Implementierung eines Cluster-Algorithmus für Codebücher auf der MasPar MP-1

Studienarbeit von

Marcus Jürgens

am
Institut für Logik, Komplexität und Deduktionssysteme

25. August 1995

Referent: Prof. Alex H. Waibel
Betreuer: Dipl.-Inform. Tilo Sloboda, Dipl.-Inform. Ivica Rogina

Inhaltsverzeichnis

1 Warum werden Codebücher geclustert? 3

2 Wie wird geclustert? 4
- 2.1 Der Algorithmus von Duda 4
- 2.2 Der Algorithmus von Kai-Fu Lee 5
- 2.3 Darstellung eines Triphons im Rechner 6
- 2.4 Das Abstandsmaß 6

3 Die MasPar 8
- 3.1 Die Hardware der MP-1 8
- 3.2 Das lokale Netz 8
- 3.3 Das globale Netz 8
- 3.4 Das Betriebssystem 9
- 3.5 Die Programmiersprache 9

4 Der Algorithmus auf der MasPar MP-1 10
- 4.1 Der sequentielle Teil 10
- 4.2 Der parallele Teil 11
  - 4.2.1 Datenstruktur auf dem BE 11
- 4.3 Die Distanzmatrix: 13
  - 4.3.1 Die Initialisierung der Distanzmatrix 13
  - 4.3.2 Minimumsuche 14
- 4.4 Der Merge-Schritt 15
- 4.5 Der Improve-Schritt 17
  - 4.5.1 Improve-Schritt I 17
  - 4.5.2 Improve-Schritt II 18
  - 4.5.3 Die Move-Operation 18
- 4.6 Probleme mit arithmetischen Ungenauigkeiten 18
- 4.7 Ausgabe der parallelen Ergebnisse 20
- 4.8 Sequentielle Nachbearbeitung 20

5 Ergebnisse 21

6 Ausblick 22

1 Warum werden Codebücher geclustert?

Ausgangspunkt ist die Verarbeitung gesprochener Sprache. Um gesprochene Sprache zu erkennen, wird ein Sprachmodell aufgestellt. Die kleinsten Informationseinheiten dieses Sprachmodells sind die Phoneme. Ihre Anzahl ist je nach Sprachmodell unterschiedlich. In der deutschen Sprache werden meist ca 50 Phoneme verwendet.

Bei genauerer Untersuchung der Phoneme wurde festgestellt, daß sich gleiche Phoneme z. T. erheblich voneinander unterscheiden, je nachdem welche Phoneme ihnen vorausgehen bzw. welche Phoneme ihnen folgen. Diese benachbarten Phoneme werden auch Kontext eines Phonems genannt. Ein Phonem zusammen mit seinem letzten Vorgänger und mit seinem nächstem Nachfolger zusammen wird auch Triphon genannt.

Da sich die Triphone eines Phonems voneinander unterscheiden, erwartet man eine höhere Erkennungsleistung bei der Verarbeitung von Triphonen zur Erkennung gesprochener Sprache, als bei einer reinen Betrachtung nur von den Phonemen ohne Kontext.

Soll nun jedes mögliche Triphon betrachtet werden, so steigt die Datenmenge stark an. Aus 55 Phonemen können theoretisch 55^3 Phoneme werden. In der Praxis kommen nicht alle Folgen von Phonemen vor. Zum Beispiel kommt in der Praxis nicht dreimal hintereinander das Phonem X vor. Die hier benutzten Beispieldaten hatten ca 7000 Triphone.

Diese Triphone haben sich folgendermaßen auf die Phoneme verteilt:

?	117	A	217	AEH	56	AEHR	9
AH	148	AHR	47	AI	142	AR	30
AU	71	B	185	CH	77	D	181
E	154	E2	187	EH	113	EHR	42
ER	81	ER2	185	EU	42	F	274
G	184	H	113	I	203	IE	147
IHR	48	IR	15	J	74	K	249
L	365	M	286	N	417	NG	53
O	114	OE	26	OEH	37	OH	141
OHR	26	OR	24	P	161	R	288
S	287	SCH	178	T	479	TS	226
TSCH	34	U	131	UE	58	UEH	43
UEHR	9	UH	80	UHR	14	UR	18
V	170	X	57	Z	153		

Verteilung von 7266 Triphonen auf 55 Phoneme

Diese große Anzahl von Triphonen zu verarbeiten hat zwei Nachteile. Erstens ist es wesentlich schwieriger die Modelle zu schätzen, da einige Triphone nur sehr selten in der Datenbasis vorkommen, und so keine zuverlässigen Schätzungen gemacht werden können. Zweitens ist es natürlich auch ein höherer Rechenaufwand, wenn mehr Daten verarbeitet

werden müssen.

Also ist es nun das Ziel, die Anzahl der Triphone wieder zu reduzieren. Allerdings möchte man dabei natürlich so wenig Informationen wie möglich verlieren. Daher geht man so vor, daß die Triphone, die sich kaum voneinander unterscheiden, zusammengefaßt, und die Triphone, die sich stark voneinander unterscheiden, dementsprechend nicht zusammengefaßt werden, sondern einzeln repräsentiert werden.

Eine zusammengefaßte Menge von Triphonen wird auch als Cluster bezeichnet.

Ziel dieser Studienarbeit ist es, einen effizienten Algorithmus auf dem Parallelrechner MasPar MP-1 zu implementieren, der eine möglichst optimale Zuordnung der Triphonen zu Clustern findet.

2 Wie wird geclustert?

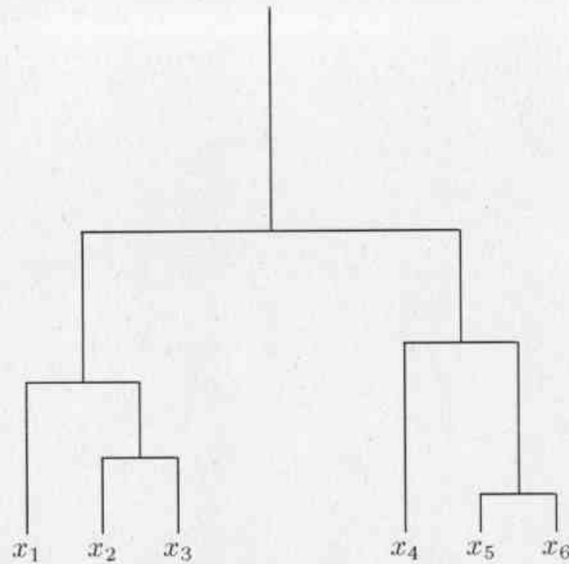
2.1 Der Algorithmus von Duda

Im letzten Kapitel wurde erörtert, warum es sinnvoll ist zu clustern. Nun werde darauf eingegangen, wie geclustert wird. Ziel ist es, die Triphone die sich kaum unterscheiden zusammenzufassen. Dazu gibt es in Duda einen Algorithmus, der dort agglomaratives hierarchisches Clustern genannt wird.

