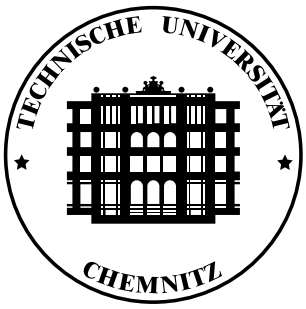
Technische Universität Chemnitz, Fakultät Informatik,
Professur Datenverwaltungssysteme



Diplomarbeit

Parallelisierung des Indexaufbaus im ICIx System

Bearbeiter:	Stefan Krumbiegel
Hochschullehrer:	Prof. Dr. W. Benn
Betreuer:	Dipl.-Inf. Otmar Görlitz
Abgabe:	29.07.2001

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung und Motivation

Wir leben in einem Informationszeitalter. Ständig werden Informationen gespeichert und verarbeitet. Zur Vereinfachung dieser Abläufe existieren Informationssysteme (IS). Heutige IS verwenden Datenbankmanagementsysteme zur Datenhaltung.

Eine häufige Operation in Informationssystemen ist das Suchen von Daten nach bestimmten Kriterien. Beispielsweise werden die Verkaufszahlen eines bestimmten Produktes in einem vorgegebenen Zeitraum gesucht, z.B. alle Verkäufe für Produkt = CT750 zwischen 1.1.2000 und 31.12.2000'.

Da die Sekundärspeicherzugriffe sehr kosten- und zeitintensiv sind, versucht man die zu lesende Datenmenge einzuschränken. Dazu werden sogenannte Indexierungstechniken angewandt.

Neben den eigentlichen Nutzdaten werden ein oder mehrere Indexe gespeichert. Mit diesen kann sehr effizient nach bestimmten Schlüsseln sortiert oder gesucht werden. So können die gesuchten Daten ohne einen aufwendigen sequentiellen Scan über die gesamte Datenbank gefunden werden.

Der hier behandelte Intelligent Cluster Index stellt eine solche Entwicklung auf dem Gebiet der Indexierungstechniken dar. Zum Verständnis von Indexen wird in Kapitel 2 ein Überblick über dieses Thema gegeben.

Ein de-facto Standard ist der von Rudolf Bayer erfundene B-Baum und seine Varianten. Auf seine Funktionsweise wird in Kapitel 2.1 noch genauer eingegangen.

Herkömmliche Indexe sind auf eindimensionale Suchanfragen beschränkt. Das heißt, es werden nur Suchanfragen nach einem Schlüssel unterstützt. Für heutige Anwendungen wie Geografische Informationssysteme oder Information Retrieval genügt das natürlich nicht. Folglich haben sich verschiedene mehrdimensionale Indexierungstechniken entwickelt. Auf diese Indexierungstechniken wird in Kapitel 2.2 näher eingegangen.

In [NG99] wurde ein solcher Index für mehrdimensionale Datenstrukturen vorgestellt. Dieser versucht die Nachteile zu vermeiden, mit denen andere mehrdimensionale Indexstrukturen behaftet sind.

Der Intelligent Cluster Index (ICIx) arbeitet mit wachsenden Neuronalen Netzen. Dies hat den Vorteil, daß er auf beliebige Daten angewendet werden kann. Die künstlichen Neuronalen Netze (KNN) passen sich dynamisch den vorhandenen Strukturen an. Es ist also a priori kein oder nur bedingt Wissen über die Daten nötig. In Kapitel 3 wird die Funktionsweise des ICIx näher erläutert.

In dieser Arbeit soll die Erstellung des Indexes parallelisiert werden. Der meiste Aufwand des Aufbaus liegt in der immer feineren Clusterung der Datenmenge.

Um den zeitaufwendigen Prozeß zu verkürzen, wurden die verschiedenen Möglichkeiten zurr Parallelisierung des anfallenden Berechnungsaufwandes untersucht. Dabei wird auf die in [Kru99] gewonnenen Erfahrungen bei der Parallelisierung von Neuronalen Netzen aufgebaut.

In Kapitel 4 werden die verschiedenen Möglichkeiten der Parallelisierung erläutert. Dabei erfolgt auch eine Untersuchung der Vor- und Nachteile. Der verwendete Ansatz wird in Kapitel 5 dargestellt. Er leitet sich aus den genannten Vor- und Nachteilen der Parallelisierungsmöglichkeiten ab.

1.1 Parallelisierung

Grundlegender Gedanke einer Parallelisierung ist die Verteilung des Berechnungsaufwandes auf mehrere parallele Prozesse. Damit kann die Berechnung bei n Prozessen idealerweise n-mal so schnell ausgeführt werden.

Wie in [Kru99] erläutert, wird diese Beschleunigung durch den zusätzlich nötigen Kommunikations- und Synchronisationsaufwand kaum erreicht. Dieser Aufwand steigt, je größer die Abhängigkeiten zwischen den parallelen Prozessen sind. Eine Parallelisierung sollte also eine Zerlegung in weitgehend unabhängige Teilaufgaben anstreben.

Während des Entwurfes des parallelen Algorithmuses wird die notwendige Berechnung zuerst in die einzelnen Teilschritte zerlegt. Anschließend können diese dann entsprechend der Funktionalität wieder zusammengefaßt werden. Es entsteht eine Gruppierung, welche die jeweils für eine Teilaufgabe benötigten Schritte enthält. Die so entstandenen Berechnungen können dann parallel ausgeführt werden, da sie voneinander unabhängig sind.

Bei realen Problemen ist eine solche ideale Zerlegung nur selten möglich. Zumindest bei einer notwendigen Zusammenfassung von Teilergebnissen entsteht Kommunikations- und Synchronisationsaufwand.

Wie in Kapitel 4 noch gezeigt wird, ist der Trainingsprozeß des ICIx sehr gut in unabhängige Teilaufgaben zerlegbar. Dies impliziert einen geringen Kommunikations- und Synchronisationsaufwand und damit eine gute Beschleunigung des Indexaufbaus durch Parallelisierung.

In dieser Arbeit wurde ein Rechnercluster als Plattform für die parallele Ausführung der Berechnung verwendet. Solch ein Cluster läßt sich prinzipiell auf jedem Computernetzwerk einrichten. Damit ist eine derartige Parallelisierung ohne großen zusätzlichen Aufwand möglich.

Da der Aufbau der Indexstrukturen des ICIx gut parallelisierbar ist, kann eine Parallelisierung einen deutlichen Geschwindigkeitszuwachs bringen. Sie ist also in jedem Fall empfehlenswert.

Ein Anwender mit Einzelplatzrechner kann zwar die hier verwendete parallele Verarbeitung auf einem Rechnercluster nicht direkt nutzen. Falls der Rechner aber über mehrere Prozessoren verfügt, kann ein solcher Cluster simuliert werden. Die Kommunikation zwischen den parallelen Prozessen läuft in diesem Falle sogar schneller ab, da sie über den internen Speicher erfolgt.

Wenn nur ein Prozessor zur Verfügung steht, ist offensichtlich keine Parallelisierung möglich. Durch seine Funktionsweise eignet sich ICIx besser für eher statische Datenmengen und bulk load Verfahren. Daher fällt der größte Aufwand während der Initialisierung des Indexes an.

Für diesen Fall kann der Rechner kurzzeitig in ein Netzwerk integriert werden. Dadurch ist eine Ausnutzung des Vorteils, den eine Parallelisierung bringt, während der initialen Erstellung der Indexstrukturen möglich. Bei der Benutzung des Indexes ist dann im Regelfall keine Parallelisierung mehr nötig.

Für evtl. umfangreiche Rekonfigurationen des Baumes nach einer Änderung am Datenbestand kann wie beim Aufbau des Indexes eine kurzzeitige Verbindung mit anderen Rechnern vorgenommen werden.

2 Arten von Indexen

Da ICIx eine Weiterentwicklung bewährter Indexierungstechniken darstellt, soll in diesem Kapitel ein Überblick über die verschiedenen Arten von Indexen gegeben werden.

Ziel von Indexen ist die Minimierung der Sekundärspeicherzugriffe. Im Folgenden sind die dazu notwendigen Strukturen und Algorithmen erläutert.

Ursprung der Entwicklung waren Indexe, die über die Werte eines einzelnen Attributes aufgebaut werden. Als typischer Vertreter wird hier der B-Baum vorgestellt. Da diese eindimensionalen Indexe für viele Anwendungen nicht ausreichend sind, wurden mehrdimensionale Indexierungstechniken entwickelt. Beispielhaft wird hier der R-Baum erläutert.

Typische Anfragen an IS betreffen semantisch ähnliche Objekte. Deshalb sollten die Daten nach ihren Ähnlichkeitsbeziehungen gruppiert werden. Solche Gruppierungen werden von Taxonomien, wie sie z.B. aus der Biologie bekannt sind, vorgenommen. Da ICIx solche taxonomischen Strukturen nachbildet, werden sie am Ende dieses Kapitels näher erläutert.

Das Suchen von Datensätzen mit bestimmten Werten erfordert ein Durchmustern des gesamten Datenbestandes. Dies impliziert das Laden sämtlicher Datensätze in den Speicher, um jeweils die Erfüllung der Suchkriterien zu überprüfen.

Eine Möglichkeit zur Minimierung dieses Aufwandes besteht in der Verwendung von Indexen. Bei diesen werden zusätzlich zu den eigentlichen Daten Indexinformationen gespeichert. Ziel ist die Minimierung der zu ladenden Datenbankseiten.

Bei Anfragen, die sich auf indizierte Attribute beziehen, können die zu ladenden Seiten ohne Sekundärspeicherzugriff bestimmt werden. Im Index werden alle Datensätze bestimmt, welche die Suchkriterien erfüllen. Alle Datenbankseiten, auf die der Index dann verweist, werden schließlich geladen. Abhängig von den vorhandenen Ausprägungen des angefragten Attributes kann das ein kleiner Bruchteil der Gesamtmenge sein. Als Einführung in die Datenorganisation ist [SS83] zu empfehlen.

Indexstrukturen bilden einen sogenannten Suchbaum. In der Informatik bezeichnet Baum eine Datenstruktur, welche aus einer Menge von zyklenfrei verbundenen Knoten besteht. Der Anfangsknoten heißt Wurzel und die gegenüberliegenden Knoten Blätter. Dazwischen befinden sich die sogenannten inneren Knoten.

Alle Knoten enthalten gewisse Daten. Wurzel und innere Knoten speichern zusätzlich Verweise auf ihre Söhne. Durch diese Verweisstruktur entsteht eine hierarchische Ordnung unter den Knoten. In Abbildung 1 ist ein solcher

Baum dargestellt. Da der Verzweigungsgrad immer 2 beträgt, spricht man von einem binären Suchbaum.

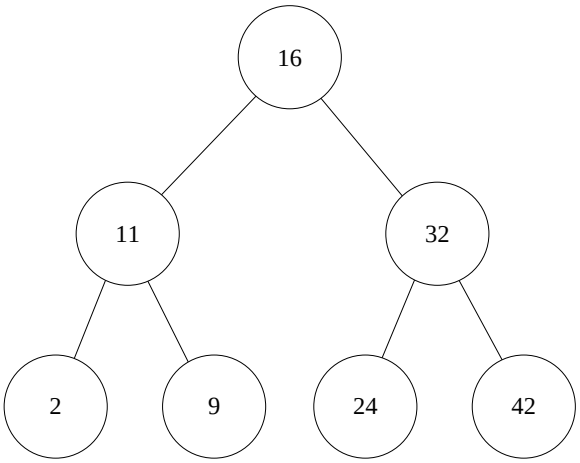


Abbildung 1: binärer Suchbaum

Ein Suchbaum enthält in seinen Knoten Schlüsselwerte, welche während einer Suche die Verzweigung innerhalb des Baumes ermöglichen. Alle Werte im Teilbaum des linken Sohnes sind kleiner als der im Vaterknoten gespeicherte Wert. Im rechten Teilbaum sind sie entsprechend größer.

Durch Vergleiche zwischen dem gesuchtem Wert und dem jeweiligen Schlüsselwert des Knotens kann im Baum abgestiegen werden. Entweder man wird innerhalb des Baumes fündig oder die Suche endet in einem Blatt.

Gelesen werden müssen dann nur noch die Sätze, auf die in dem gefundenen Knoten verwiesen wird. Das Verhältnis zwischen dem Umfang dieser Menge und der Gesamtmenge aller Datensätze in der Datenbank wird als Selektivität bezeichnet.

Indexe sollten nur für bestimmte Attribute angelegt werden. Zuviele Indexe sind eher hinderlich. Zum einen sind Indexe redundante Informationen, die Speicherplatz benötigen. Zum anderen müssen Indexe bei Update- oder Insertoperationen angepaßt werden. Bei zu vielen Indexen kann die Performance bei solchen Operationen dramatisch sinken.

Indexe sollten deshalb nur für Attribute angelegt werden, deren Wertebelegung eine niedrige Selektivität bedingen. Außerdem ist es günstig, wenn man bei der Indexerstellung die üblichen Anfragen an das System bereits kennt. Nur die darin enthaltenen Attribute kommen für eine Indexierung in Frage. Bei Beachtung dieser Regeln ist das Verhältnis zwischen dem Aufwand für die Pflege der Indexe und der Einschränkung der zu lesenden Datenmenge optimal.

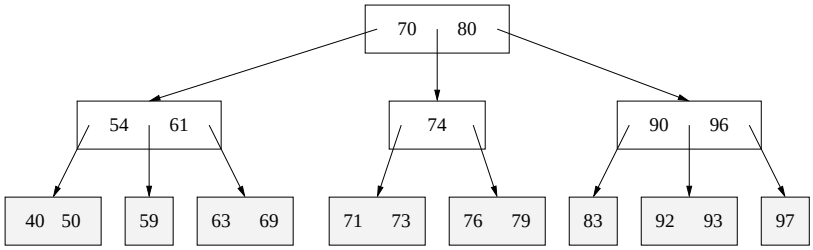


Abbildung 2: B-Baum der Ordnung 1

2.1 Eindimensionale Indexe

In herkömmlichen kommerziellen DBMS stehen dem Nutzer nur eindimensionale Indexstrukturen zur Verfügung.

Der bekannteste Vertreter der eindimensionalen Indexe ist der B-Baum. Bei dieser Art Suchbaum ist die Anzahl Söhne pro Knoten nicht so restriktiv wie beim oben genannten binären Suchbaum festgelegt. Es gibt eine obere und eine untere Grenze. Diese wiederum sind abhängig von der Ordnung des Suchbaumes.

Ein Knoten besitzt eine gewisse Anzahl m an Nachfolgern und entsprechend $m - 1$ Schlüssel zur Trennung der Unterbäume. Die Knoten müssen aber mindestens halb gefüllt sein. In einem Baum der Ordnung k hat ein Knoten also mindestens $k + 1$ und maximal $2k + 1$ Nachfolger. Abbildung 2 zeigt einen B-Baum erster Ordnung.

Durch die Algorithmen zur Verwaltung eines B-Baumes wird gesichert, daß der Baum stets vollständig balanciert ist. Dadurch ist der Weg von der Wurzel zu den Blättern immer gleich lang.

Die Anzahl der Knoten, die bei einer Suche durchlaufen werden müssen, ist logarithmisch durch die Anzahl der Schlüssel im Baum begrenzt. Bei n Schlüsseln und einem Baum der Ordnung k beträgt die Weglänge bei der Suche maximal $log_k n$.

Die Balance des Baumes wird von verschiedenen Operationen beeinflußt. Eine davon ist das Einfügen eines neuen Schlüssels. Tritt dabei eine Überfüllung auf, so wird der Knoten geteilt und die Werte entsprechend zugeordnet.

Das Überlaufen und Teilen der Knoten kann sich im ungünstigsten Fall bis an die Wurzel fortsetzen. Dann würde auch diese geteilt und der Baum bekommt eine neue Wurzel. In der Praxis tritt das aber selten auf.

Beim Löschen eines Schlüssels werden entsprechend Knoten zusammengefaßt, wenn sie weniger als halb gefüllt sind.

In der ursprünglichen Beschreibung des B-Baum [BM72] ist keine Aussage über die Ablage der Daten innerhalb des Baumes gemacht. In der Folgezeit wurden dann Varianten vorgeschlagen, bei denen nur in den Blättern Verweise auf die Datensätze enthalten sind. Diese sogenannten B*-Bäume werden in der Regel in heutigen DBMS verwendet.

2.2 Mehrdimensionale Indexe

Häufig beziehen sich Suchanfragen auf mehrere Attribute. In diesem Fall kann man die betreffenden Attribute hintereinander hängen und nach dem ersten Attribut sortieren, bei Wertgleichheit nach dem zweiten und so weiter. Es ergibt sich damit ein zusammengesetzter Schlüssel, der die Daten lexikografisch in der Reihenfolge der Indexattribute ordnet.

Das Telefonbuch ist ein Beispiel dafür. Es ist erst nach dem Ort, dann dem Namen und schließlich nach dem Vornamen geordnet. Solche zusammengesetzten Schlüssel eignen sich aber nur für Anfragen, die sich auf die führenden Schlüsselattribute beziehen. Im Telefonbuch kann zum Beispiel nach Ort und Namen gesucht werden. Bei einer Suche nur nach dem Vornamen ist die lexikografische Ordnung nutzlos.

In der Praxis kommen häufig Bereichsanfragen vor, die sich auf mehrere Attribute beziehen. Ein Beispiel wäre eine Anfrage nach allen vorhandenen Daten zu München vom Januar.

Die Spalten einer Datentabelle spannen mathematisch gesehen einen mehrdimensionalen Raum auf. Im Beispiel wäre dieser zweidimensional und hat die Wertebereiche [0,99999] für die Postleitzahlen und [0,365] für die Tage des Jahres. Bei einem weiteren Attribut würde entsprechend ein Quader im dreidimensionalen Raum gebildet.

Die Anfrage bildet darin ein Rechteck mit der Größe [80000,81999] x [0,30]. Eine solche Anfrageregion wird als *query box* bezeichnet.

Man könnte mehrere Sekundärindexe zur Beantwortung solcher Anfragen heranziehen, indem sie einzeln angewendet und ihre Ergebnisse zusammen-

gefaßt werden. Das Resultat würde zwar nur noch die Datensätze enthalten, welche alle Suchkriterien erfüllen. Es entsteht aber ein hoher Aufwand beim Zugriff auf die einzelnen Indexe. Außerdem müssen die Sätze einzeln geladen werden, da die gefundenen Datensätze auf dem Sekundärspeicher verstreut liegen können.

In manchen Fällen ist es günstiger, die Daten nur mit einem Index zu laden und die restlichen Anfragekriterien danach zu prüfen. Dies kann der Fall sein, wenn in der Anfrage ein Index mit sehr niedriger Selektivität enthalten ist. Dann ist der Aufwand für das unnötige Laden von Datensätze geringer als der Aufwand bei der Zusammenfassung mehrerer Indexergebnisse.

In Anwendungen wie Geografische Informationssysteme (GIS) und Information Retrieval (IR) sind die Anfragen naturgemäß mehrdimensional.

Im Information Retrieval werden Dokumente durch sogenannte Deskriptoren beschrieben. Dies sind charakteristische Merkmale, die in den Dokumenten enthalten sind und diese beschreiben. Eine Dokumentation kann damit durch die Menge aller möglichen Deskriptoren dargestellt werden.

Man kann sich ein einzelnes Dokument als binären Vektor vorstellen, dessen Länge dem Umfang der Deskriptorenmenge entspricht. Der Wert 1 besagt, daß der entsprechende Deskriptor in diesem Dokument enthalten ist. Ist der Deskriptor nicht in dem Dokument enthalten, so erhält die entsprechende Komponente des Vektors den Wert 0.

Eine Anfrage kann auch auf einen solchen Vektor abgebildet werden. Die Elemente, die eine 1 enthalten, bilden dann die Suchkriterien für eine Anfrage an die Datenbank. Bei einer mathematischen Betrachtung spannen die Vektoren einen Raum auf. Jede Komponente entspricht einer Achse eines möglichen Koordinatensystems in diesem Raum.

Im Laufe der Zeit haben sich verschiedene mehrdimensionale Indexierungstechniken entwickelt. Allen liegt das gleiche Prinzip zugrunde. Der Datenraum wird in mehrere Partitionen zerlegt. Diese Aufteilung wird dann in einem Suchbaum verwaltet. Dadurch kann die Suche anhand des Indexes auf bestimmte Teilräume eingeschränkt werden.

2.2.1 Der R-Baum

Eine Art der mehrdimensionalen Indexierung ist der von Guttmann vorgeschlagene R-Baum [Gut84]. Er stellt eine Erweiterung des B-Baumes auf mehrere Dimensionen dar.

Die Blattknoten enthalten Verweise auf die entsprechenden Datenbankseiten und die sogenannte *Bounding Box*. Dies ist das Rechteck, welches alle in dem Knoten referenzierten Objekte enthält.

