Objektorientierte parallele Ein-/Ausgabe auf Höchstleistungsrechnern

Simon Pinkenburg

Objektorientierte parallele Ein-/Ausgabe auf Höchstleistungsrechnern

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Phys. Simon Pinkenburg
aus Leonberg

Tübingen
2006

Tag der mündlichen Qualifikation:	28.06.2006
Dekan	Prof. Dr. Michael Diehl
1. Berichterstatter:	Prof. Dr. Wolfgang Rosenstiel
2. Berichterstatter:	Prof. Dr. Dietmar Kaletta

Danksagung

Die vorliegende Arbeit entstand am Lehrstuhl für Technische Informatik der Fakultät für Informations- und Kognitionswissenschaften der Universität Tübingen. An dieser Stelle möchte ich mich daher bei meinem Doktorvater Herrn Prof. Dr. Wolfgang Rosenstiel für die sehr gute wissenschaftliche Betreuung, Beratung und Unterstützung während meiner Arbeit bedanken. Darüber hinaus gilt mein Dank Herrn Prof. Dr. Dietmar Kaletta für die Übernahme des Korreferats und die sorgfältige Begutachtung meiner Dissertation.

Weiterhin bedanke ich mich bei meiner Arbeitsgruppe, namentlich Sven Ganzenmüller, Michael Hipp, Steffen Holtwick, Andreas Nagel und Dr. Marcus Ritt, für die zahlreichen fruchtbaren Diskussionen im Laufe der letzten Jahre. Insbesondere danke ich Herrn Dr. Marcus Ritt für die kritische Prüfung und die zahlreichen Korrekturhinweise, die zum Gelingen dieser Arbeit beigetragen haben.

Meinen Kolleginnen und Kollegen danke ich für das stets gute Arbeitsklima und die zahlreichen Diskussionen und Gespräche auch abseits der Arbeit.

Der herzlichste Dank gebührt jedoch meiner Lebensgefährtin Katja für die seelische und moralische Unterstützung meiner Arbeit, ihre Liebe und Geduld - insbesondere in den letzten Monaten - und ihr stets vorhandenes Verständis.

Nicht zu letzt danke ich meinen Eltern, die mir durch ihr Vertrauen stets die Möglichkeit gegeben haben meine Ziele bestmöglich zu erreichen.

Kurzfassung

Unterschiedliche Leistungssteigerungen bei Prozessoren und Festplatten einerseits und gestiegene Anforderungen an Genauigkeit, Interaktivität und Visualisierung paralleler Simulationen andererseits führten in den letzten Jahren dazu, dass sich die Ein-/Ausgabe von Daten auf Höchstleistungsrechnern zum Flaschenhals entwickelt hat. Auf Systemebene wird dies durch eine Vielzahl von Festplatten und parallelen Dateisystemen kompensiert. Damit jedoch die Anwendungsebene von dieser aggregierten Bandbreite profitieren kann, müssen geeignete Schnittstellen zur parallelen Ein-/Ausgabe vorhanden sein. Als weit verbreiteter Standard hat sich die auf Nachrichtenaustausch basierende Bibliothek MPI-IO etabliert. Diese prozedurale Bibliothek kann jedoch die Anforderungen der wachsenden Anzahl objektorientierter paralleler Anwendungen nicht erfüllen. Der darüber hinaus hohe Funktionsumfang führt zu einer komplexen und umfangreichen Programmierung für den Anwender.

Mit TPO-IO wurde eine objektorientierte, benutzerfreundliche und sehr effiziente Schnittstelle entwickelt, die auf MPI-IO aufsetzt und es ermöglicht, neben Objekten und Standard-Datentypen auch Container der Standard Template Library persistent zu machen. Dabei folgen Funktionsumfang und Namenskonventionen soweit wie möglich und sinnvoll dem Standard MPI-2. Das Design der Schnittstelle vereinfacht und restrukturiert erheblich die Funktionalität von MPI-IO und stellt sich dem Benutzer deutlich transparenter und übersichtlicher dar. Leistungsstarke Techniken, wie z.B. die freie Definition einer beliebigen Sicht eines Prozesses auf die Daten, kollektive Zugriffe mehrerer Prozesse auf eine Datei und asynchrone Ein-/Ausgabe, wurden auf objektorientierte Konstrukte übertragen und zur Steigerung der Benutzerfreundlichkeit teilweise bereits innerhalb der Schnittstelle automatisiert.

Synthetische Leistungsmessungen zeigen einen im Vergleich zum Gewinn der objektorientierten Darstellung geringen Verlust von TPO-IO gegenüber MPI-IO. Darüber hinaus konnte die Schnittstelle bereits in drei Anwendungen unserer Arbeitsgruppe erfolgreich eingesetzt werden. Es handelt sich dabei um zwei Teilchenmethoden und einer Methode aus dem Bereich der Gensequenzanalyse.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Numerical Tokamak's Parallel Platform Paradoxon:
The average time required to implement a moderate-sized application on a parallel computer architecture is equivalent to the half-life of the latest parallel supercomputer.
J.D. Oldham [64]

Das Ziel einer wissenschaftlichen Simulation liegt in der möglichst exakten Abbildung der realen Welt durch ein Programm. Die Anforderungen an solche Simulationen bezüglich ihrer Genauigkeit, Interaktivität und der Visualisierung der Ergebnisse sind dabei im Laufe der Zeit kontinuierlich gestiegen. Man begegnet dieser Problematik mit Hilfe von Höchstleistungsrechnerarchitekturen, die aus einem Zusammenschluss mehrerer Prozessoren und Hauptspeicherelemente bestehen, um in angemessener Zeit zu aussagekräftigen Ergebnissen zu gelangen.

Immer aufwändigere Simulationen einerseits und die steigende Gesamtleistung eines Parallelrechners andererseits führten in den letzten Jahren zu einer regelrechten Explosion der im Laufe einer Simulation entstehenden Daten. Die Entwicklungen der Festspeicherindustrie konnten dabei jedoch mit denen der Prozessortechnik nicht Schritt halten. Dies ist insbesondere auch auf die geringe Beachtung der Ein-/Ausgabe im Bereich der Forschung zurückzuführen. Allein die Tatsache, dass im wissenschaftlichen Rechnen die Leistung eines Systems meist lediglich anhand der pro Sekunde durchführbaren Anzahl von Fließkommaoperationen (engl. floating point operations per second, FLOPS) beurteilt wird und nur selten anhand der Menge von Daten, die vom System pro Sekunde gespeichert werden können, zeigt, worauf sich die Forschung in den letzten 30 Jahren fokussiert hat. Es verwundert daher auch nicht, dass in dieser Zeit die Leistung eines Prozessors nahezu regelmäßig alle 18 Monate verdoppelt werden konnte, während die Zugriffszeiten der Festplatten sich lediglich um $8 - 10\%$ pro Jahr verbesserten. Als Folge dieser divergierenden Entwicklung hat sich die Ein-/Ausgabe in Höchstleistungsrechnern mehr und mehr zum Flaschenhals entwickelt und stellt damit heutzutage den leistungsbegrenzenden Faktor eines solchen Systems dar.

Erste Forschungsaktivitäten auf dem Gebiet der Ein-/Ausgabe begegneten diesem Problem zunächst auf Systemebene durch den Zusammenschluss einer Vielzahl von Festplatten. Die effiziente Verteilung der Daten trieb dabei die Entwicklung von parallelen Dateisystemen voran, deren Hauptziel darin bestand, die aggregierte Bandbreite der Komponenten möglichst auszuschöpfen. Um jedoch auch auf Anwendungsebene von der gesteigerten Bandbreite profitieren zu können, müssen geeignete Schnittstellen vorhanden sein, die eine Formulierung der parallelen Ein-/Ausgabe zulassen. Die Analogie zwischen einer Punkt-zu-Punkt-Kommunikation zweier Prozessoren und dem Datentransfer eines Prozessors auf die Festplatte ermöglichte es, die Entwicklung von Bibliotheken zur parallelen Ein-/Ausgabe an bereits vorhandenen Standards, wie z.B. MPI, auszurichten.

Im Bereich des Softwareentwurfs war der nachhaltige Einzug objektorientierter Methoden in den Bereich der parallelen Programmierung von entscheidender Bedeutung. Die Verwendung abstrakter Datenstrukturen in Form von Klassen, die nicht nur die Datentypen festlegen, aus denen die mit Hilfe der Klassen erzeugten Objekte bestehen, sondern zusätzlich die Algorithmen definieren, die auf diesen Daten operieren, ermöglichte erstmals eine klare Strukturierung einer Anwendung. Diese verbesserte Modularisierung hat den Vorteil einer höheren Wartbarkeit und Wiederverwendbarkeit der Einzelmodule und steigert damit sowohl die Qualität der Software als auch die Effizienz nachfolgender Softwareentwicklung. Die Anwendung objektorientierter Konzepte auf den Bereich der parallelen Ein-/Ausgabe ist daher erstrebenswert und Gegenstand der vorliegenden Arbeit.

Problemstellung

Im Rahmen des interdisziplinären Sonderforschungsbereichs 382 mit dem Titel „Verfahren und Algorithmen zur Simulation physikalischer Prozesse auf Höchstleistungsrechnern“ forschen und entwickeln Arbeitsgruppen der Institute für Numerik, Informatik, Mathematik und Physik der Universitäten Stuttgart und Tübingen. Das Teilprojekt C6 stellt dabei mit dem Thema „Objektorientierte Parallelisierung“ das zentrale Bindeglied zwischen der Entwicklung wissenschaftlicher Simulationen und deren Parallelisierung auf Höchstleistungsrechnern dar. Im Fokus der Arbeiten steht hauptsächlich die Umsetzung objektorientierter Implementierungstechniken zur Parallelisierung numerischer Teilchensimulationen, wie z.B. der Modellierung geeigneter Klassenstrukturen oder die Verwendung von Entwurfsmustern (engl. design patterns), die es ermöglichen, die Parallelisierung vom physikalischen Modell zu entkoppeln. Am Ende dieser Entwicklung steht dann eine objektorientierte Bibliothek oder ein Rahmengerüst (engl. framework), das die Implementierung der Modelle deutlich vereinfacht.

Zur Programmierung der parallelen Ein-/Ausgabe existieren für Höchstleistungsrechner weit verbreitete Standard-Bibliotheken. Diese folgen jedoch alle einem prozeduralen Ansatz und unterstützen den Einsatz moderner objektorientierter Entwicklungsstandards nicht. Dadurch entsteht eine Lücke zwischen objektorientierter Softwareentwicklung und deren Implementierung auf Parallelrechnern. Insbesondere ermöglichen die prozeduralen Standards lediglich die Übertragung einfacher Datenstrukturen, so dass Objekte nur unter Verwerfung des Objektmodells gespeichert werden können. Dies stellt einen erhöhten Aufwand für den Entwickler dar und gefährdet die gerade im Höchstleistungsrechnen äußerst wichtige Stabilität der Anwendung.

Das Ziel der vorliegenden Arbeit liegt daher in der Entwicklung einer objektorientierten Schnittstelle zur parallelen Ein-/Ausgabe. Von zentraler Bedeutung ist dabei die Übertragung objektorientierter Datenstrukturen selbst, deren Implementierung sich möglichst in ein weit verbreitetes Konzept integrieren lassen soll. Dabei ist stets die im Bereich des parallelen Rechnens notwendige hohe Effizienz der Schnittstelle zu berücksichtigen. In den nun folgenden Kapiteln wird der Entwicklungsprozess der Schnittstelle von der Konzeption über die Implementierung bis hin zum praktischen Einsatz mit Leistungsmessungen, mit deren Hilfe sich die Effizienz der Schnittstelle belegen lässt, beschrieben.

Aufbau der Arbeit

Die Arbeit gliedert sich dabei in drei Teile: Den ersten Teil bilden die Grundlagen des parallelen Rechnens und der Stand der Technik. Im zweiten Teil folgt die Darstellung der Entwicklung und Implementierung der objektorientierten Schnittstelle. Im abschließenden Teil belegen zahlreiche Anwendungen den praktischen Einsatz der Schnittstelle.

Der erste Teil stellt in Kapitel 2 die allgemeinen Grundlagen des parallelen Rechnens speziell im Kontext der parallelen Ein-/Ausgabe dar. Abschnitt 2.5 behandelt dabei bereits existierende Schnittstellen zur parallelen Ein-/Ausgabe und bildet - dem Rahmen der Arbeit entsprechend - den Schwerpunkt dieses Kapitels. Daran anschließend folgt in Kapitel 3 die Darstellung des aktuellen Stands der Technik und dessen unterschiedliche Ansätze zur Formulierung einer parallelen Ein-/Ausgabe. Die Bewertung der vorgestellten Ansätze erfolgt in Kapitel 4. Dazu erläutert Abschnitt 4.1 zunächst die Anforderungen, die an eine objektorientierte Schnittstelle zur parallelen Ein-/Ausgabe gestellt werden müssen, bevor in den folgenden Abschnitten eine eingehende Analyse und Bewertung der Ansätze erfolgt. Schließlich werden die gewonnenen Erkenntnisse zusammengefasst und entsprechende Kriterien für den Entwurf einer objektorientierten Schnittstelle zur parallelen Ein-/Ausgabe daraus abgeleitet.

Der zweite Teil in Kapitel 5 und 6 mit der Beschreibung der Konzeption und Implementierung der Schnittstelle unter Berücksichtigung der aufgestellten Ziele bildet den Kern der vorliegenden Arbeit. Die einzelnen Abschnitte behandeln dabei die Portabilität und Effizienz, die Umsetzung der Übertragung objektorientierter Datenstrukturen, die Funktionalität und die Verbesserung der Bedienbarkeit der Schnittstelle.

Der dritte Teil belegt in Kapitel 7 mit Messungen und Beispielen aus der Praxis die Anwendbarkeit und Leistung der vorgestellten Schnittstelle. Die Abschnitte 7.1, 7.2 und 7.3 beschreiben die Integration der Schnittstelle in drei Anwendungen, deren Ein-/Ausgabe parallelisiert wurde. Daran wird deutlich, wie der praktische Einsatz der Schnittstelle den Entwicklungsprozess vereinfachen kann, welche gängigen Strategien zur Parallelisierung der Ein-/Ausgabe existieren und inwiefern die Ziele der Konzeption und Implementierung erreicht werden konnten. Zur Beurteilung der Effizienz stellt Abschnitt 7.4 synthetische Leistungsmessungen dar und vergleicht die gefundenen Resultate mit denen anderer Schnittstellen.

Schließlich fasst Kapitel 8 die Arbeit zusammen und beschreibt Forschungsthemen, die nahtlos an die vorliegende Arbeit anknüpfen könnten.

2 Grundlagen

The most misleading assumptions are the ones
you don't even know you're making.
Douglas Adams

Dieses Kapitel beschreibt die für das weitere Verständnis der Arbeit notwendigen Grundlagen und Definitionen im parallelen Rechnen. Der erste Abschnitt erläutert dazu zunächst die verwendeten Begriffe. Eine Klassifikation paralleler Rechnerarchitekturen folgt im zweiten Abschnitt. Dem Fokus der Arbeit entsprechend bildet Abschnitt 2.3 über parallele Ein-/Ausgabearchitekturen den umfangreichsten Teil dieses Kapitels. Im Anschluss erläutern die Abschnitte 2.4 und 2.5 die auf parallelen Ein-/Ausgabearchitekturen eingesetzte Software, wie parallele Dateisysteme und darauf aufsetzende Schnittstellen, die es den Anwendungen ermöglichen, parallele Ein-/Ausgabe zu betreiben. Schließlich werden in Abschnitt 2.6 einige im Rahmen dieser Arbeit verwendete Beispielarchitekturen vorgestellt.

2.1 Begriffe

Innerhalb der vorliegenden Arbeit hat der Begriff *parallel* eine zentrale Bedeutung. Er soll daher zunächst definiert werden [21]:

Definition parallel (engl. parallel): Arbeitsabläufe bzw. deren Einzelschritte heißen parallel, wenn sie gleichzeitig und voneinander unabhängig durchgeführt werden können.

Diese Definition berücksichtigt dabei nicht den Zeitpunkt der Durchführung, sondern lediglich die Bedingung der *möglichen* gleichzeitigen Ausführbarkeit. Erst der Terminus *Parallelverarbeitung* fasst den Begriff der Parallelität deutlich enger:

Definition Parallelverarbeitung (engl. parallel processing): Gleichzeitige Verarbeitung von parallelen Arbeitsabläufen oder Einzelschritten durch mehrere Prozessoren.

Diese Definition stellt damit die zwingende Forderung der *tatsächlichen* gleichzeitigen Durchführung der Arbeitsabläufe. Eine Parallelverarbeitung ermöglicht damit die nebenläufige Ausführung von Prozessen, die bei Verwendung mehrerer Prozessoreinheiten im Allgemeinen zu einer Laufzeitbeschleunigung führt. Die nebenläufigen Prozesse werden dabei von einem parallelen Programm mit wohl definiertem

Start und Ende gesteuert. Die Verteilung der Prozesse auf die einzelnen Prozessoren erfordert die Verwendung eines Parallelrechners. Darunter versteht man im Allgemeinen die Kopplung mehrerer Prozessoren mit Hilfe eines Kommunikationsmediums [4]:

> **Definition Parallelrechner:** Als Parallelrechner bezeichnet man einen Zusammenschluss mehrerer Ausführungseinheiten unter Verwendung eines gemeinsamen Kommunikationsmittels.

Unterschiedliche Prozessorarten und Kommunikationsmedien können die Leistung einer parallelen Anwendung stark beeinflussen. Die Skalierbarkeit einer parallelen Anwendung ist daher ein sehr wichtiges Kriterium für die Leistungsfähigkeit eines Parallelrechners [37]:

> **Definition Skalierbarkeit:** Unter Skalierbarkeit versteht man die Eigenschaft, nach der die Hinzunahme weiterer Ausführungseinheiten eine Programmbeschleunigung bewirkt.

An dieser Stelle soll eine für die vorliegende Arbeit wichtige Abgrenzung zwischen parallelem und verteiltem Rechnen vorgenommen werden. Das parallele Rechnen wird auf meist homogenen Architekturen mit einem eng gekoppelten Kommunikationsmedium eingesetzt und stellt damit die Beschleunigung einer parallelen Anwendung in den Vordergrund. Im Gegensatz dazu zeichnet sich das verteilte Rechnen durch eine ortsunabhängige Ausführung einer Anwendung aus. Dabei steht vor allem die Problematik des Zusammenschlusses heterogener Architekturen mit zum Teil unterschiedlichen Betriebssystemen unter Verwendung geeigneter Middleware, wie z.B. CORBA oder DCOM, im Vordergrund und weniger die gleichzeitige Ausführung der Anwendung. Ein Beispiel hierfür wäre das bekannte Seti@Home-Projekt, das die Suche nach extraterrestrischem Leben in kleine Teilprobleme zerlegt und über das Internet an partizipierende Rechner verteilt, um anschließend die Ergebnisse wieder zusammenzuführen.

Um die Güte einer parallelisierten Anwendung quantifizieren zu können, werden Laufzeitmessungen durchgeführt. Die gewonnenen Ergebnisse werden dann mit den Laufzeiten der sequentiellen Version der Anwendung verglichen. Die Laufzeit T ist abhängig von der Anzahl der verwendeten Prozessoren p. Die *Beschleunigung* (engl. speedup) S einer Anwendung ergibt sich dann aus dem Quotienten der sequentiellen Laufzeit (Laufzeit mit einem Prozessor) und der parallelen Laufzeit mit p Prozessoren. Eine höhere Beschleunigung bedeutet eine bessere Parallelisierung:

$$S(p) = \frac{T_{seq}}{T(p)} \tag{2.1}$$

Wenn eine Anwendung von Beginn an parallelisiert wurde, liegt keine sequentielle Version vor. In diesem Fall betrachtet man die *relative Beschleunigung* der Anwendung. Man setzt dabei die Laufzeit der parallelen Anwendung mit nur einem Prozessor in das Verhältnis zur Laufzeit mit p Prozessoren:

$$S_{rel}(p) = \frac{T(1)}{T(p)} \tag{2.2}$$

Der Verlauf der Beschleunigungskurve in Abhängigkeit von der Prozessorzahl gibt damit das Skalierungsverhalten der parallelen Anwendung wieder. Abhängig von dem verbleibenden sequentiellen Anteil an einer parallelisierten Anwendung stellte *Gene Amdahl* in dem nach ihm benannten Gesetz eine theoretische Obergrenze für die Beschleunigung auf [6]:

$$S \leq \frac{1}{\alpha + (1 - \alpha)/p} < 1/\alpha \tag{2.3}$$

Die Grenzwertbetrachtung für $p \to \infty$ liefert damit abhängig vom sequentiellen Anteil α eine obere Schranke der möglichen Beschleunigung von $1/\alpha$. Ist es aus algorithmischer Sicht z.B. nur möglich, 90% einer Anwendung zu parallelisieren, kann die sequentielle Anwendung maximal um den Faktor 10 beschleunigt werden.

Neben dem sequentiellen Anteil hat auch die Verteilung der Teilprobleme einer parallelen Anwendung auf die begrenzte Anzahl von Prozessoren Auswirkungen auf die Beschleunigung. Eine gute Parallelisierung sieht daher auch Algorithmen zu einer effizienten *Lastverteilung* (engl. load balancing) vor. Den dabei erreichten Nutzungsgrad der verwendeten Prozessoren bezeichnet man als *parallele Effizienz* (engl. parallel efficiency). Sie ist definiert als der Quotient von Beschleunigung und Anzahl der Prozessoren:

$$E(p) = \frac{S(p)}{p} \tag{2.4}$$

Zwei weitere auf die Beschleunigung einflussnehmende Faktoren sind der *parallele Mehraufwand* (engl. parallel overhead) und die *Granularität* (engl. granularity) der parallelen Anwendung. Der Mehraufwand kommt durch die zur Parallelisierung der Anwendung notwendigen zusätzlichen Anweisungen innerhalb des Programms zustande. Diese nicht zur Problemlösung notwendigen Anweisungen verlangsamen die Anwendung im Vergleich zur sequentiellen Lösung. Ziel einer effizienten Umsetzung ist es daher, diesen Mehraufwand so gering wie möglich zu halten. Der Aspekt der Granularität resultiert aus der begrenzten Kommunikationsleistung eines Systems. Je feingranularer die zu lösenden Teilprobleme einer Anwendung sind, desto häufiger ist eine Kommunikation notwendig. Diese stellt in gewisser Weise einen sequentiellen Anteil dar und beschränkt daher die Beschleunigung der Anwendung. Um eine effiziente Parallelisierung zu erreichen sollte daher eine Granularität vorliegen, bei welcher der Berechnungsaufwand höher ist als der Kommunikationsaufwand.

2.2 Klassifizierung paralleler Rechnerarchitekturen

Die große Vielfalt der vorhandenen parallelen Rechnerarchitekturen erfordert den Entwurf einer Klassifikation. Das wohl bekannteste Modell von *Flynn* sieht eine Unterteilung der parallelen Architekturen in drei Klassen vor [24]. Die Rechner werden dabei nach der Anzahl ihrer *Instruktions*- und *Datenströme* unterschieden.