Er sieht folgerndermaßen aus:

1. $c := n$ und $X_i = \{x_i\}$, i=1,...,n
2. Finde die ähnlichsten Cluster X_i und X_j
3. Vereinige X_i und X_j und verringere c um eins [MERGE]
4. Wenn $c < C_{stop}$, Ende, sonst gehe zu Schritt 2

Es werden also die sich immer am nächsten liegenden Cluster zusammengefaßt. Die Ergebnisse dieses Algorithmus lassen sich in sogenannten Dendogrammen gut beschreiben.



Dendogramm

Am unteren Rand sind die initialen Elemente x_1 bis x_6. Die Schnittpunkte stehen jeweils für eine Verschmelzung von zwei Elementen, bzw. Clustern. Je weiter oben eine Verschmelzung stattfindet, umso mehr Informationen gehen verloren.

In diesem Beispiel sind also die Elemente x_5 und x_6 die ähnlichsten Elemente und sie werden als erste miteinander verschmolzen bzw. germergt. Als nächstes werden x_2 und x_3 miteinander vereinigt und das ganze setzt sich dementsprechend fort.

Dieser oben beschriebene Algorithmus ist jedoch leider nicht optimal, d. h. er findet nicht immer die optimalen Verschmelzungen zu Clustern so, daß die Abstände der übriggebliebenen Clustern maximal sind.

2.2 Der Algorithmus von Kai-Fu Lee

Der Algorithmus von Kai-Fu Lee ist wesentlich rechenintensiver, liefert aber bessere Ergebnisse.

Er sieht folgendermaßen aus:

1. $c := n$ und $X_i = \{x_i\}$, i=1,...,n
2. Finde die ähnlichsten Cluster X_i und X_j
3. Vereinige X_i und X_j und verringere c um eins [MERGE]

4. Untersuche: Bringt das Verschieben irgendeines Elementes aus irgendeinem Cluster in irgendeinen anderen Cluster eine Verbesserung? Falls ja, führe die Verschiebeoperation aus. Wiederhole diesen Schritt solange, bis keine Verbesserung mehr möglich ist. [IMPROVE und MOVE]

5. Wenn $c < C_{stop}$, Ende, sonst gehe zu Schritt 2

Er unterscheidet sich vom ersten Ansatz von Duda nur darin, daß der 4. Schritt ergänzt wurde. Es wird zusätzlich überprüft, ob es eine Verbesserung bringt, wenn aus irgendeinem Cluster ein Element herausgenommen und in ein anderes Element geschoben wird. Das ist eine starke Verteuerung des Algorithmus gegenüber der ersten Version, da jetzt sehr viel mehr "ausprobiert" werden muß.

2.3 Darstellung eines Triphons im Rechner

Ein Triphon wird durch eine bestimmte Anzahl von Codebüchern repräsentiert. Die hier benutzten Beispieldaten hatte 3 FFTs und 3 Δ-FFts, also insgesamt 6 Codebücher, die jeweils eine Länge von 50 Einträgen hatten. Also wird jedes Triphon durch 300 Floats dargestellt.

Sei $N_{a,d}(i)$ das Vorkommen des Codewortes i in Verteilung d des Kontextes a eines Monophons. Dann wird ein gesammtes Triphon folgendermaßen dargestellt, wobei ein Triphon immer komplett auf einem Prozessorelement gespeichert wird.

$N_{a,0}(0)$	$N_{a,0}(2)$	$\cdots$	$N_{a,0}(49)$
$N_{a,1}(0)$	$N_{a,1}(2)$	$\cdots$	$N_{a,1}(49)$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
$N_{a,5}(0)$	$N_{a,5}(2)$	$\cdots$	$N_{a,5}(49)$

Ein Triphon auf einem PE

2.4 Das Abstandsmaß

Es stellt sich nun die Frage, wie festgestellt werden kann, ob zwei Triphone ähnlicher sind als andere. Als Abstandsmaß wird die "Verlorene Information" benutzt. Das heißt, der Abstand von zwei Triphonen ist umso größer, je mehr Informationen verlorengehen, falls die Triphone zusammengeclustert werden.

Sei, wie oben, $N_{a,d}(i)$ das Vorkommen des Codewortes i in Verteilung d des Kontextes a eines Monophons. Entsprechendes soll für $N_{b,d}(i)$ gelten. Dann läßt sich weiter berechnen:

Summe der Vorkommen:

$$N_{a,d} = \sum_i N_{a,d}(i)$$

Normalisierung:

$$P_{a,d}(i) = \frac{N_{a,d}(i)}{N_{a,d}}$$

Entropie:

$$H_{a,d} = -\sum_i P_{a,d}(i) log(P_{a,d}(i))$$

"Gemischte" Verteilung:

$$N_{m,d}(i) = N_{a,d}(i) + N_{b,d}(i)$$

"Verlorene" Entropie:

$$L_d(a,b) = N_{m,d} * H_{m,d} - N_{a,d} * H_{a,d} - N_{b,d} * H_{b,d}$$

Über alle Codebücher:

$$L(a,b) = \sum_d L_d(a,b)$$

Da für die Abstandsberechnungen die Summe der Vorkommen eines Codebuchs ($N_{a,d}$) und die Entropie ($H_{a,d}$) immer wieder nötig sind, werden diese Werte am Anfang einmal berechnet und dann immer wieder mitbenutzt. Daher gibt sich die um die Summe und Entropie leicht erweiterte Darstellung eines Triphon im Speicher pro PE:

$N_{a,0}(0)$	$N_{a,0}(2)$	$\cdots$	$N_{a,0}(49)$	$N_{a,0}$	$H_{a,0}$
$N_{a,1}(0)$	$N_{a,1}(2)$	$\cdots$	$N_{a,1}(49)$	$N_{a,1}$	$H_{a,1}$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
$N_{a,5}(0)$	$N_{a,5}(2)$	$\cdots$	$N_{a,5}(49)$	$N_{a,5}$	$H_{a,5}$

3 Die MasPar

Die Maspar MP-1 gehört zu der Klasse der Feldrechner. Sie besteht aus sehr vielen sehr einfachen gleichartigen Verarbeitungselementen, die jeweils die gleiche Operation ausführen. Daher spricht man auch von einer SIMD-Architektur. SIMD steht für Single-Instruction-Multiple-Data.

3.1 Die Hardware der MP-1

Im einzelnen besteht die MasPar aus zwei Komponenten: Einem Vorrechner, dem Front-End (FE), und dem eigentlichen Feldrechner, dem Back-End (BE). Das Back-End zerfällt in zwei Teile: in das Prozessorfeld und die Prozessorfeldsteuereinheit. Die einzelnen Verarbeitungselemente des Prozessorfeldes sind über ein globales und ein schnelles lokales Netz miteinander verbunden.