Die Bounding Box der inneren Knoten umschließt alle Söhne, welche jeweils referenziert werden. In Abbildung 3 ist ein solcher Baum dargestellt. In der untersten Ebene sind die Datenbankseiten mit den enthaltenen Objekten zu sehen.

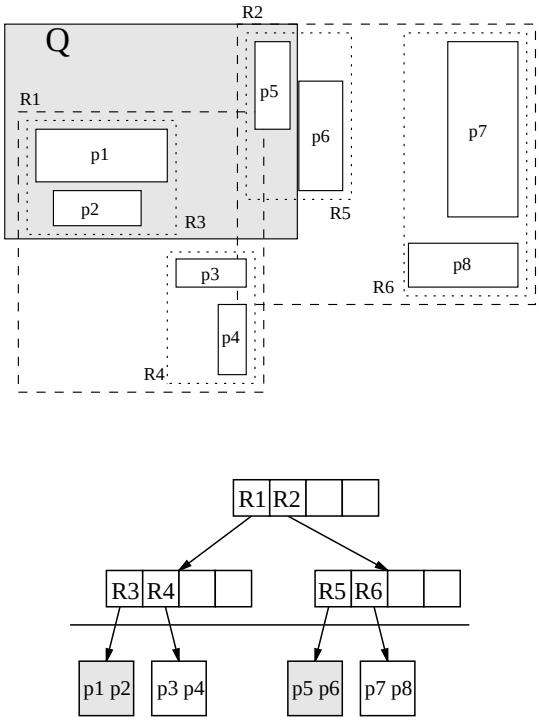


Abbildung 3: R-Baum

Bei einer Suche wird im Baum abgestiegen. Dabei werden alle Knoten betrachtet, deren Bounding Box von der Anfrage geschnitten wird. Alle Daten in den gefundenen Blättern müssen dann gelesen und nochmals einzeln geprüft werden.

Dies entspricht einer mehrstufigen Suche. Die Datenmenge, welche von der Datenbank gelesen werden muß, wird dabei minimiert. Im Beispiel von Abbildung 3 muß nur die Hälfte der Datenobjekte (p1, p2, p5, p6) gelesen werden. Ein Viertel der gelesenen Daten (p6) wird bei der anschließenden Prüfung wieder aussortiert. Ohne Index müßten alle Datenobjekte gelesen und geprüft werden. Fast zwei Drittel (p3, p4, p6, p7, p8) würden dabei umsonst gelesen werden, da sie die Anfrage nicht erfüllen.

Bei einer Suche wird in alle Teilbäume abgestiegen, deren Bounding Box von der Anfrage geschnitten wird. Dies kann natürlich auf mehrere Söhne zutreffen und es müssen mehrere Teilbäume untersucht werden. Aus die-

sem Grund ist die Minimierung der Überlappung der Bounding Boxen von Geschwistern beim Aufbau des R-Baumes und bei Einfüge- und Löschoperationen sehr wichtig.

Vor allem Einfüge- und Splitalgorithmus haben Einfluß auf diese Überlappung. Letzterem kommt dabei die höhere Bedeutung zu. Es existieren mehrere Algorithmen zum Splitten eines Knotens.

Bei einem Split werden mit den vorhandenen Datenobjekten zwei neue Blätter gebildet. Diese sollen weit voneinander entfernt liegen und ein minimales Volumen besitzen. Ausgehend von den zwei am weitesten entfernt liegenden Objekten werden die restlichen Daten dort eingefügt, wo jeweils eine geringere Erweiterung der Bounding Box erforderlich ist.

Mit dem Ziel der Minimierung von Volumen und Überlappung der Bounding Box wurden aufbauend auf den R-Baum viele weitere mehrdimensionale Indexstrukturen entwickelt.

Anwendungsgebiete wie Data Mining oder Information Retrieval benutzen häufig Anfragen nach semantisch ähnlichen Objekten. Typisch ist eine immer stärkere Verfeinerung der Auswahlkriterien.

Im Information Retrieval sucht man beispielsweise englische Dokumente zum Thema Kochen. Da die gelieferte Ergebnismenge noch zu umfangreich ist, nimmt man weitere Einschränkungen vor. Zum Beispiel sind nur noch nach Dokumente von Interesse, die speziell auf Pasta eingehen. Eine weitere Verfeinerung nach Pastaart oder Autor wäre denkbar.

Alle Datensätze, auf die in den Ergebnisknoten einer Suche verwiesen wird, müssen gelesen und untersucht werden. Folglich ist es günstig, wenn sich die tatsächlich betroffenen Sätze innerhalb weniger Knoten befinden. Bei Systemen, die häufig Anfragen nach semantisch ähnlichen Daten verwenden, sollte sich die Partitionierung der Indexstrukturen deshalb an den semantischen Gruppen innerhalb der Daten orientiert.

Dafür geeignet sind taxonomische Strukturen, da sie Objekte nach ihren verwandschaftlichen Beziehungen in ein System von Gruppen einordnen. Dabei sind die Gruppen hierarchisch geordnet. Die Relation Ober-/Unterklasse entspricht Verallgemeinerung/Spezialisierung.

2.2.2 Taxonomische Strukturen

Taxonomien, auch Systematiken genannt, findet man in allen Wissenschaftszweigen. Zu den bekanntesten Vertretern zählen die Systematiken der Biologie. Taxonomien wurden während der gesamten Wissenschaftsgeschichte erforscht und weiterentwickelt.

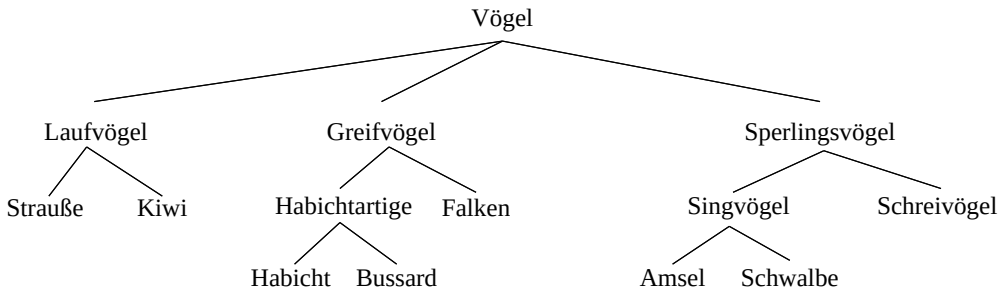


Abbildung 4: Ausschnitt aus der Taxonomie der Vögel

Ihre Leistungsfähigkeit ist die semantische Strukturierung großer Mengen anhand von vielen Merkmalen oder Merkmalskombinationen. Objekte werden nach ihren inhaltlichen Merkmalen in disjunkte Gruppen eingeteilt. Die einzelnen Merkmale wirken dabei unterschiedlich stark strukturierend. Sowohl die Gruppierung der Daten als auch die Anzahl der entstehenden Gruppen kann verschieden sein.

So ist es möglich, Hierarchien aufzubauen. Man gliedert die Menge anhand eines Merkmales. Die entstehenden Gruppen stehen auf einer Ebene der Hierarchie. Sie können dann anhand der restlichen Merkmale weiter unterteilt werden. Die daraus resultierenden Teilmengen werden der jeweiligen Obermenge untergeordnet. So entstehen unterschiedliche Abstraktionsniveaus der Ausgangsmenge. Solange noch differenzierende Merkmale vorhanden sind, kann die rekursive Unterteilung fortgesetzt werden. Gewöhnlich wird sie aber bei einer bestimmten gewünschten Granularität abgebrochen.

Neue Objekte lassen sich mit einem solchen hierarchischen System leicht einer Gruppe zuordnen. Falls dies nicht gelingt, gehört es zu einer neuen Kategorie. In der Biologie entspricht das der Entdeckung einer neuen Art. Ohne Taxonomien ist eine solche Aussage nur schwer nachzuweisen.

Sind nicht alle Merkmale eines Objektes zugänglich, können sie trotzdem anhand der bestimmbaren in die Taxonomie eingeordnet werden. Anhand dieser werden im Hierarchiebaum Gruppen gesucht, welche die Merkmale ebenfalls enthalten. Der Grad der Übereinstimmung aller Merkmale repräsentiert die Stärke der Verwandschaft zwischen dem Objekt und den Gruppen.

Die Gruppen können benannt werden. Semantisch ähnliche Objekte sollten dabei ähnlich aufgebaute Namen erhalten. Menschlichen Nutzern wird damit die Navigation in der Hierarchie erleichtert.

Taxonomien stellen einen Mechanismus zur mehrdimensionalen semantischen Indexierung dar. Typisch für die Einsatzgebiete dieser Indexierungsart sind explorative und inhaltsbezogene Anfragen. Das heißt, sie bezie-

hen sich auf inhaltlich ähnliche Objekte und werden schrittweise verfeinert. Herkömmliche mehrdimensionale Indexe unterstützen derartige Anfragen nur unzureichend. Deshalb suchen aktuelle Forschungsprojekte nach neuen Ansätzen.

Ein wichtiger Aspekt ist die Einteilung der Daten in Gruppen semantisch ähnlicher Daten. Neuronale Netze erlernen die Unterscheidung von Klassen anhand von Beispielen in einem Trainingsprozeß. Nach dem Training können sie die Trainingsbeispiele und unbekannte Daten entsprechend einordnen. Sie erfordern eine Kodierung symbolischer Daten, da sie nur mit Zahlenwerten arbeiten. Aus entsprechend kodierten Daten werden die semantischen Verwandschaftsbeziehungen extrahiert. Ebenso lassen sich dann neue Datensätze entsprechend einordnen.

Sie erscheinen daher als geeignet, bei einer semantischen Indexierung die Einteilung in Gruppen vozunehmen. Aus einer solchen Einteilung kann eine künstliche Systematik konstruiert werden, indem die Gruppen iterativ weiter unterteilt werden. ICIx stellt eine solche Art der Indexierung dar.

3 Funktionsweise des ICIx

Wie in Kapitel 2.2 dargestellt, benötigen Anwendungen wie Geografische Informationssysteme oder Information Retrieval durch ihre charakteristischen mehrdimensionalen Anfragen Indexe für mehrere Attribute. Mehrere eindimensionale Indexe sind bei Updateoperationen hinderlich. Außerdem ist das Zusammenfassen der Teilergebnisse der verschiedenen Indexe für eine Anfrage sehr aufwendig. Aus diesem Grunde haben sich mehrdimensionale Indexierungstechniken entwickelt.

Der beschriebene R-Baum (2.2.1) und seine Varianten werden im Bereich der geometrischen Anwendungen eingesetzt. ICIx hingegen stellt einen Ansatz für die semantische Indexierung dar.

Beim Aufbau des Indexbaumes wird die Datenmenge in Gruppen semantisch ähnlicher Datenobjekte unterteilt. Diese Gruppen werden iterativ weiter geclustert. Ziel ist es, den Umfang der gefundenen Cluster auf die Größe einer Datenbankseite einzuschränken.

Die Gruppen werden in einer Baumstruktur angeordnet. Je tiefer die Gruppen in dem Baum eingeordnet sind, desto spezieller sind die Merkmale der enthaltenen Datenvektoren. Ein Abstieg im Baum entspricht einer Spezialisierung und ein Aufstieg einer Verallgemeinerung.

Anfragen an Datenbanken betreffen gewöhnlich semantisch ähnliche Objekte. Der ICIx bietet dafür also optimale Unterstützung. Löschen, Einfügen oder Ändern der Daten erfordert aber einigen Aufwand.

Beim Löschen kann eine Unterfüllung, beim Einfügen eine Überfüllung eines Knotens des Indexbaumes auftreten. Beim Ändern eines Datenobjektes wird dieses evtl. in einen anderen Knoten verlagert. Im Ergebnis können auch hier Unter- und Überfüllung auftreten. In jedem Fall ist ein Neuaufbau des Zweiges notwendig. Durch evtl. Verschiebungen der Daten kann sich das nach oben fortplanzen, so daß der ganze Baum neu aufgebaut werden muß.

ICIx eignet sich deshalb eher für statische Datenmengen wie sie im Information Retrieval auftreten. Dort treten im Regelfall lesende Zugriffe auf. Diese werden optimal unterstützt. Die Datenbasis wird selten erweitert. Löschen und Ändern von Daten treten ebensowenig auf.

3.1 Neuronale Netze

Wie in der Einleitung bereits gesagt wurde, verwendet der ICIx neuronale Netze zur Einteilung der Daten in semantische Gruppen. Deshalb soll hier auf deren Arbeitsweise näher eingegangen werden.

Die grundlegenden Begriffe von Neuronalen Netzen werden hier nicht behandelt und können in der Literatur nachgelesen werden.[1] Zum besseren Verständnis der formellen Erläuterungen soll zuerst die verwendete Notation erläutert werden.

3.1.1 Notationen

Ein Neuronales Netz besteht aus einer Menge A von N Neuronen.

$$A = c_1, c_2, \ldots, c_N$$

Zu jedem Neuron c gehört ein Referenzvektor, der auf die Position des Neurons im Eingaberaum zeigt.

$$w_c \in R^n$$

Zwischen den Neuronen kann eine Menge ungewichteter, symmetrischer Verbindungen existieren.

$$C \subset A \times A$$

$$(i,j) \in C \Longleftrightarrow (j,i) \in C$$

Daraus resultiert eine Menge direkter topolgischer Nachbarn für jedes Neuron.

$$N_c = i \in A | (c,i) \in C$$

Für das Training des Netzes wird eine endliche Menge n-dimensionaler Eingabevektoren (Trainingsmenge) gebraucht.

$$D = \xi_1, \ldots, \xi_M \; \xi_i \in R^n$$

Für jedes Eingabesignal ξ existiert ein Gewinnerneuron $s(\xi)$, dessen Referenzvektor dem Eingabesignal am nächsten liegt.

$$s(\xi) = \arg\min_{c \in A} \|\xi - w_c\|$$

$\|\cdot\|$ bezeichnet dabei die euklidische Abstandsnorm.

3.1.2 Lernverfahren

Um ein bestimmtes Problem lösen zu können, müssen die Netze einen Trainingsprozeß durchlaufen. Das heißt, die freien Parameter des Netzes werden durch Stimulation aus der Umgebung angepaßt. In einem iterativen Prozeß erfolgen Änderungen an den Referenzvektoren der Neuronen.

$$w_i(t+1) = w_i(t) + \Delta w_i(t)$$

[1] [Hay94], [Roj96] ,[Zel94]

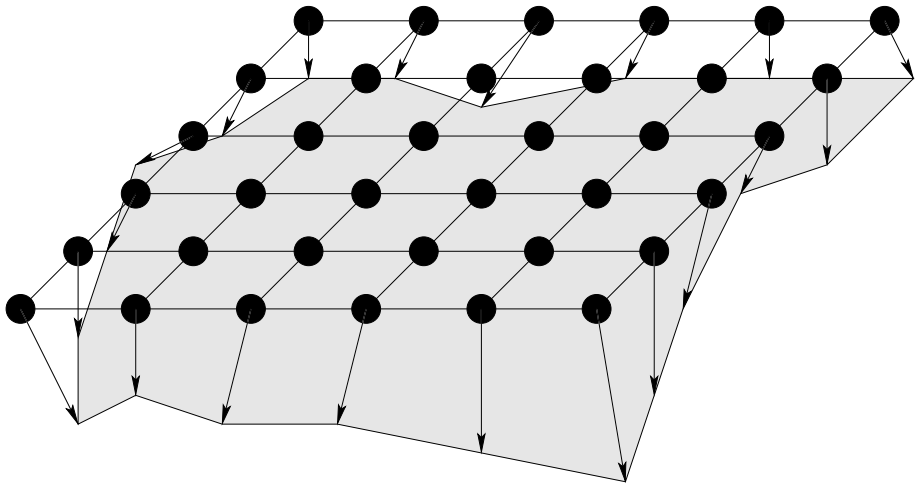


Abbildung 5: SOM

Dabei bestimmen Art und Weise dieser Anpassung den Typ des Lernens.

Zum einen gibt es die Gruppe der überwachten Lernverfahren. Dabei bewertet ein Lehrer die für die Eingabevektoren erzeugten Ausgaben des Netzes. Der Lehrer hat also Wissen über die Umgebung, welches auf das Netz übertragen wird. Dadurch sind diese Lernverfahren für das Clustering ungeeignet. Es kann kein Wissen vorausgesetzt werden und es sollen auch unbekannte Beziehungen gefunden werden.

Die zweite Gruppe der unüberwachten Lernverfahren benutzt sogenanntes Wettbewerbslernen. Allen Neuronen des Netzes wird das gleiche Beispiel präsentiert. Daraufhin ändert sich deren Aktivierungspotential. Das Neuron mit der höchsten Aktivierung wird so angepaßt, daß es bei nochmaliger Präsentation desselben Beispiels noch stärker reagieren würde. Die Neuronen des Netzes spezialisieren sich also auf bestimmte Beispiele.

3.1.3 Selbstorganisierende Neuronale Netze

ICIx verwendet selbstorganisierende Neuronale Netze. Die Entwicklungen auf diesem Gebiet werden in diesem Abschnitt kurz aufgezeigt.

Die Gruppe der selbstorganisierenden Netze verwenden unüberwachte Lernverfahren. Bekanntester Vertreter sind die von Teuvo Kohonen entwickelten Self Organizing Maps (SOM). [Koh89]

Hier ist Anzahl und Verbindungsstruktur fest vorgegeben. Nachteilig daran ist die schlechte Repräsentation der Topologie des Eingabesraumes.

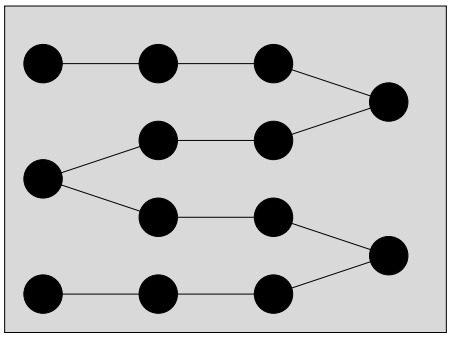


Abbildung 6: Topologiedefekt

In der Folge haben sich Lernverfahren für wachsende neuronale Netze entwickelt. Sie fügen während des Trainings Neuronen und Kanten in das Netz ein. Beispiele von direkten Erweiterungen der SOM-Architektur sind Growing SOM [BV95], Growing Grid [Fri95] und Incremental Growing Grid [BM95]. Auch diese Verfahren verwenden eine feste Gitterstruktur. Lernverfahren ohne Gitterstruktur sind Growing Cell Structures (GCS) [Fri91] und Adaptive Cell Struktures (ACS) [KS96].

Die GCS arbeiten mit einer k-dimensionalen Simplexstruktur. Während des Trainings werden Neuronen hinzugefügt und auch wieder gelöscht. Dabei wird immer auf den Erhalt der Simplexstruktur geachtet. Bei geeigneter Wahl von k wird eine gute Topologieerhaltung erreicht.

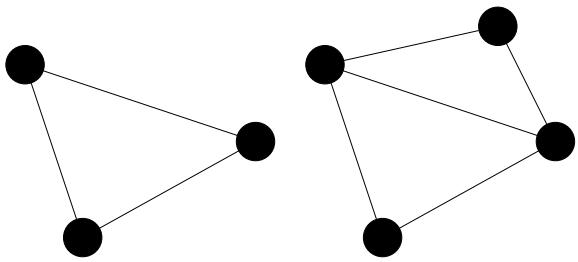


Abbildung 7: GCS

Eine dritte Gruppe von wachsenden neuronalen Netzen arbeitet ohne Initialstruktur. Die Verbindungsstruktur bildet sich erst während des Trainings heraus. Grundlegend für diesen Architekturtyp ist das Neuronengas Modell [MS91]. Dieses verwendet lediglich das Best-Matching Kriterium zur Sortierung der Referenzvektoren. Da keine Verbindungsstrukturen definiert sind, können sich die Neuronen ähnlich Gasmolekülen frei bewegen. Somit erreicht man eine Anpassung an beliebig dimensionale Trainingsdaten.

Darauf aufbauend entwickelte Bernd Fritzke das Growing Neural Gas (GNG) [Fri94]. Bei Tests [NG98] erwies sich diese Architektur als sehr flexibel. Sie wurde deshalb im ICIx eingesetzt und soll näher betrachtet werden.

3.2 Growing Neural Gas

Beginnend mit zwei verbundenen Neuronen werden während des Trainings in Regionen mit hohem Fehlermaß weitere Neuronen eingefügt. Die Topologie wird durch Hebbsches Lernen erzeugt. Dieses beruht auf dem Postulat von D.O. Hebb, welches besagt, daß die Verbindung zweier Neuronen, die nahe beieinander liegen und sich gegenseitig beeinflußen, im Laufe der Zeit stärker wird.