MISD-Rechner Dieser *Multiple-Instruction-Single-Data*-Rechner könnte mehrere Instruktionen auf ein und demselben Datum gleichzeitig ausführen. Kein bisher existierender Rechner lässt sich jedoch dieser Kategorie zuordnen. Nach vorherrschender Meinung ist daher diese Klasse auch leer [4, 37, 88].

SIMD-Rechner Ein *Single-Instruction-Multiple-Data*-Rechner führt eine Instruktion auf mehreren Datenströmen gleichzeitig aus. Dies ist die einfachste Art paralleler Architekturen, die in Vektorrechnern (z.B. NEC SX-5, SX-8) oder Feldrechnern implementiert ist. Haupteinsatzgebiet sind die Berechnungen von Vektoroperationen oder Matrizen im Rahmen der Numerik, Mathematik oder Physik, da sich diese besonders leicht und effizient auf Vektorrechnern implementieren lassen.

Eine deutliche Beschleunigung der Anwendung gegenüber einer sequentiellen Version kann jedoch nur bei datenparallelen Operationen erzielt werden, bei denen die Vektoroperationen zum Einsatz kommen. Alle anderen Befehle, wie z.B. Sprungbefehle oder Adressberechnungen, können nur sequentiell ausgeführt werden. Bevor eine Anwendung auf einem Vektorrechner parallelisiert wird, sollte daher eine Analyse des Anteils der datenparallelen Berechnungen vorgenommen werden, um die Effizienz der Implementierung abschätzen zu können.

MIMD-Rechner Die allgemeinste Form eines Parallelrechners bildet der *Multiple-Instruction-Multiple-Data*-Rechner. In dieser Architektur sind mehrere Prozessoren über ein Kommunikationsnetz miteinander verbunden. Daher kann es sich sowohl um ein System mit gemeinsamem Speicher (z.B. NEC SX-4) als auch ein Clustersystem mit verteiltem Speicher handeln (z.B. Kepler-Cluster, Cray-Opteron-Cluster). Die Parallelität liegt in der gleichzeitigen Ausführbarkeit mehrerer Operationen auf mehreren Datenströmen.

Da die meisten parallelen Architekturen in diese Klasse fallen, existieren Erweiterungen der flynnschen Arbeit, die eine genauere Untergliederung vornehmen. Die unterschiedlichen Merkmale und ihre Ausprägungen sind in Tabelle 2.1 dargestellt.

Merkmal	Ausprägung 1	Ausprägung 2
Speicheranordnung (physikalisch)	gemeinsam	verteilt
Adressraum	global, gemeinsam	lokal, privat
Programmiermodell	globaler Adressraum	nachrichtenorientiert
Kommunikations-struktur	Speicherkopplung	Nachrichtenaustausch
Synchronisation	gemeinsame Variablen	synchronisierte Nachrichten
Latenzbehandlung	versteckt	minimiert

Tabelle 2.1: Merkmale einer genaueren Klassifikation von MIMD-Architekturen nach [78].

Das wichtigste Merkmal ist dabei die Art der Speicheranordnung. Dabei unterscheidet man zwischen Parallelrechnern mit gemeinsamem Speicher (engl. shared memory), die einen direkten Zugriff der Prozessoren auf den Speicher realisieren, und Architekturen mit verteiltem Speicher (engl. distributed memory), die einem Prozessor nur einen lokalen Speicherzugriff ermöglichen. Trotz dieser physikalischen Trennung kann die logische Sicht jedoch neben einem lokalen durchaus auch einen gemeinsamen Adressraum darstellen. Ein weiteres Merkmal ist das Programmiermodell, das sich an der verwendeten Adressierung orientiert. Architekturen mit gemeinsamem Speicher verwenden eine direkte Adressierung, wohingegen Rechner mit verteiltem Speicher ein nachrichtenorientiertes Modell verwenden.

2.3 Parallele Ein-/Ausgabearchitekturen

Die Klassifizierung der Rechnerarchitekturen spielt im Zusammenhang der vorliegenden Arbeit eine eher untergeordnete Rolle. Zwar bestimmt die Klassifikation der Architektur das verwendete Programmiermodell, doch ist für die parallele Ein-/Ausgabe in erster Linie die Ein-/Ausgabearchitektur des Rechnersystems von zentraler Bedeutung. Dieser Abschnitt beschreibt daher detailliert die Hardwarekomponenten eines parallelen Ein-/Ausgabesystems. In den ersten vier Unterabschnitten wird daher auf aktuelle Speichergeräte, RAID-Systeme, Topologien und Verbindungsnetzwerke in diesen Architekturen eingegangen. Schließlich erläutert Unterabschnitt 2.3.5 standardisierte Möglichkeiten der Leistungsbewertung der Architekturen.

2.3.1 Speichergeräte

Die von Rechnern erzeugten Daten können auf einer Vielzahl von Medien gespeichert werden. Diese Medien klassifiziert man in die drei Ebenen der primären, sekundären und tertiären Speichergeräte. Ein *primärer Speicher* zeichnet sich dadurch aus, dass eine ständige Stromzufuhr notwendig ist, um die Daten zu erhalten (z.B. Arbeitsspeicher). Die Zugriffszeiten sind kurz und die Kosten pro Byte im Verhältnis zu den anderen Ebenen relativ hoch. Ein klassischer *sekundärer Speicher* ist die Festplatte, die auch nach Abschalten eines Rechners die Daten erhält und deren Zugriffszeiten im Millisekundenbereich liegen. *Tertiäre Speicher* sind Magnetbänder und optische Medien, die vom System vollkommen entfernt und zu einem späteren Zeitpunkt wieder hinzugefügt werden können. Die Zugriffszeiten liegen dabei im Sekundenbereich.

Aufgrund des guten Preis-Leistungs-Verhältnisses werden in parallelen Höchstleistungsrechnern vorwiegend Festplatten zur Speicherung von Anwendungsdaten installiert. Die verwendeten Bussysteme wie SCSI, UltraATA oder Serial ATA bieten zwar technisch mit bis zu 150 MB/s (SATA-I) bzw. 300 MB/s (SATA-II) eine sehr hohe Bandbreite, doch beschreibt dieser Wert lediglich die theoretische Maximalbandbreite der Schnittstelle zwischen Laufwerk und Rechnersystem. In der Praxis ist diese nur dann interessant, wenn Daten aus dem Laufwerkszwischenspeicher ausgelesen oder in diesen geschrieben werden. Anderenfalls ist das magnetische Speichermedium nach wie vor der beschränkende Faktor. Moderne 3,5-Zoll-Festplatten erzielen maximale Transferraten von gerade einmal $60 - 70$ MB/s. In der Praxis sind diese Werte sogar deutlich geringer, da jedes Dateisystem einen Zusatzaufwand darstellt und damit die Leistung reduziert. Diese ist damit für moderne Parallelrechner, die im Allgemeinen eine Transferrate von mehreren hundert Megabyte pro Sekunde erfordern, zu gering.

Eine naheliegende Lösung des Problems liegt in der Parallelisierung der Ein-/Ausgabe. Die eingesetzten Techniken verteilen dabei die Daten in Form von kleinen Teilstücken gleichzeitig auf mehrere Festplatten (engl. disk striping). Im Allgemeinen werden die Daten in Blöcke vordefinierter Größe zerlegt und zyklisch auf die einzelnen Platten verteilt. Die Anzahl der eingesetzten Platten bezeichnet man als *Stripe-Faktor* und die Größe eines Blocks als die *Stripe-Größe*. Der Stripe-Faktor bestimmt den Grad der Parallelität und damit die maximal erreichbare aggregierte Bandbreite des Systems. Im Prinzip könnte man mit dieser Methode durch den Einsatz einer Vielzahl von Platten eine sehr hohe Bandbreite erzielen. Das Problem dabei liegt jedoch in der Zuverlässigkeit, da der Ausfall einer einzigen Platte die verbleibenden Daten unbrauchbar machen könnte.

2.3.2 RAID-Systeme

Um das Problem der Zuverlässigkeit in einem Festplattenverbund zu lösen, entwickelte eine Forschergruppe der Universität von Kalifornien in Berkeley im Jahre 1988 das RAID-System (engl. redundant array of inexpensive[1] disks) [66]. Man postulierte fünf Strategien mit unterschiedlicher Ein-/Ausgabeleistung und Art der Speicherung der Daten - die *RAID-Ebenen*. Die Ebenen RAID 1 bis RAID 5 wurden im Laufe der Zeit um die Ebenen RAID 0, RAID 6, RAID 53 und RAID 10[2] ergänzt.

Die bereits erwähnte Verteilung der Daten auf mehrere Festplatten stellt die Ebene RAID 0 dar. Trotz der fehlenden Datensicherheit wird RAID 0 aufgrund der hohen Transferraten häufig als temporärer Anwendungsspeicher eingesetzt.

Gerade entgegengesetzt verhält es sich mit der Charakteristik eines RAID 1, das in der Regel aus zwei oder auch mehr Festplatten besteht, die dieselben Daten enthalten (engl. mirroring oder duplexing). Somit wird volle Redundanz der gespeicherten Daten gewährleistet, während die Kapazität des Arrays höchstens so groß ist wie die kleinste beteiligte Festplatte. Fällt eine der gespiegelten Platten aus, können die anderen weiterhin die Daten liefern. Dies ist besonders für Echtzeit-Anwendungen unverzichtbar. Erst der Ausfall aller Platten führt zum Totalverlust der Daten. Zwar bietet RAID 1 keine höhere Schreibleistung als eine einzelne Platte, doch kann eine erhöhte Leseleistung erzielt werden, weil Daten von mehreren Festplatten parallel angefordert werden können.

Die Kombination aus RAID 1 und RAID 0 wird RAID 10 genannt. Dabei verbindet man die beiden Faktoren Sicherheit und Leistung, indem die Daten auf mehrere Platten verteilt werden, die ihrerseits wiederum auf jeweils eine weitere Platte gespiegelt werden. Damit sind mindestens vier Platten notwendig, um ein RAID 10-System zu realisieren.

RAID 2 und RAID 3 spielen in der Praxis keine Rolle mehr. Bei RAID 2 werden die Daten in Bitfolgen fester Größe zerlegt und mittels eines *Hamming-Codes* auf größere Bitfolgen abgebildet. Durch Aufteilung der einzelnen Bits des Codeworts auf alle Festplatten kann prinzipiell ein hoher Durchsatz erzielt werden. Jedoch muss dabei die Anzahl der Platten ein Vielfaches der Codewortlänge sein. RAID 3 teilt die Daten byteweise über einzelne Festplatten auf. Die Paritätsinformationen, die sich durch XOR-Verknüpfung der einzelnen Datenbytes ergeben, werden auf einer entkoppelten Platte gespeichert. Die Ineffizienz dieser Methode hat zwei Gründe: Zum einen stellt die dedizierte Paritätsfestplatte einen Flaschenhals dar, zum anderen werden in modernen Systemen Ein-/Ausgabe-Operationen mit größeren Blöcken - bis hin zu mehreren Megabytes - bevorzugt, wie sie in den RAID-Ebenen 4 und 5 realisiert sind.

[1] Aufgrund der fallenden Festplattenpreise durch Massenproduktion spricht man heute von unabhängigen (engl. *independent*) Platten

[2] auch bekannt als RAID 0&1

RAID 4 berechnet ebenfalls Paritätsinformationen, die auf eine dedizierte Festplatte geschrieben werden. Zwar werden dabei größere Blöcke als bei RAID 3 verwendet, doch bleibt der Flaschenhals aufgrund der fest definierten Paritätsplatte bestehen.

RAID 5 bietet sowohl gesteigerte Leistung als auch Redundanz und ist damit die beliebteste RAID-Variante. Darüber hinaus ist es die kostengünstigste Möglichkeit, Daten auf mehr als zwei Festplatten mit Redundanz zu speichern. Bei n Platten sind $(n-1)/n$ der Gesamtkapazität nutzbar[3]. Die verbleibende Kapazität wird für Paritätsdaten verwendet. Die Nutzdaten werden wie bei RAID 0 auf alle Festplatten verteilt. Die Paritätsinformationen werden jedoch nicht wie bei RAID 4 auf einer einzigen Platte konzentriert, sondern ebenfalls verteilt. Die Berechnung der Parität erfordert leistungsfähige RAID-Controller und führt beim Schreiben, im Vergleich zu RAID 0, zu leichter bis erheblicher Verminderung der Datentransferrate. Da die Paritätsinformationen beim Lesen nicht benötigt werden, stehen alle Platten für einen parallelen Zugriff zur Verfügung. Des Weiteren ist die Datensicherheit beim Ausfall einer Platte noch gewährleistet.

Im Unterschied zu RAID 5 können bei RAID 6 bis zu zwei Festplatten ausfallen. Hier werden nicht ein, sondern zwei Fehlerkorrekturwerte berechnet und so verteilt, dass Daten und Paritätsblöcke auf unterschiedlichen Platten liegen.

Genau wie RAID 10 ist RAID 53 eine Kombination aus zwei RAID-Ebenen. Überraschenderweise besteht es jedoch aus den Ebenen RAID 0 und RAID 3. Hierbei werden die Daten auf mehrere virtuelle Platten verteilt, wobei jede virtuelle Platte ein RAID 3-System mit drei Festplatten ist. Dadurch wird bezüglich der Leistung ein ausgeglichenes System geschaffen.

Eine Zusammenfassung der vorgestellten RAID-Ebenen bietet Tabelle 2.2.

2.3.3 Topologien

Mit Hilfe von RAID-Systemen ist es möglich, die Bandbreite von Festplattensystemen zu erhöhen. Entscheidend für die gesamte Ein-/Ausgabeleistung eines Höchstleistungsrechners ist jedoch die topologische Anordnung der Festplatten, welche die Erreichbarkeit der Daten durch die Prozessoren und die Kommunikationsleistung erheblich beeinflusst. Dieser Teilabschnitt beschreibt daher die in der Praxis üblichen Topologien [23].

Rechenknoten mit eigener Platte In dieser Architektur verfügt jeder Rechenknoten über ein eigenes, direkt an ihn angeschlossenes Speichergerät. Dieses kann neben einer einzelnen Platte natürlich auch ein RAID-System sein, um die Datensicherheit oder Leistung zu erhöhen. Die Vorteile liegen im exklusiven Zugriff des

[3] Zum Vergleich: bei RAID 1 lässt sich nur die Hälfte der realen Kapazität verwenden.

RAID	Sicherheit	Speicherplatz	Vorteile	Nachteile
0	keine	N	Leistung	Datensicherheit
1	Spiegelung	$2N$	Datensicherheit, Leseleistung	Speicherbedarf
2	Hamming Code	$\approx 1,5N$	Transferrate	Speicherbedarf, Anfragerate
3	Parität	$N+1$	Transferrate	Anfragerate
4	Parität	$N+1$	Leseanfragerate	Schreibleistung
5	Parität	$N+1$	Anfragerate	Transferrate
6		$N+2$	Datensicherheit	Schreibleistung
10	Spiegelung	$2N$	Leistung	Speicherbedarf
53	Parität	$N + Stripe - Faktor$	ausgeglichene Leistung	

Tabelle 2.2: Zusammenfassung der RAID-Levels nach [57].

Knotens auf die Daten und in der direkten Verbindung zwischen dem Speichergerät und dem Knoten, so dass keine zusätzliche Belastung des im System vorhandenen Kommunikationsnetzwerkes notwendig ist. Gleichzeitig stellt diese Exklusivität jedoch auch einen Nachteil dar, da jeder andere Knoten, der Daten von diesem Speichergerät benötigt, eine Anfrage stellen muss. Dies belastet sowohl die Rechenleistung des Knotens als auch das Netzwerk, das die Daten anschließend kommuniziert. Die Topologie eignet sich daher am besten für Anwendungen, deren Ein-/Ausgabebedarf sich auf die lokal berechneten Daten beschränkt oder die zur Laufzeit lediglich temporäre Daten auf die lokale Platte speichern müssen.

Am Netzwerk angeschlossene Platten Ein Nachteil der gerade dargestellten Topologie liegt in der reduzierten Rechenleistung eines Knotens, die durch Bearbeitung externer Ein-/Ausgabeanfragen entsteht. Dieser Teilabschnitt beschreibt daher die Möglichkeit, Speichergeräte direkt an ein Netzwerk anzuschließen [29]. Dazu ist es natürlich notwendig, dass ein geeignetes Verbindungsnetzwerk vorhanden ist (siehe Abschnitt 2.3.4). Die Vorteile liegen dabei in einer direkten Erreichbarkeit der Daten von allen Prozessoren aus, ohne dabei das vorhandene Kommunikationsnetz zusätzlich zu belasten, und darin, dass keine zwischengeschalteten Knoten für den Zugriff auf die Daten benötigt werden. Jedoch ist zu einem solchen Aufbau zusätzliche Hardware notwendig, die nicht unerhebliche Kosten verursacht.

Spezielle Ein-/Ausgabeknoten Die als letzte beschriebene Topologie verwendet spezielle Datenserver, die als Ein-/Ausgabeknoten fungieren und über ein Verbindungsnetzwerk mit den anderen Knoten kommunizieren können. Dabei können durchaus mehrere Ein-/Ausgabeknoten in einem System vorhanden sein, um eine höhere Bandbreite zu erzielen. Die Verteilung der Daten auf mehrere Server ist dann Hauptaufgabe der im Abschnitt 2.4 erläuterten parallelen Dateisysteme.

2.3.4 Verbindungsnetzwerke

Die Aufgabe des Verbindungsnetzwerkes besteht darin, die Datenkommunikation zwischen einem Speichergerät und einem Rechenknoten zu ermöglichen. Der Transfer zwischen diesen beiden stellt hohe Anforderungen an die Transferrate, da oft große Datenblöcke übertragen werden müssen. Diese Art von Netzwerken nennt man daher auch Systemnetzwerke (engl. *system area network*) oder Speichernetzwerke (engl. *storage area network*) - kurz SAN. Im Folgenden werden nun sowohl die in der Praxis des Höchstleistungsrechnens am häufigsten eingesetzten als auch die im Rahmen der vorliegenden Arbeit verwendeten Verbindungsnetzwerke dargestellt. Zunächst erfolgt jedoch eine kurze Beschreibung von Bussystemen, die die Schnittstelle zwischen Rechner und Verbindungsnetzwerk innerhalb eines Knotens bilden.

Bussysteme

Eines der wohl bekanntesten Bussysteme ist PCI (engl. Peripheral Component Interconnect), das ein 32-Bit-Bussystem für Erweiterungskarten darstellt und an der Southbridge angeschlossen ist. Es bietet eine Datentransferrate von 133 MB/s bei einer Taktung von 33 MHz. Diese Bandbreite ist z.B. beim Einsatz von RAID-Controllern schnell erreicht. Infolgedessen wurde eine 64-Bit-Version entwickelt, die eine Bandbreite von 266 MB/s bei 33 MHz und 532 MB/s bei 66 MHz bietet.

Durch die Entwicklung von PCI-X („PCI Extended“) [68] als kompatibler Weiterentwicklung mit 66, 100 oder 133 MHz Bustakt konnte die Bandbreite auf maximal 1066 MB/s erhöht werden. Dabei kann jedoch nur ein Schacht (engl. slot) mit 133 MHz betrieben werden. Alle weiteren können höchstens mit 66 MHz getaktet werden. Die Anzahl der Schächte kann variiert oder mehrere unabhängige Busse können implementiert werden. Die Version 2.0 von PCI-X sieht 266 und 533 MHz als Bustakt bei einer maximalen Bandbreite von 4,3 GB/s vor. Moderne Ultra320-SCSI, 10-Gigabit-Ethernet- und Fibre-Channel-Karten werden für PCI-X angeboten und in hochwertigen Servern eingesetzt. Seit 2002 ist der Standard PCI-X 1066 in Planung, der Transferraten von 8,5 GB/s erreichen soll. Aufgrund der Entwicklung von PCI-Express wurde dieses Projekt jedoch eingestellt.

Im Gegensatz zum PCI-X-Bus ermöglicht PCI-Express [69, 3] - ehemals 3GIO genannt - auf der elektrischen Ebene eine serielle Punkt-zu-Punkt-Verbindung. Durch die Verwendung von PCI-Signalisierung und -Programmiertechniken kann es von

Betriebssystem und Software jedoch wie ein PCI-Bus behandelt werden. PCI-Express ist vollduplexfähig und arbeitet mit einer Taktrate von 1,25 GHz DDR (engl. double data rate transfer). Daraus berechnet sich die Datenrate einer Verbindung (engl. lane) zu 250 MB/s pro Richtung. Die Spezifikation definiert die Anzahl vorhandener Verbindungen durch die Bezeichnungen von *x1* bis *x32*. Durch diese Erweiterbarkeit erzielt PCI-Express unter Verwendung von 32 seriellen Verbindungen eine bisektionale Bandbreite von 16 GB/s. Durch Anschluss an die Northbridge können darüber hinaus sehr geringe Latenzen erzielt werden. Die dritte Generation von PCI-Express soll sogar einen theoretischen Durchsatz von über 60 GB/s realisieren können. Seit 2005 hat PCI-Express daher sowohl PCI als auch AGP als Standard abgelöst.

Ein weiteres System stellt HyperTransport (HT) [44, 5] dar. Es handelt sich dabei um ein bidirektionales serielles oder paralleles Punkt-zu-Punkt-Verbindungssystem, das integrierte Schaltkreise mit dem Motherboard verbindet. HT ist paketbasiert, unterstützt Taktraten von 200 - 1600 MHz und besitzt die im Vergleich zu anderen Systemen geringste Latenz für Chip-zu-Chip-Verbindungen. Die Busbreiten reichen von 2 bis 32 Bit und erzielen damit theoretisch eine maximale Transferrate von 12,8 GB/s. Darüber hinaus sind unterschiedliche Übertragungsmodi möglich, wie z.B. PIO oder DMA. In den meisten Systemen wird HT an die Northbridge angeschlossen. Es gibt jedoch auch Systeme, in denen HT sowohl North- als auch Southbridge ersetzt. Eine Reihe von Herstellern, wie z.B. nVidia, AMD, Apple (Power Mac G5) und Microsoft (Xbox), unterstützen diese Technologie und ersetzen damit jeweils ihre eigenen proprietären Bussysteme.

1/10-Gigabit-Ethernet

Mit einem Anteil von 28,4% der TOP500-Rechner stellt Gigabit-Ethernet (GE) [56, 30] eines der am häufigsten eingesetzten und gleichzeitig kostengünstigsten Verbindungsnetzwerke dar. GE ist eine Erweiterung des 10/100-Mb/s-Ethernet Standards und unterstützt eine Vollduplex-Übertragung mit Raten von bis zu 1000 Mb/s. In der Praxis erreichen GE-Karten an einem PCI-Bus bis zu 995Mb/s [43]. Ebenso wie seine Vorgänger verwendet GE das CSMA/CD-Protokoll (engl. Carrier Sense Multiple Access with Collision Detection), das den Zugriff der Systeme auf das gemeinsame Medium regelt. Sowohl im Kepler-Cluster als auch im Cray-Opteron-Cluster wird GE als Speichernetzwerk eingesetzt.