Der Vorrechner ist eine DEC 5000 Workstation, die eine graphische Oberfläche und einen Netzwerkanschluß zur Verfügung stellt. Auf diesem Vorrechner sind auch die MasPar Werkzeuge, wie zum Beispiel Compiler und Hilfsprogramme installiert.

Die Prozessorfeldsteuereinheit ist ein vollständiger skalarer Prozessor mit einem Risc-ähnlichen Befehlssatz und 32 Bit breiten Zweiadressbefehlen. Skalare Operationen werden auf der Prozessorfeldsteuereinheit direkt ausgeführt. Die vektorwertigen Funktionen werden von der Prozessorfeldsteuereinheit als Mikroinstruktionen an die Prozessorfeldelemente geschickt. Ausschließlich die Prozessorfeldsteuereinheit kommuniziert mit dem Front-End.

Das Prozessorfeld besteht aus $2^{14} = 16384$ Prozessorelementen (PEs), die zu einem quadratischem Feld von 128x128 (PEs) angeordnet sind. Durch die Einfachheit der einzelnen PEs ist es möglich, das Prozessorfeld preisgünstig zu bauen. Bei der MasPar MP-1 handelt es sich um einfache 4-Bit Prozessoren. Eine 32-Bit-Addition muß deshalb in mehrere 4-Bit-Additionen umgewandelt werden.

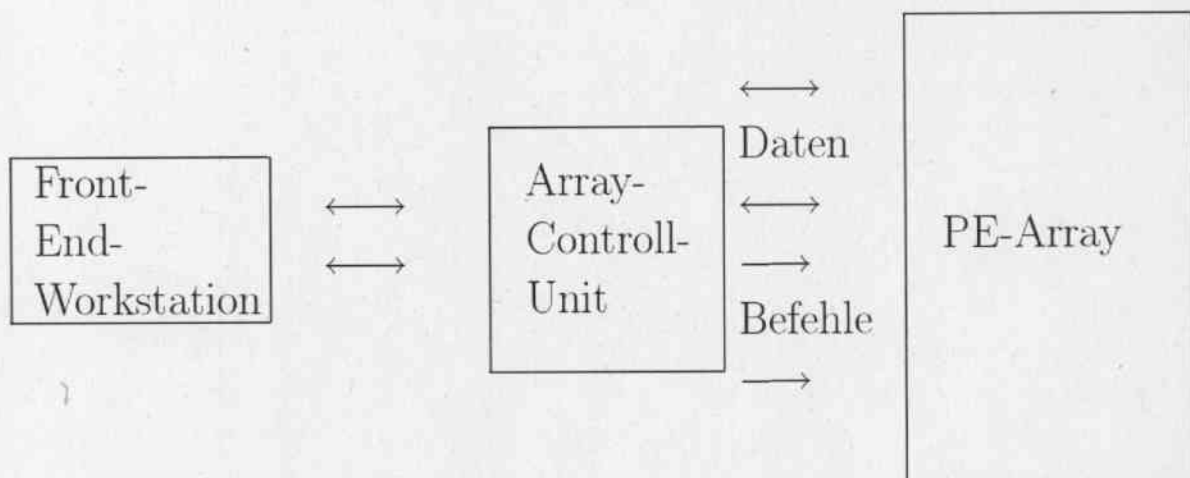
3.2 Das lokale Netz

Das lokale Netz verbindet die 8 benachbarten Prozessorfeldelemente. Die Verbindungen laufen waagerecht, senkrecht und diagonal. Jeweils eine Verbindungsleitung ist nur ein Bit breit. Das lokale Netz ist nicht konfigurierbar. Die Gitterstruktur der MP-1 ist ein zweidimensionaler Torus, in dem die Prozessorfeldelemente zu 128x128 Elementen angeordnet sind.

3.3 Das globale Netz

Das globale Verbindungsnetz, oder auch "global router" genannt, dient zur wahlfreien Verbindung der Prozessorfeldelemente untereinander. Es ist von der Architektur her als universelles Permutationsnetzwerk ausgelegt. Jeweils 16 PEs sind zu einem Cluster zusammengefaßt. Jeder Cluster verfügt über eine Verbindung in und eine Verbindung aus dem

3-stufigen Clos-Netzwerk. Wollen zum gleichen Zeitpunkt mehrere PEs eines Clusters kommunizieren, so sind entsprechend mehrere Schritte nötig. Die Daten werden hier ebenfalls wie beim lokalen Netz über 1-Bit breite Leitungen ausgetauscht.



3.4 Das Betriebssystem

Das Front-End läuft unter ganz normalem Unix. Das Back-End ist sogar in der Lage mehrere Programme gleichzeitig im Time-Sharing-Betrieb laufen zu lassen. Der 16 KB große Speicher der Prozessorelemente wird in 4 KB große Teile aufgespalten. Wenn nun z. B. ein Programm nur eines dieser Teile benötigt und ein anderes Programm die anderen 12 KB Speicher braucht, so können diese Programme gleichzeitig im Speicher bleiben und die Prozessorfeldsteuereinheit kann mal an dem einen und mal an dem anderem Programm ohne swappen weiterarbeiten. Wenn allerdings alle aktiven Programme mehr als die 16 KB Speicher brauchen. So müssen die Programme ein- bzw. ausgeswappt werden. Das dauert wegen der großen Datenmengen leider ziemlich lange, kommt in der Praxis aber recht häufig vor.

3.5 Die Programmiersprache

Als Programmiersprache stand mpl-cc zur Verfügung. (mpl steht für Mas-Par-Language.) Das ist eine Erweiterung von C um einige für die Parallelprogrammierung nötigen Konstrukte. Die wichtigste Erweiterung ist sicherlich das Anlegen von Variablen auf allen Prozessorelementen.

Soll z.B. eine Integervariable auf allen Prozessorelementen angelegt werden, so geschieht das durch den Zusatz: `plural`

Beispiel:

Mit

```
plural int a;
```

wird eine Integer Variable a auf jedem Prozessorelement angelegt. Also produziert die obige Anweisung 16384 Integervariablen.

Das eigentliche Problem bei der Implementierung von Algorithmen auf Rechner dieser Architektur ist, daß man bestrebt sein muß, möglichst viele PEs gleichzeitig auszulasten, sie also an den verschiedenen Aufgaben mit unterschiedlichen Daten mit gleichen Befehlen (da nur 1 Befehlssstrom) arbeiten zu lassen. Zusätzlich wird die Sache erschwert, daß (wie meistens bei Computern) die Kommunikation verhältnismäßig teuer ist und daher nur auf das unbedingt notwendige Maß reduziert werden sollte.

4 Der Algorithmus auf der MasPar MP-1

Das Programm auf der MasPar gliedert sich entsprechend der Architektur in zwei Teile. Der eine Teil läuft sequentiel auf dem Front-End und der parallele Teil auf dem Back-End.