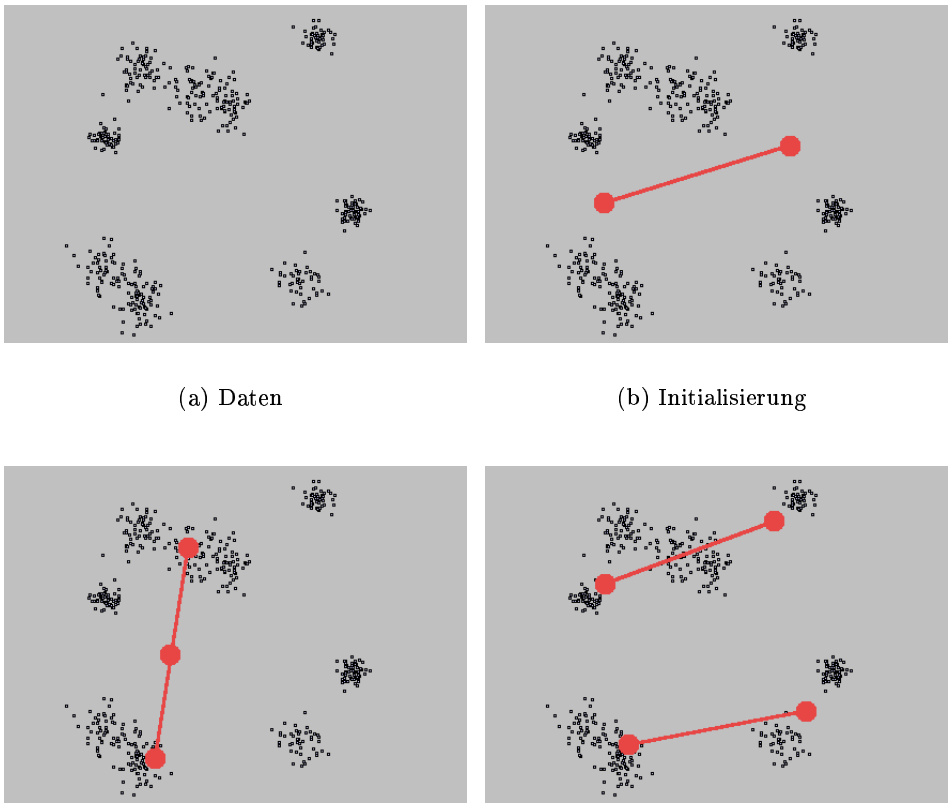
Für jeden Eingabevektor werden die beiden naheliegendsten Referenzvektoren und die dazugehörigen Neuronen bestimmt. Falls noch keine Verbindung zwischen ihnen existiert, wird eine generiert. An jeder Kante wird vermerkt, wie oft sie getroffen wurde. Daraus kann auf ihre Bedeutung geschlossen werden. Weniger wichtige Kanten werden wieder entfernt. Adaptiert werden die Referenzvektoren des Gewinnerneurones und seiner direkten topologischen Nachbarn.

Hauptvorteil ist die weitestgehend topologieerhaltende Verbindungsstruktur, welche während des Trainings generiert wird. Es müssen keine Festlegungen zu Beginn des Trainings erfolgen. Die Struktur kann am Ende des Trainings aus mehreren Teilnetzen bestehen.

Während des Trainings werden weniger wichtige Kanten gelöscht. Durch die ständige Adaption der Referenzvektoren können sich zwei verbundene Neuronen wieder voneinander wegbewegen. So entstehen im Laufe des Trainingsprozesses Kanten, die für keinen Eingabevektor Gewinnerneuron und Zweitbesten verbinden. Neuronen, die innerhalb eines Clusters in den Eingangsdaten liegen, sind alle jeweils paarweise Gewinner und Zweiter für die Datenvektoren des Clusters. Die Verbindungen zwischen ihnen besitzen also ein gewisse Bedeutung. Nur Kanten, die Neuronen in verschiedenen Clustern verbinden, sind eher unwichtig. Sie können demzufolge auch gelöscht werden. Wenn das GNG-Netz während des Trainings in Teilnetze zerfällt, können diese im Umkehrschluß als Repräsentanten von Datenclustern angesehen werden.

3.2.1 Algorithmus

Die formale Beschreibung des Algorithmus wurde der Neural Network Objects Bibliothek (NNO) [Ste98] entnommen. Hier wird ein anderer Algorithmus zur Alterung der Kanten verwendet, als sonst üblich.



(a) Daten

(b) Initialisierung

(c) erste Einfügung

(d) am Ende

Abbildung 8: Growing Neural Gas

Zum einen werden Kanten nicht ständig sondern in expliziten Löschschritten entfernt. Zum anderen besitzen die Kanten einen Zähler, der bei jedem Treffer um einen bestimmten Betrag β erhöht wird. Als Fehlermaß für die Neuronen wird die Varianz verwendet.

Bei jedem Einfüge- oder Löschschritt werden sowohl die Zähler der Kanten als auch die Varianzsummen der Neuronen auf einen Wert kleiner 1.0 normiert. Diese Normierung ist abhängig von der Anzahl der Adaptionen seit der letzen Anpassung. Die Anzahl der Einfügeschritte λ_{Ins} und der Löschschritte λ_{Del} sind üblicherweise unterschiedlich groß. Deshalb muß die Zeit explizit gemessen werden.

Kanten mit einem Zähler kleiner dem Schwellwert β_{min} werden in einem Löschschritt nur entfernt, wenn die verbundenen Neuronen nicht allein stehen bleiben würden. Das bedeutet, es werden prinzipiell keine Neuronen gelöscht.

Algorithmus 1 GNG

1. Initialisierung der Neuronenmenge, Verbindungen und des Zeitparameters

$$A = \{c_1, c_2\} \tag{1}$$
$$C = \{(c_1, c_2)\} \tag{2}$$
$$t = 0 \tag{3}$$

2. zufällige Wahl eines Eingabevektors $\xi \; inD$

3. Bestimmung des Gewinnerneurons s_1 und des Zweibesten s_2

$$s_1(\xi) = \arg\min_{c \in A} \|\xi - w_c\| \tag{4}$$
$$s_2(\xi) = \arg\min_{c \in A \setminus \{s_1\}} \|\xi - w_c\| \tag{5}$$

4. Erstellung einer Kante zwischen s_1 und s_2 und Initialisierung des Kantenzählers, falls noch keine Verbindung exisitert

$$C = C \cup (s_1, s_2) \tag{6}$$
$$F_{(s_1,s_2)} = \frac{1}{(1-\beta)^t} \tag{7}$$

sonst Erhöhung des Kantenzählers

$$F_{(s_1,s_2)} = F_{(s_1,s_2)} + \beta \tag{8}$$

5. Erhöhung der lokalen Fehlervariable des Gewinnerneurons s_1 um den quadratischen Abstand zwischen Eingabe- und Referenzvektor

$$\Delta E_{s_1} = \|\xi - w_{s_1}\|^2 \tag{9}$$

6. Anpassung der Referenzvektoren von s_1 und seinen topologischen Nachbarn unter der Verwendung der Lernraten ϵ_b und ϵ_n

$$\Delta w_{s_1} = \epsilon_b(\xi - w_{s_1}) \tag{10}$$
$$\Delta w_i = \epsilon_n(\xi - w_i) \quad (\forall i \in N_{s_1}) \tag{11}$$

7. Einfügen eines neuen Neurones, wenn die Zahl der Trainingsschritte ein Vielfaches von λ_{Ins} ist

   - Bestimmung des Neurons q mit dem größten lokalen Fehler

$$q = \arg\max_{c \in A} E_c \tag{12}$$

   - Bestimmung des topologischen Nachbarn f mit dem größten lokalen Fehler

$$f = \arg\max_{c \in N_q} E_c \tag{13}$$

   - Einfügen eines neuen Neurons r und Bestimmung dessen Referenzvektors aus den Refernzvektoren von q und f

$$A = A \cup \{r\}, \quad w_r = \frac{(w_q + w_f)}{2} \tag{14}$$

   - Löschen der Kante (q, f) und Einfügen von (q, r) und (r, f)

$$C = C \backslash \{(q, f)\}, \quad C = C \cup \{(q, r), (rf,)\} \tag{15}$$

     sowie der Verbindungen von r zu allen gemeinsamen Nachbarn von q und f

$$C = C \cup \{(r, p)\}, \quad (\forall p \in A | (q, p), (q, f) \in C) \tag{16}$$

   - Initialisierung des lokalen Fehlermaßes von r durch Interpolation der Fehler seiner Nachbarn

$$E_r = \frac{1}{|N_r| + 1} * \sum_{i=1}^{|N_r|} E_i \tag{17}$$
$$E_i = E_i * \left(1 - \frac{1}{|N_r + 1|}\right) \tag{18}$$

- Normierung der Fehler und Zähler

$$E_i = E_i * (1-\alpha)^t \quad (\forall i \in A) \tag{19}$$
$$F_j = F_j * (1-\beta)^t \quad (\forall j \in C) \tag{20}$$

und Zurücksetzen des Zeitparameters

$$t = 0 \tag{21}$$

8. Entfernung weniger wichtiger Kanten, wenn die Zahl der Trainingsschritte ein Vielfaches von λ_{Del} ist

- Normierung der Fehler und Zähler

$$E_i = E_i * (1-\alpha)^t \quad (\forall i \in A) \tag{22}$$
$$F_j = F_j * (1-\beta)^t \quad (\forall j \in C) \tag{23}$$

und Zurücksetzen des Zeitparameters

$$t = 0 \tag{24}$$

- Löschen unbedeutender Kanten

$$C = C \backslash C_{rem} \tag{25}$$

$$C_{rem} = \{(c_i, c_j) \in C | F_(c_i, c_j) < \beta_{min}, |N_{c_i}| > 1, |N_{c_j}| > 1)\}$$

9. Erhöhung des Zeitparameters

$$t = t + 1 \tag{26}$$

10. Wiederholung ab Schritt 2, solange das Stopkriterium noch nicht erfüllt ist

---

Als Stopkriterium wird in der Regel eine maximale Neuronenzahl festgelegt. Die Parameter $\lambda_{Ins}, \lambda_{Del}, \alpha, \beta, \beta_{min}, \epsilon_b, \epsilon_n$ müssen zu Beginn festgelegt werden. Allerdings gibt es dafür keinen analytischen Weg.

3.3 Hierarchische Strukturen

Die im letzten Abschnitt vorgestellten Netze haben in realen Anwendungen ein unzureichendes Skalierungsverhalten. Das heißt, mit wachsender Problemgröße steigt der Aufwand für das Best-Matching auf unzumutbare Größe an. Erfolgversprechend für eine Beschleunigung sind hierarchische Architekturen. Auch ICIx verwendet eine Hierarchie. Die Datenmenge wird

durch Neuronale Netz in immer feinere Cluster aufgeteilt. Im Hierarchiebaum kommt dann diese Ober-/Untermengenbeziehung zum Ausdruck.

Als Kriterium für eine notwendige Erweiterung der Hierarchie dient die Neuronenanzahl. Beim Erreichen eines bestimmten Grenzwertes während der Clusterung einer Datenmenge wird das Training auf dieser Ebene eingestellt. Es erfolgt eine weitere Unterteilung aller gefundenen Cluster, die noch zu viele Eingabevektoren enthalten. Im Baum ergibt sich daraus pro Cluster ein neuer Teilbaum.

ICIx kommt für Datenbanken zum Einsatz. Daraus ergibt sich für die Datenmengen, die von einem Blatt des Hierarchiebaumes repräsentiert werden, die Einschränkung der Größe auf eine Datenbankseite. Der Grenzwert für eine erforderliche weitere Clusterung ist von der Größe der einzelnen Datenobjekte und der Größe der Datenbankseiten abhängig. Beides steht zu Beginn des Indexaufbaues fest. So kann der Grenzwert am Anfang bestimmt werden.

Beschleunigend für den Indexaufbau wirkt der Fakt, daß in den Substrukturen nur noch mit einer eingeschränkten Datenmenge gearbeitet werden muß. Durch die Clusterung auf der vorhergehenden Ebene sind die Daten bereits vorklassifiziert. Eine weiter Clusterung ist nur noch eine Spezialisierung ähnlicher Objekte.

Außerdem kann die Wahl der maximalen Größe der Netze abhängig vom Umfang der Datenmenge erfolgen. Für die Gesamtmenge findet nur eine Grobgliederung mit kleinen Netzen statt. Diese Netze sind auch für große Datenmengen performant.

3.4 Strukturen des ICIx

Mehrdimensionale Objekte oder Punktmengen haben gewöhnlich keine regelmäßige Begrenzung. Diese komplexen Strukturen lassen sich daher bei Anfragen nur sehr aufwendig mit der Anfrageregion vergleichen. In Indexen werden deshalb nur Approximationen der Objekte verwaltet.

Wie der R-Baum verwendet ICIx umschließende Hyperboxen. Sie repräsentieren die Knoten und Blätter des Suchbaumes. Durch den Indexierungsalgorithmus entsteht eine hierarchische Struktur der Hyperboxen. In den Blättern werden die Daten umschlossen, auf die verwiesen wird. Die Hyperboxen der inneren Knoten enthalten alle Sohnknoten und damit deren Daten.

Die Knoten und Blätter werden mit fortlaufenden eineindeutigen Schlüsseln, den sogenannten *Pagekeys*, versehen. Jeder Knoten des Baumes kann durch ein Tupel (Pagekey, Hyperbox, Verweis auf Vaterknoten, Verweise auf

Sohnknoten) dargestellt werden. Die Pagekeys stellen das Bindeglied zur Datenbank dar.

Eine Möglichkeit der Anbindung des erstellten Hierarchiebaumes an ein Datenbanksystem (DBS) ist es, die Pagekeys in die zu indexierenden Relationen aufzunehmen. Die Relationen werden dabei um die Attribute *pk* und *tid* erweitert. *tid* ist eine eindeutige ID für alle Tupel, die als Primärschlüssel verwendet wird. *pk* gibt Auskunft über die Ähnlichkeitsgruppe, zu welcher der Datensatz gehört. Für beide Attribute werden normale eindimensionale Indexe angelegt.

Zuerst wird die Anfrage mit Hilfe des Indexbaumes verarbeitet. Diese Anfragebearbeitung liefert eine Menge $\{pk_i\}$ von Pagekeys, in deren Gruppen bzw. Datenbankseiten die gesuchten Tupel gefunden werden können. Die Suche in der Datenbank kann also auf diese Gruppen beschränkt werden. Dazu wird die WHERE-Klausel der ursprünglichen Anfrage um "AND pk in $(pk_1, \ldots, pk_n)$" erweitert und an das DBS weitergereicht. Für die Anfragebearbeitung auf der Datenbank kann dann der eindimensionale Index für das Attribut *pk* verwendet werden.

Dieser Ansatz hat aber keinerlei Einfluß auf die physikalische Speicherung der Relation. Ziel beim Aufbau des Indexes ist es, daß die Datenmenge, die von den Blattknoten repräsentiert wird, auf eine Datenbankseite paßt. Damit bietet sich der ICIx zur Primärorganisation an. Mann kann dann eine Beziehung zwischen den Blattknoten und den repräsentierten Clustern zu den Datenbankseiten herstellen. Die Blattknoten verweisen jeweils auf eine Datenbankseite.

Die Herstellung dieser Beziehung kann auf zwei Wegen geschehen. Zum einen kann man nach dem Aufbau des Indexes alle Daten aus der Datenbank löschen und anschliesend geordnet nach den Pagekeys wieder einfügen. Durch die Größenbeschränkung in den Blattknoten liegen dann Datensätze einer Gruppe auf sehr wenige Datenbankseiten verteilt. Diese Ordnung wird aber bei UPDATE, INSERT oder DELETE von Datensätzen wieder zerstört.

Die andere Möglichkeit der Primärorganisation mit den Pagekeys ist abhängig vom zugrundeliegenden System. Einige DBS bieten die Möglichkeit, die Datensätze geordnet nach einem ausgewählten Attribut zu speichern. Diese Mechanismen werden *clustered table* (Oracle Server Administrators Guide: Managing Clusters) oder *clustered index* (INFORMIX-OnLine Dynamic Server Performance Guide: clustering) genannt. Dabei werden die Datensätze vom DBMS nach dem Attribut *pk* geordnet. Aber auch diese Ordnung wird von Änderungen im Datenbestand beeinflußt. Eine permanente Sicherung der Ordnung kostet sehr viel Perfomance. Deshalb wird

eine zeitweise Verletzung der Ordnung inkaufgenommen. In bestimmten Intervallen erfolgt eine komplette Reorganisation.

3.5 Aufbau des Baumes

Der ICIx stellt eine taxonomische Struktur bereit. Das bedeutet, beim Aufbau des Indexes werden die Daten in immer speziellere Gruppen eingeteilt. Die Wurzel des Baumes repräsentiert alle Datensätze. Je weiter man im Baum absteigt, desto kleiner werden die von einem Knoten repräsentierten Datenmengen.

Der Aufbau beginnt mit der Clusterung der Gesamtdatenmenge. Anschließend erfolgt in einem iterativen Prozeß die weitere Unterteilung der gefundenen Cluster. Für jede ermittelte Menge wird die umschließende Hyperbox bestimmt und ein eindeutiger Pagekey vergeben. Die so gewonnen Daten kommen im Indexbaum als neuer Knoten unter die Obermenge.

3.5.1 Clustering

Für die Begriffe Cluster oder Clustering gibt es keine allgemeingültige Definition. Hier soll die intuitive Erklärung aus [NG99] verwendet werden.

Wenn man die Merkmale eines Objektes betrachtet, so kann man sich die Objekte als Punkte im mehrdimensionalen Raum vorstellen. Die Achsen eines Koordinatensystems in diesem Raum werden durch die Merkmale bestimmt. Die jeweiligen Ausprägungen ergeben dann die Koordinatenwerte.

Cluster sind dann Regionen mit einer hohen Dichte von Punkten, getrennt durch Bereiche mit einer geringeren Dichte. Clustering ist nun das Problem des Auffindens solcher Regionen. Dabei gilt es, Gruppen von Datenpunkten zu finden, bei denen die Abstände zwischen den Punkten sowie der jeweilige Abstand zum Clusterzentrum minimal sind.

Da die Anzahl der Gruppen a priori nicht bekannt ist, haben sich selbstorganisierende Neuronale Netze mit unüberwachten Lernverfahren für die Lösung des Problems bewährt.

3.5.2 Algorithmus

Bezeichnungen	
pk	pagekey $\in N$
D	Datenvektorenmenge
ξ	Datenvektor
$\xi.pk$	pagekey des Clusters, zu dem der Datenvektor gehört
$D.pk$	Menge der Datensätze, die zu einem Cluster gehören
d_{max}	Maximalanzahl Datenvektoren in einem Blatt des Index
CL	Menge von Datenclustern

Der Aufbau des Indexbaumes läßt sich in drei Algorithmen teilen. Zuerst erfolgt die Initialisierung der Wurzel des Baumes. Dieser Schritt besteht aus der Zuordnung aller Datenvektoren, der Ermittlung der umschließenden Hyperbox und der Initialisierung und Zuweisung des Zählers für die Pagekeys pk. Anschließend wird die weitere Unterteilung der Datenmengen gestartet.

Algorithmus 2 create Index

$pk = 0$
$root = T(pk++, HB(D), \emptyset, \emptyset)$
for all $\xi \in D$ **do**
  $\xi.pk = T.pk$
end for
Expand($root$)

Für die Unterteilung gibt es zwei Grenzwerte, die zu Beginn festgelegt werden müssen. Der Indexbaum verweist in seinen Blättern auf Datenbankseiten. Diese können nur eine begrenzte Anzahl Datensätze aufnehmen. Die Daten sind also ausreichend geclustert, wenn alle Datensätze, die den einzelnen Blattknoten des Indexbaumes zugeordnet sind, auf jeweils eine Datenbankseite passen. Ein Grenzwert ist also die maximale Anzahl Datensätze pro Cluster.

ICIx ist bei der Anzahl der Söhne im Gegensatz zum Binärbaum variabel. Da der Index in der Datenbank effizient verwaltet werden soll, wird aber wie beim B-Baum eine obere Schranke benötigt. Dieser Grenzwert S_{max} kann direkt bei der Clusterung getestet werden. Dazu wird die Anzahl der verwendeten Neuronen beschränkt.

Jeder neue Sohn repräsentiert einen gefundenen Cluster. Um einen solchen Cluster zu finden, muß während des Trainings ein Teilnetz entstehen. Dazu sind mindestens zwei Neuronen erforderlich. Wenn man die Gesamtzahl auf $2 * S_{max}$ begrenzt, so können nur S_{max} Cluster und damit Söhne entstehen.