10-Gigabit-Ethernet (10-GE) [1] stellt die Weiterentwicklung von GE dar und bietet Übertragungsraten von bis zu 10 Gb/s. Es ermöglicht einen kosteneffizienten Einsatz eines Hochgeschwindigkeitsnetzwerkes sowohl für an ein Netzwerk angeschlossene Speichergeräte (NAS) als auch für SANs. Die Latenz konnte durch den Einsatz eines RDMA-Modus (engl. Remote Direct Memory Access) auf einen Wert reduziert werden, der vergleichbar ist mit anderen Speichersystemen, wie z.B. Fiber-Channel, Ultra320 oder HIPPI (engl. High-Performance Parallel Interface). Zahlreiche Hersteller, wie z.B. Chelsio, Level5, Intel oder Broadcom, produzieren und vertreiben bereits 10-GE-Karten. Leistungsmessungen [43] einer 10-GE-Karte von Intel (Intel Pro/10GbE LR) an einem PCI-X-Bus mit 133 MHz (max. 1066 MB/s) zeigen Übertragungsraten von bis zu 5,7 Gb/s, was einer Effizienz von ca. 68% entspricht.

Fibre Channel

Viele SANs basieren heute auf der Implementierung des Standards *Fibre Channel* [81], dessen erreichte Bandbreiten bei 4 Gb/s liegen. Der Vollduplex-Betrieb erreicht damit theoretische Datentransferraten von bis zu 800 MB/s. Generell können zwei Arten von Implementierungen unterschieden werden: Die *Switched Fabric*, die meist als Fibre Channel (FC) bezeichnet wird, und die *Arbitrated Loop* (FC-AL).

Bei der Switched Fabric handelt es sich um die leistungsfähigste und ausfallsicherste Implementierung von Fibre Channel. Im Zentrum steht der Switch oder der Direktor (engl. director), der alle anderen Geräte mittels direkter Punkt-zu-Punkt-Verbindung zusammenschließt. Durch Verwendung mehrerer Adapter in einem Server kann die Bandbreite durch Bündelung vervielfacht werden. Die Kombination mehrerer Switches erlaubt es, die vorhandene Topologie intelligent zu verwenden, so dass stets der am geringsten belastete Weg genutzt wird. Verfügt ein Server über mehr als einen Adapter, kann ein Speichersubsystem auf mehreren Wegen erreicht werden. Diese Fähigkeit wird als *Multi-Pathing* bezeichnet und erhöht die Leistung und Ausfallsicherheit des SANs. Letzere kann durch den Einsatz einer zweiten, redundanten Fabric, die vollkommen unabhängig arbeitet, weiter gesteigert werden. Das Gesamtsystem kann nun neben dem Ausfall einzelner Datenwege sogar den Ausfall einer ganzen Fabric verkraften. Diese Fähigkeit spielt besonders im Bereich des Totalausfalls (engl. *Disaster Recovery*) eine wichtige Rolle.

FC-AL ist sehr kostengünstig und bildet daher oft den Einstieg in die Welt der SANs. Häufig findet man diese Implementierungen bei kleineren Clustern, in denen es mehreren Knoten möglich ist, auf einen gemeinsamen Massenspeicher direkt zuzugreifen. Es können dabei bis zu 127 Geräte an einem logischen Bus betrieben werden, die sich die verfügbare Bandbreite von 1 GB/s bzw. 2 GB/s teilen. Die Verkabelung erfolgt bevorzugt sternförmig über einen Hub. Eine ringförmige Anordnung ist jedoch ebenso möglich, da viele Geräte über zwei Ein- bzw. Ausgänge verfügen.

Zum Einsatz kommt Fibre Channel z.B. im Cray-Opteron-Cluster in Stuttgart. Die von QLogic verwendete Karte erreicht hier eine maximale Bandbreite von 2 Gb/s. Darüber hinaus bieten noch weitere Hersteller, wie z.B. HP, Emulex, Compaq oder Seagate, entsprechende Fibre-Channel-Komponenten an.

Infiniband

Infiniband [70, 45] ist ein serieller Bus mit sehr hoher Geschwindigkeit zum internen und externen Einsatz. Es ist das Resultat der Vereinigung zweier konkurrierender Systeme: Future I/O von Compaq, IBM und Hewlett-Packard und Next Generation I/O (ngio), das von Intel, Microsoft und Sun Microsystems entwickelt wurde. Bevor der Name gewählt wurde war Infiniband bekannt als System I/O. Intel hat sich seitdem eher auf die Entwicklung der Alternative PCI-Express fokussiert, so dass die zukünftige Entwicklung von Infiniband ungewiss bleibt. Bisher wird Infiniband in lediglich knapp 2% der Superrechner der TOP500-Liste eingesetzt.

Infiniband verwendet einen bidirektionalen seriellen Bus zur kostengünstigen und latenzarmen Datenübertragung. Trotzdem ist es sehr schnell und erreicht Datenübertragungsraten von bis zu 10 Gb/s in beide Richtungen. Der große Vorteil von Infiniband gegenüber TCP/IP/Ethernet liegt in der Minimierung der Latenzzeit durch Auslagerung des Protokollstacks in die Netzwerkhardware.

Um zeitaufwendige Wechsel zwischen Betriebssystem- und Benutzerkontext zu vermeiden, werden zunächst für die Benutzung vorgesehene Speicherbereiche bei der Karte registriert. Dies ermöglicht der Karte eine selbständige Übersetzung von virtuellen in physikalische Adressen. Beim Senden von Daten wird durch das Abbilden verschiedener Kontrollregister des HCAs in den Speicher des Prozesses (*Doorbell-Mechanismus*) die Sendeoperation ohne Umweg über den Betriebssystemkern vorgenommen. Dabei holt sich der HCA die Daten aus dem Hauptspeicher durch Ansteuerung des DMA-Controllers. Das Versenden der so auf dem HCA vorhandenen Daten wird durch den Protokollstack der Karte übernommen.

Um die Latenzzeiten zu minimieren, stellt Infiniband zwei Verbindungsmodi zur Verfügung, die Daten in den Hauptspeicher eines anderen Knotens übertragen oder von dort lesen, ohne das Betriebssystem oder den Prozess auf der Gegenseite zu involvieren. Diese beiden Operationen werden als RDMA Write/RDMA Read (Remote DMA) bezeichnet.

Myrinet

Myrinet ist mittlerweile der Marktführer bei Cluster-Netzwerken. Insgesamt 31,1% aller TOP500-Superrechner verwenden im Juni 2005 ein Myrinet. Das System wird ebenfalls im Kepler-Cluster in Tübingen und im Cray-Opteron-Cluster in Stuttgart eingesetzt. Die Firma Myricom entwickelte 1994 die erste Version des Myrinet [13] als Alternative zum klassischen Ethernet. Neben einem guten Preis-Leistungs-Verhältnis liegt der Hauptvorteil gegenüber Ethernet darin, dass die gesamte Übertragung im Anwenderbereich (engl. user space) durchgeführt und so Störungen mit dem Betriebssystem verhindert werden können. Daher erreicht Myrinet bei kleinen Nachrichten eine sehr geringe Latenz von $10 - 15\mu$s.

Ebenso wie Infiniband bietet Myrinet einen RDMA-Modus, um direkt Daten in den oder von dem Speicher eines anderen Adapters zu übertragen. Zur Verbindung der einzelnen Knoten stehen Crossbar-Switches mit 8 oder 16 Kanälen zur Verfügung. Sämtliche größeren Topologien werden durch ein *Clos*-Netzwerk realisiert, das durch eine hierarchische Anordnung mehrerer Switches das Gesamtnetzwerk realisiert. Dadurch wird die bisektionale Bandbreite zwischen zwei Endpunkten maximiert.

Die vier Generationen von Myrinet erreichen Transferraten von 512 Mb/s, 1,28 Gb/s, 2 Gb/s und 10 Gb/s. In der Praxis liegen die tatsächlichen Werte immer nahe der theoretisch maximalen Leistung der physikalischen Ebene. So erreicht das 2 Gb/s-Myrinet Verbindungen von stabilen 1,98 Gb/s, wohingegen z.B. die Transferrate von Ethernet in Abhängigkeit von der Auslastung des Prozessors zwischen 0,6 und 1,9 Gb/s variiert. Die aktuellste Version Myri-10G ist auf physikalischer Ebene vollständig kompatibel mit dem 10 Gb-Ethernet. Die Auslieferung der ersten Produkte erfolgte im September 2005.

2.3.5 Bewertung der Ein-/Ausgabeleistung

In den letzten Abschnitten wurden zahlreiche Möglichkeiten der Speicherung von Daten in einem Höchstleistungsrechner vorgestellt. Die Vielzahl der Kombinationsmöglichkeiten erschwert daher einen Vergleich zwischen unterschiedlichen Architekturen. Neben der Leistungsoptimierung eines einzelnen Speichergerätes oder eines Verbundes von Speichergeräten durch Parameter wie z.B. Stripe-Größe oder Stripe-Faktor spielen vor allem die topologische Anordnung und das Skalierungsverhalten der einzelnen Geräte in einem Höchstleistungsrechner die bedeutende Rolle. Eine einfache Bewertung der Ein-/Ausgabe, z.B. anhand der theoretisch möglichen aggregierten Bandbreite der im System vorhandenen Geräte, scheidet damit für einen Leistungsvergleich aus.

Abhilfe für diese Problematik schaffen standardisierte Benchmarks, die auf den einzelnen Maschinen übersetzt und ausgeführt werden. Der Fokus der Leistungsbewertung einer parallelen Architektur ist allgemein auf die Gesamtleistung in Form von Fließkommaoperationen pro Sekunde (engl. FLOPS) ausgerichtet. Das zeigt sich auch daran, dass die Rangfolge der TOP500-Liste der schnellsten Rechner der Welt nach diesem Kriterium festgelegt wird. Jedoch ist diese Zahl nicht ausreichend, um ein System vollständig zu charakterisieren. Ein ausgeglichenes System erfordert neben der Bewertung der Rechenleistung auch die Berücksichtigung anderer Aspekte, wie z.B. die Bewertung der Ein-/Ausgabeleistung. Allerdings hat das entsprechende Komitee dies noch nicht berücksichtigt, so dass im Bereich der parallelen Ein-/Ausgabe nur wenig standardisierte Messmethoden existieren. Die meisten Benchmarks zielen dabei auf die verwendete Hardware und die parallelen Dateisysteme ab [74].

Einen eher anwendungsbezogenen Ansatz verfolgt die standardisierte und verbreitete an der Universität Stuttgart entwickelte Benchmark „Effective I/O Bandwidth Benchmark", kurz `b_eff_io` [74]. Sie beinhaltet eine Reihe von synthetischen Messverfahren, die das Verhalten der Bandbreite des Systems bei Variation von Prozessorzahl und Datenmenge bestimmen, und anwendungsorientierten Messverfahren, die die im wissenschaftlichen Rechnen oft verwendeten Zugriffsmuster simulieren. Als Ergebnis liefert das Programm einen Gesamtwert des Systems, der unter Verwendung aller gemessenen Bandbreiten ermittelt wird. Dadurch lassen sich auch sehr unterschiedliche Architekturen miteinander vergleichen. Einziger Nachteil der Benchmark ist jedoch, dass sie auf dem Standard MPI aufsetzt und damit an diesen gebunden ist. Aufgrund der Standardisierung existiert allerdings nahezu kein paralleles Rechnersystem ohne eine Implementierung von MPI. Die alleinige Messung der Bandbreite ist im Bereich der Ein-/Ausgabe meist ausreichend, da die Latenz in diesem Bereich aufgrund der verhältnismäßig hohen Datenmengen und der Zugriffszeiten eines Speichergerätes eher unkritisch ist.

2.4 Parallele Dateisysteme

Im letzten Abschnitt wurden die im Höchstleistungsrechnen am häufigsten eingesetzten Topologien von Ein-/Ausgabesystemen beschrieben. Allen gemein ist dabei die Verwendung mehrerer Speichergeräte, die über ein Netzwerk miteinander verbunden sind. Zur Nutzung dieser Systeme bedarf es nun einer Software zur Speicherung und Wiederherstellung der auf den Geräten abgelegten Daten. An dieses Dateisystem werden zwei zentrale Anforderungen gestellt: Erstens den gleichzeitigen Zugriff auf dieselbe Datei durch mehrere Prozesse zu ermöglichen, wobei nicht notwendigerweise alle Prozesse auf dieselbe Stelle innerhalb der Datei zugreifen müssen, und zweitens eine möglichst effiziente Verteilung der Daten auf die zur Verfügung stehende Ein-/Ausgabehardware zu garantieren.

Die *verteilten Dateisysteme*, wie z.B. NFS (Network File System) [31] oder AFS (Andrew File System) [41], stellen keine geeignete Lösung dar, da sie unter der Prämisse konzipiert wurden, dass ein gleichzeitiger Zugriff mehrerer Prozesse auf dieselbe Datei eher selten ist. NFS besteht z.B. aus NFS-Servern, die das lokale Dateisystem auf die NFS-Klienten exportieren. Dabei werden simultane Anfragen an das Dateisystem serialisiert abgearbeitet, so dass diese Architekturen nicht skalieren.

Im Gegensatz dazu werden in *parallelen Dateisystemen* die Daten typischerweise auf mehrere Ein-/Ausgabeknoten verteilt [57]. Greifen nun mehrere Prozesse auf eine Datei zu, können die Anfragen meist an mehrere Ein-/Ausgabeknoten weitergeleitet werden. Das Skalierungsverhalten des Systems kann daher durch Erhöhung der Anzahl dieser Knoten verbessert werden. Der zusätzliche Nutzen eines weiteren Ein-/Ausgabeknotens hängt dabei natürlich von vielen Faktoren ab, z.B. der Anzahl der pro Sekunde zu speichernden Daten, der Bandbreite des Verbindungsnetzwerks, der Ein-/Ausgabeleistung eines Ein-/Ausgabeknotens, dem Plattendurchsatz und der Anzahl an Platten pro Knoten. Ist z.B. die Bandbreite des Netzwerks auf 100 MB/s limitiert und der Durchsatz eines Ein-/Ausgabeknotens liegt bei 25 MB/s, wäre eine Anzahl von 4 Ein-/Ausgabeknoten optimal. Eine weitere Einheit würde aufgrund des erhöhten Verteilungsaufwandes sogar zu einer Leistungsreduktion des gesamten Systems führen.

2.4.1 Funktionalität paralleler Dateisysteme

Die Anforderungen an ein paralleles Dateisystem wurden weiter oben bereits genannt. Das Problem einer effizienten Verteilung der Daten auf die vorhandenen Platten wird in der Regel durch *Declustering* oder einen *Striping*-Mechanismus unter Angabe einer festen Stripe-Größe gelöst [20, 17]. Die Stripe-Tiefe ergibt sich automatisch aus der Anzahl der vorhandenen Speichergeräte. Der Begriff des Declustering stellt eine allgemeinere Form des Striping dar und ermöglicht im Gegensatz zum Striping die Verteilung der Daten mit variabler Stripe-Größe.

Insgesamt verhält sich eine Architektur dieser Art wie ein RAID 0-System: Die aggregierte Bandbreite ist zwar sehr hoch, aber bei Ausfall nur eines Ein-/Ausgabeknotens sind die Daten unter Umständen verloren. Aus diesem Grund bietet es sich an, die Ein-/Ausgabeknoten selbst mit einem RAID-System, z.B. RAID 1, auszustatten. Dies kann sowohl durch ein spezielles Hardware-RAID als auch durch ein Software-RAID in Verbindung mit zusätzlichen Speichergeräten realisiert werden. Es wäre auch vorstellbar, einen RAID-ähnlichen Mechanismus über sämtliche Ein-/Ausgabeknoten zu implementieren. Doch würde der Zusatzaufwand z.B. für die Paritätsberechnung eine hohe Kommunikationsleistung zwischen den Knoten erfordern und damit die Leistung stark beeinträchtigen.

Eine wesentlich größere Herausforderung an ein paralleles Dateisystem stellt die Unterstützung des gleichzeitigen Zugriffs mehrerer Prozesse dar. Aufgrund der Verteilung der Daten auf unterschiedliche Ein-/Ausgabeknoten ist es Aufgabe des parallelen Dateisystems, einerseits die Abbildung der Zugriffe mehrerer Prozesse auf eine gemeinsame Datei und andererseits die Abbildung der Verteilung einer Datei auf mehrere Ein-/Ausgabeknoten zu organisieren. Die Leistung wird dabei wesentlich durch die Datenverteilung auf die Prozesse und die Stripe-Größe beeinflusst. Stimmen beide Größen überein, kann eine direkte Abbildung der Datenblöcke der Prozesse auf die Ein-/Ausgabeknoten erfolgen. Eine geringe Abweichung erfordert jedoch bereits eine Umverteilung der Blöcke auf unterschiedliche Ein-/Ausgabeknoten und reduziert damit die Leistung des Dateisystems. Dieser Aufwand kann jedoch reduziert werden, wenn das Verteilungsmuster der Blöcke bekannt ist. Jeder Prozess kann dann die seinen Blöcken entsprechenden Ein-/Ausgabeknoten selbst bestimmen und mit ihnen direkt kommunizieren.

Ein Problem entsteht jedoch, wenn mehrere Prozesse auf denselben Datenbereich, der auf mehrere Ein-/Ausgabeknoten verteilt ist, schreibend zugreifen und das Dateisystem sequentielle Konsistenz gewährleisten muss. Dies erfordert eine wohldefinierte Abfolge der Schreibvorgänge. Zur Lösung des Problems bieten parallele Dateisysteme zwei unterschiedliche Zugriffsmuster an: Einen exklusiven Zugriff für einen Prozess durch Sperren der Datei (engl. locking) und damit verbunden eine Einhaltung der Konsistenz oder einen kollektiven Zugriff durch mehrere Prozesse. Die Einhaltung der sequentiellen Konsistenz ist dann Aufgabe des Programmierers.

Eine weitere Möglichkeit der Lösung des Problems liegt in der Verwendung von *Tokens*. Dieser Mechanismus garantiert die richtige Reihenfolge der Abarbeitung von Schreibanfragen an das Dateisystem und wird z.B. vom Dateisystem Vesta von IBM [18] verwendet. Der zusätzliche Aufwand reduziert jedoch die Leistung des parallelen Dateisystems.

2.4.2 Leistungssteigernde Techniken

Ebenso wie sequentielle Dateisysteme verwenden auch parallele Dateisysteme leistungssteigernde Techniken zur Reduktion von Festplattenzugriffen, wie z.B. das Cachen und Puffern von Daten [51, 53]. In Systemen mit getrennten Ein-/Ausgabeknoten und Rechenknoten kann eine Pufferung an beiden Seiten vorgenommen werden. Bei einer Pufferung durch den Rechenknoten spricht man von *client buffering*, anderenfalls von *server buffering*.

Dateisysteme mit *client buffering* halten eine lokale Kopie des gerade verwendeten Datenbereichs vor. Eine Änderung der Daten führt dann lediglich zu einer Modifikation des Puffers ohne diese an den Ein-/Ausgabeknoten weiterzuleiten. Die damit verbundene Problematik bei einem Zugriff durch mehrere Prozesse entsteht auch bei Zwischenspeichern. Lösungsansätze, wie z.B. Cache-Kohärenz-Protokolle, werden in ähnlicher Form auch von den Dateisystemen eingesetzt. Dabei wird festgestellt, welche Prozesse gerade lesend oder schreibend auf welchen Datenbereich zugreifen. Ein Prozess kann dann eine Datei oder eine Stelle in der Datei während eines Schreibvorgangs sperren und andere Prozesse können erst nach Freigabe des Bereichs lesend oder schreibend darauf zugreifen. Die Sperrung der Bereiche wird in der Regel mit Hilfe von Tokens durchgeführt, deren Verwaltung und Vergabe durch einen zentralen Serverprozess durchgeführt wird.

Die Alternative dazu ist das *server buffering*. Bei dieser Strategie werden einzelne Blöcke von den Ein-/Ausgabeknoten zwischengespeichert und können bei entsprechender Anfrage ohne einen weiteren Festplattenzugriff zu den Rechenknoten versendet werden. Die serverseitige Pufferung verhindert somit das Problem der Inkonsistenz, da zu jedem Zeitpunkt lediglich *eine* Kopie des Datenblocks existiert. Darüber hinaus können bei gleichzeitigen Schreibzugriffen mehrerer Prozesse auf einen Block die Daten vom Ein-/Ausgabeknoten zusammengefasst und auf einen einzigen Schreibvorgang reduziert werden. Der wesentliche Nachteil dieser Art der Pufferung liegt in dem erhöhten Kommunikationsaufwand zwischen Rechen- und Ein-/Ausgabeknoten, da sämtliche Anfragen über das Netzwerk kommuniziert werden müssen. Viele Anfragen auf denselben Block könnten dann durch Pufferung beim Rechenknoten effizienter gehandhabt werden.

Die Kombination aus beiden Strategien erlaubt es einem Dateisystem, von den jeweiligen Vorzügen zu profitieren, erfordert jedoch auch die Akzeptanz der Nachteile. Aus diesem Grund gibt es teilweise sehr unterschiedliche Realisierungen innerhalb paralleler Dateisysteme, wie die beiden folgenden Abschnitte zeigen werden.

2.4.3 Kommerzielle parallele Dateisysteme

Dieser Abschnitt beschreibt parallele Dateisysteme, die von Herstellern von Höchstleistungsrechnern entwickelt und kommerziell vertrieben wurden. Einige der Systeme sind zwar bereits veraltet, werden aber, weil sie Pionierleistungen darstellen, trotzdem kurz erwähnt. Die Hauptziele der Entwicklung lagen vorwiegend in der Beseitigung der im letzten Abschnitt dargestellten Probleme.

Intel PFS

Das parallele Dateisystem PFS (Parallel File System) [48] wurde von Intel für den Einsatz auf den Paragon-Superrechnern entwickelt. Die Paragon ist ein Rechner mit verteiltem Hauptspeicher und mehreren Ein-/Ausgabeknoten. PFS unterstützt dabei sowohl Dateien von NFS als auch von UFS (Unix File System). Die Daten werden anhand einmal festgelegter Parameter auf die Ein-/Ausgabeknoten verteilt. Das Problem des gleichzeitigen Zugriffs mehrerer Prozesse wird unter Verwendung von 6 unterschiedlichen Ein-/Ausgabemodi gelöst, wobei jeder Modus eine spezielle Zugriffssemantik hat, die sich auch auf die Leistung auswirkt (siehe Tabelle 2.3). Kollektive Methoden erfordern dabei die Teilnahme aller zugehörigen Prozesse an einem Zugriff, während einfache Methoden durch einen einzelnen Prozess unabhängig voneinander durchgeführt werden können.

Die Schnittstelle zu PFS ist eine Erweiterung des Unix-Dateisystems. PFS implementiert darüber hinaus zahlreiche neue Funktionen zum Öffnen einer Datei, Setzen des Ein-/Ausgabemodus und parallelen Zugriff. Auch nicht-blockierende (asynchrone) Zugriffe werden unter Verwendung zahlreicher Synchronisationsmodi unterstützt. Trotz dieser Vielfalt sind nicht alle Unix-Befehle implementiert. Es können z.B. keine ausführbaren Dateien gestartet werden, da PFS keine Mechanismen zur Abbildung von Dateiinhalten auf mehrere Speicherbereiche besitzt.

