansonsten wird paralleler Code benutzt.

Alle Beschleunigungen in dieser Arbeit werden mit einer sequentiellen Version des Programms, das auf einem GHC compiliert wurde, verglichen. Die folgende Tabelle zeigt den parallelen Zusatzcode für den Divide-and-Conquer Faktor auf einem Prozessor von zwei Sparc-based Architekturen. Die Laufzeiten von GUM werden als prozentuale Laufzeiten dargestellt, d.h.- die sequentiellen Laufzeiten geteilt durch die Laufzeit von GUM.

Plattform	**seq. runtime**	**seq-par efficiency**	**par**
Sun 4/15	43.2 s	93 %	92 %
SunMP	35.9 s	90 %	92 %

Die Spalte für ***seq***. ***runtime*** gibt die Gesamtverarbeitungszeit in Sekunden an. Hierbei wird die voll optimierte sequentielle Version des Programms unter dem Laufzeitsystem ausgeführt.

Die Spalte für ***seq-par Efficiency*** zeigt die Leistung, wenn die sequentiellen Version des Programms mit GUM auf einem Single-Prozessor ausgeführt wird. Die steigende Laufzeit zeigt, daß der Zusatzcode, der bei GUM nötig ist, im Bereich von 10% liegt.

Die Spalte für ***par*** zeigt die Leistung, wenn die parallele Version des Programms unter GUM auf einem Single-Prozessor ausgeführt wird.

3.2 Leistungsfähigkeit bei Multiprozessoren

Vergleicht man nun die Leistungsfähigkeit eines Programms, das sowohl auf einem Prozessor als auch auf mehreren Prozessoren ausgeführt wird, so kann man erkennen, daß die Beschleunigung mit der Anzahl an Prozessoren abnimmt. Dies bedeutet, benutzt man mehrere Prozessoren zur Ausführung eines Programms, so erreicht man dadurch nicht unbedingt eine größere Beschleunigung. Zum Beispiel nimmt die Leistung auf einer Sun 4/15 mit 6 Prozessoren von 92 % auf 73 % ab (unter Benutzung eines Prozessors). Dieser Verlust läßt sich dadurch erklären, daß bei mehr Prozessoren eine größere Kommunikation stattfindet und größere Datenmengen vorhanden sind.

4 Fazit

In dieser Arbeit wurde die portable, parallele Implementation von Haskell vorgestellt, die auf der PVM Kommunikationsstruktur aufgebaut ist.

Es ist sehr schwierig, eine effiziente Sprache für alle Architekturen zu entwicklen und es ist nicht einleuchtend, daß ein einfaches Architekturmodell für alle Maschinen entwickelt werden kann. Es ist sicher einfacher und effizienter, ein nachrichtenbasierendes Protokoll auf eine Distributed-Memory Maschine abzubilden.

Die in Kapitel 3 beschriebene Leistungsfähigkeit ist eher aufschlußreich als überzeugend. Diese zeigen, daß die Entwickler von GUM kein paralleles System entwickeln wollten, dessen Leistungsfähigkeit um einen großen Faktor langsamer ist als die Compilierungstechnologie des besten Uniprozessors. Leider wird nichts darüber ausgesagt, wie echte Programme mit brauchbaren Beschleunigungen ausgeführt werden können. In der Tat ist noch einige Arbeit an GUM notwendig, um das bereits existierende System noch um einiges zu verbessern.

5 Literatur

[1] Hammond K, Mattson JS Prtridge AS, Peyton Jones SL and Trinder PW
„GUM: a portable Parallel Implementation of Haskell „

[2] Hammond K, Mattson JS Prtridge AS, Peyton Jones SL and Trinder PW
„GUM: a portable Parallel Implementation of Haskell „
Proc 7th Intl. Workshop on Implementation of Functional Languages, Bastad, Sweden

[3] Simon L. Peyton Jones, David Lester
„Implementing Functional Languages. A Tutorial“
Prentice Hall International Ltd, 1992