Algorithmus 3 Expand(node)

```
D' = D.pk
if |D'| ≤ d_max then
  return
end if
CL = Clusterung(D')
for all D'' ∈ CL do
  son = T(pk++, HB(D''), ∅, node)
  node.addSon(son)
  for all ξ ∈ D'' do
    ξ.pk = son.pk
  end for
  Expand(son)
end for
```

Der Algorithmus zum weiteren Aufbau des Indexbaumes muß zu Beginn überprüfen ob eine weitere Clusterung nötig, d.h., die Datenmenge noch zu groß ist. Ist dies der Fall, werden die Daten weiter unterteilt. Ansonsten ist der Aufbau dieses Zweiges abgeschlossen. Bei einer weiteren Clusterung werden alle Datenvektoren einem der entstehenden Cluster zugeordnet, die Hyperboxen ermittelt und Pagekeys erzeugt. Anschließend werden die neuen Teilmengen auf der nächsten Ebene des Baumes eingehängt.

Zusätzlich erfolgt für jeden neuen Cluster ein Eintrag in einer Warteschlange zur weiteren Verarbeitung. Die Behandlung der Einträge erfolgt analog der Vorgehensweise für die Wurzel. Je nach Art der Schlange gleicht das einer Tiefen- oder Breitensuche.

Der Clusteralgorithmus verwendet Neuronale Netze zur Einteilung. Am Anfang müssen die dafür nötigen Trainingsparameter bestimmt werden. Das betrifft unter anderem die maximale Anzahl an Neuronen. Wie bereits weiter oben geschrieben, werden Cluster durch zusammenhängende Teilnetze, sogenannte Zusammenhangskomponenten (ZHK), des entstehenden Netzes interpretiert. Die maximale Neuronenanzahl ist also für die Begrenzung der Anzahl entstehender Söhne im Indexbaum notwendig.

Abhängig vom Umfang der aktuellen Datenmenge werden noch Einfüge- und Löschschritte sowie die Adaptionsraten für Gewinnerneuron und seine Nachbarn gewählt.

Die Schrittweite zum Löschen von Kanten und Einfügen von Neuronen wird so gewählt, daß jeder Datensatz unabhängig vom Umfang der Datenmenge immer gleich häufig angelegt wird. Bei kleineren Datenmengen werden die Neuronen dadurch seltener adaptiert. Sowohl die Schrittweiten als auch die Adaptionsraten müssen also abhängig vom Datenumfang gewählt werden.

Algorithmus 4 Clusterung(D')

```
1: Bestimmung der GNG-Parameter
2: Training eines GNG auf D'
3: CL = ∅
4: Bestimmung der Zusammenhangskomponenten zhk
5: for all zhk do
6:    D'' = ∅
7:    for all c ∈ zhk do
8:       for all ξ ∈ D' do
9:          if s(ξ) = c then
10:            D'' = D'' ∪ {ξ}
11:         end if
12:      end for
13:   end for
14:   CL = CL ∪ {D''}
15: end for
16: return CL
```

Mit diesen Parametern wird ein GNG auf den Daten trainiert. Dieses besteht am Ende des Trainings aus Zusammenhangskomponenten, die Bereiche mit hoher Dichte in den Trainingsdaten abdecken. Alle Datenvektoren, die einem solchen Teilnetz zugeordnet werden, gehören zu einem Cluster.

4 Parallelisierung

Die Parallelisierung eines Problemes soll die notwendige Zeit zu dessen Lösung verkürzen. Dazu wird der Berechnungsaufwand auf mehrere Prozesse verteilt. Das verkürzt zwar die Berechnungszeit, es entsteht aber zusätzlicher Kommunikations- und Synchronisationsaufwand, der bei serieller Ausführung nicht auftritt.

Nur selten ist die Aufteilung einer Berechnung in vollständig unabhängige Teilaufgaben möglich. Problematisch ist beispielsweise eine evtl. nötige Zusammenfassung der Ergebnisse. Entweder ein zentraler Verwaltungsprozeß faßt die lokalen Ergebnisse aller Prozesse zusammen. Oder die Prozesse bilden eine Kette. In dieser werden die Ergebnisse weitergereicht und jeder Prozeß fügt sein lokales Ergebnis hinzu. In beiden Fällen ist Kommunikation und Synchronisation zwischen den Prozessen notwendig.

Ziel dieser Arbeit ist es, den Aufbau des Indexbaumes durch Parallelisierung zu beschleunigen. In Kapitel 3.5.2 sind die drei Algorithmen beschrieben, in die der Aufbauprozeß gegliedert ist. Diese sollen in den nächsten Abschnitten auf Parallelisierungsmöglichkeiten untersucht werden.

Das Kapitel beginnt mit der Erläuterung des verwendeten Kostenmodelles. Im Anschluß werden die einzelnen Möglichkeiten für die gewünschte Parallelisierung näher betrachtet.

4.1 Kostenmodell

Das Kostenmodell ist dem Buch "Simulation of neural networks on parallel computers" von Urs A. Müller ([Mü93]) entnommen. Es kam bereits in der Arbeit [Kru99] zum Einsatz. Zum Verständnis der weiteren Untersuchungen soll es hier noch einmal erläutert werden.

Die folgenden Bezeichnungen tauchen in den formellen Beschreibungen auf:

n	Anzahl paralleler Prozesse
m	Anzahl der zu berechnenden Werte
t_0	Zeit zur Berechnung eines Wertes
t_c	Zeit zur Kommunikation eines Wertes zwischen den Prozessen
t_s	serieller Overhead, der nicht direkt der Berechnung dient
t_{su}	Synchronisations- und Verwaltungsaufwand

Aussage über den erzielbaren Geschwindigkeitsvorteil bei der Parallelisierung gibt der sogenannte *Speedup Faktor* $s(n)$. Er stellt das Verhältnis zwischen der Zeit, die der schnellste serielle Algorithmus benötigt, und der Zeit in paralleler Ausführung dar.

Die Zeit für die serielle Ausführung ergibt sich aus der Anzahl der zu berechnenden Werte, der Zeit, die für einen Wert benötigt wird und dem seriellen Overhead.

$$t_{ser} = m * t_0 + t_s \tag{27}$$

Die benötigte Zeit des parallelen Algorithmus setzt sich aus der Berechnungszeit t_{comp} und der Zeit für die Kommunikation t_{comm} zusammen. Die Berechnung wird dabei auf die n parallelen Prozesse aufgeteilt.

$$t_{comp} = \frac{m * t_0}{n} + t_s \tag{28}$$

In die Kommunikationszeit gehen die Zeiten zum Austausch der Werte zwischen den Prozessen und die Synchronisationszeiten ein.

$$t_{comm} = (m * t_c) + (n * t_{su}) \tag{29}$$

Der Speedup Faktor ist damit

$$\begin{aligned} s(n) &= \frac{t_{ser}}{t_{comp} + t_{comm}} && (30) \\ &= \frac{m * t_0 + t_s}{(\frac{m*t_0}{n} + t_s) + ((m * t_c) + (n * t_{su}))} \\ &= \frac{1}{\frac{1}{n} + \frac{t_c}{t_0} + \frac{t_s}{m*t_0} + n + \frac{t_{su}}{m*t_0}} && (31) \end{aligned}$$

Aus dieser Formel lassen sich drei Faktoren extrahieren, die den Speedup Faktor beeinflussen. Das ist zum einen der Komplexitätsfaktor c.

$$c = \frac{t_c}{t_0}$$

Er gibt das Verhältnis zwischen der Berechnungs- und Kommunikationszeit eines Wertes an.

Die Kommunikationsverzögerung d stellt das Verhältnis zwischen Synchronisations- und Verwaltungsaufwand und reiner Berechnungszeit des seriellen Algorithmus dar.

$$d = \frac{t_{su}}{m * t_0}$$

Der serielle Overhead e wird aus dem Verhältnis der reinen Berechnungszeit zu den zusätzlich nötigen Zeiten ermittelt.

$$e = \frac{t_s}{m * t_0}$$

Beim Einsetzen der drei Faktoren ergibt sich folgende Formel für den Speedup Faktor:

$$s(n) = \frac{1}{\frac{1}{n} + \frac{1}{c} + e + n * d} \tag{32}$$

Im Folgenden sollen die Einflüsse der Faktoren genauer untersucht werden.

4.1.1 Analyse des Speedup Faktors

Ideal wäre eine Beschleunigung der Berechnung um den Faktor n, also der Anzahl parallel arbeitender Prozesse. Aus Formel 32 ergibt sich dieser Zustand bei $c \to \infty$, $d = e = 0$. Diese Werte ergeben einen Speedup Faktor $s(n) = n$.

Wenn man nur einzelne Faktoren betrachtet, kann die Formel vereinfacht werden und eine genauere Untersuchung des Einflusses auf den Speedup Faktor ist möglich.

Zuerst soll nur der Komplexitätsfaktor c betrachtet werden. Dies ergibt nach entsprechender Umformung von Formel 32:

$$s(n) = \frac{n * c}{n + c}$$

Für eine unendlich große Prozeßanzahl ($n \to \infty$) geht der Speedup Faktor gegen den Grenzwert c. Der Komplexitätsfaktor wird durch die Kommunikationsgeschwindigkeit begrenzt. Je schneller die Kommunikation abläuft, desto größer wird der Komplexitätsfaktor. Mit steigendem Berechnungsaufwand für die einzelnen Werte steigt der Komplexitätsfaktor ebenfalls.

Bei Betrachtung des seriellen Overhead e ergibt sich für den Speedup Faktor die Gleichung:

$$s(n) = \frac{1}{\frac{1}{n} + e}$$

Sie geht mit steigender Prozeßzahl n gegen $\frac{1}{e}$. Dies ist ein Grenzwert, der durch den Algorithmus gegeben ist und nicht hardwareseitig beschleunigt werden kann.

Wenn $c \to \infty$ und $e = 0$ kann der Einfluß der Kommunikationsverzögerung näher untersucht werden. Es ergibt sich folgende Gleichung zur Bestimmung des Speedup Faktors:

$$s(n) = \frac{n}{1 + n^2 * d}$$

Diese hat für eine steigende Prozeßanzahl den Grenzwert 0. Das heißt, der Speedup Faktor wird durch die Kommunikationsverzögerung vermindert.

Aus diesen drei genannten Faktoren ergeben sich verschiedene Aspekte, die beim Entwurf paralleler Algorithmen beachtet werden müssen. Grundsätzlich sollte der Komplexitätsfaktor c maximiert, serieller Overhead e und Kommunikationsverzögerung d hingegen minimiert werden.

Sowohl c als auch d hängen vom insgesamt notwendigen Kommunikationsaufwand ab. Für den Komplexitätsfaktor spielt die Beschleunigung der Kommunikation eine Rolle. Dies kann durch die Reduktion der notwendigen Kommunikationvorgänge geschehen. Zum anderen kann auch durch technische Mittel der eigentliche Kommunikationsvorgang beschleunigt werden (z.B. höhere Übertragungsgeschwindigkeiten).

Die Notwendigkeit zur Synchronisation entsteht immer, wenn Prozesse Werte austauschen müssen. Mit anderen Worten ist auch der Synchronisationsaufwand von der Anzahl der Kommunikationsvorgänge abhängig.

Die Minimierung der Kommunikation stellt einen zentralen Aspekt bei der Maximierung des Speedup Faktors dar. Entscheidend für die Anzahl der notwendigen Kommunikationsvorgänge sind die Abhängigkeiten zwischen den parallelen Prozessen. Umso unabhängiger die zu bearbeitenden Teilaufgaben voneinander sind, desto weniger Kommunikation ist unter den Prozessen notwendig. Wichtig ist also die Aufteilung in weitgehend unabhängige Teilaufgaben.

Da Kommunikation nicht vollständig zu vermeiden ist, spielt auch die Lastbalancierung eine wichtige Rolle. Wenn zwei Prozesse am Ende der Berechnungen miteinander kommunizieren müssen, ist eine Synchronisation zwischen ihnen notwendig. Im Falle einer ungleichen Verteilung des Berechnungsaufwandes ist ein Prozeß eher fertig und muß auf den anderen warten. Diese Wartezeiten werden umso kürzer, je besser der Berechnungsaufwand verteilt wird. Durch leerlaufende Prozesse wird außerdem Parallelisierungspotential verschenkt, das die Berechnung weiter beschleunigen könnte.

Wichtige Aspekte der Parallelisierung sind also:

- die Minimierung der Kommunikation
- die Vermeidung von Abhängigkeiten zwischen den Teilaufgaben
- die Lastbalancierung zwischen den Prozessen

In den folgenden Abschnitten sollen die verschiedenen Algorithmen zum Indexaufbau des ICIx auf mögliche Parallelisierungen untersucht werden. Dabei werden auch Vor- und Nachteile der Möglichkeiten genannt.

4.2 Möglichkeiten der Parallelisierung

Wie in Kapitel 3.5.2 beschrieben, läßt sich die Erstellung des Indexbaumes in drei Algorithmen unterteilen. Der Aufbau besteht aus einer Initialisierung (Alg 2) und der wiederholten rekursiven Erweiterung der Baumknoten (Alg 3). Bei dieser Expandierung wird der dritte Algorithmus (Alg 4) zur Clusterung der Datenmenge verwendet. Bei der Betrachtung dieser Algorithmen lassen sich verschiedene mögliche Parallelisierungsvarianten finden, welche in den folgenden Abschnitten untersucht werden.

Bei der Clusterung werden zur Einteilung der Daten wachsende künstliche Neuronale Netze vom Typ Growing Neural Gas verwendet. Der Trainingsprozeß dieser Netze bietet eine Möglichkeit zur Parallelisierung.

Nach jeder Clusterung erfolgt eine Zuordnung der Eingabevektoren zu den gefundenen Clustern. Ebenso werden die jeweiligen umschließenden Hyperboxen bestimmt. Beide Aufgaben lassen sich zerlegen und in parallelen Prozessen ausführen.

Der Kern des Indexaufbaues ist die immer feinere Unterteilung der Datenmenge in einem iterativen Prozeß. Dazu wird der Algorithmus 3 rekursiv für die Datenmengen der bereits gefundenen Cluster aufgerufen. Auch diese Rekursion ist zur Parallelisierung geeignet.

In den folgenden Abschnitten werden die verschiedenen Möglichkeiten näher betrachtet.

4.3 Trainingsprozeß der Neuronalen Netze

Grundsätzlich gibt es beim Trainingsprozeß der Neuronalen Netze drei Ebenen der Parallelisierung. Zum einen können mehrere Netze gleichzeitig trainiert werden. Dies entspricht einer Parallelisierung im Experimentraum. Eine andere Möglichkeit ist die Aufteilung der Trainingsdaten, was einer Parallelisierung im Datenraum entspricht. Und schließlich kann die Parallelisierung auch auf Netzebene ansetzen. Dabei erfolgt eine Verteilung der Neuronen des Netzes auf verschiedene Prozesse.

In der Arbeit [Kru99] wurden bereits erste Erfahrungen mit der Parallelisierung von GNG-Netzen gesammelt. Damals erfolgte eine gleichzeitige Aufteilung auf Netz- und Datenebene. Sobald sich das Netz teilt, werden die beiden entstandenen Teilnetze mit den zugeordneten Datenvektoren unabhängig in parallelen Prozessen weiterverarbeitet.

Diese Herangehensweise ist nach dem jetzigen Kenntnisstand kritisch zu beurteilen. Sie schließt ein späteres Zusammenwachsen zweier Teilnetze völlig aus. Die Möglichkeit für so eine Entwicklung des Netzes besteht jedoch. Es würde dann ein Teilnetz in zwei verschiedenen Prozessen trainiert. Je

nach Aufteilung der dazugehörigen Daten kann ein Teil des Netzes in einem Prozeß vollkommen unterrepräsentiert und überhaupt nicht mehr in der Netzstruktur ersichtlich sein. Außerdem müßte beim Zusammenfassen der Ergebnisse das verteilte Teilnetz erkannt und vereinigt werden. In dieser Arbeit sollen deshalb weitere Möglichkeiten der Parallelisierung des Trainingsprozesses untersucht werden.

Für die Bestimmung des Speedup Faktors ist die Laufzeit des seriellen Algorithmus (Alg. 1) notwendig. Am Anfang der Untersuchungen der Parallelisierungmöglichkeiten wird deshalb der Aufwand für eine serielle Ausführung des Trainingsprozeß bestimmt. Darauf aufbauend erfolgt in den nächsten Abschnitten eine Berechnung der Aufwände und Speedup Faktoren für die verschiedenen Varianten der Parallelisierung.

Für die Betrachtungen von Laufzeit und Speedup Faktor werden folgende Werte verwendet:

Anzahl Datenvektoren	m	100000
Länge von Daten- und Referenzvektoren	l_ξ	10
Anzahl paralleler Prozesse	n	10
maximale Anzahl an Neuronen	x_{max}	10
Schrittweite Einfügungen	λ_{Ins}	$10 * m$
Schrittweite Löschungen	λ_{Del}	$5 * m$

Die Untersuchungen sollen unabhängig von der verwendeten Hardware sein. Aus diesem Grund wird mit Zeiteinheiten (ZE) gerechnet. Der Zugriff auf einen Wert (z.B. ein Vektorelement) entspricht einer Zeiteinheit, unabhängig von der Art der Operation (Vergleich, Addition, ...). Ebenso wird bei den parallelen Algorithmen die Übertragung eines Wertes mit einer Zeiteinheit angesetzt.

Für die Bestimmung des Aufwandes einzelner Schritte ist die Anzahl der Neuronen und Verbindungen nötig. Diese ändern sich aber bei Einfüge- und Löschschritten im Laufe des Trainings. Deshalb wird von Mittelwerten ausgegangen, die eine homogene Verteilung der Daten voraussetzen. Dabei soll ein vollständig verbundenes Netz entstehen.

Das Training beginnt mit zwei Neuronen. Im Laufe des Trainingsprozesses werden bis zum Erreichen des Grenzwertes x_{max} Neuronen hinzugefügt. Der Mittelwert der verwendeten Neuronenanzahl ergibt sich als arithmetisches Mittel aus dem Startwert und der maximalen Anzahl.

$$\overline{x} = \frac{2 + x_{max}}{2} = \frac{2 + 10}{2} = \mathbf{6}$$

Bei der Bestimmung der Anzahl an vorhandenen Kanten wird von diesem Wert ausgegangen. In einem vollständig verbundenen Netz hat jedes Neuron

eine Verbindung zu allen anderen Neuronen, wobei diese bidirektional sind. Die Gesamtzahl der Verbindungen ergibt sich durch Aufsummieren der ganzen Zahl von 1 bis zur Anzahl der Neuronen $\overline{x} - 1$.

$$\overline{v} = \sum_{i=1}^{\overline{x}-1} i = \sum_{i=1}^{5} i = \mathbf{15}$$

Nach diesen Vorbetrachtungen soll jetzt die Laufzeit des seriellen Algorithmus (Alg 1) untersucht werden.

4.3.1 Laufzeit des seriellen Algorithmus

Schritt 1 untergliedert sich in die Initialisierung der Neuronen- und Verbindungsmenge sowie des Zeitparameters. Die Neuronen benötigen zufällige Referenzvektoren. Der Aufwand dafür beträgt $2 * l_\xi = 20$. Die Verbindung besteht aus den Identifikatoren der Neuronen und dem Startwert des Kantenzählers. Insgesamt sind das drei Werte. Zur Initialisierung des Zeitparameters ist nur ein Wertzugriff notwendig. Zusammengefaßt beträgt der Aufwand der Initialisierung:

$$2 * l_\xi + 3 + 1 = \mathbf{24\ ZE}$$

Für die Auswahl eines Referenzvektors ist die Bestimmung einer Zufallszahl im Bereich $[1 : m]$ erforderlich. Dafür ist nur ein Wertzugriff notwendig. Schritt 2 von Algorithmus 1 ist also in einer Zeiteinheit berechenbar.

Für die Ermittlung von Gewinnerneuron und Zweitbestem müssen sämtliche Referenzvektoren mit dem Eingabevektor verglichen werden. Bei einer Vektorlänge l_ξ von 10 Elementen sind das 60 Vergleichsoperationen.

$$\overline{x} * l_\xi = 6 * 10 = \mathbf{60}$$

Falls bereits eine Verbindung besteht, muß der Kantenzähler erhöht werden. Das entspricht einem Wertzugriff und damit einer Zeiteinheit. Wie bereits bei der Initialisierung erläutert, dauert das Einfügen einer neuen Kante drei Zeiteinheiten. Bei der angenommenen homogenen Datenmenge und dem vollständig verbundenen Netz ist dieser Fall aber selten. Nur nach dem Einfügen eines neuen Neuron wird eine Verbindung aufgetrennt, die sich im Laufe des Trainings wieder neu bilden muß. Der Aufwand für Schritt 4 wird deshalb mit einer Zeiteinheit angesetzt.

Die lokale Fehlervariable des Gewinnerneurones wird in Schritt 5 um den quadratischen Abstand von Referenz- und Eingabevektor erhöht. Dieser Abstand wurde bereits bei der Ermittlung des Gewinners in Schritt 3 bestimmt.

Folglich ist die Anpassung der lokalen Fehlervariable mit einem einzigen Wertzugriff verbunden. Schritt 5 benötigt also nur eine Zeiteinheit.