4.1 Der sequentielle Teil

Das Front-End übernimmt die Vorverarbeitung der Daten und startet das Programm auf dem Back-End.

Die Vorverarbeitung der Daten gliedert sich in die folgenden Abschnitte:

- Einlesen der Daten im Janus-Format. Dazu gab es bereits zwei bestehende Routinen `ds_read` und `ds_load_file`. Probleme gab es nur sie so einzubinden, daß auch der Compiler auf der MasPar alles "verstanden" hat.

- Sortieren der Verteilungen nach ihrem Namen. Alle Verteilungen haben einen Namen, der beispielsweise so aussieht: `A(D,U)m-b-0`. Das `A` bedeutet, daß es ein Phonem A ist. In der Klammer steht der Kontext. Es ist also ein A, das zwischen einem `D` und einem `U` steht. Eine alphabetische Sortierung der Namen garantiert also, daß auf die Triphone, die zusammengehören auch nacheinander zugegriffen werden kann.

- Filtern der wichtigen Informationen. Einige Triphone brauchen nicht geclustert zu werden, weil es nur ein Triphon pro Phonem gibt. Diese Triphone brauchen selbst verständlich nicht auf das Back-End kopiert zu werden.

- Transformation der zusammengehörigen Daten in benachbarte Speicherbereiche. Da die Kommunikation mit dem Back-End am schnellsten durchzuführen ist, wenn große Speicherbereiche kopiert werden, werden jeweils die Triphone eines Phonems hintereinander in einen Speicherbereich auf dem Front-End geschrieben. Das Back-End braucht dann nur die Information wo die Daten stehen und wieviele es sind und kann sie dann schnell auf die Prozessorelemente kopieren.

- Übergabe der Parameter und Start des Programms auf dem Back-End.

4.2 Der parallele Teil

Die eigentliche Ausführung des Programms geschieht auf dem Back-End. Hier wird der parallele Teil ausgeführt.

4.2.1 Datenstruktur auf dem BE

Auf jedem Prozessor-Element (PE) sind ein Triphon und sein zugehöriger Cluster gespeichert. Der zugehörige Cluster ist derjenige Cluster, zu dem das Element im Moment gehört. Da am Anfang $c := n$ und $X_i = \{x_i\}$, i=1,...,n gilt, stimmen das Element und der Cluster überein.

Zusätzlich werden noch für Zwischenberechnungen 3 Speicherbereiche für Cluster gebraucht.

Für die "Buchhaltung", welche Aktionen ausgeführt werden können und tatsächlich ausgeführt wurden, werden noch folgende Informationen gebraucht:

- MNr: Nummer des Phonems (konstant)
- CNr: Nummer des Clusters
- ENr: Nummer des Triphons (konstant, im folgenden wird auch von Element geredet)
- EAnz: Anzahl der Elemente pro Cluster (Am Anfang 1)
- MC: Anzahl der Triphone pro Phonem
- MC2: Hälfte von MC, Größe der Distanzmatrix pro PE
- PIP: primus inter pares, jeweils bei genau einem Prozessorelement, aller Prozessorelemente, die Triphone speichern, die zu einem Cluster gehören ist PIP == 1, bei allen anderen PEs dieses Clusters ist $PIP == 0$.

$E = x_i$ Initialer Cluster		
$C = \{x_i\}$ gemergter Cluster		
Hilfsspeicher 1		
Hilfsspeicher 2		
Hilfsspeicher 3		
Phonem-Nr	Cluster-Nr	Element-Nr
EAnz	MC/MC2	PIP

Datenstruktur auf einem PE

Am Programmstart müssen die Triphone in einer bestimmten Ordnung auf das Prozessorfeld verteilt werden: Das Programm fängt von oben an, das Prozessorfeld zu füllen. Pro Phonem wird jeweils eine neue Zeile genommen. Reicht eine Zeile alleine nicht aus, so werden die nächsten Zeilen dazugenommen. Das wiederholt sich solange, bis das Phonem komplett mit allen Triphonen gespeichert ist. Beim nächsten Phonem wird genauso verfahren, bis keine Phoneme mehr da sind, oder bis das Prozessorfeld voll ist. Falls das Prozessorfeld zu klein ist um alle Triphone auf einmal aufzunehmen, muß das Prozessorfeld mehrmals entsprechend gefüllt und der Algorithmus auf dem Back-End mehrmals ausgeführt werden.

Im allgemeinen kann man nicht davon ausgehen, daß die Anzahl der Triphone pro Phonem genau durch 128 teilbar ist, also werden die Zeilen nicht alle ganz gefüllt. Das führt zu einem sogenannten Verschnitt, da das Prozessorfeld nicht optimal ausgelastet werden kann. Einige PEs werden während der ganzen Berechnung also nie benutzt. Es ist oft aber vorteilhafter diesen Verschnitt zu tolerieren um dafür das Programm einfacher und durch weniger Fallunterscheidungen schneller zu machen.

	Spalte 0	Spalte 1	Spalte 2	Spalte 3	Spalte 4	Spalte 5	Spalte 6	Spalte 7
Zeile 0	A	A	A	A	A	A	A	A
Zeile 1	A	A	A	A	A	A	A	A
Zeile 2	A	A	A	A	A	A	A	A
Zeile 3	A	A	A					
Zeile 4	AEH	AEH	AEH					
Zeile 5	AU	AU	AU	AU	AU	AU	AU	AU
Zeile 6	AU	AU	AU	AU	AU	AU	AU	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Aufteilung der Triphone eines Phonems auf das quadratische Prozessorfeld

4.3 Die Distanzmatrix:

Um nicht immer alle Distanzen neu berechnen zu müssen, werden die Distanzen zu den anderen Clustern in einer Distanzmatrix abgespeichert. Normalerweise ist eine Distanzmatrix ein Array mit n x n Einträgen.

Es sind aber von den n^2 Angaben nur die Hälfte, oder genauer $\frac{n(n-1)}{2}$, nötig. Um auf diese Werte schnell zugreifen zu können, werden sie verteilt auf verschiedenen PEs gespeichert. Jedes PE speichert die Distanzen zu seinen nächsten $\frac{MC}{2}$ Clustern.

Daraus ergibt sich das folgende Aussehen der verteilten Distanzmatrix. Die Distanzmatrix ist jeweils auf die PEs eines Phonems verteilt. Die jeweilige Anzahl der Triphone pro Phonem wird mit n bezeichnet.