E/A-Modus	Dateizeiger	Zugriffsart	Zugriffskontrolle
M_UNIX	lokal	einzeln	Serialisierung
M_LOG	gemeinsam	einzeln	Serialisierung
M_SYNC	gemeinsam	kollektiv	Serialisierung
M_RECORD	lokal	kollektiv	Partitionierung
M_GLOBAL	gemeinsam	kollektiv	identischer Zugriff
M_ASYNC	lokal	einzeln	keine

Tabelle 2.3: Ein-/Ausgabemodi von Intels PFS nach [57].

IBM GPFS

Das von IBM entwickelte parallele Dateisystem GPFS (General Parallel File System) [9] basiert auf dem bereits zuvor entwickelten Dateisystem Tiger Shark [33], das aufgrund hoher Transferraten für Multimediaanwendungen eingesetzt wurde. Während bisherige parallele Dateisysteme über eine standardisierte Schnittstelle, wie z.B. Unix, in Verbindung mit speziellen Funktionen zur parallelen Ein-/Ausgabe verfügen, liegt GPFS nur eine Standard-Unix-Schnittstelle zugrunde. Die Funktionen des parallelen Dateisystems sind, für den Anwender nicht sichtbar, innerhalb der Schnittstelle verborgen. Der Nachteil dieser Vereinfachung liegt jedoch in der etwas verminderten Leistung des Systems.

Ebenso wie andere Dateisysteme für Rechner mit verteiltem Hauptspeicher partitioniert auch GPFS die Daten auf mehrere Ein-/Ausgabeknoten. Zur Sicherung der Cache-Kohärenz verwendet GPFS ein client-buffering in Verbindung mit einem globalen Sperrmechanismus, der die einzelnen Dateibereiche mit Hilfe eines Token-Managers sperrt bzw. freigibt.

Durch kontinuierliche Protokollierung von Veränderungen am Dateisytem (engl. journaling) und den möglichen Einsatz eines Software-RAID 10 verfügt GPFS über eine hohe Zuverlässigkeit und Nutzbarkeit.

SGI XFS

Mit dem Dateisystem XFS [87] von SGI wird nun ein System für Rechner mit gemeinsamem Hauptspeicher vorgestellt. Durch diesen speziellen Einsatz entfallen viele der bereits dargelegten Probleme. Da keine externen Ein-/Ausgabeknoten und getrennten Puffer existieren, gestaltet sich die Zugriffskontrolle konkurrierender Prozesse als sehr einfach. Das Problem der sequentiellen Konsistenz bleibt jedoch bestehen und wird in XFS durch einfaches Sperren der gesamten Datei realisiert. Dadurch ist es unmöglich, dass parallele Zugriffe auf dieselbe Datei durchgeführt werden können.

XFS versucht die Datentransferrate durch ein zusammenhängendes Lesen angrenzender Blöcke auf dem Festspeicher zu maximieren. Darüber hinaus verzögert XFS die Speicherung von einzelnen Blöcken so lange wie möglich, um die Wahrscheinlichkeit dafür zu erhöhen, dass die Anwendung weitere Daten in angrenzende Blöcke schreibt. Durch die Bündelung der kleinen Blöcke zu einem großen zusammenhängenden Block kann die Leistung deutlich gesteigert werden.

2.4.4 Parallele Dateisysteme auf Clustern

Ein Trend im Bereich des Höchstleistungsrechnens bestand in den letzten Jahren in der Entwicklung von Clustersystemen. Dabei werden Standardrechner verwendet, die über ein schnelles Kommunikationsnetzwerk zu einem leistungsstarken Rechner mit verteiltem Hauptspeicher verbunden sind. Typischerweise findet man als Betriebssystem dieser kostengünstigen Alternative Windows oder Linux vor. Die Entwicklung eines parallelen Dateisystems für Cluster muss daher neben der Sicherung von sequentieller Konsistenz auch client-buffering und/oder server-buffering unterstützen, um eine hohe Leistung zu erreichen.

In vielen Clustersystemen findet man keine speziellen Ein-/Ausgabeknoten, sondern eine Architektur von Rechenknoten, von denen eine Teilmenge - oder auch alle - zusätzlich als Ein-/Ausgabeknoten fungiert. Erfolgen sowohl Kommunikation als auch Ein-/Ausgabe über dasselbe Kommunikationsmedium, bietet sich ein client-buffering an, um die Leistung des Systems nur gering zu beeinträchtigen. Zur Sicherung der Konsistenz wird üblicherweise ein Token-Mechanismus zur Sperrung von einzelnen Dateiblöcken eingesetzt.

Die Zuverlässigkeit innerhalb eines Clustersystems ist in der Regel geringer als in speziellen Höchstleistungsrechnern, so dass der Ausfall eines Ein-/Ausgabeknotens durchaus realistisch ist. Die meisten Dateisysteme replizieren aus diesem Grund die Daten auf mehrere Ein-/Ausgabeknoten mit Techniken, die auch in RAIDs eingesetzt werden.

Schließlich liegt ein weiteres Problem in der Art und Weise der Verteilung der Daten auf die Ein-/Ausgabeknoten. Die maximale aggregierte Bandbreite wird zwar erreicht, wenn sämtliche Rechenknoten auch Ein-/Ausgabeknoten darstellen, doch erhöht sich damit das Risiko eines Datenverlustes durch den Ausfall eines Knotens und zusätzlich führt der hohe Übertragungsaufwand zu einer insgesamt reduzierten Kommunikationsleistung des Netzwerks. Die nun folgenden Unterabschnitte zeigen verschiedene Lösungsmöglichkeiten für dieses Problem auf.

xFS

Das Dateisystem xFS [7] wurde in Berkeley entwickelt und zeichnet sich dadurch aus, dass die Daten nicht auf alle verfügbaren Ein-/Ausgabeknoten, sondern nur auf eine zuvor definierte Untermenge verteilt werden. Jede Gruppe agiert dabei wie ein RAID-System und garantiert so eine hohe Datensicherheit und Zuverlässigkeit.

Da die gesamte Software für das Dateimanagement auf jedem Ein-/Ausgabeknoten verfügbar ist und auch jede Datei von genau einem dieser Knoten verwaltet wird, sind keine speziellen Dateiserver notwendig. Zur Sicherung der Cache-Konsistenz werden Tokens zur Sperrung auf Datenblockebene verwendet. Dabei kann das Problem des *false sharing* auftreten, wenn zwei konkurrierende Prozesse zwar auf denselben Datenblock, nicht aber auf denselben Bereich innerhalb des Blocks zugreifen. Führen nun die Prozesse mehrere kleine Schreibzugriffe auf den jeweiligen Block durch, springt die Sperrung zwischen den beiden hin und her, obwohl sich die Datenbereiche nicht überlappen. Dies beeinträchtigt die Leistung des Systems sehr stark. Aus diesem Grund nehmen andere Systeme eine Sperrung auf Byte-Ebene vor. Der Nachteil liegt dann in dem deutlich höheren Verwaltungsaufwand der Tokens, so dass insgesamt keine optimale Lösung für dieses Problem existiert.

PVFS

Das von der Clemson University entwickelte parallele Dateisystem PVFS (Parallel Virtual File System) [16] wurde speziell für selbsterstellte Clustersysteme konzipiert. Es handelt sich um ein weit verbreitetes Dateisystem, das auf einigen der während dieser Arbeit zur Verfügung stehenden Rechnersysteme installiert ist. Im Unterschied zu xFS können in PVFS auch separate Ein-/Ausgabeknoten eingesetzt werden. Trotzdem kann die dafür notwendige Serversoftware auch auf jedem normalen Rechenknoten mit Anbindung an einen Festspeicher eingesetzt werden. Die grundlegende Architektur ist in Abbildung 2.1 dargestellt. Sie besteht aus Ein-/Ausgabeknoten, Ein-/Ausgabeklienten und einem Metadaten-Server. Ein wesentlicher Vorteil liegt darin, dass die Knotentopologie dabei frei wählbar ist.

Die Ein-/Ausgabeklienten entsprechen in der Regel den Rechenknoten und der Metadaten-Server wird auf einem beliebigen Rechner im System installiert. Er koordiniert die Verteilung der Daten auf die Ein-/Ausgabeknoten und gibt den Speicherort eines Datenblocks an die Klienten weiter. Der Zweck der Ein-/Ausgabeknoten liegt im Zugriff auf die PVFS-Daten. Dazu werden unter Verwendung der Metadaten direkte Verbindungen zwischen einem Klienten und dem Ein-/Ausgabeknoten hergestellt, um möglichst effizient eine Schreib- oder Leseoperation durchführen zu können. Das Problem der klientenseitigen Dateninkonsistenz umgeht PVFS durch eine serverseitige Pufferung der Daten.

PVFS setzt auf dem bestehenden Dateisystem auf und verwendet das TCP/IP-Protokoll zur Kommunikation zwischen den drei Instanzen. Darüber hinaus kann der Anwender für jede einzelne Datei getrennte Striping-Parameter angeben. Dies ermöglicht bis zu einem gewissen Grad eine Optimierung der Leistung des Systems.

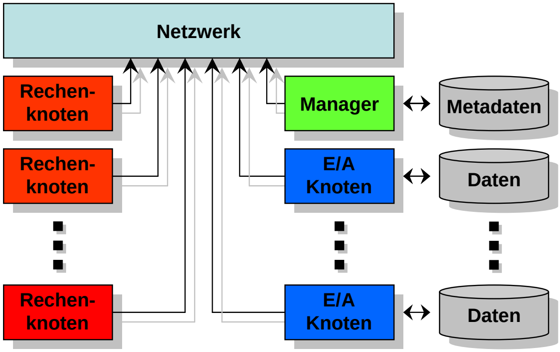


Abbildung 2.1: Schematischer Aufbau der Architektur von PVFS.

Ein wesentlicher Nachteil von PVFS besteht jedoch darin, dass es keine Datenredundanz oder Datenwiederherstellungsmechanismen gibt, die zum Zeitpunkt eines Knotenausfalls die Sicherheit der Daten gewährleisten würden. Bei Ausfall eines einzigen Ein-/Ausgabeknotens ist das gesamte System nicht mehr nutzbar. Ein weiterer Nachteil ist, dass nur ein Metadaten-Server eingesetzt werden kann. Mit einer steigenden Zahl von Klienten entwickelt sich dieser Server zum Flaschenhals des Systems. Darüber hinaus ist bei der Verwendung von TCP/IP sowohl die Anzahl gleichzeitig geöffneter Kommunikationskanäle als auch aufgrund langer Kopfdaten die Leistung beschränkt. PVFS ist zusätzlich an die von Linux vorgegebene Dateigrößengrenze von 2 GB gebunden.

PVFS 2

Die doch teilweise erheblichen Mängel von PVFS wurden in der nachfolgenden Version PVFS 2 nahezu vollständig beseitigt. Die Entwicklung von PVFS 2 begann im Jahre 2001, und eine erste Release-Version ist seit November 2004 verfügbar. Die wesentlichen Merkmale von PVFS 2, die jedoch noch nicht vollständig umgesetzt wurden, sind:

- Modulare Unterstützung unterschiedlicher Netzwerksysteme, wie z.B. Ethernet oder Myrinet, um das Dateisystem von TCP/IP zu entkoppeln.

- Modulare Unterstützung unterschiedlicher Speichermechanismen zur Optimierung von Zugriffen (z.B. durch asynchrone Ein-/Ausgabe).
- Unterstützung mehrerer Metadaten-Server zur Vermeidung eines Flaschenhalses.
- Redundanz sowohl der Daten als auch der Metadaten für den Fall eines Systemabsturzes. Der Anwender kann dabei selbständig die redundante Speicherung der Dateien einleiten und den Algorithmus wählen, der dabei eingesetzt werden soll.
- Verbesserung der Kompatibilität zu UNIX.
- Verbesserte Optionen zur Datenverteilung, die von entsprechenden Modulen gesteuert wird. Es ist möglich, diese zu erweitern bzw. neue hinzuzufügen, so dass die Verteilung den Anforderungen der Anwendung genügt.

Die Notwendigkeit der Umsetzung dieser Merkmale beweist die hohe Relevanz eines parallelen Dateisystems für die Effizienz eines Höchstleistungssystems. Jedoch sind parallele Dateisysteme im Allgemeinen nicht für spezielle Anwendungsprobleme optimiert, wie z.B. die im wissenschaftlichen Rechnen sehr häufig auftretenden Zugriffe auf unzusammenhängende, aber äquidistante Blöcke (engl. strided access). Insgesamt lässt sich das Anwendungsmodell sehr schwer mit dem Dateisystemmodell vereinbaren. Aus diesem Grund existieren Schnittstellen zur parallelen Ein-/Ausgabe, die auf den parallelen Dateisystemen aufsetzen. Diese stellen dem Anwender Methoden zur Verfügung, mit deren Hilfe sich die parallele Ein-/Ausgabe der Anwendung besser, effizienter und komfortabler formulieren lässt. Der nächste Abschnitt beschreibt diese als *Ein-/Ausgabe-Middleware* bekannte Schicht.

2.5 Schnittstellen zur parallelen Ein-/Ausgabe

Die insgesamt erzielte Ein-/Ausgabeleistung von Hardware (Speichermedien, Netzwerk etc.) und Software (parallele Dateisysteme) hängt von der Struktur der zu übertragenden Daten ab. Eine optimale Leistung wird erreicht, wenn möglichst große, zusammenhängende Datenblöcke gespeichert werden sollen. In der Realität gestalten sich die Zugriffsmuster paralleler Anwendungen jedoch derart, dass meist viele kleine Datenmengen übertragen werden müssen.

Ein Ziel der in diesem Abschnitt vorgestellten Schnittstellen zur parallelen Ein-/-Ausgabe ist daher der Einsatz geeigneter Techniken zur Optimierung der parallelen Ein-/Ausgabeleistung. Die *kollektive Ein-/Ausgabe* koordiniert z.B. den simultanen Zugriff mehrerer Prozesse auf eine Datei und kann viele einzelne Anfragen in eine größere zusammenfassen. Auch durch die Angabe zusätzlicher *Informationen* (engl.

hints) über das verwendete Ein-/Ausgabesystem, wie z.B. die Stripe-Größe oder den Stripe-Faktor, kann die Ein-/Ausgabeleistung optimiert werden.

Die Standardschnittstellen von Unix oder Fortran I/O stellen solche Techniken nicht zur Verfügung und können daher im Bereich der parallelen Ein-/Ausgabe nicht eingesetzt werden. Aus diesem Grund beschreibt dieser Abschnitt drei weit verbreitete Schnittstellen, die eine parallele Ein-/Ausgabe unterstützen: High Performance Fortran (HPF), das Scalable I/O Initiative Low-Level API (SIO LLAPI) und die im Standard MPI-2 integrierte Schnittstelle zur parallelen Ein-/Ausgabe (MPI-IO). Auch wenn diese drei Schnittstellen auf unterschiedlichen Ebenen der Abstraktion zum Einsatz kommen, handelt es sich doch bei allen um eine Low-Level-Schnittstelle in dem Sinne, dass keine zusätzlichen Informationen über die Struktur der zu übertragenden Daten gespeichert werden.

Die Wahl fiel auf die genannten Schnittstellen, da sie sich einerseits als Standardschnittstellen zur parallelen Ein-/Ausgabe entwickelt haben und andererseits ihr Design unabhängig von der darunterliegenden Implementierung ist. In diesem Zusammenhang ist es wichtig den Unterschied zwischen einer Schnittstelle und der Implementierung einer Ein-/Ausgabebibliothek oder eines Dateisystems zu verdeutlichen: Während spezielle Dateisysteme, wie z.B. GPFS von IBM oder PFS von Intel, darauf ausgelegt sind die Fähigkeiten des vorliegenden Systems bestmöglich zu unterstützen, liegt das Ziel bei HPF I/O, MPI-IO und SIO LLAPI in der Realisierung einer möglichst plattformunabhängigen Schnittstelle, um einen weitreichenden Einsatz zu garantieren.

Aus diesen Standardisierungsbemühungen entwickelte sich jedoch der Konflikt zwischen dem Ausschöpfen der vorhandenen Möglichkeiten eines speziellen Systems einerseits und dem Erhalt der Portabilität der Schnittstelle andererseits. Gelöst werden konnte dieser Konflikt, indem die Funktionalität der Schnittstellen zur Sicherung der Portabilität auf ein Mindestmaß reduziert wurde und zur Optimierung der Leistung zusätzliche Systeminformationen herangezogen werden können.

2.5.1 High Performance Fortran (HPF)

HPF stellt funktionale Erweiterungen von Fortran zur Verfügung, die es dem Anwendungsentwickler erlauben, mit Hilfe von Übersetzerdirektiven datenparallele Bereiche, wie z.B. unabhängige Schleifen, automatisch zu parallelisieren oder die Art und Weise der Verteilung von Arrays auf die einzelnen Prozesse zu definieren. Das Programmiermodell folgt daher überwiegend dem Ansatz der *Datenparallelität*, d.h., jeder Prozess einer parallelen Anwendung führt dieselbe Operation auf einem unterschiedlichen Teil der Daten aus. Obwohl dieses Modell restriktiver ist als die Programmierung über Nachrichtenaustausch (siehe Abschnitt 2.5.3), vereinfacht es doch die Optimierung der parallelen Ein-/Ausgabe, da globale Ein-/Ausgabeoperationen sehr leicht zu erkennen sind.

Bis heute sind zwei Versionen von HPF erschienen: HPF-1 im Jahre 1993 [39] und HPF-2 im Jahre 1997 [40]. HPF-1 ermöglicht lediglich die Behandlung regulärer Arrays und den Einsatz einfach gestalteter Muster zur Datenverteilung. Die Aufteilung eines multidimensionalen Arrays einer Anwendung auf die Prozesse kann dabei in jeder Dimension blockweise (BLOCK), zyklisch (CYCLIC) oder gar nicht erfolgen. Die Festlegung der Verteilungsmuster der Daten kann dabei entweder zur Übersetzungszeit oder zur Laufzeit erfolgen.

Aufgrund der expliziten Angabe von Datenverteilungen und des datenparallelen Ansatzes ist keine zusätzliche spezielle Schnittstelle zur parallelen Ein-/Ausgabe notwendig. Ein in HPF-1 definiertes Array besitzt alle notwendigen Informationen, um darauf eine kollektive Operation durchführen zu können. HPF-2 realisiert daher lediglich kleinere Erweiterungen in Form von komplizierteren Verteilungsmustern von Arrays auf die Prozesse und einer nicht-blockierenden (asynchronen) Ein-/Ausgabe.

2.5.2 Scalable I/O Initiative Low-Level API

Das SIO-Projekt definiert die Low-Level-Schnittstelle 1996 [19] als Erweiterung von POSIX[4] [46]. Das Ziel von SIO LLAPI liegt in der Definition einer portablen Schnittstelle mit einer möglichst umfangreichen Funktionalität, um viele Modi von paralleler Ein-/Ausgabe zu unterstützen. Darüber hinaus besitzt SIO LLAPI einen umfangreichen Hint-Mechanismus und ermöglicht die explizite Kontrolle von Caches, so dass Anwendungen die Konfiguration des Dateisystems stark beeinflussen können.

Ebenso wie HPF besteht auch SIO LLAPI aus einer Menge von Grundfunktionen, die eine Implementierung unterstützen muss, um standardkonform zu sein, und zwei optionalen Erweiterungen, die eine kollektive Ein-/Ausgabe und ein schnelles Kopieren von Daten ermöglichen. Zusätzlich lässt der Standard benutzerdefinierte Erweiterungen des Hint-Mechanismus zu.

Durch seine Systemnähe ist SIO LLAPI eher für Systemprogrammierer geeignet, die, auf der Schnittstelle aufbauend, höher entwickelte Bibliotheken zur parallelen Ein-/Ausgabe konzipieren und realisieren. Darunter fällt z.B. die Möglichkeit der Realisierung von lokalen und gemeinsamen Dateizeigern. SIO LLAPI besitzt lediglich eine Schnittstelle zur Sprache C und verwendet selbst für einfache Datenstrukturen, wie z.B. Integer-Werte, ein eigenes Typsystem.

2.5.3 MPI-IO

Von den bisher beschriebenen Schnittstellen soll MPI-IO am ausführlichsten dargestellt werden. HPF-IO impliziert zwar durch den datenparallelen Ansatz die Unterstützung der parallelen Ein-/Ausgabe, doch besitzt sie keine sehr umfangreiche

[4] portable operating systems interface

Funktionalität. Die Schnittstelle SIO LLAPI bietet sich eher für Systemprogrammierer als für Anwendungsentwickler an. Daher existieren deutlich weniger Implementierungen dieser Schnittstelle, was zu einer gegenüber MPI-IO deutlich geringeren Verbreitung geführt hat. Obwohl die Komplexität von MPI-IO sehr hoch ist, führte dies nicht zu einer Reduktion ihres Einsatzes auf Parallelrechnern, so dass sie heutzutage auf nahezu jedem Parallelrechnersytem verfügbar ist.

Ihren Anfang nahm die Entwicklung von MPI-IO im Jahre 1994 bei IBM. Sie entstand aus der Erkenntnis, dass eine parallele Ein-/Ausgabe zwei grundlegende Abstraktionen benötigt, die im Standard MPI [58] bereits vorhanden sind: Die Fähigkeit, eine Gruppe von Prozessen über MPI-Kommunikatoren zu definieren, und die Möglichkeit, komplexe Zugriffsmuster mit Hilfe von MPI-Datenstrukturen festzulegen. Darüber hinaus bot sich die Analogie der Kommunikation zwischen zwei Prozessen und der Kommunikation zwischen Prozess und Festplatte geradezu an. Auf dieser Grundlage entwickelte man eine Schnittstelle zur parallelen Ein-/Ausgabe, die im Jahre 1997 in den Standard MPI-2 [59] aufgenommen wurde und in Anlehnung daran als MPI-IO bezeichnet wird.

Die Verwendung vieler Funktionen von MPI erfordert jedoch auch die Verfügbarkeit einer MPI-Bibliothek. Da bereits zu dieser Zeit MPI als De-facto-Standard weit verbreitet war, konnte sich MPI-IO mühelos als Standard zur parallelen Ein-/Ausgabe durchsetzen. Trotz der starken Bindung an MPI erfolgt die Implementierung von MPI-IO jedoch separat.

Der Funktionsumfang von MPI-IO erstreckt sich auf insgesamt 69 Funktionen, die sich in die folgenden Teilgebiete unterteilen lassen.

Datenabstraktion

Die Abstraktion der Daten erfolgt in MPI-IO unter Verwendung des aus MPI-1 bekannten Typsystems. Dabei können Datenstrukturen, ähnlich wie in C oder Fortran, aus grundlegenden Datentypen, wie z.B. Integer, Byte oder Fließkomma, zusammengesetzt werden. Diese neue Struktur kann dann verwendet werden, um sie in einem Schritt zu speichern oder zu laden. MPI-IO unterstellt dabei, dass die gelesenen Daten auch tatsächlich auf die erstellte Struktur passen. Dies ist ein wesentlicher Nachteil, da eine Typverletzung von MPI-IO nicht bemerkt werden kann. Die Zusammensetzung von Datenstrukturen erlaubt darüber hinaus in MPI-IO die Partitionierung von Daten auf mehrere Prozesse. Dies wird in einem folgenden Unterabschnitt genauer dargestellt.