In den Aufwand der Adaption der Referenzvektoren geht die Zeit für eine Anpassung und die Gesamtzahl anzupassender Vektoren ein. Bei einer Anpassung wird zuerst der Differenzvektor zwischen Eingabe- und Referenzvektor gebildet und dann zum Referenzvektor addiert. Bei einer Vektorlänge von 10 Elementen sind das 20 Operationen. Da von einem vollständig verbundenen Netz ausgegangen wird, müssen auch alle Referenzvektoren adaptiert werden. Insgesamt ergibt sich damit ein Aufwand von:

$$(2 * l_{xi}) * \overline{x} = 20 * 6 = \mathbf{120\,ZE}$$

Bei einem Einfügeschritt muß zuerst das Neuron q mit dem größten lokalen Fehlermaß ermittelt werden. Der Aufwand dafür ist von der Anzahl vorhandener Neuronen abhängig. Für die Suche des Neurons mit dem größten Fehlermaß werden also bei den hier angenommenen Werten 6 Zeiteinheiten benötigt. Bei der Suche des Nachbarn f mit dem größten Fehlermaß müssen dann nur noch 5 Neuronen betrachtet werden.

Der Referenzvektor des neuen Neurons r ergibt sich aus den Vektoren der beiden gerade ermittelten Neuronen q und f. Durch die Länge der Referenzvektoren $l_\xi = 10$ benötigt seine Bestimmung 10 Zeiteinheiten.

Die Einbindung in die Netzstruktur setzt sich aus dem Löschen der Kante zwischen q und f sowie dem Einfügen der neuen Kanten zusammen. Das Löschen der Kante ist in einer Zeiteinheit abgeschlossen. Anschließend muß r mit allen bereits vorhandenen Neuronen verbunden werden. Bei unserer Annahme von durchschnittlich 6 Neuronen und 3 Zeiteinheiten pro Verbindung summiert sich das auf 18 Zeiteinheiten. Zusammen kostet das Einbinden in die Netzstruktur also 19 Zeiteinheiten.

Das lokale Fehlermaß von r wird aus den Fehlern seiner Nachbarn interpoliert. Pro Neuron stellt dies einen Wertzugriff dar. Da es sich um ein vollständig verbundenes Netz handelt, werden 6 Zeiteinheiten für das Lesen der Fehlermaße und 1 Zeiteinheit für das Eintragen des interpolierten Wertes benötigt.

Am Ende eines Einfügeschrittes werden die lokalen Fehlermaße und Kantenzähler normiert. Nach der Einfügung sind 7 Neuronen und 20 Kanten vorhanden. Das ergibt 27 Zeiteinheiten. Das Zurücksetzen des Zeitparameters läßt sich in einer Zeiteinheit berechnen. Insgesamt benötigt ein Einfügeschritt damit einen Aufwand von:

$$
\begin{aligned}
\overline{x} + (\overline{x} - 1) &+ l_\xi \\
&+ 1 + 3 * \overline{x} \\
&+ \overline{x} + 1 \\
&+ (\overline{x} + 1) \\
&+ (\sum_{i=1}^{\overline{x}-1} i - 1) + \overline{x} \\
&+ 1
\end{aligned}
$$

$$11 + 10 + 19 + 7 + 7 + 20 + 1 = \mathbf{76\,ZE}$$

Das Löschen von unbedeutenden Kanten beginnt mit der Normierung der Fehlermaße und Kantenzähler. Bei den verwendeten Mittelwerten von 6 Neuronen und 15 Kanten werden dafür 21 Zeiteinheiten benötigt. Um unbedeutende Kanten zu ermitteln, müssen anschließend alle Kantenzähler untersucht werden. Es kommen dadurch 15 Zeiteinheiten zum Aufwand hinzu. Unter der Voraussetzung der homogenen Datenmenge kann davon ausgegangen werden, daß keine Kanten gelöscht werden. Folglich kommt zum Gesamtaufwand eines Löschschrittes nur noch eine Zeiteinheit für das Zurücksetzen des Zeitparameters hinzu.

$$\overline{x} + 2 * \overline{v} + 1 = \mathbf{37\,ZE}$$

Die Erhöhung des Zeitparameters stellt eine einfache Addition dar. Schritt 9 benötigt also nur eine Zeiteinheit. In folgender Tabelle sind die Aufwände der einzelnen Schritte noch einmal aufgelistet.

1	Initialisierung	24
2	Wahl des Eingabevektors	1
3	Bestimmung von Gewinner und Zweitbestem	60
4	Verbindungskante anpassen/erzeugen	1
5	lokale Fehlervariable des Gewinnerneurons erhöhen	1
6	Referenzvektoren adaptieren	120
7	Neuron einfügen	76
8	Kanten löschen	37
9	Zeitparameter erhöhen	1

Die Schritte 2 bis 9 werden in einer Schleife durchlaufen. Als Abbruchkriterium dient eine maximale Neuronenanzahl. Es werden innerhalb des Algorithmus keine Neuronen gelöscht. Damit läßt sich genau ermitteln, wie

oft die Schleife durchlaufen wird. Bei einem Startwert von 2 und einer maximalen Anzahl x_{max} von 10 Neuronen müssen also 8 eingefügt werden. Die Schrittweite zwischen zwei Einfügungen beträgt $\lambda_{Ins} = 10 * m = 1.000.000$. Nach der letzten Einfügung findet noch eine Feinordnungsphase gleicher Länge statt. Damit ergeben sich insgesamt

$$8 * 1.000.000 + 1.000.000 = \mathbf{9.000.000}$$

Schleifendurchläufe.

Die Schritte 2 bis 6 sowie 9 werden in jedem Schleifendurchlauf ausgeführt. Pro Schleifendurchlauf fällt somit immer ein Aufwand von

$$1 + 60 + 1 + 1 + 120 + 1 = \mathbf{184\,ZE}$$

an. Eine Einfügung findet $(x_{max} - 2) = 8$ mal statt. Die Anzahl der Löschschritte hängt von der Gesamtzahl Schleifendurchläufe und der Schrittweite für das Löschen λ_{Del} ab. Sie beträgt:

$$\frac{(x_{max} - 1) * \lambda_{Ins} * m}{\lambda_{Del} * m} = \frac{(x_{max} - 1) * \lambda_{Ins}}{\lambda_{Del}} = \mathbf{18}$$

Damit läßt sich der Gesamtaufwand des seriellen Algorithmus bestimmen. Er beträgt:

$$t_{ser} = 9.000.000 * 184 + 8 * 76 + 18 * 37 = \mathbf{1.656.001.274\,ZE} \qquad (33)$$

4.3.2 Experimentraum

Bedingung für eine Parallelisierung im Experimentraum ist eine völlige Unabhängigkeit der Experimente.

Eine Anwendung für diese Art der Parallelisierung ist die Parameterbestimmung. Da hierfür kein analytisches Modell existiert, können sie nur empirisch ermittelt werden. Zur Verkürzung der dafür benötigten Zeit ist ein gleichzeitiges Training mehrerer Netze mit verschiedenen Parametern auf den gleichen Daten möglich.

Dabei ist der Aufwand für das Training eines einzelnen Netzes identisch mit dem des seriellen Algorithmus. Bei e Experimenten entsteht ein Gesamtaufwand von $e * t_{ser}$. Dieser wird auf die n parallelen Prozesse verteilt. Kommunikation zwischen diesen ist nicht notwendig. Der Nutzer muß manuell zu Beginn des Trainings die einzelnen verwendeten Trainigsparameter festlegen und die damit erzielten Ergebnisse begutachten, um die geeignetsten Parameter zu finden.

Für den Speedup Faktor ergibt sich folgende Gleichung:

$$\begin{aligned} s(n) &= \frac{e * t_{ser}}{\frac{e*t_{ser}}{n}} \\ &= \frac{e * t_{ser} * n}{e * t_{ser}} \\ &= n \end{aligned} \tag{34}$$

Diese Art der Parallelisierung erzeugt also einen linearen Speedup. Der Trainingsprozeß der Neuronalen Netze ist aber nur ein Teilschritt bei der Erstellung des Indexbaumes. Er läßt sich nicht aus dem Gesamtalgorithmus herauslösen. Denkbar wäre aber, dieses Verfahren auf den gesamten Aufbauprozeß zu übertragen.

Eine andere Anwendung für diese Art der Parallelisierung ist es, mehrere Datenmengen gleichzeitig zu bearbeiten. Für jedes Training werden allerdings andere Trainingsparameter benötigt, da diese von den Eigenschaften der Datenmenge abhängig sind. Folglich sollte deren Bestimmung auch im parallelen Algorithmus enthalten sein. Dies entspricht dann einer Parallelisierung des Expandierens von Knoten und wird in einem späteren Abschnitt besprochen.

4.3.3 Datenebene

Parallelisierung auf Datenebene bedeutet eine Verteilung der Datenmenge auf parallele Prozesse. Die Teilaufgaben sind aber nicht voneinander unabhängig. Aus den einzelnen Ergebnissen muß ein Gesamtresultat gebildet werden. Für den verwendeten Trainingsalgorithmus sind verschiedene Varianten denkbar.

Eine Möglichkeit besteht darin, in den parallelen Prozessen identische Netzkopien mit verschiedenen disjunkten Teilmengen der Trainingsdaten zu trainieren. Dabei muß während des Trainings die Konsistenz der verteilten Netzkopien gewährleistet sein.

Grundvoraussetzung für identische Netzkopien ist eine Initialisierung mit den gleichen Werten. Diese werden in einem Prozeß erzeugt und dann an alle anderen verteilt.

Während des Trainingsprozesses findet ein synchrones Lernen der Datenvektoren statt. Das heißt, in jedem Lernschritt müssen die Änderungen an jeder Kopie des Netzes ausgeführt werden. In jedem Prozeß wird lokal ein Datenvektor zufällig gewählt, der lokale Gewinner und Zweitbeste ermittelt und die notwendigen Adaptionen an lokalem Fehlermaß, Kantenzähler und Referenzvektoren vorgenommen. Diese Schritte sind unabhängig von den Berechnungen in anderen Prozessen durchführbar.

Danach ist ein Abgleich der Netzkopien notwendig. Dazu sendet jeder Prozeß die lokalen Änderungen an alle anderen und führt eintreffende Änderungen an seiner Netzkopie durch. Nach diesem Abgleich können evtl. notwendige Einfüge- oder Löschschritte wieder unabhängig in den Prozessen erfolgen.

Der Algorithmus für diese Art der Parallelisierung unterscheidet sich durch den Abgleichschritt von Algorithmus 1. Außerdem verringert sich durch die Datenaufteilung die Zahl der Schleifendurchläufe. Die Zahl der Adaptionen in jedem Schritt ist gleich der Anzahl beteiligter paralleler Prozesse.

Die Schrittweite der Einfüge- und Löschschritte ist vom Umfang der Datenmenge abhängig. Den Prozessen wird entweder am Anfang diese Schrittweite oder der Gesamtumfang der Datenmenge mitgeteilt. Da die Anzahl der Schritte zwischen zwei Einfüge- oder Löschschritten nicht unbedingt durch die Anzahl paralleler Prozesse teilbar ist, müssen die Bedingungen in Schritt 7 und 8 angepaßt werden. Eventuell wird die Schrittweite in einem Adaptionsschritt übersprungen. Auch in diesem Fall muß die Einfügung oder Löschung stattfinden.

In die Aufwandsschätzung gehen jetzt zusätzlich zum Berechnungsaufwand des seriellen Algorithmus noch die Zeit für den Abgleichschritt und anfallende Kommunikationszeiten mit ein. Die Daten für Referenzvektoren und Verbindungsstruktur werden jeweils in zwei Arrays gehalten. Im Abgleichschritt versenden die Prozesse Felder, in denen die vorgenommenen Änderungen vermerkt sind. Das Durchführen dieser Änderungen in den einzelnen Prozessen entspricht dann Matrixadditionen.

Jeder Prozeß führt in einem Abgleichschritt $(\overline{x}-1)*2 = \mathbf{10}$ Matrixadditionen durch. Eine Matrix für die Referenzvektoren umfaßt $\overline{x} * l_\xi = \mathbf{60}$ Werte. Eine Verbindung wird durch die Identifikatoren der beteiligten Neuronen und dem Kantenzähler repräsentiert. Für die Verbindungstruktur wird eine $\overline{x} \times \overline{x}$ - Matrix verwendet, die in ihren Elementen die Kantenzähler enthält. Das sind bei den angenommenen Rahmenbedingungen $\overline{x} * \overline{x} = 36$ Werte. Der Berechnungsaufwand für einen Abgleichschritt beträgt dann

$$(60 + 36) * 5 = \mathbf{480\,ZE}$$

Der Berechnungsaufwand der einzelnen Schritte ist noch einmal tabellarisch zusammengefaßt.

1	Wahl des Eingabevektors	1
2	Bestimmung von Gewinner und Zweitbestem	60
3	Verbindungskante anpassen/erzeugen	1
4	lokale Fehlervariable des Gewinnerneurons erhöhen	1
5	Referenzvektoren adaptieren	120
6	Abgleich der Netzkopien	480
7	Neuron einfügen	76
8	Kanten löschen	37
9	Zeitparameter erhöhen	1

Die Schritte 2-6 und 9 werden bei jedem Schleifendurchlauf ausgeführt. Das ergibt pro Schleifendurchlauf ein Berechnungsaufwand von:

$$1 + 60 + 1 + 1 + 120 + 480 + 76 + 37 + 1 = \mathbf{777\,ZE}$$

In jedem Adaptionsschritt werden n Datenvektoren gelernt. Die Schleife wird nur noch

$$\frac{\lambda_{Ins} * m * (x_{max} - 1)}{n} = \frac{10 * 100.000 * 9}{10} = \mathbf{90.000}$$

mal durchlaufen.Die Anzahl der Einfüge- und Löschschritte hängt jeweils von der Anzahl der Adaptionen ab. Sie sind wie beim seriellen Algorithmus 8 und 18. Als Berechnungsaufwand der parallelen Variante ergibt sich damit:

$$90.000 * 777 + 8 * 76 + 18 * 37 = \mathbf{69.931.274\,ZE}$$

Hinzu kommt noch die Kommunikationszeit. Bei jedem Schleifendurchlauf müssen die $2*n = 20$ Matrizen zwischen den Prozessen ausgetauscht werden. Jeder übertragene Wert schlägt mit einer Zeiteinheit zu Buche. Die Matrix für die Referenzvektoren benötigt also 60 und die Matrix der Verbindungsstruktur 36 Zeiteinheiten.

Für den Ablauf der Kommunikation sind verschiedene Vorgehensweisen denkbar. Wenn jeder Prozeß den anderen seine Matrizen direkt übermittelt, so sind

$$n * (n - 1) * (60 + 36) = 5.400 + 3.240 = \mathbf{8.640\,ZE}$$

Kommunikationsoperationen notwendig. Die Prozesse können für diesen Austausch aber auch eine Kette bilden. Bei dieser Vorgehensweise werden die jeweiligen Änderungen nicht direkt bei der Ermittlung der Werte lokal ausgeführt sondern nur in dafür vorgesehene Matrizen gespeichert. Ein ausgewählter Prozeß initiiert den Austausch, indem er die Matrizen mit seinen Änderungen an einen zweiten Prozeß schickt. In der Folge addiert jeder seine lokalen Änderung hinzu und sendet die Matrizen weiter. Dadurch werden

alle lokalen Änderungen aufsummiert und landen beim initierenden Prozeß. In einem zweiten Umlauf erhalten alle die global notwendigen Änderungen und führen sie aus. Für die angenommenen Randbedingungen sind dafür

$$(10 * 60 + 10 * 36) * 2 = 960 * 2 = \mathbf{1.920\,ZE}$$

Kommunikationsoperationen notwendig. Dabei sind aber im ersten Umlauf pro Prozeß außer dem initierenden noch zwei Matrixadditionen notwendig.

$$9 * (60 + 36) = \mathbf{864\,ZE}$$

Beim Vergleich ist die Kette deutlich besser. Sie benötigt

$$1.920 + 864 = \mathbf{2.784\,ZE}$$

in jedem Schleifendurchlauf.

Der Kommunikationsaufwand beträgt bei 90.000 Schleifendurchläufen

$$t_{comm} = 90.000 * 1.920 = \mathbf{172.800.000\,ZE}$$

und der Berechnungsaufwand

$$t_{comp} = 69.931.274 + 90.000 * 864 = \mathbf{147.691.274\,ZE}$$

Der Algorithmus hat einen Gesamtaufwand von

$$t_{comp} + t_{comm} = 147.691.274 + 172.800.000 = \mathbf{320.491.275\,ZE} \tag{35}$$

In die Gleichung für den Speedup Faktor eingesetzt ergibt sich

$$\begin{aligned} s(n) &= \frac{t_{ser}}{t_{comp} + t_{comm}} \\ &= \frac{1.656.001.274}{147.691.274 + 172.800.000} \\ &\approx \mathbf{5.17} \end{aligned} \tag{36}$$

Um die notwendige Kommunikation zu minimieren wäre auch denkbar, die Teilmengen mit verschiedenen Netzen zu clustern und die Teilergebnisse erst am Ende zusammenzufassen. Das verringert zwar die nötige Kommunikation während des Trainings. Die korrekte Zusammensetzung der Teilnetze ist jedoch nach meinen Erkenntnissen nicht möglich.

Die Aufteilung der Daten kann nur willkürlich geschehen, da a priori kein Wissen über die Struktur der Datenmenge vorliegt. So können Datencluster auf verschiedene Prozesse aufgeteilt sein. In der Folge gehören verschiedene Teilnetze zu demselben Cluster der Eingangsdaten. Beim Zusammensetzen

am Ende des Trainings müsste diese Zusammengehörigkeit erkannt werden. Dies ist aber nur mit großem Aufwand bei sich schneidenden ZHK möglich.

Außerdem ist eine ungleichmäßige Aufteilung der Daten eines Clusters auf die parallelen Prozesse wahrscheinlich. Dadurch kann der Cluster in einem Prozeß gegenüber den restlichen lokalen Daten unterrepräsentiert sein. Im Laufe des Trainings werden deshalb für ihn keine Neuronen erzeugt. Beim Zusammensetzen der Teilnetze würde dieser Teil des Clusters nicht mehr erkannt.

4.3.4 Netzebene

Hierbei werden die Neuronen eines Netzes auf mehrere parallele Prozesse aufgeteilt. Damit ist die Netzstruktur über Prozeßgrenzen hinweg verteilt. Änderungen der Netzstruktur müssen demzufolge synchron ablaufen.

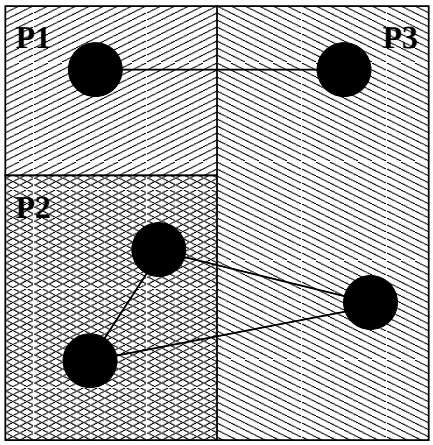


Abbildung 9: Aufteilung eines Netzes auf die Prozesse P1, P2 und P3

Der Berechnungsaufwand wird zwar auf die parallelen Prozesse verteilt. Die Erhaltung der korrekten Netzstruktur ist aber schwierig. Im Folgenden wird ein möglicher Algorithmus beschrieben.

Alle Prozesse trainieren dasselbe Netz. Die Reihenfolge, in der die Eingabevektoren gelernt werden, ist damit für alle Prozesse zwingenderweise gleich. Die Prozesse verfügen alle über die vollständige Datenmenge. Ein Prozeß trifft die zufällige Auswahl eines Vektors und informiert die anderen Prozesse über den entsprechenden Tupelidentifikator.

Diese Kommunikation läßt sich parallelisieren. Der Prozeß, der den Tupelidentifikator erzeugt hat, sendet ihn in einem ersten Schritt an einen zweiten

Prozeß. Dieser kann dann parallel die Tupel-ID ebenfalls an andere Prozesse weiterreichen.