PE	Abstand zu nächstem Triphon				
0	1	2	3	$\cdots$	$\frac{n}{2}$
1	2	3	4	$\cdots$	$\frac{n}{2}+1$
2	3	4	5	$\cdots$	$\frac{n}{2}+2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$\frac{n}{2}$	$\frac{n}{2}+1$	$\frac{n}{2}+2$	$\frac{n}{2}+3$	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
n-1	0	1	2	$\cdots$	$\frac{n}{2}-1$

Die Distanzmatrix eines Phonems auf mehreren PEs

4.3.1 Die Initialisierung der Distanzmatrix

Die gesamte Distanzmatrix muß auch geeignet initialisiert werden. Am Anfang müssen also jeweils die Abstände von jedem Triphon, das auf einem PE gespeichert ist zu den nächsten MC2 Triphonen berechnet werden.

Dazu gibt jedes PE sein Triphon an seinen linken Nachbarn weiter. Der berechnet dann den Abstand zu seinem eigenen Triphon und trägt den Abstand in die Distanzmatrix in das erste Feld ein.

Danach gibt es das Triphon an den nächsten Nachbarn weiter und der berechnet den Abstand zu seinem eigenem Triphon. Dieser Abstand wird dann in dem zweiten Feld eingetragen.

Diese Verfahren wiederholen sich genau $\frac{MC}{2}$ mal. Danach ist die komplette Distanzmatrix entsprechend initialisiert.

Beispiel mit 6 Triphonen und 3 Verschiebungen nach links.

Nach jeder Verschiebung wird jeweils der Abstand ausgerechnet und in der Distanzma-

trix abgelegt.

Anfangszustand:

Triphon	0	1	2	3	4	5
Hilfsspeicher	0	1	2	3	4	5

Verschiebung nach links und dann Abstand ausrechnen:

Triphon	0	1	2	3	4	5
Hilfsspeicher	1	2	3	4	5	0

Verschiebung nach links und dann Abstand ausrechnen:

Triphon	0	1	2	3	4	5
Hilfsspeicher	2	3	4	5	0	6

Verschiebung nach links und dann Abstand ausrechnen:

Triphon	0	1	2	3	4	5
Hilfsspeicher	3	4	5	0	1	2

Man sieht also, daß nach $\frac{MC}{2}$ Durchläufen alle Abstände ausgerechnet worden sind.

4.3.2 Minimumsuche

Da die Distanzmarix auf verschiedenen PEs verteilt ist, kann parallel gesucht werden. Der Suchevorgang gliedert sich in zwei Phasen:

Erste Phase: Lokale Minimumsuche: jedes PE sucht parallel nach seinem Minimum in dem Teil der Distanzmatrix, die es gespeichert hat. Hier wird ganz einfach sequentiell die Distanzmatrix durchsucht.

Zweite Phase: Globale Minumsuche: aus den lokalen Minima wird mit dem "Präfix-Sum" Algorithmus das globale Minimum gesucht. Dieses Verfahren ist auch unter dem Begriff der Binären Suche bekannt. Am Anfang vergleichen alle Prozessoren mit einer geraden Prozessornummer ihr lokales Minimum mit dem lokalem Minimum ihres rechten Nachbarn. Falls ihr eigenes Minimum kleiner ist als das des rechten Nachbarn behalten sie es. Sonst übernehmen sie den Wert des Nachbarn. Nach diesem Schritt ist das absolute Minimum auf jeden Fall in einem der Prozessoren mit einer geraden Prozessornummer. In der nächsten Iteration "spielen" nur die Prozessoren mit Nummer mit, die durch 4 teilbar sind. Es wiederholt sich alles, nur das sie ihren Wert nicht mit dem rechten Nachbarn, sondern mit dem Prozessor, der 2 weiter nach rechts liegt, vergleichen. Danach wiederholt sich das gleiche mit den Prozessoren, die eine durch 8 teilbare Prozessornummer haben und

sie schauen 4 Prozessoren weiter nach rechts. Dieses Verfahren läßt sich beliebig fortsetzen.

Beispiel:

Prozessor-Nr	0	1	2	3	4	5	6	7	8	9	10	11
Anfangswerte, lokales Minimum	5	4	2	5	2	1	3	8	10	4	3	4
1. Schritt	4	-	2	-	1	-	3	-	4	-	3	-
2. Schritt	2	-	-	-	1	-	-	-	3	-	-	-
3. Schritt	1	-	-	-	-	-	-	-	3	-	-	-
4. Schritt	1	-	-	-	-	-	-	-	-	-	-	-

Das "-" bedeutet, daß der Wert nicht mehr gebraucht wird und weiter berücksichtigt wird.

Man sieht, daß pro Schritt die Anzahl der Prozessoren, unter denen man das Minimum findet, verdoppelt wird. Für die Zeitkomplexität des Algorithmus in der zweiten Phase bedeutet das: er findet in $O(\log n)$ das Minimum n Zahlen.

4.4 Der Merge-Schritt

Wenn das Minimum zwischen zwei Clustern innerhalb eines Phonems gefunden worden ist, werden diese Cluster zusammengefaßt bzw. zusammengemergt

Das Mergen funktioniert, indem die Verteilungen komponentenweise aufaddiert werden. Nehmen wir an, das Cluster C_1 und Cluster C_2 gemergt werden sollen. Dazu schicken die PEs, die Cluster C_1 beherbergen, ihre Daten an die PEs die auf denen Cluster C_2 gespeichert wird, und die Cluster, die C_2 beherbergen an die PEs auf denen Cluster C_1 liegt. Das geschieht gleichzeitig über das globale Kommunikationsnetzwerk.

Dann addieren alle PEs, die gerade einen Cluster empfangen haben ihn zu dem anderen Cluster hinzu. Jetzt stimmt allerdings die Entropie nicht mehr und sie muß nach den oben angegebenen Formeln neu berechnet werden. Die Summe bräuchte eigentlich nicht neu berechnet zu werden. Bei Testberechnungen wurde festgestelt, daß bei Rechnungen auf der MasPar arithmetische Ungenauigkeiten auftreten, die die numerischen Ergebnisse stark beeinflussen. Daher wird sicherheitshalber auch die Summe neu aktualisiert.

Es müssen jetzt auch noch die oben angegebenen Variablen aktualisiert werden. Das sind im wesentlichen die Anzahl der Triphone pro Cluster und die Cluster-Nummer (CNr). Alle an dem Merge-Schritt beteiligten Cluster bekommen jetzt eine einheitliche Nummer. Es ist die kleinere der beiden bisherigen Cluster-Nummern C_1 und C_2. Die Variable PIP muß immer bei genau einem PE, der die Triphone eines Clusters speichert gleich 1 sein. Bei einem Merge wird also jedes mal ein PIP von 1 auf 0 geschaltet.