Dateimanagement

Dieses Teilgebiet beinhaltet die zur Verwendung von Dateien grundlegenden Funktionen, die aus dem POSIX-Standard bekannt sind. Dabei handelt es sich um Metho-

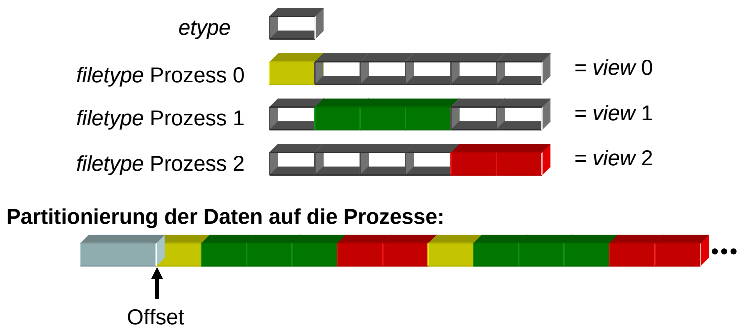


Abbildung 2.2: Schematischer Aufbau unterschiedlicher Dateiansichten verschiedener Prozesse in MPI nach [32].

den wie z.B. das Öffnen, Schließen oder Löschen von Dateien. Anwendungen übergeben den Methoden einen MPI-Kommunikator, um die auf eine Datei gemeinsam zugreifenden Prozesse zu spezifizieren, und legen mit Schlüsselwörtern den geplanten Zugriffsmodus auf die Daten fest. Als Rückgabewert erhält der Benutzer einen eindeutigen Bezeichner auf die Datei (engl. file handle), mit dessen Hilfe er auf die Daten zugreifen kann.

Datenpartitionierung

Unzusammenhängende Datenzugriffe werden durch die Definition von Zugriffsmustern ermöglicht. Diese Partitionierung der Daten (engl. file view) kann von jedem beteiligten Prozess unabhängig von den anderen definiert werden. Für den jeweiligen Prozess ist dann nur diese Stelle innerhalb einer Datei sichtbar. Abbildung 2.2 zeigt die Vorgehensweise bei der Erstellung einer Sicht. Eine wichtige Voraussetzung dabei ist, dass die Datei aus gleichen, grundlegenden Datentypen (engl. etype) besteht. Ein *etype* kann jede Art von MPI-Datentyp sein und repräsentiert die Granularität von MPI-IO. Die Zusammensetzung dieser Typen bildet dann die Sicht auf die Daten (engl. file type) und stellt damit eine Art Zugriffsmaske dar, die Zwischenräume enthalten kann, welche ebenfalls ein Vielfaches vom Basisdatentyp sein müssen. Anschließend wird die Datei durch Aktivieren der Sicht auf die Prozesse partitioniert.

Die Methoden zur Datenpartitionierung bilden die Grundlage der im folgenden Teilabschnitt dargestellten kollektiven Zugriffe.

Datenzugriff

Die Datenzugriffsmethoden bilden mit 30 Funktionen den umfangreichsten Teil von MPI-IO und lassen sich in *Zugriffsmethoden unter der Angabe von Offsets*, *kollektive Zugriffsmethoden* und *asynchrone Zugriffsmethoden* unterteilen.

Explizite Angabe von Offsets Die Datenzugriffsmethoden von MPI-IO können Offsets auf drei unterschiedliche Arten spezifizieren: Durch lokale Dateizeiger, gemeinsame Dateizeiger oder unter Angabe eines expliziten Offsets. Da die Sicht auf die Daten für einen Prozess durch ein Template definiert ist, verhalten sich Offsets relativ zu einer Sicht und sind aus diesem Grund ebenfalls in Einheiten von etype anzugeben. Daraus folgt, dass derselbe Offset in unterschiedlichen Sichten auf unterschiedliche Daten zeigen kann. Daher müssen Prozesse, die einen gemeinsamen Dateizeiger verwenden, eine identische Sicht auf die Datei definieren.

Einfache Schreib- und Leseoperationen verwenden per Definition einen lokalen Dateizeiger, d.h., jeder Prozess bekommt innerhalb derselben Datei einen eigenen Dateizeiger zugewiesen. Gemeinsame Dateizeiger werden oft dann verwendet, wenn mehrere Prozesse koordiniert auf eine Datei zugreifen sollen. Diese Zugriffe sind in der Regel weniger effizient als lokale Dateizeiger, weil MPI-IO die aktualisierte Dateizeigerposition nach jedem Zugriff an alle beteiligten Prozesse kommunizieren muss. Die drei Arten der Angabe von Offsets können - unter Berücksichtigung der Restriktionen - innerhalb einer geöffneten Datei beliebig kombiniert werden.

Kollektive Zugriffsmethoden Die gleichzeitige Bearbeitung einer Datei durch mehrere Prozesse bezeichnet man als kollektive Ein-/Ausgabe. Die Syntax in MPI-IO folgt dabei soweit wie möglich der der nicht-kollektiven Ein-/Ausgabe.

Das Hauptziel der kollektiven Zugriffe besteht darin, dass durch Bündelung vieler kleiner Anfragen an unzusammenhängende Daten zu einer einzigen, zusammenhängenden Anfrage eine Leistungssteigerung erzielt werden kann. Um dies auf jeder Ebene zu erreichen, gibt es in MPI-IO zu jeder einfachen Zugriffsfunktion ein kollektives Gegenstück.

Im Gegensatz zu MPI-1 ist in MPI-IO auch eine nicht-blockierende, kollektive Ein-/Ausgabe implementiert. Die dadurch entstehende Problematik der Synchronisation wird im folgenden Unterabschnitt näher erläutert.

Asynchrone Zugriffsmethoden Diese Methoden werden häufig zur Leistungsoptimierung eingesetzt. Dabei ist es möglich, die Ein-/Ausgabe mit weiteren Berechnungen einer Anwendung zu überlappen. Grundlegende Voraussetzung für das Gelingen ist die Unabhängigkeit der Daten. Dadurch ergibt sich gleichzeitig eine obere Schranke für die Dauer eines Überlappens. Ein weiteres Problem entsteht beim Einsatz der kollektiven Ein-/Ausgabe. Viele dieser Methoden benötigen, sowohl vor als

Positionierung	**Synchronisierung**	**Koordination** *einfach*	**Koordination** *kollektiv*
expliziter *Offset*	*blockierend*	read_at write_at	read_at_all write_at_all
	nicht *blockierend*	iread_at iwrite_at	read_at_all_begin read_at_all_end write_at_all_begin write_at_all_end
individueller *Dateizeiger*	*blockierend*	read write	read_all write_all
	nicht *blockierend*	iread iwrite	read_all_begin read_all_end write_all_begin write_all_end
gemeinsamer *Dateizeiger*	*blockierend*	read_shared write_shared	read_ordered write_ordered
	nicht *blockierend*	iread_shared iwrite_shared	read_ordered_begin read_ordered_end write_ordered_begin write_ordered_end

Tabelle 2.4: Die 30 Datenzugriffsmethoden von MPI-IO. Zur besseren Übersicht wurde das Präfix `MPI_File_` weggelassen.

auch nach einem Festplattenzugriff, einen globalen Kommunikationsschritt zur Synchronisierung der Daten. Um dabei Berechnungen und Ein-/Ausgabe überlappen zu können, darf die zweite Kommunikation erst nach Abschluss der asynchronen Methode durchgeführt werden.

MPI-IO löst dieses Problem unter Verwendung eines Mechanismus zum aufgeteilten kollektiven Datenzugriff (engl. split collective data access). Dabei werden kollektive Funktionen sowohl zur Initialisierung als auch zur Beendigung eines asynchronen Datenzugriffs verwendet. Dazu stehen zusätzliche - um die Schlüsselworte `begin` bzw. `end` erweiterte - Methoden zur Verfügung.

Eine Übersicht über alle in diesem Teilabschnitt dargestellten, durch Kombination der unterschiedlichen Modi entstehenden Zugriffsfunktionen gibt Tabelle 2.4.

Zum Abschluss dieses Teilabschnitts demonstriert das folgende Beispiel die Programmierung eines kollektiven Schreibzugriffs in MPI-IO. Dazu muss als Erstes der Datentyp `type` erzeugt werden, der als Grundlage der Sicht auf die Daten dient. Abhängig von der Anzahl beteiligter Prozesse erfolgt die Partitionierung des gesamten Blocks und wird die Sicht auf die Daten so definiert, dass jeder Prozess einen Teil der Daten exklusiv bearbeitet (Zeile 14-21). Wurde die Datenstruktur dem System bekannt gemacht (Zeile 22), erfolgt das Öffnen der Datei (Zeile 26) und Setzen der Sicht (Zeile 29). Anschließend erfolgt ein kollektiver Schreibzugriff unter Angabe eines expliziten Offsets (Zeile 32). Das Programm terminiert, nachdem die Datei geschlossen und MPI-IO beendet wurde.

```
1  #include "mpi.h"
2  main(int argc, char **argv)
3  {
4    MPI_Datatype ftype, btype;
5    int length[3];
6    MPI_Aint disp[3];
7    MPI_Datatype type[3];
8    MPI_File fh;
9
10   /* initialize MPI */
11   MPI_Init(&argc, &argv);
12
13   /* create/commit filetype */
14   length[0] = 1; length[1] = blocklen; length[2] = 1;
15   disp[0] = 0; disp[1] = blocklen * myrank; disp[2] = blocklen * commsize;
16   type[0] = MPI_LB; type[1] = MPI_CHAR; type[2] = MPI_UB;
17   MPI_Type_struct(3, length, disp, type, & filetype);
18   MPI_Type_commit(&filetype);
19
20   /* create/commit buftype */
21   MPI_Type_contiguous(buf_size, MPI_CHAR, &buftype);
22   MPI_Type_commit(&buftype);
23
24   /* open file */
25   mode = MPI_MODE_CREATE | MPI_MODE_WRONLY;
26   MPI_File_open(MPI_COMM_WORLD, filename, mode, MPI_INFO_NULL, &fh);
27
28   /* set file view */
29   MPI_File_set_view(fh, offset, MPI_CHAR, filetype, "native", MPI_INFO_NULL);
30
31   /* write buffer to file */
32   MPI_File_write_at_all(fh, offset, (void*) buf, 1, buftype, & status);
33
34   /* close file */
35   MPI_File_close(&fh);
36
37   /* finalize MPI */
38   MPI_Finalize();
39 }
```

Systeminformationen

Zur Optimierung von Zugriffen bietet MPI-IO einen Mechanismus, um dem System zusätzliche Informationen - die sogenannten *file hints* - bekannt zu machen. Die Basis stellt dabei die Datenstruktur `MPI_Info` dar, die eine Liste von *Schlüsseln* und dazugehörigen *Werten* enthält. Dabei repräsentiert jeder Schlüssel einen Konfigurationsparameter des Ein-/Ausgabesystems oder der Nachrichtenbibliothek und der Wert die gewünschte Ausprägung des Parameters.

MPI-IO definiert viele Standardschlüssel, wie z.B. die Stripe-Größe einer Datei, die Anzahl und Konfiguration der im System vorhandenen Ein-/Ausgabeknoten, die Konfiguration der kollektiven Ein-/Ausgabe oder die Angabe erwarteter Zugriffsmuster. Anwendungen können diese Informationen während des Öffnens einer Datei oder beim Setzen einer Sicht auf die Daten definieren. Wenn keine zusätzlichen Informationen vorhanden sind oder nicht angegeben werden sollen, kann die Dummyvariable `MPI_INFO_NULL` verwendet werden.

Datenportierung

Eine standardisierte Schnittstelle ermöglicht die Lauffähigkeit der Anwendungen auf einer Vielzahl von Architekturen. Dies garantiert jedoch noch nicht die Portierbarkeit der geschriebenen Daten. Die unterschiedliche Reihenfolge der Bytes (engl. little und big endian) oder die Darstellung von Fließkommazahlen führen dazu, dass auf einem System geschriebene Daten auf einem anderen nutzlos sind, da sie nicht korrekt eingelesen werden können. MPI-IO löst dieses Problem auf zwei Arten: Zum Ersten definiert MPI-IO eine standardisierte Darstellung der Daten, die von jeder MPI-IO-Bibliothek gelesen werden kann. Zum Zweiten stellt MPI-IO dem Anwender einen Mechanismus zur Verfügung, der es erlaubt, benutzerdefinierte Methoden zur Konvertierung der Daten in einen Datenzugriff zu integrieren. Jedoch speichert MPI-IO keine zusätzlichen Informationen über die Struktur oder Darstellung der Daten innerhalb der Datei. Daher ist es Aufgabe des Benutzers, eine Art „Typsicherheit der Daten“ zu garantieren.

Sowohl die standardisierte als auch die benutzerdefinierte Darstellung der Daten wird durch den Parameter `datarep` während des Setzens einer Sicht auf die Daten realisiert. MPI-IO bietet dabei drei standardisierte Darstellungen an. Die *native*-Darstellung verwendet das Format des lokalen Systems einer Anwendung. In einer heterogenen Umgebung stellt dies ein Problem dar, sobald mehrere Rechner unterschiedlicher Architekturen gemeinsam auf dieselbe Datei zugreifen. Unter Umständen entsteht dann eine von keinem der Rechner lesbare Datei. Die Daten werden bei dieser Darstellung keinerlei Umsetzung unterzogen und können dadurch mit der bestmöglichen Leistung gelesen werden. Die *internal*-Darstellung zwingt die Prozesse dazu, die Daten in einer über die Anwendung hinweg konsistenten Form zu speichern, die es erlaubt, die Daten ohne Probleme wieder einzulesen. MPI-IO spezifiziert dabei keine genaue Darstellung der Daten, sondern überlässt es der Implementierung, bis zu welchem Grad die Daten portabel sind. Schließlich definiert die *external32*-Darstellung ein spezielles Format für jeden MPI-Datentyp, so dass Daten, die mit dieser Darstellung geschrieben werden, von jeder anderen MPI-Implementierung gelesen werden können[5]. Das verwendete Format basiert dabei auf bekannten Standards (IEEE, Unicode) und ist daher in einigen Systemen mit der *native*-Darstellung identisch, während andere Systeme eine Umsetzung der Daten erfordern und damit eine geringere Leistung erreichen.

2.6 Beispielarchitekturen

Kepler-Cluster

Im November 2000 enstand durch Kooperation des Instituts für Astronomie und Astrophysik, des Arbeitsbereichs numerische Mathematik, der technischen Informatik und des Zentrums für Datenverarbeitung an der Universität Tübingen der aus Standardkomponenten aufgebaute Kepler-Cluster [93].

Das System besteht aus zwei Teilen mit insgesamt 128 Doppelprozessorknoten. 96 Knoten sind plattenlos und verfügen über Pentium-III-Doppelprozessoren mit einer Takrate von 650 MHz und 1 GB gemeinsamem Hauptspeicher. Die verbleibenden 32 Knoten bestehen aus je zwei AMD Athlons. Diese sind mit einer Taktrate von $1,667$ GHz und 2 GB gemeinsamem Speicher ausgestattet. Zur Visualisierung verfügen sie zusätzlich über leistungsstarke Grafikkarten und sie besitzen eine Festplatte mit einer Kapazität von jeweils 80 GB.

Der Datentransfer erfolgt über zwei voneinander getrennte Verbindungsnetzwerke. Das 100 Mb Ethernet wird für administrative Zwecke und als SAN verwendet. Zur Kommunikation zwischen den Knoten existiert zusätzlich ein Hochgeschwindigkeitsnetz von Myrinet. Die nominale bisektionale Bandbreite von Myrinet entspricht mit $1,28$ Gb/s der maximalen Leistung des verwendeten PCI-Bus. Messungen ergeben eine Bandbreite von 115MB/s und eine Latenz von 7μs. Die nominale Bandbreite des Ethernet liegt bei 100Mb/s und es erreicht aufgrund des Zusatzaufwands von TCP/IP eine effektive Transferrate von 11MB/s. Beide Netzwerke bilden eine mehrstufige, hierarchisch geswitchte Fat-Tree-Topologie.

Von der theoretisch maximalen Leistung von 127 GFlop/s erreichen die Pentiumknoten in einem LINPACK-Benchmark bei einer Effizienz von 75% eine Leistung von $96,2$ GFlop/s.

[5] Die Zahl 32 bezieht sich dabei auf die Anzahl von Bits, die zur Darstellung einer einfach genauen Fließkommazahl verwendet werden. Das MPI-Forum zog zwar eine weitere Darstellnug mit 64 Bits in Erwägung, hat sich dann aber doch dagegen entschieden.

Die Ansteuerung des gesamten Systems erfolgt über zwei Frontendrechner, die ebenfalls Doppelprozessormaschinen mit Pentium-III-Prozessoren und einer Taktrate von 733 MHz und 2GB Speicher sind. Außer über Grafikkarten zur Visualisierung verfügen diese auch über insgesamt 500 GB Festplattenspeicher, die als RAID-5-System konfiguriert sind.

Die parallele Ein-/Ausgabearchitektur setzt auf dem Ethernet auf und besteht aus der MPI-IO-Implementierung ROMIO [91], die unter Verwendung einer abstrakten Geräteschnittstelle (ADIO [90]) direkt auf das verwendete Dateisystem aufsetzt. Das eingesetzte parallele Dateisytem PVFS [16] wurde auf den 32 Athlon-Knoten installert, die damit als Ein-/Ausgabeknoten fungieren. Gleichzeitig stellen alle vorhandenen 128 Knoten Ein-/Ausgabeklienten dar, die über das SAN auf die Daten zugreifen können.

Auf der Softwareseite kommen das Betriebssystem Linux und die Cluster-Software Score-D [67] zum Einsatz. Letztere dient der Allokation, dem Starten, Beenden und Überwachen von Knoten und dem Checkpointing des gesamten Systems. Die nutzbaren Compiler sind neben Fortran und High Performance Fortran von GNU der C- und C++-Compiler. Darüber hinaus stehen der Debugger TotalView [22] und der Profiler Vampir-Trace [47][6] zur Verfügung. TotalView ist dabei ein speziell für Parallelrechner entwickelter Debugger, der die Daten aller Knoten auf einen Rechner zusammenführt. Bei Vampir-Trace handelt es sich um einen speziellen Profiler für parallele Rechner, der es ermöglicht, einen Überblick über den Ablauf auf allen Knoten, inklusive der Kommunikation, zu vermitteln.

Cray-Opteron-Cluster

Der seit März 2004 am Höchstleistungsrechenzentrum in Stuttgart (HLRS) installierte Cray-Opteron-Cluster [35] ist dem Kepler-Cluster sehr ähnlich. Er besteht aus einem Frontendrechner für den interaktiven Zugriff der insgesamt 125 Doppelprozessorknoten und zwei davon getrennten Ein-/Ausgabeknoten. Die Rechenknoten verfügen über jeweils zwei AMD Opterons mit 2 GHz Taktfrequenz und 4 GB Arbeitsspeicher. Die Spitzenleistung des gesamten Systems liegt bei 1024 GFlops. Als Verbindungsnetzwerk zwischen den Knoten wird Myrinet 2000 eingesetzt, das eine Transferrate von 250 MB/s erzielt.

Die Ein-/Ausgabedaten werden über ein separates Netzwerk (1 Gb Ethernet) zwischen den Ein-/Ausgabeknoten und den Rechenknoten kommuniziert. Jeder Ein-/Ausgabeknoten stellt ein eigenes Dateisystem zur Verfügung und kann daher nur einzeln angesprochen werden. Als Backendsystem ist ein RAID 5 mit insgesamt 2 TB Speicher installiert, das über Fibre Channel von den Ein-/Ausgabeknoten angesprochen werden kann. Da die Bandbreite des Fibre Channel weit über der Transfer-

[6] Intel kaufte die Software auf und vertreibt sie unter dem Namen *Trace Analyzer and Collector*.

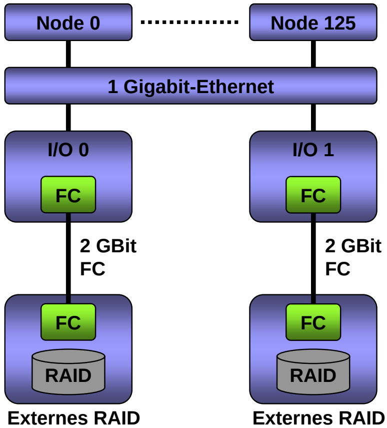


Abbildung 2.3: Schematische Darstellung der Ein-/Ausgabearchitektur des Cray-Opteron-Clusters

rate des Ethernet liegt, ist die Netzwerkkarte der Flaschenhals des Systems. Einen Überblick über die gesamte Ein-/Ausgabearchitektur gibt Abbildung 2.3.

Als Betriebssystem wurde Linux SuSE SLES 8 als 64-Bit-Version installiert. Zur Verwaltung der abzuarbeitenden Jobs steht das Stapelverarbeitungssystem PBSPro zur Verfügung. Zur Entwicklung von 64-Bit- und 32-Bit-Anwendungen stehen der Portland Group Compiler (PGI) von STMicroelectronics in Fortran, C und C++ und die GNU Compiler Collection (GCC) zur Verfügung. Darüber hinaus sind zahlreiche Werkzeuge zum Überwachen, Profiling und Debugging der Anwendungen vorhanden (Cray Utils). Einem System mit verteiltem Speicher entsprechend wird als Programmiermodell der Nachrichtenaustausch in Form von MPI eingesetzt.

Hitachi SR 8000-F1

Mit der SR 8000 Serie von Hitachi entstand ein System, dessen Hauptziele bei der Entwicklung Flexibilität und hohe Skalierbarkeit waren. Am Leibniz-Rechenzentrum (LRZ) in München ist dieses Modell seit März 2000 als Höchstleistungsrechner in Bayern (HLRB) in Betrieb [34]. Eine detaillierte Aufstellung der Ausstattung findet sich in Tabelle 2.5.

Die Architektur der SR 8000-F1 ermöglicht die Nutzung des Vektor- und des ska-

Knoten	168
Prozessoren pro Knoten	8 (9 physisch)
Anzahl Prozessoren	168*8 = 1344
Gesamtrechenleistung LINPACK-Benchmark	1,5*8*168 = 2016 GFlop/s 1645 GFlop/s
Gesamthauptspeicher	1344 GByte
Taktfrequenz	375 MHz
Gesamtplattenspeicher	10 TByte
Bandbreite zum /home/	600 MByte/s
Bandbreite zum /temp/	2.4 GByte/s
MPI-Bandbreite (2 Knoten)	950 MByte/s (max: 1 GByte/s)
Gesamtbandbreite	114 GByte/s (max: 150 GByte/s)

Tabelle 2.5: Hardware-Ausstattung der SR 8000-F1

laren Cluster-Programmiermodells im gleichen Rechner. Erreicht wird dies dadurch, dass in einem Knoten 8 von dessen 9 superskalaren 64-Bit-RISC-Prozessoren mit je 1,5 GFlop/s Spitzenrechenleistung durch Hardware und Software zu einer virtuellen Vektor-CPU mit effektiv 12 GFlop/s zusammengefasst werden können. Bei einer traditionellen Vektor-CPU werden vektorisierbare Rechenoperationen über ein Fließband abgewickelt, mit dem pro Takt ein oder mehrere Speicherinhalte an die CPU geliefert werden. Bei der Hitachi SR 8000-F1 wird die vektorisierbare Rechenoperation dagegen auf die Rechenwerke eines Knotens aufgeteilt (COMPAS[7]) oder einzelne Speicherinhalte lange vor ihrem Gebrauch in die Register geladen (PVP[8]). Dies hat den zusätzlichen Vorteil, dass sämtliche Rechenwerke achtfach nutzbar sind und nicht nur die in traditionellen Vektor-CPUs mehrfach vorhandenen Fließkommaeinheiten. Diese zwei Eigenschaften tragen besonders zur relativ hohen erzielbaren Effizienz der Knoten der SR 8000-F1 bei. Als Kommunikationsmedium kommt dabei

[7] Co-operative Micro Processors in single Address Space: Die automatische Verteilung der Rechenlast in Schleifen auf die acht Prozessoren eines Knotens durch den Compiler (Autoparallelisierung) und die begleitende Hardware-Unterstützung für die Synchronisation (z.B. Cache-Synchronisation aller acht Prozessoren am Anfang und am Ende eines parallelen Codestücks).