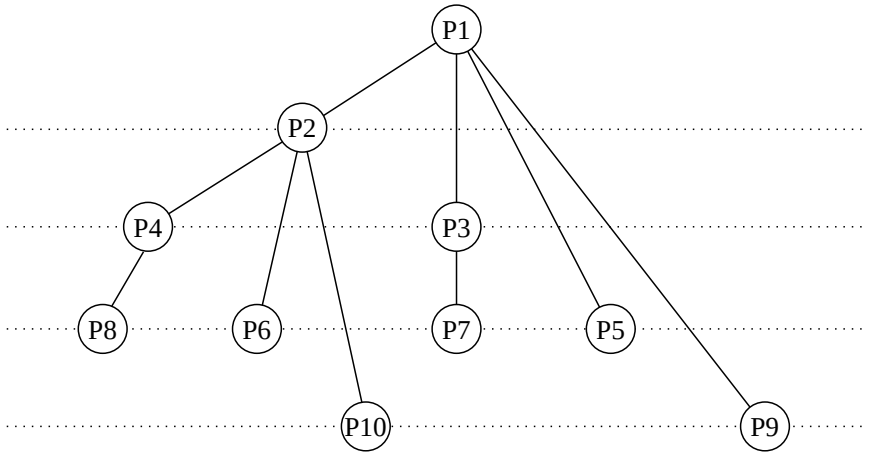


Abbildung 10: parallele Kommunikation in einer Baumstruktur

Die Parallelität der Kommunikation erhöht sich in jedem Schritt, da immer mehr Prozesse den Tupelidentifikator besitzen und weitergeben können. Bei den angenommenen 10 parallelen Prozessen benötigt dieses Schneeballprinzip nur vier Zeiteinheiten um alle zu erreichen. Die Zeit für die zufällige Auswahl eines Datenvektors beträgt zusammen:

$$1 + 4 = \mathbf{5}$$

Mit dem gewählten Eingabevektor können die Prozesse ihre jeweiligen lokalen Gewinnerneuronen und Zweitbesten ermitteln. Der Aufwand hierfür beträgt:

$$\frac{\overline{x} * l_{\xi}}{n} = \frac{6 * 10}{10} = \mathbf{6\ ZE}$$

Danach ist eine Abstimmung über die globalen Sieger notwendig. Auch hierfür ist eine baumförmige Kommunikationsstruktur einsetzbar.

Die Kommunikation erfolgt in zwei Durchläufen. Zuerst werden die lokalen Gewinner von den Blättern beginnend nach oben propagiert. Die Prozesse schicken dazu die Neuronenidentifikatoren und die dazugehörigen Abstände zwischen Referenz- und Eingabevektor von lokalem Gewinnerneuron und Zweitbestem nach oben zum im Baum darüberliegenden Prozeß. Dieser vergleicht die erhaltenen mit den eigenen lokalen Abstandswerten und reicht die geringeren Werte nach oben weiter. In der Wurzel ergeben sich so die Abstandswerte von globalem Gewinnerneuron und Zweitbestem. Im zweiten Schritt werden diese mit den dazugehörigen Neuronenidentifikatoren in umgekehrter Richtung nach unten propagiert.

Die Kommunikation benötigt 4 Schritte aufwärts und 4 abwärts. Dabei werden bei jeder Kommunikationsoperation jeweils 4 Werte (Identifikatoren, Abstandswerte für 2 Neuronen) ausgetauscht. Damit ergeben sich insgesamt 32 Zeiteinheiten für die eigentliche Kommunikation.

Beim Aufstieg im Baum sind noch Vergleichsoperationen notwendig. Es werden dabei jeweils 2 Wertepaare betrachtet. Die Operationen sind aber teilweise parallel. Es wird deshalb pro Schritt mit einem Vergleich gerechnet. Insgesamt ergibt sich so für die globale Abstimmung ein Zeitaufwand von

$$4 * 4 + 2 * 4 + 4 * 4 = \mathbf{40\,ZE}$$

Nachdem allen Prozessen die Gewinner bekanntgemacht wurden, können die Schritte zur Adaption der Referenzvektoren und Kantenzähler in den Prozessen lokal erfolgen. Auch Kantenlöschungen sind autonom durchführbar. Diese Schritte entsprechen bis auf die kleinere Neuronenzahl dem seriellen Algorithmus.

Schwieriger sind Einfügeschritte. Zuerst ist eine Bestimmung des Neurons und des entsprechenden topologischen Nachbars mit dem größten lokalen Fehlermaß erforderlich. Diese können aber in verschiedenen Prozessen liegen. Neben den Identifikatoren der Neuronen benötigt jeder Prozeß zur Bestimmung der topologischen Nachbarschaft die gesamte Verbindungsstruktur, welche die von ihm verwalteten Neuronen betrifft.

Die Prozesse stimmen erst über das Neuron mit dem größten lokalen Fehler ab. Die Kommunikation läuft analog der Ermittlung von globalem Gewinnerneuron und Zweibestem in einer baumförmigen Ordnung ab. Diesmal werden aber nur zwei Werte ausgetauscht (Neuronenidentifikator + Fehlermaß) und es ist nur ein Vergleich erforderlich. Der Zeitaufwand beträgt damit:

$$4 * 2 + 4 * 1 + 4 * 2 = \mathbf{20\,ZE}$$

Wenn dieses Neuron ermittelt und allen bekannt ist, wird bei seinen topologischen Nachbarn der größte lokale Fehler gesucht.

Eine Aufteilung anhand der ZHK würde die nötigen Adaptionen auf einen Prozeß beschränken. Wie bereits erwähnt, muß aber die Möglichkeit beachtet werden, daß ZHK im Laufe des Trainings auch wieder zusammenwachsen können. Folglich muß auch hierbei ein globaler Gewinner und ein Zweitnächster bestimmt werden, da diese ja in verschiedenen ZHK liegen können. Ein mögliches Verschmelzen von ZHK impliziert eine Neuaufteilung der Neuronen unter den Prozessen. Auch hier ist ein synchrones Abarbeiten der Datenvektoren nötig. Dies verringert wieder die Parallelität des Algorithmus auf das Suchen der jeweiligen lokalen Gewinner und Zweitnächsten.

4.4 Aufteilung der Vektoren auf die Cluster

Eine Möglichkeit der Parallelisierung ist die Verteilung des Berechnungsaufwandes anhand von Schleifen. Dabei arbeitet jeder Prozeß nur einen Teil der Schleife ab. Je nach Art der Aufteilung und der Berechnungen innerhalb der Schleife kann eine Zusammenfassung der Teilergebnisse am Ende notwendig sein. Die Aufteilung der Datenvektoren geschieht mit folgendem Algorithmus:

Algorithmus 5 divData

```
for all zhk do
  D'' = ∅
  for all c ∈ zhk do
    for all ξ ∈ D' do
      if s(ξ) = c then
        D'' = D'' ∪ {ξ}
      end if
    end for
  end for
  CL = CL ∪ {D''}
end for
```

Vor der Zuordung der Datenvektoren sind die ZHK des Netzes bereits ermittelt. Alle Neuronen einer solchen ZHK erhalten denselben Identifikator. Die Zuordnung der Datenvektoren geschieht durch Übertragung dieser ID auf die Vektoren. Das entspricht genau einem Wertzugriff pro Einordnung. Zur Ermittlung des jeweiligen Gewinnerneurons muß der Eingabevektor mit allen Referenzvektoren verglichen werden. Da die Zuordnung nach abgeschlossenem Training passiert, sind das $x_{max} = 10$ Stück. Durch die Vektorlänge von $l_\xi = 10$ ergibt sich pro Eingabevektor ein Aufwand von $10 * 10 = 100$. Jeder Datenvektor muß einem Cluster zugeordnet werden. Bei $m = 100000$ Datenvektoren ergibt sich ein Gesamtaufwand für den seriellen Algorithmus von:

$$(1 * 100.000) + (10 * 10 * 100.000) = \mathbf{10.100.000\,ZE} \tag{37}$$

In Algorithmus 5 gibt es drei ineinander verschachtelte Schleifen, die aufgeteilt werden können. Eine Parallelisierung der innersten Schleife bedeutet, daß jeder Prozeß nur einen Teil der Datenmenge erhält. Um aufwendige globale Abstimmungen über die aktuelle ZHK und das jeweilige aktuelle Neuron zu vermeiden, liegen entsprechend auch identische Kopien der Netzstruktur vor. So können die Vektoren der jeweiligen lokalen Teildatenmenge in unabhängigen Berechnungen in die Cluster eingeordnet werden. Der Aufwand

dafür entspricht dem des seriellen Algorithmus geteilt durch die Anzahl paralleler Prozesse.

$$\frac{10.100.000}{10} = \mathbf{1.010.000\,ZE}$$

Jeder Prozeß ermittelt Teilmengen der enthaltenen Cluster. Diese Teilmengen müssen dann am Ende global zusammengefaßt werden. Die Teilmengen eines Clusters haben in allen Prozessen den gleichen Identifikator und können so problemlos zusammengefaßt werden. Dabei kommt die parallele Kommunikation in der baumförmigen Struktur zum Einsatz. Damit werden 4 Schritte benötigt. Die inneren Knoten vereinigen ihre lokale Datenmenge mit den jeweils erhaltenen Mengen und schicken die Gesamtmenge weiter nach oben. Damit erhöht sich natürlich das transferierte Datenvolumen mit der Höhe im Baum. Für die gesamte Kommunikation werden

$$1 * 10.000 + 2 * 10.000 + 3 * 10.000 + 3 * 10.000 = \mathbf{90.000\,ZE}$$

benötigt. Insgesamt ergibt sich damit für den Speedup Faktor

$$s(n) = \frac{10.100.000}{\frac{10.100.000}{10} + 90.000} \approx \mathbf{9.18} \tag{38}$$

Sowohl die Parallelisierung der Schleife über alle Neuronen als auch der über alle ZHK entspricht einer Aufteilung der Neuronen des Netzes. Die Prozesse erhalten die vollständige Datenmenge und einen Teil der Neuronen. Das bringt einen hohen Synchronisations- und Kommunikationsaufwand bei der Ermittlung der globalen Gewinnerneuronen mit sich. Zur Entkopplung können die Prozesse ihre lokalen Gewinner bestimmen und den Identifikator mit dem Abstandsmaß an den Datenvektoren vermerken. Für diese Berechnungen entsteht ein Aufwand von

$$\frac{x_{max}}{n} * m * l_\xi = \mathbf{1.000.000\,ZE}$$

Am Ende werden die Datenmengen dann zusammengefaßt. Dies beinhaltet Vergleiche der Abstandsmaße und eine Kennzeichnung mit dem entsprechenden Identifikator. Durch die baumförmige Kommunikation benötigt dieser Vorgang wieder 4 Schritte. Um eine Datenmenge mit den korrekten Unterteilungen zu erhalten, sind 9 Vergleiche und entsprechend 9 Kommunikationsvorgänge notwendig. Für die Zusammenfassung zweier Datenmengen müssen die Abstandswerte aller Vektoren verglichen werden. Da die Zusammenfassungen teilweise parallel erfolgen, ergibt sich ein Aufwand von $4{*}100.000 = 400.000\,ZE$. Bei jeder Kommunikation wird die gesamte Datenmenge mit den Identifikatoren und Abstandsmaßen übertragen. Insgesamt ergibt sich für den Speedup Faktor die Gleichung:

$$s(n) = \frac{10.100.000}{(\frac{x_{max}}{n} * m * l_\xi) + (4 * m) + ((l_\xi + 2) * m * 4)} = \frac{10.100.000}{6.200.000} \approx \mathbf{1.63} \tag{39}$$

Durch den Aufwand für die Zusammenfassung der Teilergebnisse bei einer Parallelisierung auf Netzebene erscheint eine Aufteilung der Datenmenge als günstiger. Dort sind kleinere Datenmengen auszutauschen und eine Mengenvereinigung ist auch weniger aufwendig als der Vergleich aller Abstandsmaße.

4.5 Bestimmung der Hyperbox

Die Datensätze werden durch Vektoren repräsentiert, die in den Komponenten die jeweiligen Ausprägungen der Attribute enthalten. Wenn man die Vektoren räumlich betrachtet, so wird die umschließende Hyperbox eines Clusters durch die am weitesten entfernt liegenden Endpunkte der Vektoren bestimmt. Zu deren Ermittlung dient Algorithmus 6.

Algorithmus 6 Hyperbox

```
Initialisierung von min mit einem Vektor ξ ∈ D
Initialisierung von max mit einem Vektor ξ ∈ D
for all ξ ∈ D do
  for i = 0; i < ξ.length do
    if ξ.i < min.i then
      min.i = ξ.i
    end if
    if ξ.i > max.i then
      max.i = ξ.i
    end if
  end for
end for
```

Die Initialisierung benötigt 20 Zeiteinheiten.

$$2 * l_\xi = \mathbf{20\,ZE}$$

Danach werden alle Datenvektoren mit den beiden Vektoren min und max verglichen.

$$2 * l_\xi * m = \mathbf{2.000.000\,ZE}$$

Insgesamt benötigt der serielle Algorithmus:

$$2 * l_\xi + 2 * m * l_\xi = \mathbf{2.000.020\,ZE} \tag{40}$$

Eine Möglichkeit der Parallelisierung ist die Aufteilung der gesuchten Vektoren min und max. Die Prozesse erhalten die vollständige Datenmenge, betrachten aber in der inneren Schleife nur einen Teil der Vektoren. Bei 10 parallelen Prozessen beträgt der zeitliche Aufwand für die Vergleichsoperationen:

$$\frac{l_\xi}{n} * m * 2 = \mathbf{20.000\,ZE}$$

Am Ende werden aus den Teilvektoren für *min* und *max* die Gesamtvektoren zusammengefaßt. Die Kommunikation geschieht in einer baumförmigen Struktur von unten nach oben. Die Prozesse in den Blattknoten schicken jeweils zwei vollständige Vektoren nach oben, die in allen Komponenten außer der lokal berechneten den Wert 0 enthalten. In den inneren Knoten erfolgt eine Vektoraddition, so daß in den bis dahin bereits ermittelten Komponenten die richtigen Werte stehen. Die Zusammenfassung erfolgt in 4 Schritten. Die benötigte Kommunikationszeit beträgt:

$$4 * l_\xi = \mathbf{40\,ZE}$$

Die Vektoradditionen ergeben ebenfalls einen Aufwand von:

$$4 * l_\xi = \mathbf{40\,ZE}$$

Für den Speedup Faktor ergibt sich somit:

$$\begin{aligned} s(n) &= \frac{2.000.020}{\frac{l_\xi}{n} * m * 2 + 4 * l_\xi + 4 * l_\xi} \\ &= \frac{2.000.020}{200.080} \\ &\approx \mathbf{10} \end{aligned} \tag{41}$$

Bei einer Aufteilung der Datenmenge benötigt jeder Prozeß weniger Vergleiche zur Bestimmung der lokalen Vektoren *min* und *max*. Der Aufwand dafür beträgt:

$$2 * l_\xi * \frac{m}{n} = \mathbf{200.000\,ZE}$$

Bei der Zusammenfassung werden die jeweils ermittelten Vektoren übertragen. Dadurch entsteht ein Kommunikationsaufwand von

$$4 * 2 * l_\xi = \mathbf{80\,ZE}$$

In den inneren Knoten sind Vergleiche der Vektorelemente erforderlich. Dies ergibt einen Aufwand von

$$4 * 2 * *l_{xi} = \mathbf{80\,ZE}$$

Insgesamt kostet die Zusammenfassung der Teilergebnisse:

$$4 * 2 * l_\xi + 4 * 2 * l_\xi = \mathbf{160\,ZE}$$

Für den Speedup Faktor ergibt sich so ein Wert von:

$$s(n) = \frac{2.000.020}{2 * l_\xi + 2 * l_\xi * \frac{m}{n} + 4 * 4 * l_\xi} \approx \mathbf{9.99} \tag{42}$$

Außerdem kann die Ermittlung von *min* und *max* getrennt voneinander erfolgen. Die n parallelen Prozesse werden in zwei Gruppen aufgeteilt, von denen jede jeweils einen der beiden Vektoren ermittelt. Innerhalb der Gruppen kann eine der beiden vorher genannten Möglichkeiten zur Parallelisierung angewandt werden. Wenn sich die Prozesse in den Gruppen die Erzeugung der Vektoren teilen, ergibt sich ein Aufwand von

$$\frac{l_\xi}{n/2} * m + 4 * l_\xi + 4 * l_\xi = \mathbf{200.080\,ZE}$$

Eine Aufteilung der Datenmenge halbiert ebenfalls den nötigen Aufwand für die Vergleichsoperationen. Deshalb ist die benötigte Zeit identisch zur Vektorenaufteilung.

$$l_\xi * \frac{m}{n/2} + 4 * l_\xi + 4 * l_\xi = \mathbf{200.080\,ZE}$$

Bei beiden Varianten entsteht gleichviel Aufwand. Damit sind auch die Speedup Faktoren identisch:

$$\begin{aligned} s(n) &= \frac{2.000.020}{200.080} \\ &\approx \mathbf{10} \end{aligned} \tag{43}$$

4.6 Rekursiver Aufruf des Expandierens

Solange die jeweils repräsentierte Datenmenge noch zu groß ist, werden die gefundenen Söhne mit dem Algorithmus zum weiteren Aufbau des Baumes (Alg. 3) erweitert. Dies entspricht einem rekursiven Abstieg in den Baum (je nach Reihenfolge der Abarbeitung analog Breiten- oder Tiefensuche). Die Aufteilung dieser Rekursion ist eine Möglichkeit der Parallelisierung.

In einem System mit ausreichend parallelen Prozessen könnte jeder Aufruf von Expand(*node*) von einem neuen Prozeß übernommen werden. Der Aufbau des Baumes findet dann in einem gemeinsamen Speicherbereich statt. In gewöhnlichen Systemen wird dies aufgrund der begrenzten Prozessoranzahl nicht möglich sein.

Wie in Kapitel 3.5.2 beschrieben, verwendet der ICIx eine Warteschlange für Knoten, die noch zur Bearbeitung anstehen. Für jeden der Einträge darin wird ein Expand gestartet, welches evtl. neue Einträge hinzufügt. Die Rekursion kommt durch das wiederholte Einfügen und Bearbeiten der Einträge zum Ausdruck. Die Behandlung der einzelnen Knoten ist unabhängig. Deshalb erscheint eine Parallelisierung bei der Abarbeitung der Warteschlange als lohnenswert.

Bei den weiteren Betrachtungen wird von einer Datenmenge ausgegangen, die 16 Cluster enthält. Diese liegen so, daß zuerst eine Aufteilung in zwei Cluster erfolgt. Beide werden im nächsten Schritt jeweils in 4 Teilmengen geteilt. Alle Aufteilungen sollen Teilmengen gleicher Größe ergeben. Bei den so angenommenen Verhältnissen entstehen Datenmengen von 12.500 Vektoren. Unter der Annahme, daß eine Datenbankseite nur 7.000 Datensätze aufnehmen kann, muß eine weitere Unterteilung erfolgen. Die Blätter des Baumes repräsentieren so 6.250 Sätze. Der Indexbaum hat eine Tiefe von 3. Es ergeben sich folgende Clusterungen:

Anzahl	Datenumfang
1	100.000
2	50.000
8	12.500

Ein zentraler Prozeß verwaltet die Warteschlange der weiter zu bearbeitenden Knoten und baut den Indexbaum auf. Jeder Aufruf des Algorithmus 3 zum weiteren Unterteilen einer Datenmenge wird an einen arbeitsbereiten Rechenprozeß delegiert. Dieser nimmt die nötige Clusterung vor und liefert das entsprechende Ergebnis zurück. Somit kann der iterative Aufbau des Baumes in parallelen Prozessen erfolgen.

Der in den Rechenprozessen verwendete Algorithmus zur Clusterung arbeitet seriell. Dadurch sind die benötigten Gesamtzeiten für alle Clusterungen in serieller und paralleler Ausführung anhand der Datensatzanzahlen vergleichbar. Pro Clusterung fallen die Zeiten für das Training des Netzes, der Vektorzuordnung und der Hyperboxbestimmung an. Aus den Betrachtungen von Kapitel 4.3.1 kann eine datenabhängige Formel für die Laufzeit des Trainingsprozesses abgeleitet werden:

$$\begin{aligned} g(m) &= 24 + 9 * m * (1 + 60 + 1 + 1 + 120 + 1) + 8 * 76 + 18 * 37 \\ &= 9 * m * 184 + 24 + 608 + 666 \\ &= \mathbf{m * 1.656\,ZE + 1.298\,ZE} \end{aligned}$$

Aus Kapitel 4.4 ergibt sich die entsprechende Formel für die Bestimmung der Vektorzuordnung:

$$\begin{aligned} v(m) &= x_{max} * l_\xi * m + 1 * m \\ &= \mathbf{m * 101\,ZE} \end{aligned}$$

Die Formel für die Bestimmung der Hyperboxen lautet:

$$\begin{aligned} h(m) &= 2 * l_\xi + 2 * m * l_\xi \\ &= \mathbf{m * 20\,ZE + 20\,ZE} \end{aligned}$$

Zusammenfassend entsteht Aufwand für eine Clusterung in Höhe von:

$$g(m) + v(m) + h(m) = \mathbf{m * 1.777\,ZE + 1.318\,ZE}$$

Die Tabelle der notwendigen Clusterungen läßt sich mit dieser Formel um die notwendigen Aufwände bei den verschiedenen Datenmengen ergänzen.