Beispiel:

Startzustand

ENr	0	1	2	3	4	5
CNr	0	1	2	3	4	5
EAnz	1	1	1	1	1	1
PIP	1	1	1	1	1	1

MERGE Element 2 mit Element 4

ENr	0	1	2	3	4	5
CNr	0	1	2	3	2	5
EAnz	1	1	2	1	2	1
PIP	1	1	1	1	0	1

MERGE Element 3 mit Element 5

ENr	0	1	2	3	4	5
CNr	0	1	2	3	2	3
EAnz	1	1	2	2	2	2
PIP	1	1	1	1	0	0

MERGE Element 1 mit Element 2

ENr	0	1	2	3	4	5
CNr	0	1	1	3	1	3
EAnz	1	3	3	2	3	2
PIP	1	1	0	1	0	0

MERGE Element 0 mit Element 1

ENr	0	1	2	3	4	5
CNr	0	0	0	3	0	3
EAnz	4	4	4	2	4	2
PIP	1	0	0	1	0	0

MERGE Element 0 mit Element 3

ENr	0	1	2	3	4	5
CNr	0	0	0	0	0	0
EAnz	6	6	6	6	6	6
PIP	1	0	0	0	0	0

Anhand dieses Beispiels kann verfolgt werden, wie die Variabeln für die "Buchhaltung" CNr, EAnz und PIP nach den Mergeoperationen aktualisiert werden. Am Anfang ist

die Anzahl EAnz pro Cluster überall 1. Nach 5 Merge-Operationen sind alle Elemente zusammengemergt und es gibt nur noch einen Cluster, dessen Anzahl an Elementen 6 ist.

Für das weitere korrekte Voranschreiten des Programms ist es nötig, daß die Distanzmatrix aktualisiert wird. Das geschieht so, daß alle PEs, den neuen Cluster erhalten und sie rechnen gleichzeitig den Abstand von ihrem eigenen Cluster zu dem neuen Cluster aus. Anschließend speichern sie den berechneten Abstand in der Distanzmatrix wieder ab. Dabei kann es vorkommen, daß der berechnete Abstand auf einem anderem Prozessorelement abgespeichert werden muß, als das, das ihn ausgerechnet hat. Im obigen Beispiel wäre das etwa für die PEs 1, 2, 3 der Fall wenn alle den Abstand zu 0 ausgerechnet haben. Da die Abstände von 0 zu 1, 0 zu 2 und von 0 zu 3 auf dem PE 0 abgespeichert werden müssen, schicken die PEs ihre Ergebnisse an das Prozessorelement 0.

4.5 Der Improve-Schritt

Es ist jetzt zu untersuchen, ob durch das Verschieben eines Elementes aus irgendeinem Cluster in irgendeinen anderen Cluster eine Verbesserung erreicht wird. Da nur ein Cluster verändert worden ist, nämlich der durch den Merge-Schritt gerade entstandene, braucht auch nur dieser im weiteren untersucht zu werden.

Also muß nun geprüft werden, ob es eine Verbesserung bringt, wenn man in den neuen Cluster ein Element hinzufügt oder eins herausnimmt. Daher wird die Aufgabe in zwei Teile aufgeteilt.

4.5.1 Improve-Schritt I

Hier wird überprüft, ob es eine Verbesserung bringt, wenn noch ein zusätzliches Element in den neuen, den durch den Merge-Schritt entstandenen Cluster, aufgenommen wird.

Das Problem wird so gelöst, daß der neue Cluster jeweils auf alle anderen PEs verteilt wird, die ein Element speichern, das zu einem Cluster mit mindestens zwei Elementen gehört. Nur aus einem Cluster mit mehreren Elementen kann auch eine Element herrausgenommen werden.

Alle Prozessoren, die den neuen Cluster bekommen haben, führen jetzt probehalber eine Verschiebung aus, ob es eine Verbesserung bringt, wenn sie ihr eigenes Element aus ihrem Cluster herausnehmen und in den neuen Cluster verschieben.

Falls das bei einem oder mehreren Elementen der Fall ist, wird das Element mit der größten Verbesserung genommen, es wird entsprechend verschoben.

Das Suchen nach der größten Verbesserung funktioniert wie in der zweiten Phase der Minimumsuche durch das binäre Suchen diesmal nach dem größten Element.

4.5.2 Improve-Schritt II

Hier wird der umgekehrte Fall überprüft. Bringt es eine Verbesserung, wenn aus dem neuen Cluster ein Element rausgenommen wird und es in einen anderen Cluster geschoben

wird.

Das wurde so programmiert, daß der Reihe nach jedes Triphon aus dem neuen Cluster auf jeweils ein PE eines Cluster kopiert wird. Dort wird nun überprüft, ob es eine Verbesserung bringt, wenn das jeweilige Triphon aus dem neuen Cluster in den aktuellen Cluster jedes PEs verschoben wird. Auch hier gilt wie oben, daß im Falle von mehreren Möglichkeiten die mit der größten Verbesserung gewählt wird.

Beide Improve Schritte werde solange ausgeführt, bis keiner von ihnen mehr eine Verbesserung bringt.

4.5.3 Die Move-Operation

Bei einer Verschiebe- oder Move-Operation muß ein bestimmtes Element aus einem Cluster rausgenommen werden und in einen anderen Cluster geschoben werden. Dabei sind noch eine Reihe von anderen Variablen zur Buchhaltung zu aktualisieren. So muß die Anzahl der Elemente pro Cluster (EAnz) des Clusters von dem das Element kommt, auf den das Element geschoben wird und des Elementes selbst aktualisiert werden. Außerdem ist die Distanzmatrix wieder bei allen geänderten Clustern zu aktualisieren.

Beispiel:

ENr	0	1	2	3	4	5
CNr	0	1	1	3	1	3
EAnz	1	3	3	2	3	2
PIP	1	1	0	1	0	0

MOVE Element 2 von Cluster 1 nach Cluster 3

ENr	0	1	2	3	4	5
CNr	0	1	3	3	1	3
EAnz	1	2	3	3	2	3
PIP	1	1	0	1	0	0

4.6 Probleme mit arithmetischen Ungenauigkeiten

Die Formeln zur Berechnung des Abstandes zwischen zwei Clustern sind sehr rechenenintensiv. Pro Berechnung eines einzigen Abstandes gehen in meinem Beispiel unter anderem 900 Logarithmen-Berechnungen und 900 Floating-Point Divisionen ein. Das führt leider dazu, daß die berechneten Ergebnisse nicht immer mit den tatsächlichen Ergebnissen übereinstimmen. So kommt es öfter vor, daß als Abstand zwischen zwei Clustern, die sehr ähnlich waren, negative Werte berechnet worden sind. Aus der mathematischen Definition eines Abstandes hätte das gar nicht passieren dürfen.