[8] Pseudo-Vector-Processing: Erweiterungen von Hitachi am POWER-Befehlssatz von IBM zur Verbesserung der Hauptspeicherzugriffe verringern dieses derzeit größte Defizit RISC-basierter Höchstleistungsrechner. Diese von Hitachi Pseudo Vector Processing (PVP) genannte Eigenschaft wird vom Compiler dazu genutzt, je nach Struktur der Hauptspeicherzugriffe die Daten direkt aus dem Hauptspeicher oder über den Cache zur Verfügung zu stellen.

ein multidimensionaler Kreuzschienenverteiler (engl. *crossbar switch)* zum Einsatz.

Die Architektur der Hitachi ermöglicht eine hybride Parallelisierung, d.h. dass innerhalb eines Knotens auf der Basis von gemeinsamem Hauptspeicher und zwischen den Knoten aufgrund des verteilten Speichers eine Programmierung über Nachrichtenaustausch vorgenommen werden muss.

Die Ein-/Ausgabe der Hitachi verteilt sich auf mehrere parallele Dateisysteme. Insgesamt verfügen 112 Knoten der Hitachi über eine eigene Festplatte, die eine Transferrate von ca. 20 MB/s besitzt. Durch Einteilung der einzelnen Knoten in Untersysteme entstehen mehrere nutzbare Dateisysteme, von denen das kleinste aus 4 und das größte aus 96 Ein-/Ausgabeknoten besteht. Sowohl Ein-/Ausgabe als auch Kommunikation erfolgt über dasselbe Verbindungsnetzwerk.

NEC SX-6

Mit der NEC SX-6 der Universität Stuttgart [36, 50] wurde eine kommerzielle Vektorrechnerarchitektur im Rahmen dieser Arbeit verwendet. Der Zugriff auf das System erfolgt über einen Frontendrechner, bestehend aus einem 16-fach SMP mit Itanium-2 Prozessoren und 240 GB Arbeitsspeicher.

Die Architektur eines einzelnen Rechenknotens besteht aus 8 Vektoreinheiten und einer skalaren Steuereinheit. Zur Minimierung der Latenzen und Maximierung der Bandbreite verbindet ein Kreuzschienenverteiler die 64 GB gemeinsamen Hauptspeicher mit den Prozessoren. Bei einer Taktrate von 565 MHz erreicht ein Knoten eine maximale Leistung von 8 GFlop/s.

Um ein leistungsfähiges Gesamtsystem zu erhalten, können bis zu 128 Knoten über das von NEC entwickelte Hochgeschwindigkeitsnetz IXS verbunden werden. Es besteht aus einem Kreuzschienenverteiler (XSW) und einer Kreuzschienensteuereinheit (XCT). Diese Technologie ermöglicht sehr geringe Latenzen und bisektionale Bandbreiten zwischen den einzelnen Knoten von bis zu 8 GB/s.

Zur Ein-/Ausgabe von Daten verfügt ein Rechenknoten über Schnittstellen zu HIPPI, FC-AL, SCSI und Gigabit Ethernet. Die Installation am HLRS verwendet ein Festplattenarray mit einer Kapazität von 1 TB, das über Fibre Channel an jeden der insgesamt 6 Rechenknoten angeschlossen ist. Insgesamt stehen 4 parallele Dateisysteme zur Verfügung, die von den Knoten einzeln angesteuert werden können.

Insgesamt verfügt das System am HLRS über eine theoretische Leistung von maximal $6 * 8 * 8 = 384$ GFlop/s. Unabhängige Messungen [50] eines einzelnen Knotens unter Verwendung der LINPACK TPP Benchmark belegen eine Effizienz von ca. 65% der Spitzenleistung. Auf das System in Stuttgart übertragen entspricht dies einer Leistung von ca. 240 GFlop/s.

3 Stand der Technik

No one does I/O, because it's too slow.
Saphir [83]

Die im letzten Kapitel vorgestellten Methoden zur Parallelisierung der Ein-/Ausgabe durch Techniken, die sowohl auf Hardware als auch auf Software basieren, haben zum Ziel, möglichst die aggregierte Ein-/Ausgabeleistung eines Systems zu erreichen. Die Möglichkeiten orientieren sich dabei sehr systemnah und formulieren die Ein-/Ausgabe mit einem meist einfachen Datenmodell, das die Daten und Strukturen der durch ein Ein-/Ausgabesystem manipulierbaren Daten definiert. Das einfachste Modell verwendet Unix, das die Daten als einen linearen Strom von Bytes behandelt. Das Modell von MPI-IO ist hingegen deutlich besser entwickelt, da es benutzerdefinierte Strukturen und unzusammenhängende Zugriffe auf die Daten erlaubt. Jedoch werden von MPI-IO dabei keine weiteren Informationen über die Daten gespeichert, so dass es Aufgabe der Anwendung ist, für Typsicherheit zu sorgen.

Die Praxis der parallelen Anwendungsentwicklung zeigt jedoch, dass hochentwickelte Software die Verwendung stark komplexer Datenstrukturen und komplizierter Verteilungsmuster erfordert. Standardbibliotheken, wie z.B. MPI-IO, sind diesen Anforderungen nicht gewachsen und so für den Entwickler nur sehr komplex und damit zeitaufwändig einsetzbar. Aus diesem Grund wurden in den letzten Jahren parallele Ein-/Ausgabebibliotheken entwickelt, die der Ein-/Ausgabe auf Systemebene mit einer Standardbibliothek und dem Anwender gegenüber auf einem höheren Abstraktionsniveau begegnen. Die implementierten Datenmodelle ermöglichen dem Entwickler die Verwendung komplexer Datenstrukturen auf möglichst einfache Art und Weise. Dies gelingt durch automatische und möglichst effiziente Speicherung der Datenstrukturen in Form von Metainformationen.

Die gestiegenen Anforderungen an die Komplexität der Datenstrukturen sind vorwiegend auf die Verbreitung objektorientierter Methoden im Bereich des parallelen Rechnens zurückzuführen. Immer mehr Anwendungen werden in objektorientierter Form entwickelt und implementiert, da die Parallelisierung eines Problems durch das deutlich höhere Abstraktionsniveau meist deutlich einfacher durchgeführt werden kann. Darüber hinaus fördert die Wiederverwendbarkeit objektorientierter Module die Entwicklung von Bibliotheken und Frameworks für bestimmte Teildisziplinen, wie z.B. numerische oder physikalische Methoden, die bei Variation des betrachteten Problems lediglich an die veränderten Rahmenbedingungen angepasst werden müssen.

Auch auf dem Gebiet der parallelen Ein-/Ausgabe versucht man daher, die Vorteile objektorientierter Software durch die Entwicklung neuer Schnittstellen zu nutzen. Dieses Kapitel geht daher auf den Stand der Technik in diesem Bereich ein. Die Entwicklung der in dieser Arbeit vorgestellten Schnittstelle folgt der Vorgehensweise einer objektorientierten Softwareentwicklung. Abschnitt 3.1 stellt daher kurz die wesentlichen Techniken von der Analyse des Problems bis hin zur Implementierung vor. Die aktuellen Entwicklungen im Bereich der objektorientierten parallelen Ein-/Ausgabe werden in Abschnitt 3.2 beschrieben. Die objektorientierte Kommunikation stellt die Grundlage dieser Art der Ein-/Ausgabe dar. Abschnitt 3.3 beschreibt daher die Umsetzung einer solchen Kommunikation, die im Rahmen der Arbeiten von Ritt [77] realisiert wurde.

3.1 Objektorientierte Softwareentwicklung

3.1.1 Objektorientierte Analyse und Design

Die objektorientierte Analyse und Design von Softwaresystemen ist ein Prozess, der keinen strengen Vorgaben unterliegt. Es handelt sich vielmehr um einen iterativen und inkrementellen Vorgang. In diesem Abschnitt sollen die einzelnen Phasen durch Ziele, Produkte, Arbeitsgänge/Aktivitäten und Bewertungen beschrieben werden. Nach und nach werden dabei die verschiedenen Sichten auf das Gesamtsystem erschlossen.

1988 begann die Geschichte der objektorientierten Methoden. Grundlage für diese waren die objektorientierten Sprachen (Smalltalk, später C++). Die Absicht dabei war, die Vorteile der strukturierten auf die objektorientierten Methoden zu übertragen. Die wichtigsten Methoden sind von Shlaer und Mellor, Coad und Yourdon, OOAD (G. Booch [14]), OMT (J. Rumbaugh [80]) und OOSE (I. Jacobson [49]).

Booch und Rumbaugh entwickelten zusammen bei Rational eine einheitliche Methode, die sie 1995 als sogenannte *Unified Method 0.8* (UM) veröffentlichten. Kurze Zeit später stieß Ivar Jacobson hinzu, was eine Integration der von ihm geprägten Anwendungsfälle (engl. use cases) zur Folge hatte. Ihre Zusammenarbeit führte dann 1997 zur Version 1.1 der *Unified Modeling Language (UML)*, die von der Object Management Group (OMG) akzeptiert und als Standard verabschiedet wurde. Im Dezember 1999 folgte die bis heute aktuelle Version 2.0.

In der UML [15, 63] kommen als Darstellungsmittel der Klassen- und Objektmodellierung Diagramme und textliche Beschreibungen zum Einsatz. Durch die Zusammenarbeit von Booch und Rumbaugh wurden die aus ihren komplexen Methoden bekannten Notationen vereinheitlicht und weiterentwickelt, woraus sehr kompakte Darstellungsmittel entstanden, die sowohl in der Analyse als auch im Design die oben genannten Informationen visualisieren können. Im Kern besteht die Methode darin, von einer relativ hohen Abstraktionsebene durch wiederholte Ausführung der einzelnen Schritte zu einer niedrigeren Abstraktionsebene zu gelangen, welche dann eine möglichst einfache Implementierung zulässt.

Während der Modellierung der vorliegenden Schnittstelle wurden ausschließlich Klassendiagramme erstellt und deren Beziehungen untereinander festgelegt. Diese können verschiedenster Art sein, z.B. Vererbungsbeziehungen (engl. kind-of) oder Aggregationen (engl. part-of). Bei der Aggregation unterscheidet man, ob eine Klasse als Wert oder als Referenz in der anderen enthalten ist. Eine Referenz auf eine Klasse ist dann nötig, wenn mehrere Objekte auf eine Instanz zugreifen müssen. Zudem gibt es noch die Verwendungsbeziehung (engl. link), die besteht, wenn eine Klasse in einer Methode der anderen als Über- oder Rückgabeparameter eingesetzt wird.

Neben Klassendiagrammen existieren noch weitere Diagrammarten, z.B. Sequenzdiagramme oder Aktivitätsdiagramme. Diese Darstellungsmethoden kamen jedoch bei der Entwicklung nicht zum Einsatz, so dass an dieser Stelle auf eine genaue Beschreibung verzichtet und auf [15] verwiesen wird.

3.1.2 Die objektorientierte Programmiersprache C++

Die Programmiersprache C++ eignet sich für viele Arten von Anwendungen und stellt Sprachmittel für abstrakte Datentypen und die modulare, generische, objektorientierte und strukturierte Programmierung zur Verfügung. C++ basiert auf der Programmiersprache C, bietet aber zusätzlich zu den in C vorhandenen Möglichkeiten weitere Datentypen, Klassen mit Vererbung und virtuellen Funktionen, Ausnahmebehandlungen, Schablonen (engl. templates), Namensräume, Inline-Funktionen, Überladen von Operatoren und Funktionsnamen, Referenzen, Operatoren zur Freispeicherverwaltung und mit der C++-Standardbibliothek eine erweiterte Bibliothek. Neben der Kompatibilität zu C führten weitere Vorteile gegenüber anderen hohen Programmiersprachen zu einer hohen Akzeptanz von C++. Dies sind z.B. die Möglichkeit einer sehr maschinennahen und gleichzeitig stark abstrakten Programmierung, die zu effizienten Anwendungen führt, die sehr hohe Ausdrucksstärke und Flexibilität der Sprache und Möglichkeiten der Metaprogrammierung. In den folgenden Teilabschnitten werden nun die wichtigsten Merkmale von C++ beschrieben.

Generische Programmierung

Das mächtigste Programmierwerkzeug von C++ ist die generische Programmierung. Während die objektorientierte Programmierung in Java und C# ein höchstmögliches Abstraktionsniveau anstrebt, ist diese Art der Programmierung in C++ rückläufig. Zu Gunsten der Effizienz und der Minimierung des Ressourcenverbrauchs verzichtet man in vielen Fällen auf Polymorphie, den Kernmechanismus der objektorientierten Programmierung. Im Bezug auf die Wiederverwertbarkeit einmal kodierter Algorithmen übertreffen generische Techniken nach Meinung vieler Fachleute die objektorientierte Programmierung [2, 60].

Wesentlich bei der generischen Programmierung ist, dass die Algorithmen, Funktionen und Klassen möglichst allgemein geschrieben werden, so dass sie für unterschiedliche Datentypen verwendet werden können. Die generische Programmierung wird in C++ über Templates realisiert und erreicht eine hohe Flexibilität und Typsicherheit der Funktion. Am Beispiel der generischen Implementierung der Funktion `max` soll dies nun demonstriert werden:

```
template<class T>
T max(T a, T b)
{
  if (a < b)
    return b;
  else
    return a;
}
```

Unter der Voraussetzung, dass für den verwendeten Typ der Vergleichsoperator implementiert wurde, kann die so definierte Funktion `max()` nun für alle Typen verwendet werden. Zumindest die Basistypen von C++ erfüllen diese Voraussetzung und können direkt mit dieser Funktion verwendet werden. Ein weiterer Vorteil von Templates ist damit, dass ein Datentyp lediglich bestimmte Eigenschaften erfüllen muss, um in einem Template verwendet werden zu können. Datentypen, die diese Eigenschaften besitzen, können auf diese Weise auch mit einer Funktion verwendet werden, die erst zu einem späteren Zeitpunkt entwickelt wird.

Die Standard-Bibliothek von C++

Die C++-Standardbibliothek stellt verschiedene generische Container, Funktionen zu deren Manipulierung, Funktionsobjekte, generische Zeichenketten (engl. strings), Datenströme u.a. für den Dateizugriff, die Unterstützung von Sprachmitteln sowie einfache Funktionen, z.B. zur Bildung der Quadratwurzel, zur Verfügung.

Die meisten Komponenten der C++-Standardbibliothek liegen in Form von Templates vor. Das Template-Konzept hat den großen Vorteil der Wiederverwendbarkeit. Es können z.B. durch einfache Deklaration Container für beliebige Datentypen oder Algorithmen, die für eine ganze Reihe von Datentypen gelten, erzeugt werden.

Container (Behälterklassen) sind Datenstrukturen, die aus einer Anzahl verschiedener Objekte desselben Datentyps bestehen. Die grundlegenden Container der C++-Standardbibliothek sind die sequentiellen Container Vektor (vector), doppelt verkettete Liste (list), Warteschlange (queue), Warteschlange mit zwei Enden (deque), Warteschlange mit Priorität (priority_queue) und Stapel (stack). Darüber hinaus existieren die assoziativen Container Menge (set), assoziatives Feld (map) und Bitmenge (bitset).

Während in sequentiellen Containern die Objekte linear angeordnet sind und über ihre Position angesprochen werden, ordnen die assoziativen Container die Objekte in einer Baumstruktur. Der Zugriff erfolgt dann mit Hilfe von Schlüsseln. Jeder Container verfügt über verallgemeinerte Zeiger, die sogenannten Iteratoren, mit denen über die Elemente eines Containers iteriert und auf einzelne Elemente des Containers zugegriffen werden kann. Damit stellen die Iteratoren reine Zugriffsobjekte dar und entkoppeln die Algorithmen von den Daten, so dass diese typenunabhängig werden. Man unterscheidet bei den Iteratoren Eingabe-Iteratoren (lesender Zugriff für einen einzelnen Durchlauf), Ausgabe-Iteratoren (schreibender Zugriff für einen einzelnen Durchlauf), Forward-Iteratoren (sequentieller Zugriff mit relativem Bezug auf Iteratoren in eine Richtung), bidirektionale Iteratoren (wie Forward-Iteratoren jedoch in beide Richtungen) und Iteratoren mit wahlfreiem Zugriff (auch mit Operator []).

3.2 Objektorientierte parallele Ein-/Ausgabe

Durch den verstärkten Einsatz objektorientierter Anwendungen im wissenschaftlichen Rechnen wurde es notwendig, auch die parallele Ein-/Ausgabe objektorientiert zu formulieren. Die wissenschaftlichen Bibliotheken und Frameworks versuchen, dazu die anfallenden Daten auf einem höheren Abstraktionsniveau zu verarbeiten. Anwendungen können direkt die komplexen Daten verändern, ohne die Verteilung der Daten oder deren Struktur innerhalb einer Datei zu kennen. Dies wird z.B. durch die zusätzliche Speicherung von Typinformationen und anderen nützlichen Metadaten realisiert. Die Grundlage dazu bildet das innerhalb der Bibliothek verwendete Datenmodell zur Speicherung der Informationen. In Tabelle 3.1 werden die bekanntesten Ein-/Ausgabesysteme und ihre Datenmodelle dargestellt. Die weite Verbreitung von MPI als nachrichtenbasierte Kommunikationsbibliothek auf Höchstleistungsrechnern hat dazu beigetragen, dass die meisten wissenschaftlichen Bibliotheken auf diesem Standard aufsetzen.

Die folgenden Unterabschnitte beschreiben nun die am häufigsten eingesetzten Schnittstellen zur objektorientierten parallelen Ein-/Ausgabe, deren zugrunde liegende Kommunikation stets über MPI realisiert ist. Neben den C++-Bindungen von MPI selbst [59] sind dies die Ansätze netCDF, HDF und OpenMPI, die versuchen, eine parallele Ein-/Ausgabe objektorientiert zu formulieren. Mit Hilfe eines einheitlichen Beispiels sollen dabei die wesentlichen Merkmale der Ansätze verdeutlicht werden. Es beschreibt die blockierende Speicherung eines Objekts mit zwei Integerwerten als Attribute durch einen einzelnen Prozess.

3.2.1 C++-Bindungen von MPI-IO

Um den Anforderungen an eine objektorientierte Schnittstelle gerecht zu werden, definiert der Standard MPI-2 Bindungen an C++. Unter Verwendung von C++-Hüllen

Ein-/Ausgabesystem	Datenmodell
Unix	Bytesequenz
MPI-IO (MPI-IO/C++, OpenMPI)	Liste typisierter Datenelemente
netCDF	Beschreibung multidimensionaler Arrays mit typisierten Elementen
HDF5	Beschreibung multidimensionaler Arrays, die Strukturen mit mehreren Elementen besitzen; Hierarchisch geordnete Gruppen von Objekten

Tabelle 3.1: Ein-/Ausgabesysteme und die von ihnen verwendeten Datenmodelle

(engl. wrapper) für die MPI-Typen können sämtliche Funktionen von MPI in dem gleichnamigen Namensraum verwendet werden. Die dadurch entstandene Klassenstruktur bildet damit die Funktionen der einzelnen MPI-Typen auf Methoden der jeweiligen Klasse ab. Die Implementierung der C++-Bindungen ist daher der C-Implementierung von MPI sehr ähnlich, wie das folgende Beispiel zeigt:

```
#include "mpi.h"
using namespace mpi;

void main(int argc, char **argv)
{
  struct { int x; int y; } Point;
  int blocklength[2]={1,1};

  Aint displacement[2];
  displacement[0] = 0;
  displacement[1] = Get_address(&Point.x) - Get_address(&Point.y);

  /* initialize MPI */
  Init(&argc, &argv);

  /* create/commit filetype */
  Datatype types[2]={INT, INT};
  Datatype typ = Datatype::Create_struct(2, blocklengths, displacement, types);
  typ.Commit();

  /* open file */
  File fh = File::open(MPI_COMM_WORLD, filename, MPI_MODE_CREATE |
                       MPI_MODE_WRONLY, MPI_INFO_NULL);

  /* write buffer to file */
  fh.Write(Point, 1, typ, &status);
```

```

 /* close file */
 fh.Close();

 /* finalize MPI */
 Finalize ();
}
```

3.2.2 netCDF

Die Network Common Data Form - kurz *netCDF* [75, 76] - wurde von Unidata Ende 1980 entwickelt. Ein Ziel lag dabei in der Definition eines Dateiformats, das den architekturunabhängigen Austausch von Daten zwischen Wissenschaftlern ermöglichen sollte. Die hohe Akzeptanz hat dazu geführt, dass mittlerweile Schnittstellen zu C, C++, Fortran und Perl verfügbar sind. Aktuell wird an einer Schnittstelle zu Java gearbeitet.

Das in netCDF realisierte Datenmodell besteht aus einer multidimensionalen Arraystruktur, die Objekte des gleichen Basisdatentyps beinhaltet (z.B. 32-Bit-Integerwerte). Ein skalarer Wert wird folglich durch ein 0-dimensionales Array beschrieben. Die Werte des Arrays können direkt bearbeitet werden, ohne dass der Benutzer die physikalische Art und Weise der Speicherung der Daten kennen muss. Dies liegt daran, dass die zur Wiederherstellung der Struktur notwendigen Informationen zusätzlich im Dateikopf mit abgespeichert werden. Die Dateien bestehen damit aus sich selbst beschreibenden Objekten und können unabhängig von Architektur und Anwendung weiterverarbeitet werden.

Die Erzeugung eines netCDF-Datenobjekts erfolgt in zwei Phasen: Als Erstes wird eine netCDF-Datei geöffnet, und die zu speichernden Dimensionen und Variablen werden definiert (engl. define mode). Anschließend erfolgt die Speicherung der Daten in den Variablen (engl. data mode). Die Funktionen von netCDF können die so definierten Objekte entweder über deren Namen oder ihre Identifikationsnummer ansprechen.