Anzahl	Datenumfang	Aufwand in ZE
1	100.000	177.701.318
2	50.000	88.851.318
8	12.500	22.213.818

Für den seriellen Algorithmus ergibt sich so Berechnungsaufwand von

$$\begin{aligned} & 1 * (1.777 * m + 1.318) \\ + \quad & 2 * (1.777 * \frac{m}{2} + 1.318) \\ + \quad & 8 * (1.777 * \frac{m}{8} + 1.318) \\ = \quad & 5.331 * m + 14.498 \\ = \quad & \mathbf{533.114.498\,ZE} \end{aligned}$$

Beim parallelen Algorithmus werden die Clusterungen einer Stufe parallel ausgeführt. Damit ergibt sich bei ihm ein Aufwand von:

$$\begin{aligned} & 1 * (1.777 * m + 1.318) \\ + \quad & 1 * (1.777 * \frac{m}{2} + 1.318) \\ + \quad & 1 * (1.777 * \frac{m}{8} + 1.318) \\ = \quad & \frac{23.101 * m}{8} + 3.954 \\ = \quad & \mathbf{288.766.454\,ZE} \end{aligned}$$

Zur Erhaltung der Vater-Sohn-Beziehung muß der Pagekey des Vaters immer mitgeführt werden. Somit werden nicht nur die Datenvektoren kommuniziert, sondern auch der Schlüssel des jeweiligen Vaterknotens. Zur Initialisierung einer Clusterung werden dem Rechenprozeß $m * l_\xi + 1$ Werte geschickt. Im Ergebnis besitzen alle Datenvektoren zusätzlich den Identifikator des zugeordenten Clusters, also $m * (l_\xi + 1) + 1$ Werte. Zusammen werden für eine Clusterung

$$m + l_\xi + 1 + m * (l_\xi + 1) + 1 = 21 * m + 2$$

Werte ausgetauscht. Auch diese Werte sind tabellarisch dargestellt.

Anzahl	Datemumfang	Kommunikationsumfang
1	100.000	2.100.002
2	50.000	1.050.002
8	12.500	262.502

Der Kommunikationsaufwand beträgt zusammengefaßt für alle Clusterungen:

$$\begin{aligned} & 1 * 2.100.002 \\ + \; & 2 * 1.050.002 \\ + \; & 8 * 262.502 \\ = \; & \mathbf{6.300.022ZE} \end{aligned}$$

Damit ergibt sich für den Speedup Faktor

$$\begin{aligned} s(n) &= \frac{533.114.498}{288.766.454 + 6.300.022} \\ &= \frac{533.114.498}{295.066.476} \\ &\approx \mathbf{1.8} \end{aligned} \tag{44}$$

Der Verwaltungsprozeß benötigt zusätzlich ein Mittel, um arbeitsbereite Prozesse zu verwalten. Auch dafür kann eine Warteschlange verwendet werden. Alle Rechenprozesse melden sich nach der Initialisierung beim zentralen Dispatcher. Dieser vermerkt sie mit einem eindeutigen Identifikator in seiner Warteschlange. Zur Identifikation kann beispielsweise der vom System vergebene Prozeßidentifikator (pid) verwendet werden. Solange noch Datenmengen zu clustern sind und Rechenprozesse warten, nimmt der Verwaltungsprozeß die jeweils ersten Einträge der Warteschlangen und schickt dem entsprechenden Prozeß die Daten. Dieser nimmt die Clusterung vor und liefert das Ergebnis zurück. Damit meldet er sich gleichzeitig wieder arbeitsbereit. Der Dispatcher nimmt die notwendigen Einträge in den Warteschlangen vor und arbeitet die Ergebnisse der Clusterung in den Baum ein.

Der zentrale Verwaltungsprozeß benötigt also folgende Strukturen zum Handling der zu clusternden Datenmengen und parallelen Rechenprozesse:

```
queue<pid> prozesslist
struct item{
            pk Vater
            Datenmenge}
queue<item> waitqueue
```

Algorithmus 7 dispatch Expand

```
while waitqueue not empty do
  current = prozesslist.next
  send(current, waitqueue.next)
  for all ξ from worker do
    enqueue(ξ, waitqueue)
  end for
end while
```

Die Rechenprozesse können vollkommen unabhängig voneinander arbeiten. Kommunikation findet jeweils nur zwischen Dispatcher und Rechner statt. Die Rechenprozesse erhalten alle zu Beginn die vollständige Datenmenge. So müssen während des Aufbaus nur noch die Tupelidentifikatoren (tid) ausgetauscht werden.

Die Kommunikation zwischen Rechen- und Verwaltungsprozeß umfaßt beim Initialisieren der Clusterung die Tupelidentifikatoren der zu clusternden Datenmenge und den Pagekey des Vaterknotens. Dieselbe Menge an Daten wird als Ergebnis zurückgeliefert, mit dem Unterschied daß die Daten diesmal entsprechend der gefundenen Clusterung strukturiert sind.

Von Vorteil ist die weitgehende Unabhängikeit der Prozesse. Denkbar wäre, daß die Rechenprozesse die gefundenen Cluster selbst weiterbearbeiten. Hier müssen temporäre Identifikatoren verwendet werden, da Pagekeys nur vom Verwaltungsprozeß vergeben werden. Diese müssen zwingend innerhalb des Baumes eindeutig sein.

5 Gewählter Ansatz

Im Kapitel 4 sind verschiedene Möglichkeiten zur Parallelisierung genannt worden. Das sind

1) Trainingsprozeß der Neuronalen Netze
   - Experimentraum
   - Datenebene
   - Netzebene
2) Aufteilung der Vektoren auf die Cluster
3) Bestimmung der Hyperboxen
4) Rekursiver Aufruf des Expandierens

Unter den in Kapitel 4.1 genannten Gesichtspunkten sollte die Erzeugung des Baumes in große, weitgehend unabhängige Teilaufgaben zerlegt werden. Jede dieser Berechnungen umfaßt dann einen großen Teil des Gesamtalgorithmus und die aufwendige Initialisierung der parallelen Prozesse ist weniger häufig notwendig. Wenn man die untersuchten Parallelisierungsmöglichkeiten betrachtet, besitzen diese teilweise gute Werte für den Speedup Faktor. Doch sowohl der Trainingsprozeß der Neuronalen Netze, die Aufteilung der Vektoren nach der Clusterung als auch die Bestimmung der Hyperboxen sind nur kleine Teilschritte bei der Erweiterung eines Baumknotens. Gegen den verbleibenden seriellen Anteil der Berechnungen ist der parallelisierbare Aufwand gering.

So bleiben noch die Parallelisierung im Experimentraum bezogen auf den gesamten Aufbaualgorithmus, also ein paralleler Aufbau mehrerer Indexe über derselben Datenmenge mit verschiedenen Parametern, und die Aufteilung der Rekursion. Beide Vorgehensweisen erzeugen große unabhängige Teilaufgaben. Eine Parallelisierung der rekursiven Aufrufe wirkt sich direkt auf den Algorithmus zum Indexaufbau aus. Sie wirkt also auch, falls während der Nutzung des Indexes aufgrund von Änderungen am Datenbestand eine teilweise Restrukturierung des Indexbaumes erforderlich ist. Da die Parallelisierung im Experimentraum hauptsächlich bei der initialen Parameterbestimmung Verwendung findet, wird hier die Aufteilung der Rekursion untersucht. Die Durchführung mehrerer paralleler Experimente erzeugt unabhängigere Teilaufgaben. Diese Möglichkeit der Parallelisierung sollte deshalb in zukünftigen Arbeiten untersucht werden.

Ein Rekursionsschritt besteht aus den Teilen:

1. Clusterung der Datenmenge

2. Einreihen der entstandenen Cluster in eine Warteschlange
3. Anhängen von neuen Knoten im Baum

Die Berechnungen, die innerhalb eines Rekursionsschrittes ausgeführt werden, sind in sich geschlossen, d.h., sie sind in einem Prozeß gekapselt ohne Interprozeßkommunikation durchführbar. Abhängigkeiten entstehen durch den zu erstellenden Indexbaum. Dieser muß global verwaltet werden. Genauer betrifft das die Baumstruktur und die Datenmengen, die noch weiter unterteilt werden müssen.

Im Rahmen dieser Arbeit kam für den Algorithmus 3 Expand(*node*) ein serieller zentraler Verwaltungsprozeß zum Einsatz. Dort wird der Baum aufgebaut und die Warteschlange der weiter zu bearbeitenden Datenmenge verwaltet. Für die eigentliche Clusterung schickt er die Daten an einen von mehreren parallelen Rechenprozessen. Diese verwenden den Algorithmus 4 Clusterung(D') zur Unterteilung und schicken die so strukturierte Datenmenge wieder zurück. Der zentrale Verwaltungsprozeß erzeugt daraus die notwendigen Einträge im Indexbaum und ordnet zu große Datenmengen in die Warteschlange ein.

Da die Änderungen nur die eigentliche Clusterung betreffen, konnten die bestehenden Algorithmen und Strukturen aus [NG99] weitgehend übernommen werden. Auf die notwendigen Erweiterungen für die Verwaltung der globalen Warteschlange und der Rechenprozesse sowie der Interprozeßkommunikation wird im nächsten Abschnitt eingegangen.

5.1 Strukturen

Die Strukturen für den aufzubauenden Baum wurden ungeändert aus [NG99] übernommen.

Cluster, bei denen weitere Unterteilungen nötig sind, werden in eine Warteschlange eingereiht. Zur eigentlichen Clusterung werden die Daten an einen Rechenprozeß geschickt. Dieser liefert das Resultat am Ende seiner Berechnungen wieder zurück. Da mehrere Prozesse gleichzeitig Datenmengen unterteilen und die entstehenden Clusterungen an den Verwaltungsprozeß zurücksenden, ist ein Mittel zur Repräsentation der Vater-Sohn-Beziehung notwendig. Dazu wird der Pagekey des jeweiligen Vaterknotens immer mitgeführt. Das heißt, in der Warteschlange ist zusätzlich zu den Tupelidentifikatoren der Datensätze der Pagekey gespeichert. Er wird mit an den Rechenprozeß geschickt und ist so auch im zurückgelieferten Ergebnis enthalten. Der Verwaltungsprozeß kann so entscheiden, unter welchem Knoten die neu gefundenen Cluster anzuordnen sind.

Neben dem Baum und den weiter zu bearbeitenden Datenmengen sind auch die Rechenprozesse zu verwalten. Dazu existiert eine Queue, in der arbeitsbereite Rechenprozesse vermerkt werden. Durch das 'First in - First out' Prinzip erhalten die Prozesse in der Reihenfolge ihrer Bereitschaftsmeldung wieder neue Datenmengen.

Das ICIx System wurde in [NG99] objektorientiert aufgebaut. Das heißt, die Algorithmen und Strukturen zur Clusterung sind in dem Baum selbst enthalten. Damit die Rechenprozesse diese verwenden können, legen diese einen "Baumstumpf" an. In diesem werden auch die Trainingsparamenter wie z.B. maximale Anzahl Söhne pro Knoten gespeichert.

Für die Kommunikation zwischen den Prozessen wurde eine eigene Klasse geschaffen. Diese wickelt die Übertragung der gewünschten Daten ab. Die Algorithmen enthalten an den entsprechenden Stellen Methodenrufe dieser Klasse. Diese erledigen die Codierung und Decodierung der Daten und die eigentliche Kommunikation. In diesen Operationen findet auch die Synchronisation zwischen Sender und Empfänger statt. So wird in den Rechenprozessen eine Operation zum Empfang neuer Daten gestartet. Die Prozesse schlafen solange, bis ein neues Datenpaket eintrifft.

5.2 Algorithmus

Der zentrale Verwaltungsprozeß initialisiert den Baum sowie die ToDo-Liste und wartet auf Meldung der Rechenprozesse. Bei der Initialisierung eines Rechenprozesses wird ein Speicher für die Daten angelegt. Anschließend meldet er sich beim Dispatcher arbeitsbereit und wartet auf eintreffende Aufgaben. Sobald er ein Datenpaket zur Clusterung erhält, führt er den Algorithmus zur Clusterung der Daten (Algorithmus 4) darauf aus und schickt die entstandene Clusterung zurück.

Sobald sich alle Rechenprozesse gemeldet haben, beginnt die Hauptschleife des Verwaltungsprozesses. In dieser werden Datenpakete an wartende Prozesse verschickt, der Empfangsspeicher geprüft, Ergebnisse in Baum und ToDo-Liste eingefügt und arbeitsbereite Prozesse in einer Warteschlange vermerkt. Am Ende jedes Schleifendurchlaufes wird geprüft, ob weitere Clusterungen notwendig sind.

In Abbildung 11 ist die Interaktion zwischen den Prozessen dargestellt. P1 ist der zentrale Verwaltungsprozeß. P2 und P3 sind zwei parallele Rechenprozesse. Die Kommunikationen zwischen den Prozessen sind durch die Pfeile dargestellt. Zeiten, in denen die Prozesse arbeiten, sind mit durchgezogenen Linien gekennzeichnet. Gepunktete Linien stellen Wartezeiten dar.

Zuerst erfolgt bei allen Prozessen die Initialisierung. Der Verwaltungsprozeß richtet die Strukturen für den Baum, der aufgebaut werden soll, ein. Die Re-

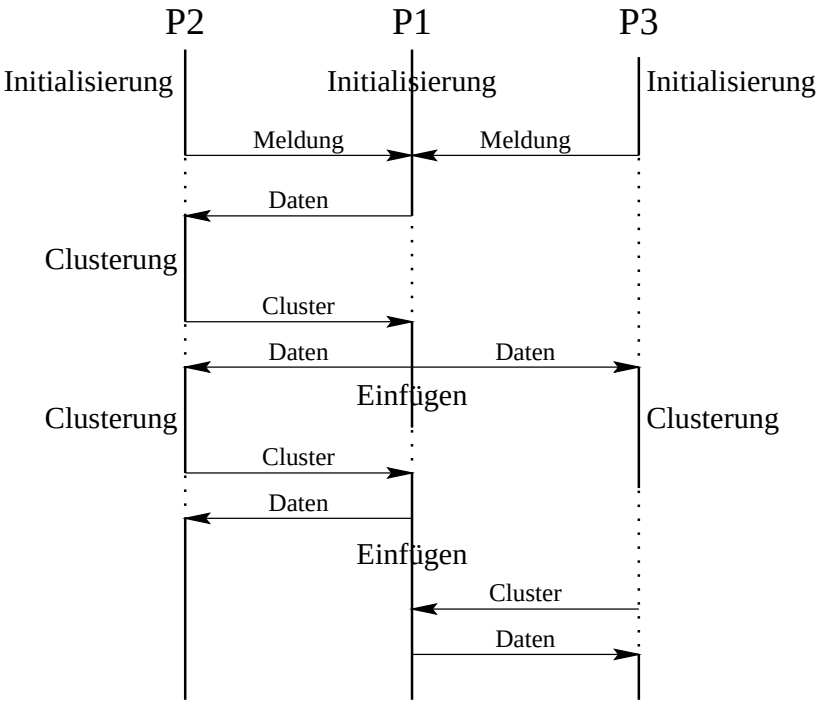


Abbildung 11: Interaktion der Prozesse

chenprozesse legen einen Baumstumpf an. Dieser enthält die Trainingsdaten, die Parameter und Methoden zur Clusterung vorgegebener Datenmengen.

Die Rechenprozesse schicken am Ende ihrer Initialisierung eine Bereitschaftsmeldung an den Dispatcher. Nachdem sich alle Rechenprozesse dort gemeldet haben, wird die ungeclusterte Datenmenge zur ersten Unterteilung an einen dieser Prozesse (P2) geschickt. Der Rechenprozeß sendet die entstandene Clusterung zurück an der Verwaltungsprozeß.

Da weitere Clusterungen nötig sind, erhalten die verfügbaren Rechenprozesse entsprechenden Datenmengen. Anschließend fügt der Verwaltungsprozeß die nötigen Einträge in den Baum ein. Jeder Rechenprozeß schickt die gefundenen Cluster an den Dispatcher zurück. Falls weitere Unterteilungen notwendig sind, erhält er im Gegenzug ein neues Datenpaket. Der Dispatcher nimmt für jeden erhaltenen Cluster einen entsprechenden Eintrag im Baum vor.

Wenn eine Datenmenge bereits klein genug ist, wird sie nicht mehr in die ToDo-Liste eingefügt. Der Aufbau des Baumes ist also beendet, wenn diese Liste leer ist und kein Rechenprozeß mehr an weiteren Clustern arbeitet.

Die Rechenprozesse warten im Ruhezustand auf neue Datenpakete. Sie erhalten am Ende des Baumaufbaues eine spezielle Nachricht, die sie veranlaßt,

sich zu beenden. Als letzter arbeitender Prozeß speichert der Dispatcher den erzeugten Baum in einer Datei zur weiteren Verwendung ab und beendet sich.

6 Evaluierung

Zu Beginn der Testauswertung sollen die verwendete Testumgebung und die Testdaten beschrieben werden. Dazu gehört auch die Beschreibung einer Beispielsession. Im Anschluß daran erfolgt die Auswertung der ermittelten Testergebnisse. Ideen für mögliche Verbesserungen und Erweiterungen bilden den Abschluß des Kapitels.

6.1 Testumgebung

Aufbauend auf die Erfahrungen aus [Kru99] fiel die Wahl auf einen Computercluster mit mehreren PC's. Die Leistungsfähigkeit dieser Art von Parallelrechnern läßt sich in der Regel ohne großen Aufwand der Problemstellung anpassen. Das bietet auch eine gewisse Flexibilität, wenn die Problemgrößen mit der Zeit wachsen. Durch die Verwendung von gewöhnlichen PC's in so einem Cluster ist das Preis-Leistungsverhältnis viel günstiger als bei einer einzelnen Mehrprozessormaschine gleicher Leistungsfähigkeit.

Der Computercluster bestand insgesamt aus vier Rechnern. Das waren zum einen drei identische Rechner mit AMD Athlon Prozessoren. Sie verfügten über 256MB RAM und eine IDE Festplatte. Die Angaben zur Ausstattung sind in Tablle 1 aufgeführt.

Prozessor:	AMD Athlon 750MHz
Motherboard:	Gigabyte GA-7ZX
RAM:	256MB
Festplatte:	
Controller:	onboard VIA VT82C686A
HD:	Seagate ST320420A
	Barracuda ATA II (20.4GB)

Tabelle 1: Ausstattung der Rechner

Der vierte Rechner besaß einen AMD K6/2 Prozessor. Auch er verfügte über 256MB RAM. Im Gegensatz zu den anderen drei Rechnern besaß er aber sowohl eine IDE als auch eine SCSI Festplatte. Seine Ausstattung ist in Tabelle 2 aufgelistet.

Wie im vorangegangenen Kapitel bereits gesagt, spielt die Minimierung des Kommunikationsaufwandes eine wichtige Rolle. Zum einen ist das durch die Reduzierung der Anzahl notwendiger Kommunikationsvorgänge möglich. Eine andere Möglichkeit besteht in der Beschleunigung der Kommunikation durch Erhöhung der Transferraten.

Prozessor:	AMD K6/2 450MHz
RAM:	256MB
Festplatte:	
IDE	
Controller:	onboard VIA VT82C586
HD:	Maxtor 91531U3 (14.6GB)
SCSI	
Controller:	Adaptec AHA-2940 Ultra
HD:	FUJITSU M1606S (1GB)

Tabelle 2: Ausstattung der Rechner

Die Rechner des Computerclusters waren zu diesem Zweck mit einem Myrinet verbunden. Dies ist ein Hochleistungsnetzwerk zur Bewältigung großer Datenmengen. Der Umfang der transferierten Datenmenge hat deshalb keinen so großen Einfluß auf die Geschwindigkeit der Kommunikation wie bei langsameren Ethernetzen. Die Anzahl der Kommunikationsvorgänge hat durch den Synchronisationsaufwand einen wesentlich größeren Einfluß auf die Performance des parallelen Algorithmus.

Als Betriebssytem kam Red Hat Linux 6.2 zum Einsatz. Der dort mitgelieferte Standardkernel war allerdings ungeeignet. Es fehlte die Unterstützung für NFS-Server. Für die parallele Ausführung von Berechnungen ist der gemeinsame Zugriff auf identische Sekundärspeicherbereiche sehr günstig. Außerdem mußten noch Anpassungen für die Athlon-Prozessoren vollzogen werden. Diese verfügen über keine CPUID, was der Linuxkernel mit den Standardparametern aber voraussetzt.

6.1.1 Myrinet

Myrinet ist eine Netzwerktechnologie, welche zur schnellen verzögerungsarmen Übertragung großer Datenmengen entwickelt wurde. Es besteht aus Netzwerkkarten und Switches. Durch die Switches sind verschieden Netzwerktopologien realisierbar. Myrinet beinhaltet eine Flußkontrolle und Fehlererkennung auf der Hardwareebene.

Die Interfacekarten besitzen eine eigene programmierbare CPU und einen Speicher. Bei der Initialisierung der Karte wird das Betriebsprogramm vom Wirtsrechner geladen. Dadurch kann die Software leicht geändert und den Erfordernissen angepaßt werden.