Dieser Effekt führte in der Ausführung des Programmes zu Problemen, da es vorkam,

daß ein Improve-Step mit Verschiebung eines Elementes von einem Cluster in einen anderen Cluster durchgeführt worden ist, weil es eine Verbesserung brachte. Direkt danach wurde aber ausgerechnet, daß die Verschiebung des gleichen Elementes wieder zurück aus dem Cluster in den es gerade geschoben wurde in den Cluster aus dem es gerade gekommen ist auch eine Verbesserung bringt. Das führte leider zu zyklischen Verschiebungen und das Programm "hing" in einer Endlosschleife.

```
MOVE 2 von 1 nach 3
MOVE 2 von 3 nach 1
MOVE 2 von 1 nach 3
MOVE 2 von 3 nach 1
MOVE 2 von 1 nach 3
MOVE 2 von 3 nach 1
⋮
```

Die Lösung dieses Problem liegt in der Einführung eines Schwellwertes. Es wird ein gewisser Wert, der Schwellwert, festgelegt. Wenn eine Verbesserung ausgerechnet wird, wird überprüft, ob diese Verbesserung größer ist als der Schwellwert. Falls nicht, wird diese Verbesserung ignoriert und der entsprechende Move-Befehl nicht ausgeführt. Falls die Verbesserung größer ist, scheint es sich um eine echte Verbesserung zu handeln und der zugehörige Move-Befehl wird nicht ausgeführt.

Bei der Wahl des Schwellwertes muß man allerdings sehr vorsichtig sein:

- Wird der Schwellwert zu niedrig gewählt, so kann es trotzdem noch zu zyklischen Verschiebungen kommen und der Schwellwert hat seine Wirkung vollkommen verfehlt.

- Wird der Schwellwert zu hoch gewählt, so werden zu wenige, oder im Extremfall gar keine Move-Operationen ausgeführt, da die berechnete Verbesserung zu oft (oder immer) unter dem angegebenen Schwellwert liegt.

Phonem	# Triphone	100	200	500	1000	2000	5000
A	217	Zyklus	Zyklus	113	103	71	44
AEH	56	42	36	34	30	24	12
H	113	Zyklus	77	70	55	40	30
T	479	739	453	336	283	191	93

Man sieht leicht, wie auch zu erwarten war, daß mit sinkendem Schwellwert immer mehr Improve-Schritte als vernünftig angesehen werden und damit die Anzahl der Moves steigt. Allerdings besteht dann auch höhere Gefahr, daß der Algorithmus in einen Zyklus gerät.

Es ist nicht möglich die Genauigkeit der Zahlendarstellung zu erhöhen, da sich sonst auch der Speicherplatzbedarf annähernd verdoppelt, aber der Speicherplatz pro PE auf 16 KB beschränkt ist.

4.7 Ausgabe der parallelen Ergebnisse

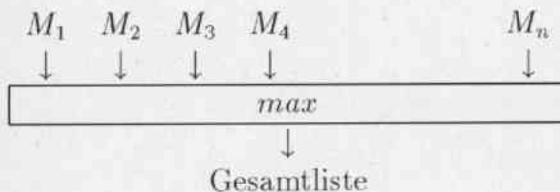
Immer wenn ein PE einen Move oder ein Merge ausführt, muß das mitgespeichert werden. Das geschieht durch ein "paralleles" schreiben der PEs auf in eine sequentielle Datei. Jedes PE speichert einen Datensatz ab. Dieser beinhaltet, welche Aktion ausgeführt wird (Move oder Merge), welches Phonem betroffen ist, und welche Cluster zusammengeclustert werden bzw. welches Triphon von wo nach wo geschoben wird. Zusätzlich wird noch abgespeichert, wieviel Informationen verloren gingen oder Informationen gewonnen wurden. Es ist klar, daß bei einem Merge nur Informationen verloren gehen können und ein Move nur sinnvoll ist, wenn Informationen gewonnen wurden.

Alle diese Aktionen liegen also am Ende eines Programmdurchlaufs in einer Datei. Für ein Phonem mit n Triphonen wird genau n-1 mal geclustert. Das bedeutet, daß die Hauptschleife genausooft durchlaufen wird, wieviele Triphone das größte Phonem hat. Die PEs, welche Triphone aus Phonemen mit weniger Triphonen haben, schalten sich ab, sobald alles auf einen Cluster heruntergeclustert ist. Das bedeutet leider, daß die Auslastung der Maschine mit zunehmender Anzahl von Iterationen sinkt. Das ist aber bei Algorithmen auf SIMD-Rechnern oft nicht zu vermeiden.

4.8 Sequentielle Nachbearbeitung

Es müssen generell alle Phoneme bis auf ein Cluster heruntergeclustert werden. Das ergibt sich daraus, daß erst ganz zum Schluß, wenn alles geclustert ist, entschieden werden kann, in welcher Reihenfolge die Operationen wirklich ausgeführt werden sollen.

Dazu wird aus den verschiedenen Einzel-Listen pro Phonem eine lange Gesamtliste erstellt. Es wird das maximale erste Element aller Listen gesucht. Das bildet das erste Element der Gesamtliste. Das Element wird aus der Einzelliste gelöscht und das nächst größere Element wird gesucht. Das wiederholt man solange, bis alle Listen leer sind.



Um jetzt n Triphone aus mehreren Phonemen bis auf m Cluster herunterzuclustern, geht man in der Gesamtliste einfach alle Schritte bis vor das n-m-1. Merge. Alle bis dahin in der Liste stehenden Aktionen merkt man sich. So hat man dann die genaue Zuordnung,

welche Triphone man jetzt zusammenfassen kann, wobei möglichst wenig Entropie verloren geht.

Jetzt werden die Verteilungen im Janus-Format geladen und geprüft, welche Verteilungen zusammengelegt werden müssen. Diese Verteilungen werden entsprechend aufaddiert und dann im Janus-Format abgespeichert.

Für die letzten genannten Schritte wurden Programme in C auf "normalen" sequentiellen Workstations implementiert. Da diese Berechnungen relativ harmlos sind, ergeben sich hier keine hohen Bearbeitungszeiten.

5 Ergebnisse

Die Laufzeit des parallelen Programms auf der Maspar MP-1 ist natürlich ganz wesentlich abhänigig von der Anzahl der Improve-Schritte und damit von der Wahl der Schwellwerte.

Nachfolgend ist eine Tabelle mit den Laufzeiten des sequentiellen und der parallelen Version des Programms in Abhängigkeit des Schwellwertes abgebildet. Es wurden jeweils Daten mit 7266 Triphonen in 55 Phonemen heruntergeclustert.

Schwellwert	Zeit in h auf der MasPar	Zeit in h auf HP-Workstation
500.0	19:34	7:35
1000.0	18:55	6:41
2000.0	18:19	5:20
10000.0	14:59	4:12

Das parallele Programm auf der MasPar ist also langsamer als die sequentielle Version auf der HP-Workstation. Warum kann das passieren?

Die MasPar hat eine theoretische Spitzenleistung von 1200 MFlops. Eine HP-Workstation ungefähr von 50 MFlops. Das bedeutet, daß im optimalen Fall ein Speed-Up von $\frac{1200}{50} = 24$ möglich ist. Da das parallele Programm auf jedem Prozessorelement ein Triphon speichert, können nur soviele PEs aktiv sein, wie Triphone zu clustern sind. Da in meinem Beispiel nur ca 7000 Triphone vorgegeben sind, werden von den 16384 PEs also weniger als die Hälfte benutzt. Das bedeutet eine weitere Leistungssenkung.