Das ursprüngliche Design von netCDF besitzt jedoch keine Zugriffsmechanismen zur parallelen Ein-/Ausgabe. Dieser Mangel führt dazu, dass parallele Anwendungen die Ein-/Ausgabe seriell durchführen müssen. Dabei werden die Daten an einen einzelnen Prozess kommuniziert, welcher die Daten anschließend auf einem Festspeicher ablegt. Um diese sehr ineffiziente und für den Anwender umständliche Vorgehensweise zu unterbinden, wurde *pnetCDF* [54] entwickelt. Es handelt sich dabei um eine Schnittstelle zur parallelen Ein-/Ausgabe, die das Datenformat von netCDF unterstützt und auf dem Standard MPI-IO aufsetzt. Dadurch wird es möglich, von den MPI-IO-internen Optimierungen der parallelen Ein-/Ausgabe vollständig zu profitieren.

Die Implementierung der Schnittstelle lässt sich in zwei Teile gliedern: Erstens die Ein-/Ausgabe der Kopfdaten einer Datei und zweitens die Formulierung der parallelen Ein-/Ausgabe der Daten. Zur Vermeidung von Inkonsistenzen werden die Kopfdaten nur von einem einzelnen Prozessor gelesen und geschrieben. Dieser verteilt die Informationen an die verbleibenden Prozessoren. Der parallele Zugriff auf die Daten erfolgt unter Konstruktion einer Sicht (engl. view) auf die Daten. Die Sicht wird von jedem Prozessor unter Verwendung der Metainformationen aus dem Dateikopf selbst generiert und ermöglicht dadurch eine kollektive Ein-/Ausgabe.

Die Vorgehensweise zur Implementierung der parallelen Ein-/Ausgabe gliedert sich in insgesamt 6 Teilschritte. Im folgenden Beispiel sind diese Schritte an der entsprechenden Stelle im Quelltext kommentiert.

```
#include <netcdf.h>

void main()
{
  int point[2];
  int pdimid, pvarid;
  int mpi_size, mpi_rank;
  int ncfile;

  /* initialize MPI */
  MPI_Init(&argc, &argv);

  /* 1. Create and open file ; enter define mode
  ncmpi_create(MPI_COMM_WORLD, "series.nc", NC_CLOBBER, MPI_INFO_NULL, &ncfile);

  /* 2. Define dimensions
  ncmpi_def_dim(ncfile, "Point", 2, &pdimid);

  /* 3. Define variables
  ncmpi_def_var(ncfile, "Point", NC_INT, 2, &pdimid, &pvarid);

  /* 4. Switch to data mode
  ncmpi_enddef(ncfile);

  /* 5. Store data
  int start = 0;
  int count = 2;
  ncmpi_put_vara_int(ncfile, pvarid, start, count, point, 2, MPI_INT);

  /* 6. Close file
  ncmpi_close(ncfile);
}
```

3.2.3 HDF5

HDF (*Hierarchical Data Format*) [61, 62] ist eine weitaus umfassendere und komplexere Bibliothek als netCDF. 1988 wurde es am „U.S. National Centrum for Supercomputing Applications“ an der Universität Illinois entwickelt. Seitdem hat es sich ebenso weit verbreitet wie netCDF und unterstützt in der aktuellen Version HDF5 auch eine parallele Ein-/Ausgabe.

Das von HDF5 verwendete Datenmodell beinhaltet *Datensätze* und *Gruppen*. Ein Datensatz lässt sich am besten mit einer Variable von netCDF vergleichen. Die Verknüpfung mehrerer Datensätze bezeichnet man als Gruppe, deren Struktur dem Dateisystem von Unix sehr ähnlich ist. Jede Gruppe kann Datensätze oder weitere Gruppen in einer hierarchischen Struktur beinhalten.

Ein Datensatz besteht aus einem Kopf und den zu speichernden Daten. Der Kopf beinhaltet den Namen, den Datentyp, ein Datenarray und das Speicherlayout. Der Name dient der Identifikation des Datensatzes. Der Datentyp kann ein Basisdatentyp oder eine Verknüpfung mehrerer Basisdatentypen sein (ähnlich einer C-Struktur). Die in HDF5 verwendeten Basisdatentypen werden über vordefinierte Namen angesprochen, wie z.B. `H5F_NATIVE_FLOAT`. Das Datenarray definiert Größe und Form der gespeicherten multidimensionalen Struktur. Schließlich definiert das Speicherlayout, ob die Daten in einer Datei linear hintereinander oder blockweise gespeichert werden sollen.

Die Schnittstelle zur parallelen Ein-/Ausgabe verwendet ebenfalls MPI-IO, um einen kollektiven Datentransfer zu realisieren. Die Auswahl des Kommunikationsmodus erfolgt dabei jedoch nicht wie bei MPI-IO durch Aufruf der entsprechenden Funktion, sondern unter Verwendung von Parametern, die der Bibliothek den gewünschten Kommunikationsmodus übergeben. Dieses Verfahren erschwert jedoch die Nachvollziehbarkeit von Quelltext, da die Stellen, an denen die Auswahl des Kommunikationsmodus und der Datenzugriff stattfinden, oft weit auseinander liegen.

Die Vorgehensweise der Implementierung ist der von pnetCDF sehr ähnlich. In einem ersten Schritt erfolgt neben dem Öffnen der Datei die Definition der Datenstrukturen. Anschließend werden die Daten in die entsprechenden Strukturen und damit auf den Festspeicher übertragen. Die einzelnen Teilschritte sind im folgenden Beispiel wieder an entsprechender Stelle kommentiert.

```
#include <hdf5.h>

void main(int argc, char **argv)
{
    /* initialize MPI */
    MPI_Init(&argc, &argv);

        /* Create structure of Point(x,y)
```

```
    int *data = ( int *) malloc( sizeof ( int )*2);

    /* Set up file access property list with parallel I/O access
    hid_t plist_id = H5Pcreate(H5P_FILE_ACCESS);
                     H5Pset_fapl_mpio( plist_id , MPI_COMM_WORLD, MPI_INFO_NULL);

    /* Create a new file and release property list identifier .
    hid_t file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
                    H5Pclose( plist_id );

    /* Create the dataspace for the dataset .
    hid_t filespace = H5Screate_simple(RANK, dimsf, NULL);

    /* Create the dataset with default properties and close filespace .
    hid_t dset_id = H5Dcreate( file_id , DATASETNAME, H5T_NATIVE_INT, filespace,
                               H5P_DEFAULT);

    /* Create property list for dataset write .
     plist_id = H5Pcreate(H5P_DATASET_XFER);
                H5Pset_dxpl_mpio( plist_id , H5FD_MPIO_INDEPENDENT);

        /* Write data .
    herr_t status = H5Dwrite(dset_id , H5T_NATIVE_INT, H5S_ALL, H5S_ALL,
                             plist_id , data );
    free ( data );

    /* Close/ release resources .
    H5Dclose(dset_id );
    H5Sclose( filespace );
    H5Pclose( plist_id );
    H5Fclose( file_id );

    MPI_Finalize ();
}
```

3.2.4 Open MPI

Eine der neuesten Implementierungen des Standards MPI-2 ist *OpenMPI* [25, 10]. Dabei wird versucht, die Vielzahl der unterschiedlichen Implementierungen von MPI zusammenzuführen. Das Design von OpenMPI baut dabei auf einer Architektur aus Komponenten - der *MPI Component Architecture (MCA)* - auf. Während die Programmierung mit Komponenten bisher eher in der Industrie eingesetzt wurde, steigt die Akzeptanz dieser Methode nun auch im wissenschaftlichen Höchstleistungsrechnen [12, 86].

Die Architektur von OpenMPI setzt sich aus drei Hauptfunktionsbereichen zusammen:

1. **MCA**: Sie bildet die Grundlage und verwaltet die verwendeten Komponenten.

2. **Komponenten:** Jeder Hauptfunktionsbereich von OpenMPI bildet eine eigenständige Komponente.

3. **Module:** Diese bilden die Schnittstellen zur Verwendung der einzelnen Komponenten.

Eine dieser Komponenten ist die parallele Ein-/Ausgabe, die ihrerseits wiederum aus einer Vielzahl von Modulen besteht. Die Einteilung der Funktionen von MPI-IO in einzelne Module folgt einem objektorientierten Ansatz. Die Klasse `MPI::File` bildet dabei die Basisschnittstelle in MPI-IO. Die einzelnen Funktionen von MPI-IO werden in dieser Basisklasse gekapselt und wenn möglich objektorientierten Konzepten angepasst. Ein Beispiel dieser Umsetzung stellt die MPI-Funktion `Get_size()` dar. Diese wird innerhalb der Klasse `File` in eine lesende Zugriffsfunktion umgesetzt, die die Größe der Datei in Form eines Integerwerts zurückgibt.

Die Implementierung von OpenMPI verwendet ein auf der Sprache C basierendes objektorientiertes System mit der Möglichkeit einer einfachen Ableitung und der Verwendung eines auf Referenzzählern basierenden Speichermanagement. Ein „Objekt" in OpenMPI besteht aus einer Struktur und einem einfach instanziierten „Klassen"-Deskriptor. Das erste Element der Struktur beinhaltet einen Zeiger auf die Struktur der Elternklasse. Die Verwendung von Makros ermöglicht eine zu C++ ähnliche Semantik (z.B. `new`, `delete` etc.).

OpenMPI befindet sich momentan noch in der Entwicklungsphase, so dass die Komponente zur parallelen Ein-/Ausgabe auf die Standard-Implementierung ROMIO beschränkt ist. Auf die Darstellung des Anwendungsbeispiels wird aus diesem Grund verzichtet, da der Quelltext der Implementierung dem Beispiel der MPI-C++-Bindungen entspricht.

3.3 Objektorientierte Kommunikation mit TPO++

Die Grundlage aller vorgestellten Schnittstellen zur parallelen Ein-/Ausgabe bildet die Kommunikation. Erst die Übertragung von Objekten ermöglicht damit eine objektorientierte parallele Ein-/Ausgabe. Diesem Problem begegnen die Arbeiten von Ritt [77, 71], die die Kommunikationsbibliothek TPO++ (Transmittable Parallel Objects in C++) hervorgebracht haben. Sie bildet die Grundlage der in der vorliegenden Arbeit entwickelten Schnittstelle und wird daher im Folgenden dargestellt.

Den Kern der Übertragung objektorientierter Datenstrukturen bildet dabei der Serialisierungs- bzw. Deserialisierungsmechanismus, der es ermöglicht, Objekte in ihre Attribute zu zerlegen und auf den einfacheren, von MPI unterstützten Datentyp `MPI_Byte` umzusetzten. Um diese Zerlegung in einer möglichst effizienten Art und Weise durchführen zu können, ordnet TPO++ jedem Objekt eine der folgenden Speicherkategorien zu:

1. *Triviale Objekte*
   Dies sind die Basisdatentypen und aus Basisdatentypen zusammengesetzte Strukturen, die einen konstanten und linearen Speicherbedarf besitzen. Diese können daher auf homogenen Architekturen ohne weitere Speicherkopie veschickt werden.

2. *Konstante Objekte*
   Diese besitzen Beziehungen fester Kardinalität zu trivialen oder weiteren konstanten Objekten. Der Speicherbedarf ist konstant, jedoch nicht zwingend linear. Eine Übertragung ist daher mit maximal einer Speicherkopie möglich.

3. *Variable Objekte*
   In diese Kategorie fallen alle restlichen Objekte, die folglich einen variablen Speicherbedarf besitzen. Dieser muss daher erst bestimmt und kommuniziert werden, bevor eine Kommunikation möglich ist.

Die jeweilige Speicherkategorie eines Objekts bestimmt die Vorgehensweise der Zerlegung des Objekts. TPO++ wählt dabei die effizienteste Variante aus und ermöglicht durch dieses Konzept die Übertragung aller objektorientierten Datenstrukturen.

Die dazu notwendige Schnittstelle bildet in TPO++ die Klasse `Message_data`. Diese kapselt die zu kommunizierenden Daten in einem linearen Bytestrom. Mit Hilfe der beinhalteten Methoden `insert` und `extract` kann der Anwender die zu übertragenden Attribute einer Klasse festlegen. Die folgenden Quelltextbeispiele zeigen die Implementierung der Übertragung eines benutzerdefinierten Objekts. Im ersten Beispiel soll ein Objekt der Klasse `Point` mit den Integerwerten `x` und `y` als Attribut kommuniziert werden. Da es sich bei den Attributen lediglich um Basisdatentypen handelt, fällt diese Klasse in die Speicherkategorie der trivialen Objekte.

```
class Point {
public:
  Point () : x (0), y(0) {}
private :
  int x, y;
};
TPO_TRIVIAL(Point);
```

Ein Objekt der Klasse `Point` besitzt den bekannten Speicherbedarf der beiden Integer-Werte. Daher ist eine Übertragung direkt möglich und eine Kapselung der Attribute in einem Objekt `Message_data` nicht notwendig.

Das zweite Beispiel überträgt ein Objekt der Klasse `Circle`, das als Attribute den Double-Wert `center` und eine Referenz auf das Objekt `Point` besitzt. Zwar ist die Referenz auf den Speicherbereich variabel, doch ist die Größe des Objekts `Point` durchaus bekannt. Daher ist diese Klasse der Speicherkategorie der konstanten Objekte zuzuordnen.

```
class Circle {
public:
  Circle () : radius (0.0) {
    center = new Point (0.0);
  }
  ~Circle () { delete center ;}
  void
   serialize (Message_data& m) const {
    m.insert (*center );
    m.insert ( radius );
  }
  void
   deserialize (Message_data& m) {
    m.extract (*center );
    m.extract ( radius );
  }
private :
  Point* center ;
  double radius ;
};
TPO_MARSHALL(Circle);
```

In diesem Fall handelt es sich um ein konstantes Objekt, da es eine feste Beziehung zur Klasse `Point` besitzt. Demnach ist das Speicherlayout zwar nicht linear, aber die Größe des Objekts aufgrund der trivialen Eigenschaft des Objekts `Point` bekannt. Ein konstantes Objekt erfordert jedoch die Kapselung der zu übertragenden Daten in einem Objekt der Klasse `Message_data`. Die Festlegung der zu übertragenden Attribute erfolgt innerhalb der Methoden `insert` und `extract`.

Sind die zu übertragenden Klassen entsprechend erweitert, gestaltet sich die Implementierung der Kommunikation für den Anwender sehr einfach. Im folgenden Beispiel ist die Übertragung eines Objekts der Klasse `Point` dargestellt. Die Implementierung auf Senderseite lautet unter Angabe des Zielprozesses:

```
Particle p;
CommWorld.send(p, dest_rank);
```

Analog gestaltet sich die Seite des Empfängers folgendermaßen:

```
Status status ;
status =CommWorld.recv(p);
```

Container der STL können ebenso einfach mit Hilfe von zwei Iteratoren versendet werden. Die Angabe von Iteratoren ermöglicht dabei auch ein Verschicken von Teilbereichen eines Containers. Das folgende Beispiel zeigt das Versenden eines Containers, der Objekte der Klasse `Point` beinhaltet:

```
vector<Point> vp;
CommWorld.send(vp.begin(),
               vp.end (),
               destination_rank );
```

Um einen Container der STL zu empfangen, ist lediglich ein Iterator notwendig, der den Einfügepunkt innerhalb des Containers definiert. Die Daten können dabei in einem bereits bestehenden Container empfangen werden, wobei die beinhalteten Daten überschrieben werden, oder mit Hilfe eines Einfügeiterators, der den notwendigen Speicherbedarf automatisch bestimmt, in einen leeren oder an das Ende eines bestehenden Containers eingefügt werden. Das folgende Beispiel zeigt beide Varianten:

```
vector<Point> vp1(x);
vector<Point> vp2;

// vp1 must provide enough space
CommWorld.recv(vp1.begin());

CommWorld.recv( tpo_back_insert_iterator (vp2 ));
```

Beide Beispiele zeigen den Modus der blockierenden Punkt-zu-Punkt-Kommunikation. Daneben unterstützt TPO++ auch die in MPI verwendeten Kommunikationsmodi einer asynchronen (nicht-blockierenden) oder kollektiven Kommunikation.

Insgesamt ermöglicht TPO++ unter Verwendung der generischen Programmierung eine einfache Übertragung von Objekten jeglicher Art und den Datenstrukturen der STL. Das Konzept der Speicherkategorien realisiert dabei eine effiziente Kommunikation von Objekten. Die Semantik des darunterliegenden Standards MPI wird in sinnvollstem Maße erhalten, um den Wechsel eines Anwenders von MPI auf TPO++ möglichst unkompliziert zu gestalten.

4 Beurteilung des Standes der Technik

Die im vorherigen Kapitel vorgestellten Bibliotheken zeigen mehrere mögliche Ansätze zur Umsetzung einer parallelen Ein-/Ausgabe. Bevor jedoch eine Bewertung dieser Bibliotheken vorgenommen werden kann, sollen im folgenden Abschnitt zunächst die Anforderungen erörtert werden, die an eine objektorientierte Schnittstelle zur parallelen Ein-/Ausgabe gestellt werden müssen. Die Analyse der einzelnen Ansätze wird im zweiten Abschnitt die bestehenden Defizite zeigen, aus denen sich die im letzten Abschnitt dieses Kapitels dargestellten Ziele ableiten.

4.1 Anforderungen

Die vorliegende Arbeit beschäftigt sich mit der Konzeption und Implementierung einer objektorientierten Schnittstelle zur parallelen Ein-/Ausgabe. Damit motiviert sie sich aus den beiden Bereichen der Objektorientierung und der Parallelisierung. Die Anforderungen der Objektorientierung liegen einerseits in einer *objektorientierten Modellierung* der Schnittstelle, andererseits in der Nutzung objektorientierter Techniken zur Vereinfachung der *Bedienbarkeit*. Im Mittelpunkt der Parallelisierung stehen neben der *Portabilität* der Schnittstelle eine effiziente Speicherung der Objekte und die Unterstützung einer *Funktionalität*, die sich an gängigen Standards orientiert. Die folgenden Unterabschnitte gehen nun näher auf diese vier Anforderungen ein.

4.1.1 Objektorientierung

Die Anforderungen seitens der Objektorientierung liegen in erster Linie in der Möglichkeit, beliebige objektorientierte Datenstrukturen auf einen Festspeicher zu übertragen. Während die gängigen Standards zur parallelen Ein-/Ausgabe lediglich die Übertragung von Basisdatentypen und zusammengesetzten Strukturen ermöglichen, muss ein objektorientierter Ansatz neben statischen und dynamischen Relationen zwischen Objekten auch die Datenstrukturen der „Standard Template Library“ (STL) unterstützen.

Weiterhin sollte der Vorteil der Vereinfachung der Schnittstelle für den Anwender durch die Objektorientierung genutzt werden. Dabei kann sowohl die Verwendung aus Sicht des Benutzers als auch die Implementierung innerhalb der Schnittstelle vereinfacht werden. Dies kann einerseits durch eine klare Strukturierung der Methoden, andererseits durch die Verwendung bekannter Sprachmittel der Programmiersprache erreicht werden. Letztere sind im Falle von C++ z.B. die Überladung, das Parametrisieren von Klassen, die Verwendung der STL oder die Vorgabe von Standardwerten (engl. default argument).

Ein weiteres Maß für die Güte einer objektorientierten Schnittstelle ist der Grad der Wiederverwendbarkeit, der sich z.B. aus der Häufigkeit der Verwendung einzelner Klassen oder Methoden innerhalb der Schnittstelle ergibt. Darüber hinaus sind die Möglichkeiten der Erweiterbarkeit und Wartbarkeit zu analysieren.

4.1.2 Bedienbarkeit

Neben der gerade dargestellten Vereinfachung der Schnittstelle durch einen objektorientierten Ansatz gibt es weitere Faktoren, die die Anwendung einer Schnittstelle vereinfachen können. Gerade im Bereich der kollektiven Datenzugriffe sind oft zahlreiche zusätzliche Berechnungen notwendig, die zu einem höchst unübersichtlichen Quelltext führen können. Viele dieser Berechnungen gestalten sich sehr generisch, so dass sie implizit durch die Schnittstelle durchgeführt werden können und damit die Zahl der Quelltextzeilen reduziert werden kann. Eine Automatisierung ist jedoch nur möglich, wenn die zur Berechnung notwendigen Informationen dem System bekannt sind. Die Schnittstellen zur parallelen Ein-/Ausgabe sind daher auch danach zu bewerten, wie Metainformationen zu dieser Art der Vereinfachung genutzt werden.

Die Entwicklungskosten einer Anwendung hängen proportional von der für sie aufgewendeten Zeit ab. Durch eine Produktivitätssteigerung des Entwicklers kann eine Senkung dieser Kosten erzielt werden. Eine letzte Anforderung an eine Schnittstelle liegt daher in der Maximierung ihrer Produktivität. Unter der Annahme, dass ein geringerer Programmieraufwand zu einer gesteigerten Transparenz und damit zu einer höheren Produktivität führt, bietet sich als Maßstab hierfür die Zahl der Quelltextzeilen (engl. lines of code) an. Den Vergleichsindex stellt bei der Bewertung der einzelnen Schnittstellen der prozedurale Standard MPI-IO dar.

4.1.3 Portabilität und Effizienz

Kapitel 2 stellte eine Reihe von Entwicklungen in den Bereichen der parallelen Rechner- und Festspeicherarchitekturen vor. Die Kombination der unterschiedlichen Ausprägungen führte in den letzten Jahren zu einer Vielfalt von Höchstleistungsrechnern. Da es für eine neu entwickelte Schnittstelle besonders wichtig ist, eine weite Verbreitung anzustreben, stellt die Portabilität eine der wichtigsten Anforderungen dar. Man unterscheidet dabei einerseits die Portabilität der Schnittstelle selbst, andererseits die Portabilität der Daten, die mit der Schnittstelle generiert wurden. Ein weit verbreiteter Einsatz kann schließlich nur erfolgen, wenn beide Faktoren gleichzeitig unterstützt werden.

Die Portabilität der Schnittstelle sollte auf nahezu alle Architekturen ohne Anpassung der Implementierung erreicht werden können. Dies impliziert gleichzeitig die Portabilität der parallelen Ein-/Ausgabe der Anwendung. In diesem Zusammenhang ist auch eine möglichst starke Unabhängigkeit des Quelltextes von Compilern zu fordern. Da die Portierung von Daten eher eine Anforderung an die Funktionalität der Schnittstelle darstellt, soll sie erst im entsprechenden Unterabschnitt behandelt werden.

Das Ziel einer Parallelisierung liegt im Grunde in der Beschleunigung der Ausführungszeit einer Anwendung. Daher ist eine möglichst effiziente Implementierung der Schnittstelle anzustreben, die einen hohen Datendurchsatz erreicht. Darüber hinaus sollte die Schnittstelle leichtgewichtig sein, um den Speicherverbrauch der Anwendung geringstmöglich zu erhöhen.

4.1.4 Funktionalität

Der Funktionsumfang einer Schnittstelle zur parallelen Ein-/Ausgabe sollte dem Umfang des weit verbreiteten Standards MPI 2 folgen. Dieser sehr umfangreiche Standard setzt sich aus vier Teilbereichen zusammen, aus denen sich die folgenden Anforderungen ergeben.

Das Dateimanagement von MPI 2 ermöglicht neben einfachen Operationen, wie z.B. dem Öffnen und Schließen von Dateien, auch Funktionen zur Verwaltung der Dateien, z.B. das Setzen von Zugriffsrechten. Die Anforderungen an eine Schnittstelle liegen hier zumindest in der Unterstützung der einfachen Funktionen.