Das mitgelieferte Betriebsprogramm erlaubt TCP/IP Übertragungen. Die IP-Prüfsummen können direkt auf der Karte berechnet werden. Die Übertragung wird durch das TCP/IP Protokoll beschränkt. Durch die Programmier-

barkeit der Karten sind noch viele andere Protokolle entstanden, welche die Möglichkeiten der Karte zum Teil besser ausnutzen.

In Tabelle 3 sind die genauen Bezeichnungen der verwendeten Hardware aufgeführt:

Interface-Karten	M2L-PCI64B-2 (LANai 9)
Switch	M2F-SW4 (4 Ports)

Tabelle 3: verwendete Netzwerkhardware

Auf den Seiten von Myricom im Internet [MYRI] wird eine erreichbare durchschnittliche Datenrate von 240 MB/s bei Kommunikation in eine Richtung angegeben. Sie ist von der Größe der übertragenen Daten abhängig. Für kleine Datenpakete spielt der Transfer zwischen dem RAM des Wirtsrechners und der Karte eine stärkere Rolle als bei großen Nachrichten. Die Abhängigkeit von Nachrichtengröße und Übertragungsrate ist in Bild 12 dargestellt. Die Geschwindigkeit von 240 MB/s stellt einen Grenzwert für sehr große Datenpakete dar. In der Praxis liegt die erzielte Bandbreite zwischen 60 und 120 MB/s.

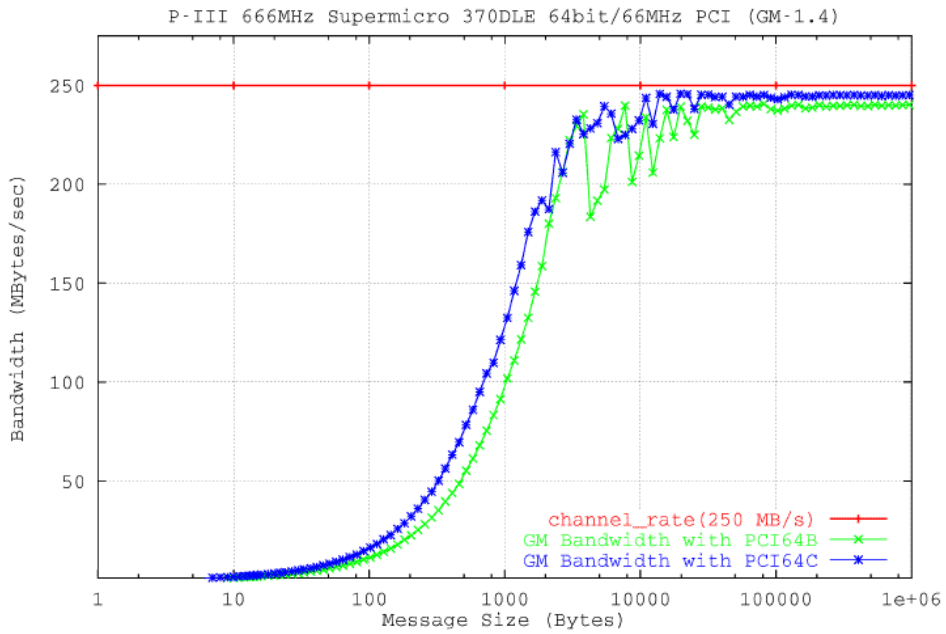


Abbildung 12: Übertragungsrate im Myrinet

Ein weiteres wichtiges Leistungskriterium für Netzwerke ist die Verzögerungszeit. Sie entsteht durch die physikalischen Gegebenheiten der Hardware sowie der softwareseitigen Initialisierung und Adressierung der zu sendenden Datenpakete. Myricom gibt als untersten Grenzwert 9 μsec an. Auch die

Verzögerungszeit ist von der Größe der Nachrichten abhängig. In der Praxis treten 10 - 60 μsec auf. Der Verlauf der Verzögerungszeit mit steigender Nachrichtengröße ist in Bild 13 dargestellt.

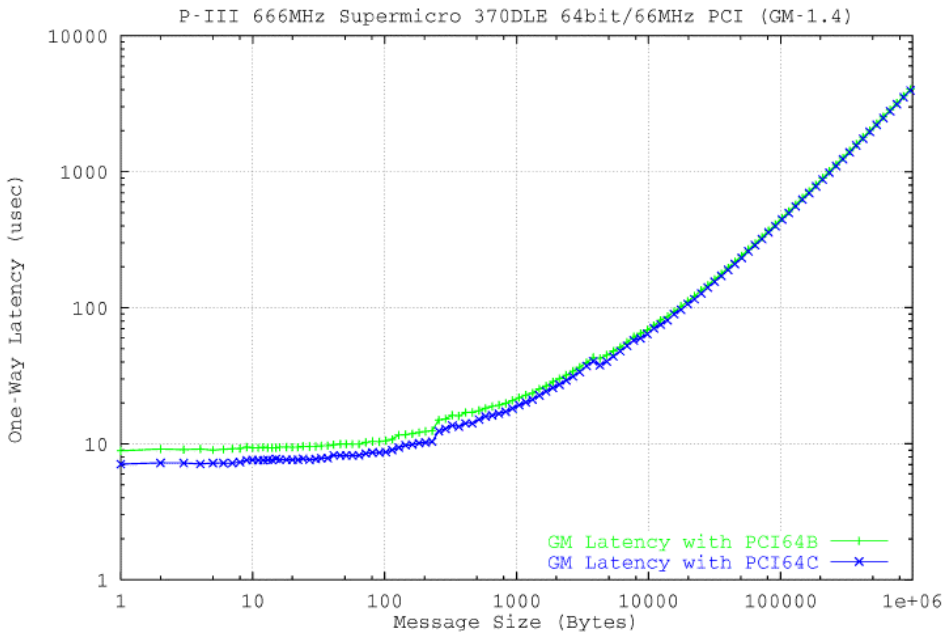


Abbildung 13: Verzögerungszeit im Myrinet

Gegenüber dem Fast Ethernet mit einer Übertragungsrate von ca. 11 MB/s und einer Verzögerungszeit von 100 μsec ist ein Myrinet deutlich schneller. Es eignet sich damit besser für Computercluster.

6.2 Message Passing Interface

Das Message Passing Interface MPI ist ein vom Message Passing Interface Forum (MPIF) verabschiedeter Standard. Es stellt eine Programmierschnittstelle zur Programmierung nachrichtenaustauschender Parallelrechner und Cluster bereit. Im MPIF sind verschiedene Hersteller (IBM, Intel, Cray Research,...), Organisationen (GMD,...) und Universitäten (Yale University, Michigan State University,...) vertreten.

Es stellt eine Bibliothek von Funktionen für die verbindungslose Kommunikation in parallelen Rechnersystemen bereit. Dabei wird davon ausgegangen, daß die Anzahl der parallelen Prozesse während der gesamten Laufzeit fest ist. Sie können in Gruppen zusammengefaßt werden, wobei ein Prozeß zu mehreren Gruppen gehören kann.

Die Kommunikation erfolgt über sogenannte Kommunikatoren. Diese sind in beliebiger Anzahl erzeugbar. Sie bilden den Kontext für die Kommunikation.

Ihnen liegt immer eine Prozeßgruppe zugrunde. Dabei erhält jeder Prozeß einen eindeutigen Rang innerhalb des Kommunikator. Über diesen lassen sich die Nachrichten entsprechend adressieren.

MPI bietet Operationen für die Punkt-zu-Punkt Kommunikation. Diese kann blockierend oder nicht blockierend sein. Blockierend bedeutet, daß der sendende Prozeß solange wartet, bis sein Sendepuffer wieder frei ist. Dies ist je nach eingestelltem Übertragungsmodus der Fall, wenn die Nachricht in einen Zwischenpuffer oder den Puffer des Empfängers kopiert wurde. Eine blockierende Empfangsoperation wartet solange, bis eine Nachricht eintrifft.

Die Nachrichten werden mit einem sogenannten *message tag* versehen. Dieses spezifiziert die Art der Nachricht. Bei Empfangsoperationen kann so selektiv nur nach bestimmten Nachrichten gefragt werden.

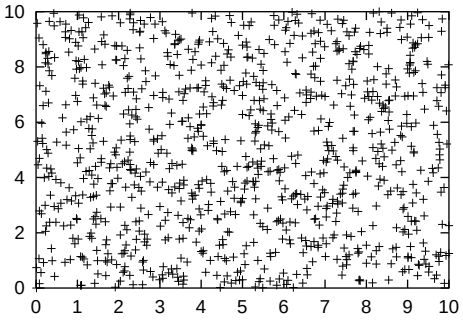
Daneben bietet MPI auch kollektive Operationen. Beispielsweise ist ein Broadcast möglich, mit dem ein Prozeß alle anderen innerhalb des verwendeten Kommunikators erreicht. Ebenso können Ergebniszusammenfassungen durch den Aufruf einer globalen Operation geschehen. Zur globalen Synchronisation aller Prozesse kann eine Barriere auf den Kommunikator gelegt werden, die alle Prozesse gleichzeitig überwinden müssen.

Auf dem Testsystem wurde das mitgelieferte MPICH over GM in der Version 1.2.3 verwendet. Dieses bietet Unterstützung für MPI 1.2.

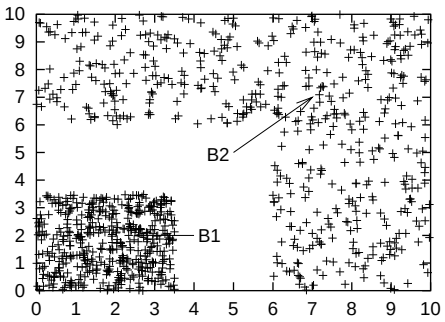
6.3 Testdaten

Die in den Testdaten enthaltenen Cluster mußten bekannt sein, um die Korrektheit der erzeugten Ergebnisse überprüfen zu können. Es wurden zufällig erzeugte 6-dimensionale Daten verwendet. Dabei sind immer nur zwei Dimensionen zufällig. Die anderen vier enthielten jeweils feste Werte. Dadurch werden drei verschiedene Ebenen aufgespannt, die jeweils im rechten Winkel zueinander stehen.

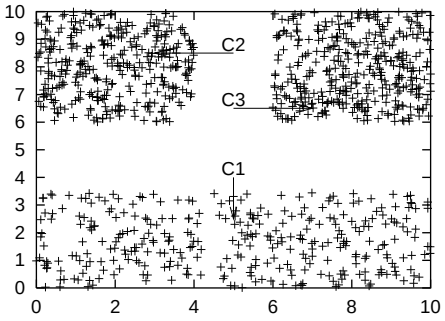
Die in den Ebenen enthaltenen Cluster hatten eine unterschiedliche Form und Ausdehnung. Dadurch war auch die Wahrscheinlichkeitsverteilung verschieden. Die erste Ebene enthielt nur einen rechteckigen Cluster. Auf Ebene 2 lagen zwei Cluster, von denen einer den anderen teilweise umschließt. In der dritten Ebene sind 3 Cluster mit unterschiedlicher Größe enthalten.



(a) Ebene1



(b) Ebene2



(c) Ebene3

6.4 Beispielsession

Das Programm wird über den Befehl mpirun von MPICH gestartet. Dabei wird MPI die gewünschte Anzahl paralleler Prozesse mitgegeben. Als zweiter Parameter ist der Name des gewünschten Datenfiles notwendig. Für 3 Prozesse und die Datei "6d600000.RVC" sieht der Aufruf folgendermaßen aus:

```
mpirun -np 3 Application -f 6d600000.RVC
```

Die Datendatei enthält am Ende neben den Datenvektoren die Pagekeys der zugeordneten Datenbankseiten. Außerdem wird der erzeugte Indexbaum in die Datei geschrieben und kann so weiterverwendet werden.

6.5 Testergebnisse

Bei den Tests konnte der erwartete Vorteil durch die parallel arbeitenden Prozesse nicht belegt werden. Der Verwaltungs- und Kommunikationsaufwand verbraucht den Zeitgewinn, welcher durch die parallele Clusterung ensteht, gleich wieder.

	2	3	4
200.000	2:37:39	2:35:35	2:36:46
300.000	7:15:14	7:14:42	7:09:34
400.000	10:56:18	10:46:44	10:50:23
500.000	15:45:32	15:52:50	15:52:56

In Abbildung 14 ist zum Vergleich der lineare Anstieg eingezeichnet. Die Differenzen der Trainingszeiten bei unterschiedlicher Prozeßanzahl liegen innerhalb der Meßgenauigkeit. Es läßt sich kein Trend zu einem Geschwindigkeitszuwachs bei steigender Prozeßanzahl feststellen. Vielmehr treten bei großen Datenmengen eher Probleme auf. Die erste Clusterung muß ein Prozeß allein vornehmen. Diese verwendet die gesamte Datenmenge. Bei großen Datenmenge treten deshalb zu Beginn sehr lange Wartezeiten auf.

In Abbildung 11 sind deutlich die Wartezeiten der Rechenprozesse zu erkennen. Besonders auffällig wird dies an dem Punkt, wenn P3 mit der Clusterung seiner Datenmenge fertig ist. Da P1 sein Ergebnis eher abgeliefert hat, ist der Dispatcher noch mit den Eintragungen im Baum beschäftigt. Folglich kann er nicht gleichzeitig die geclusterte Datenmenge von P3 entgegennehmen und neue Daten verschicken. Dieser Synchronisationsaufwand stellt den entscheidenden bremsenden Faktor dar.

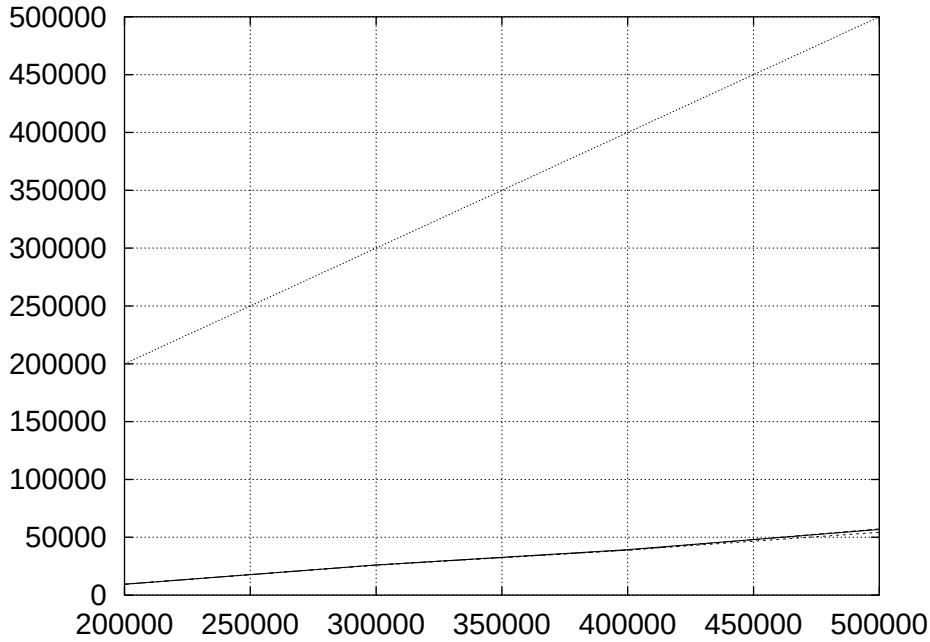


Abbildung 14: Trainingszeiten bei steigender Vektorenanzahl

Diese Art der Parallelisierung erscheint nach den Testergebnissen als ungeeignet. Besonders die mangelnde Lastbalancierung am Anfang des Aufbauprozesses hat sich als sehr hinderlich herausgestellt. Die Parallelisierung sollte eher im eigentlichen Trainingsprozeß der Neuronalen Netze ansetzen.

6.6 Ausblick

Um den Speedup Faktor der parallelen Variante des Indexaufbaus zu erhöhen, müssen die in Kapitel 4 genannten Dinge forciert werden. Wichtig ist hauptsächlich die Minimierung der Kommunikation. Wie bereits weiter oben gesagt, spielt die Menge der transferierten Daten eine eher untergeordnete Rolle. Hauptaugenmerk liegt auf der Minimierung der Anzahl von Kommunikationsvorgängen. Den größten Aufwand bei den Interaktionen stellt die nötige Synchronisation dar. Mit der Minimierung der nötigen Kommunikationen sinkt auch dieser Synchronisationsaufwand.

Kommunikation und Synchronisation entstehen durch Abhängigkeiten der parallelen Prozesse. Bei dem hier beschriebenen Vorgehen der Parallelisierung sind alle Rechenprozesse vom zentralen Verwaltungsprozeß abhängig. Je größer die Anzahl gleichzeitig clusternder Prozesse ist, desto mehr Ergebnisse muß der Dispatcher entgegennehmen und in den Baum einbauen. Bei

sehr vielen Rechenprozessen entsteht folglich ein Stau bei der Ablieferung der Ergebnisse.

Eine mögliche Lösung zur Entkoppelung der Prozesse wäre, die Rechenprozesse eigene lokale Teilbäume aufbauen zu lassen. Das heißt, sie clustern bei Bedarf ihre gefundenen Teildatenmengen einfach weiter. Wenn überall lokale Pagekeys verwendet werden, so können die Teilbäume vom Dispatcher auch eindeutig in den Gesamtbaum integriert werden.

Literatur

[BM72] Bayer, R. / McCreigth, E.M.: "Organisation and maintenance of large ordered indexes". Acta Informatica 1, 1972:173-189

[BM95] Blackmore, J. / Miikkulainen, R.: "Visualizing high-dimensional structure with the incremental grid growing neural network". Proc. 12th Int'l Conf. on Machine Learning, Tahoe City (Kalifornien), 1995:55-63

[BV95] Bauer, H.-U. / Villmann,T.: Growing a hypercubical outputspace in a selforganizing feature map. Berkley: International Computer Science Institute, 1995

[Fri91] Fritzke, B.: "Unsupervised Clustering with Growing Cell Structures". Proc. IJCNN-91, Seattle, 1991

[Fri94] Fritzke, B.: "A growing neural gas network learns topologies". NIPS 7, Proc. 1994 Conf., Denver, 1994

[Fri95] Fritzke, B.: "Growing Grid - a self-organizing network with constant neighborhood range and adaption strength" Neural Processing Letters Vol. 2 No. 5, 1995:9-13

[Gut84] Guttman, R.: "A Dynamic Index Structure for Spatial Searching". Proceedings ACM SIGMOD Conference, Boston, 1984:47-57

[Hay94] Haykin, S.: Neural Networks - A Comprehensive Foundation. New York: Macmillan College Publishing Company, 1994

[Koh89] Kohonen, T.: Self-Organization and Associative Memory. 3. Auflage. Berlin, Heidelberg, New York: Springer Verlag, 1989

[Kru99] S. Krumbiegel: Performanzvergleich künstlicher neuronaler Netze bei unterschiedlicher Hardwareunterstützung. Studienarbeit, Technische Universität Chemnitz, 1999

[KS96] Kawahara, S. / Saito, S.: "On a novel adaptive self organizing network". Proc. 4th IEEE Int'l Workshop on Cellular Networks and their Applications (CNNA-96), Sevilla, 1996:41-46

[MPI95] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. Version 1.1. [Online] URL: http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[MS91] Martinez, T. / Schulten, K.: "A 'neural gas' network learns toplogies". Artifical Neural Networks, Amsterdam (North-Holland), 1991:397-402

[MYRI] Myrinet Performance Measurements. [Online] URL: http://www.myri.com/performance

[Mü93] U.A. Müller: Simulation of neural networks on parallel computers, Konstanz: Hartung-Gorre, 1993

[NG98] Neubert, R. / Gilg, S.: Schemavergleich mit Hilfe Neuronaler Netze. Studienarbeit, Technische Universität Chemnitz, 1998

[NG99] Neubert, R. / Gilg, S.: Semantische Indexierung mittels dynamisch-hierarchischer Neuronaler Netze. Diplomarbeit, Technische Universität Chemnitz, 1999

[Pac97] Pacheco, P.: Parallel Programming with MPI. San Francisco: Morgan Kaufmann Publishers, 1997

[Roj96] Rojas, R.: Theorie der Neuronalen Netze. Berlin, Heidelberg, New York: Springer-Verlag, 1996

[SS83] Schlageter, G. / Stucky, W.: Datenbanksysteme: Konzepte und Modelle. Stuttgart: Teubner, 1983

[Ste98] Steffens, J.: Neural Network Objects Bibliothek. Version 1.3. [Online] URL: ftp://tau.ep1.ruhr-uni-bochum.de/pub/john/nno.1.3.tar.gz

[Zel94] Zell, A.: Simulation neuronaler Netze. München: Addison-Wesley (Deutschland) GmbH, 1994

Hiermit erkläre ich diese Arbeit ohne fremde Hilfe und nur mit den angegebenen Hilfmitteln erstellt zu haben.

Chemnitz, d. 29.07.2001 Stefan Krumbiegel