Der Algorithmus funktioniert so - wie die meisten Algorithmen für SIMD-Rechner - , daß im Laufe der Berechnung immer mehr PEs abgeschaltet werden. Ganz zum Schluß rechnen nur noch sehr wenige PEs einsam weiter. Das führt natürlich auch dazu, daß sehr viel Verarbeitungskapazität ungenutzt bleibt und man sich immer mehr von der theoretisch erreichbaren Verbesserung entfernt.

Nicht berücksichtigt wurde die Nachbearbeitung der Ergebnisse auf einer sequentiellen Workstation. Diese Nachbearbeitung liegt aber nur im Minutenbereich und ist deshalb nicht sehr interessant.

Leider steht ein so teurer Rechner wie die MasPar nicht einem Benutzer allein zur Verfügung, sondern man muß sie im Time-Sharing-System mit den anderen Usern teilen.

Dabei ist zu beachten, daß ein Programm, je länger es rechnet eine immer niedrige Priorität bekommt, und damit auch der Anteil Rechenzeit an der insgesamt zur Verfügung stehenden Zeit abnimmt. Jemand, der viele kleine Programme rechnen läßt, ist also jemandem mit einem langem Programm, das genausoviel Rechenzeit benötigt im Vorteil.

So hat das einmalige Durchlaufen eines Programmes sehr oft mehrere Tage gebraucht.

Die Programmierwerkzeuge, wie Compiler, Linker und Debugger stellt das Front-End zur Verfügung. Das Front-End ist leider "nur" eine DEC 5000 Workstation und wenn mehrere Benutzer gleichzeitig ihre Programme compilieren, debuggen und auf dem Front-End laufen lassen, so führt das zu Verzögerungen, die verglichen mit heutigen Workstations äußerst lang sind. Dies macht sich aber zum Glück nur bemerkbar, wenn wirklich mehrere User das Front-End nutzen.

6 Ausblick

Am Ende jedes Projektes stellen sich natürlich zwei Fragen:

- Steht der betriebene Aufwand im Verhältnis zu den erzielten Ergebnissen?
- Lohnt es sich in dem hier untersuchten Bereich weiterzuarbeiten?

Der Aufwand ein für einen "sequentiellen Programmierer" einen Algorithmus auf einem Parallelrechner zu implementieren ist enorm. Das Debuggen ist trotz einer Programmierumgebung auf dem Front-End sehr mühsam und zeitintensiv. (Es hat den größten Teil meiner Zeit ausgemacht.)

Insgesamt hat mich die Implementierung des Algorithmus auf der MasPar mehrere Monate reine Programmier- und Debuggingzeit gekostet. Und dieser Aufwand war nötig, um ein einziges Programm zu implementieren. Leider ist dieses Programm absolut architekturspezifisch nur für die MasPar geeignet und kann auf keinem anderem Parallelrechner eingesetzt werden.

Ein ganz anderer Ansatz um den beschriebenen Algorithmus schnell zu implementieren ist, daß man die Aufgaben, verschiedene Phoneme zu clustern, auf verschiedene Prozessoren oder sogar verschiedene vernetzte Rechner aufteilt. Die Multiprozessorsysteme von DEC und SUN eignen sich beispielsweise um einzelne Prozesse abzuspalten und gleichzeitig auf verschiedenen Prozessoren rechnen zu lassen. PVM oder DCE sind geeignete Umgebungen um Prozesse auf verschiedenen Rechnern auszuführen.

Ein mögliches Vorgehen zu einer Implementierung auf einer Multiprozessormaschine wird im folgenden skizziert.

Es wird ein Masterprozess gestartet. Dieser Masterprozess startet genausoviele Workerprozesse wie Prozessoren zur Verfügung stehen. Falls ein Workerprozess keine Arbeit hat, (das sind am Anfang alle), meldet er sich bei dem Masterprozess. Dieser versorgt ihn dann mit Arbeit. In diesem konkreten Beispiel bekommt er ein Phonem. Der Workerprozess clustert dann das ihm zugewiesene Phonem bis zum Schluß und speichert die Ergebnisse

ab. Danach meldet er sich wieder beim Masterprozess und holt sich die nächste Arbeit ab. Am Ende muß dann der Masterprozess die Ergebnisse zusammenfassen und auswerten.

Dieser Ansatz hätte mehrere Vorteile:

1. Das Programm führt automatisch eine dynamische Lastverteilung durch. Sobald ein Prozessor nichts mehr zu tun hat, meldet er sich beim Masterprozess und bekommt ein neues Stück Arbeit. Bei der MasPar bleibt einem oft nichts anderes übrig als den betreffenden Prozessor auszuschalten.

2. Das Programm ließe sich beliebig skalieren. Es wäre egal, ob man es auf einer Zwei-Prozessormaschine oder auf einer 16-Prozessormaschinen laufen läßt. Es könnte immer das gleiche Programm verwendet werden. Es müßten nur die Anzahl der Workerprozesse angepaßt werden.

3. Die Implementierung eines solchen Programmes ist sicherlich wesentlich einfacher als die Implementierung eines Programmes auf der MasPar. Man kann im groben das schon bestehende Clusterprogramm als Worker-Programm nehmen und muß nur noch ein Master-Programm schreiben, das die Arbeit entsprechend aufteilt.

4. Ein solches Programm wäre architekturunabhängig. Solange die Befehle zum Abspalten von Prozessen und zur Kommunikation mit ihnen gleich sind, läßt sich ein solches Programm auf verschiedenen Plattformen einsetzen.

Wenn man diese Vorteile gegenüber der sehr aufwendigen Programmierung von SIMD-Rechnern sieht, ist es in Zukunft sicherlich sinnvoller, solche Probleme wie das Clustern von Codebüchern, auf Multiprozessorsystemen zu implementieren.

Literatur

[1] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, New York, Chichester, Brisbane, Toronto, Singapore, 1973.

[2] Kai-Fu Lee. *Automatic Speech Recognition.* Kluwer Academic Publishers, Boston, Dordrecht, London, 1992.

[3] Widmayer Oddmann. *Algorithmen und Datenstrukturen.* BI-Wiss.-Verlag, Mannheim, 1993.

[4] Prechelt. *Measurements of MasPar MP-1216A Communication Operations.* Universitaet Karlsruhe, Karlsruhe, 1994.

[5] Goerke Ungerer, Schmid. *Rechnerarchitektur.* Universitaet Karlsruhe, Karlsruhe, 1994.

[6] Worsch Vollmar. *Modelle der Parallelverarbeitung.* B. G. Teubner, Stuttgart, 1995.