Die Datenzugriffsfunktionen bilden den zweiten und gleichzeitig umfangreichsten Teilbereich, der in MPI 2 aus insgesamt 30 Funktionen besteht. Sie bestimmen den Übertragungsmodus der Daten, wie z.B. eine einfache oder kollektive Operation. Der wesentliche Unterschied zwischen einer sequentiellen und einer parallelen Ein-/Ausgabe liegt im gleichzeitigen Zugriff mehrerer Prozesse auf eine Datei. Diese kollektiven Zugriffe bilden daher die Mindestanforderung an eine Schnittstelle zur parallelen Ein-/Ausgabe.

Die grundlegenden Funktionen zur parallelen Ein-/Ausgabe sind mit den ersten beiden Teilen bereits abgedeckt. Gerade im parallelen, wissenschaftlichen Rechen ist die Ausführung einer Operation auf nur einem Teilbereich der Daten häufig notwendig. Aus diesem Grund unterstützt MPI 2 zusätzlich Funktionen zur Partitionierung der Daten auf die einzelnen Prozessoren. Die Bewertung einer Schnittstelle zur parallelen Ein-/Ausgabe sollte darum auch die Unterstützung der Partitionierung von Daten berücksichtigen.

Der letzte Teilbereich ergibt sich aus der Existenz zahlreicher Unterschiede zwischen den einzelnen Rechnerarchitekturen, z.B. in Bezug auf die Byte-Reihenfolge (engl. endianess) oder Wortbreite der Prozessoren. Dadurch werden hohe Anforderungen an die Möglichkeit einer Portierung gespeicherter Daten von einer Architektur auf eine andere gestellt. Eine Konvertierung der Daten in ein architekturunabhängiges Format geht meist zu Lasten der Effizienz der Anwendung. Es sollte daher im Entscheidungsbereich des Benutzers liegen, ob er die Möglichkeiten der Portierung nutzt oder nicht.

Die vollständige Unterstützung des Funktionsumfangs von MPI 2 einerseits und das Ziel des Erreichens einer hohen Effizienz andererseits stellen insgesamt hohe Anforderungen an die Entwickler einer Schnittstelle zur parallelen Ein-/Ausgabe.

4.2 Beurteilung der Schnittstellen

Die im Kapitel 3 vorgestellten Schnittstellen zur parallelen Ein-/Ausgabe werden nun gemäß den im letzten Abschnitt angegebenen Anforderungen bewertet. Die Bindungen von MPI-IO an C++ wurden auf Grundlage der Implementierung ROMIO [91] in der Version $1.2.5.1$ untersucht. Die Schnittstelle HDF5 lag in der Version $5-1.6.4$ und OpenMPI in der Version 1.0 vor. Gemeinsame Entwicklungen des Argonne Nationallabors und der Northwestern Universität führten zu der Version 1.0 von pNetCDF. Zusätzlich wurde zur Bewertung die angegebene Literatur herangezogen.

4.2.1 Objektorientierung

Die Grundlage zur Bewertung der Schnittstellen bildet eine Implementierung des prozeduralen MPI-IO. Eine Übertragung objektorientierter Datenstrukturen erfordert daher die Umsetzung der Objekte auf das Typsystem von MPI-IO. HDF5 und pNetCDF verwenden ein eigenes, arrayorientiertes Typsystem, das aus Basisdatentypen besteht. Zu jedem Datentyp von MPI existiert ein korrespondierender Datentyp. Diese 1:1-Umsetzung führt daher zu keiner Vereinfachung der Schnittstelle und ermöglicht lediglich die Übertragung zusammengesetzter Datenstrukturen. Das jeweils unterstützte Dateiformat HDF bzw. NetCDF speichert zur typsicheren Wiederherstellung der lediglich zusammengesetzten, statischen Struktur zusätzliche Informationen, die jedoch vom Benutzer zusätzlich angegeben werden müssen. OpenMPI und die C++-Bindungen von MPI-IO verwenden ebenfalls das Typsystem von MPI-IO. Die Kapselung der Objekte in Klassen ist zwar bei OpenMPI möglich, doch müssen innerhalb der Klassen entweder Basisdatentypen oder Datentypen von MPI-IO verwendet werden. Darüber hinaus speichert keine der beiden Schnittstellen Strukturinformationen, so dass die bei einem Lesevorgang notwendige Wiederherstellung von objektorientierten Datenstrukturen nicht möglich ist.

Von einer objektorientierten Schnittstelle kann daher in keinem der Fälle gesprochen werden. Dem Anwender stehen lediglich Funktionen zur Verfügung, die eine Übertragung einfacher zusammengesetzter Datenstrukturen ermöglicht. Die Unterstützung sowohl statischer als auch dynamischer Objekte, die durch Aggregation, Assoziation oder Komposition entstanden sind, oder der Datenstrukturen der STL ist folglich ebenso wenig gegeben.

Eine Wiederverwendbarkeit ist damit lediglich bei statischen Strukturen möglich. Wird nur ein einziger Teil der Struktur verändert, muss somit eine neue Struktur definiert werden. Dies ist bei allen Schnittstellen der Fall und erfordert einen hohen zusätzlichen Programmieraufwand, der zu einer schlechten Wartbarkeit der Anwendung führt. Ein objektorientierter Ansatz kann diesen z.B. durch Verwendung von Vererbungsbeziehungen vermeiden.

4.2.2 Bedienbarkeit

Die Bedienbarkeit und Transparenz für den Benutzer hängen sowohl von der Struktur des Programms als auch vom Grad der Automatisierung einzelner Rechenvorgänge ab. Die Programmstruktur unter Verwendung der C++-Bindungen von MPI-IO und von OpenMPI gestaltet sich sehr unübersichtlich, da die Definition einer Datenstruktur, die Generierung einer daraus deklarierten Sicht und die Programmierung der Ein-/Ausgabe an nahezu beliebiger Stelle im Quelltext durchgeführt werden können. Im Gegensatz dazu befinden sich die genannten Programmteile bei HDF5 und pNetCDF an wohl definierten Textstellen. Sie werden unter Angabe des jeweiligen Programmiermodus klar unterschieden. Dadurch erhält der Quelltext eine bessere Struktur und gestaltet sich für den Anwender übersichtlicher. Die Zusammensetzung einer komplexeren Datenstruktur aus einzelnen Basisdatentypen führt trotz dieser Trennung jedoch schnell zu einer unübersichtlichen Programmierung.

Eine Automatisierung interner Berechnungen, z.B. von Offsets innerhalb einer Datei, wird hingegen in keiner der genannten Schnittstellen verwendet. Der gesamte Programmieraufwand liegt folglich beim Programmierer selbst, wodurch die Übersichtlichkeit des Quelltextes weiter verschlechtert wird. Bemerkenswert ist, dass von HDF5 trotz der Fülle an Metadaten, die generiert werden, keine Mechanismen zur Automatisierung unterstützt werden. Mit Hilfe der Informationen über die verwendeten Datenstrukturen wäre es ein Leichtes, diverse Berechnungen vom Benutzer fern zu halten.

Letzlich ermöglicht eine bessere Bedienbarkeit auch eine höhere Produktivität des Anwenders. Anhand des Kriteriums der Quelltextzeilen erfolgte eine Bewertung der Schnittstellen im Vergleich mit dem Standard MPI, deren Ergebnis sich auch aus den in Kapitel 3 vorgestellten Beispielprogrammen ableiten lässt: Jede der verwendeten Schnittstellen erfordert mindestens genauso viele Quelltextzeilen wie MPI und führt damit definitionsgemäß nicht zu einer Steigerung, sondern eher zu einer Reduktion der Produktivität.

4.2.3 Portabilität und Effizienz

Die Parallelisierung der Ein-/Ausgabe erfolgt in allen vorliegenden Fällen unter Verwendung von MPI-IO. Da es sich hierbei um einen weit verbereiteten Standard handelt, für den auf nahezu jeder parallelen Rechnerarchitektur eine Implementierung existiert, ist die Portabilität der Schnittstellen insgesamt akzeptabel. Zwar gibt es hin und wieder spezielle Implementierungen, die auf eine Architektur genau zugeschnitten sind und damit die Leistung eines Standards übertreffen. Doch geht der Trend insgesamt weg von Speziallösungen und hin zum Zusammenschluss von Standardkomponenten (engl. Cluster), für die meistens bereits eine Implementierung von MPI-IO vorliegt.

Der Vorteil der Portabilität wird jedoch durch den Nachteil der beschränkten Leistung relativiert. Die Effizienz einer auf MPI-IO basierenden Schnittstelle kann im besten Fall die maximale Leistung von MPI-IO erreichen. Die mangelnde Unterstützung der Übertragung von Objekten ermöglicht sowohl den C++-Bindungen von MPI-IO als auch OpenMPI im Vergleich zu den anderen Schnittstellen die höchstmögliche Leistung. Angesichts dieses Mangels ist das Ergebnis jedoch nicht befriedigend und daher als neutral einzustufen. Im Vergleich von pNetCDF und HDF5 zeigt sich der hohe Zusatzaufwand beider Schnittstellen, der durch die Umsetzung der verwendeten eigenen Datenstrukturen auf Datentypen von MPI-IO entsteht. In beiden Fällen reduziert dies die Leistung der Schnittstelle im Vergleich zu MPI-IO. Jedoch erzielt pNetCDF dabei eine deutlich bessere Leistung, da ein lineares Speicherlayout verwendet wird, das die Daten in einer zusammenhängenden Form an MPI-IO weitergibt. Häufige Kommunikation zwischen den Prozessen, interne Dateikopfzugriffe und eine hierarchische Dateistruktur verweisen in diesem Vergleich die Effizienz von HDF5 auf den letzten Platz. Das hohe Maß an Flexibilität geht damit zu Lasten der Leistung, wie auch andere Studien beweisen [54, 55, 79].

4.2.4 Funktionalität

Der Funktionsumfang der betrachteten Schnittstellen orientiert sich größtenteils am Standard MPI-IO. OpenMPI, HDF5 und die C++-Bindungen von MPI-IO unterstützen den vollen Funktionsumfang von MPI-IO. Es ist anzumerken, dass HDF5 die Unterscheidung der einfachen und kollektiven Funktionen unter Verwendung von Übergabeparametern durchführt. Der Funktionsumfang wird dadurch jedoch nicht beeinträchtigt. Als einzige Schnittstelle weist pNetCDF ein Defizit auf, da keine asynchrone Ein-/Ausgabe unterstützt wird.

Die Partitionierung der Daten erfolgt bei OpenMPI und den C++-Bindungen auf die bereits in Kapitel 2 vorgestellte Art. Sowohl HDF5 als auch pNetCDF verwenden zur Aufteilung der Daten jedoch Abstände (engl. offset) und die Angabe eines Teilbereichs innerhalb des betrachteten Datensatzes. Dieser Teilbereich wird mit Hilfe

Anforderung	**MPI-C++**	**pNetCDF**	**HDF5**	**OpenMPI**
Objektorientierung				
- Design	o	-	-	+
- Objektübertragung	-	-	-	-
- dynamische Objekte/STL	-	-	-	-
- Strukturinformationen	-	o	o	-
- Wiederverwendbarkeit/ Wartbarkeit	-	-	-	-
Bedienbarkeit				
- Strukturierung	-	o	o	-
- Automatisierung	-	-	-	-
- Produktivität	o	-	-	o
Portabilität und Effizienz				
- Portabilität	+	+	+	+
- Effizienz	o	-	o	o
Funktionalität				
- Funktionsumfang	+	o	+	+
- Datenportierung	+	+	+	+
Bewertung (max. 24)	9	7	9	10

Tabelle 4.1: Bewertung der unterschiedlichen Schnittstellen

eines zu definierenden Arrays festgelegt und kann anschließend exklusiv vom verwendeten Prozess bearbeitet werden. Die notwendigen Angaben sind vom Benutzer zu ermitteln und führen dazu, dass sich die Partitionierung der Daten insgesamt als äußerst aufwändig darstellt.

Die Möglichkeit einer Portierung der Daten ist bei allen Schnittstellen gegeben. OpenMPI und die C++-Bindungen von MPI-IO verwenden dazu das `external32`-Format, um die Daten problemlos zwischen 32-Bit-Architekturen zu portieren. Ebenso unproblematisch gestaltet sich die Portierung bei pNetCDF und HDF5, da beide Schnittstellen das jeweils eigene architekturunabhängige Dateiformat NetCDF bzw. HDF verwenden.

4.2.5 Zusammenfassung

Eine Zusammenfassung der Bewertungen der einzelnen Schnittstellen anhand der aufgestellten Anforderungen in Abschnitt 4.1 enthält Tabelle 4.1. Zwar bestehen viele Bereiche, wie z.B. die Portabilität und die Funktionalität, in denen die gestellten Anforderungen ganz oder teilweise erfüllt werden. Doch bilden die objektorientierten Merkmale die Anforderungen höchster Priorität. Diese können jedoch von keiner der beschriebenen Schnittstellen erfüllt werden. Damit ist die Konzeption und Implementierung einer neuen, objektorientierten Schnittstelle zur parallelen Ein-/Ausgabe erforderlich, deren Zielsetzung im folgenden Abschnitt erläutert wird.

4.3 Zielsetzung der Arbeit

Die Entwicklung einer objektorientierten Schnittstelle zur parallelen Ein-/Ausgabe sollte in erster Linie die im letzten Abschnitt aufgezeigten Mängel bestehender Ansätze beheben und den aufgestellten Anforderungen vollständig genügen. Um dieses Ziel zu erreichen müssen die folgenden vier Probleme gelöst werden:

1. *Objektorientierte parallele Ein-/Ausgabe*
   Die Lösung dieses Teilproblems erfordert die Entwicklung von Mechanismen, die eine Übertragung objektorientierter Datenstrukturen auf einen Festspeicher ermöglichen. Die Objekte sollen dabei in jeder Art von statischer oder dynamischer Beziehung zueinander stehen können. Eine Implementierung der Schnittstelle in der Sprache C++ erfordert darüber hinaus die Unterstützung der Datenstrukturen der STL.

   Weiterhin ist ein Design zu entwickeln, das die bisherige prozedurale Schnittstelle der Ein-/Ausgabe hinter einem objektorientierten Modell vor dem Anwender verbirgt. In Verbindung mit dem Einsatz objektorientierter Sprachmittel wird es dem Anwender dann möglich, die parallele Ein-/Ausgabe auf eine sehr einfache Art und Weise zu formulieren. Dies erhöht die Übersichtlichkeit und damit auch die Wartbarkeit der Programmierung.

   Zur Wiederherstellung objektorientierter Datenstrukturen sind Typinformationen notwendig. Es ist daher ein effizienter Mechanismus zum Speichern und Lesen dieser Strukturinformationen zu finden und zu implementieren. Idealerweise können diese Informationen vor dem Anwender verborgen ermittelt werden, um dessen Zusatzaufwand zur Speicherung von Objekten zu minimieren.

2. *Verbesserung der Bedienbarkeit*
   Die Zusammensetzung einer Datenstruktur aus Basisdatentypen führt zu sehr unübersichtlichen und damit fehleranfälligen Programmen. Die Vorgehensweise zur Bildung einer objektorientierten Datenstruktur muss daher objektorientierten Maßstäben angepasst werden. Dadurch ergeben sich zwei weitere Ziele: Die zur Umsetzung der parallelen Ein-/Ausgabe notwendigen Erweiterungen müssen zum einen innerhalb der zu übertragenden Klasse und zum anderen für

den Anwender in einer möglichst kompakten Form darstellbar sein. Mechanismen zur Automatisierung von Berechnungen, die innerhalb der Schnittstelle durchgeführt werden, sollen den Programmierer dabei zusätzlich entlasten. Beide Ansätze führen zu einer besseren Übersichtlichkeit und höheren Produktivität, da sich im Zuge der Zentralisierung der Datenstrukturen bzw. Internalisierung der Mechanismen in die Schnittstelle die Zahl der Quelltextzeilen deutlich reduzieren lässt.

3. *Portabilität und Effizienz*
   Ein weit verbreiteter Einsatz der Schnittstelle kann nur durch eine hohe Portabilität erzielt werden. Nur die Entkopplung von einer bestimmten Architektur ermöglicht es, auch zukünftige Rechnerarchitekturen weitestgehend zu unterstützen. Daher ist für die Schnittstelle eine Architektur zu entwickeln, die eine möglichst hohe Portabilität ermöglicht.

   Damit die Schnittstelle überhaupt im parallelen wissenschaftlichen Rechnen eingesetzt wird, ist gleichzeitig eine im Vergleich zu anderen Schnittstellen sehr hohe Leistung anzustreben.

4. *Funktionalität*
   Die Funktionalität der Schnittstelle soll sich an dem gängigen Standard MPI-IO orientieren. Damit wird den Benutzern dieser Bibliothek ein Umstieg auf die objektorientierte Schnittstelle erleichtert. Die Schwierigkeit liegt nun darin, diesen sehr hohen Funktionsumfang mit einer möglichst übersichtlichen Schnittstelle in Einklang zu bringen.

   Zur Förderung der Verbreitung der Schnittstelle ist es notwendig, dass nicht nur die Schnittstelle, sondern auch die Daten selbst portabel sind. Ein weiteres Ziel liegt daher in der Entwicklung eines Mechanismus zur architekturunabhängigen Speicherung objektorientierter Datenstrukturen. Die Umwandlung der Daten in ein solches Format sollte jedoch die Leistungsfähigkeit der Schnittstelle nur minimal beeinträchtigen.

5 Entwurf einer objektorientierten Schnittstelle

A ship in port is safe, but that is not what ships are for.
Sail out to sea and do new things.
Grace Murray Hopper

Ausgehend von den in Kapitel 4 aufgestellten Anforderungen an eine objektorientierte Schnittstelle zur parallelen Ein-/Ausgabe wird in diesem Kapitel deren Entwurf beschrieben. Der Aufbau verfolgt dabei die in Kapitel 4.3 aufgestellten Ziele und gliedert sich in die vier Abschnitte über objektorientierte parallele Ein-/Ausgabe, Verbesserungen der Bedienbarkeit, Portabilität und Effizienz der Schnittstelle und Funktionalität.

5.1 Objektorientierte parallele Ein-/Ausgabe

Wichtige Aspekte einer objektorientierten parallelen Ein-/Ausgabe bilden die Bereitstellung einer objektorientierten Schnittstelle für den Anwender und die bidirektionale Übertragung von objektorientierten Datenstrukturen zwischen dem Arbeitsspeicher und dem Festspeicher. Neben der Übertragung selbst und der dafür notwendigen Speicherung von Strukturinformationen stellt auch die Vereinfachung der Schnittstelle unter Verwendung objektorientierter Techniken einen wesentlichen Aspekt dar. Die folgenden drei Abschnitte beschreiben die dafür notwendigen zentralen Konzepte von TPO-IO.

5.1.1 Ein-/Ausgabe von objektorientierten Datenstrukturen

Die Konzeption einer portablen Schnittstelle führte zu dem Ergebnis, dass TPO-IO auf den weit verbreiteten Standard MPI-IO aufsetzen sollte. Dadurch ergibt sich für die Übertragung der objektorientierten Datenstrukturen das Problem, dass eine Transformation der Objekte auf einfache Datenstrukturen vorgenommen werden muss, die auch von der prozeduralen Schnittstelle MPI-IO verarbeitet werden können. Dieses Umsetzungsproblem wurde für die Kommunikation von Objekten bereits in TPO++ gelöst. Bei der Konzeption der Ein-/Ausgabe objektorientierter Datenstrukturen ist daher zu prüfen, inwiefern diese Lösung wiederverwendet werden kann. In Kapitel 3.3 wurde die Vorgehensweise von TPO++ zur Kommunikation objektorientierter Datenstrukturen bereits umfasssend beschrieben.

Grundvoraussetzung für den Erfolg dieser Vorgehensweise ist jedoch die Kenntnis aller Kommunikationspartner über die Struktur des Objekts. Die Unterscheidung von TPO++ in die drei Speicherkategorien *trivial, konstant* und *variabel* zeigt, dass die Struktur nur bei letzterer vor der Kommunikation den beteiligten Partnern nicht bekannt ist und daher anderweitig ermittelt werden muss.

Ein Versuch der Übertragung dieser Konzepte auf die objektorientierte parallele Ein-/Ausgabe zu übertragen, gelingt nur bedingt. Denn die zur Speicherung der Objekte notwendige Serialisierung unter Verwendung der Mechanismen von TPO++ ermöglicht zwar die Persistenz dieses Datenstroms auf einem Festspeicher, sie ist aber nur scheinbar erfolgreich. Das Problem liegt in der Terminierung der Anwendung und dem damit verbundenen Verlust der Strukturinformationen eines Objekts. Genau auf diese Strukturinformationen wird in TPO++ bei der Kommunikation im Gegensatz zu anderen Kommunikationsbibliotheken aus Effizienzgründen verzichtet, da der Datenaustausch zwischen den Prozessen lediglich zur Laufzeit einer Anwendung stattfindet und somit eine zusätzliche Übertragung von Strukturinformationen nicht notwendig ist. Die parallele Ein-/Ausgabe hingegen benötigt diese Informationen, da ansonsten ein zu einem späteren Zeitpunkt durchgeführter Einlesevorgang der Daten in Abhängigkeit von der Speicherkategorie des Objekts zu einem Fehler führen kann. Eine triviale Datenstruktur ist zwar bereits bei der Initialisierung der Anwendung bekannt, die Struktur im Speicher verteilter Objekte konstanter und variabler Größe muss allerdings zu Laufzeit erst wiederhergestellt werden, bevor ein Lesevorgang erfolgreich sein kann. Dazu ist es jedoch notwendig, entsprechende Strukturinformationen der Objekte ebenfalls persistent zu machen, um auch nach Beendigung einer Anwendung ein Objekt rekonstruieren zu können.

Bei der Konzeption von TPO-IO wurde daher neben einer Wiederverwendung des Serialisierungs- und Deserialisierungsmechanismus von TPO++ als Basis die Erweiterung dieses Konzepts um eine effiziente Verwaltung von Strukturinformationen vorgenommen. Damit sind alle objektorientierten Datenstrukturen mit TPO-IO mittels Schreibvorgang auf einen Festspeicher übertragbar und mittels Lesevorgang rekonstruierbar. Das Ablaufschema eines Schreib- und Lesevorgangs von Objekten ist in Abbildung 5.1 dargestellt. Bei einem Schreibvorgang wird das gesamte Objekt durch die Serialisierung in einen einheitlichen Datenstrom transformiert. Die zur Wiederherstellung des Objekts benötigten Informationen werden dabei extrahiert und zusätzlich abgespeichert. Ausgangspunkt bei einem Lesevorgang ist ein initialisiertes Objekt, das eine für die zu lesenden Daten noch nicht passende Struktur besitzt. Diese wird durch Einlesen und Verwenden der Strukturinformationen erfolgreich rekonstruiert. Anschließend kann die Deserialisierung des Datenstroms durchgeführt werden, und das Objekt ist vollständig wiederhergestellt. Die detaillierte Konzeption der Vorgehensweise zur Erstellung und Verwendung von Strukturdaten wird im folgenden Abschnitt erläutert.

Neben der Kommunikation objektorientierter Datenstrukturen bietet TPO++ auch

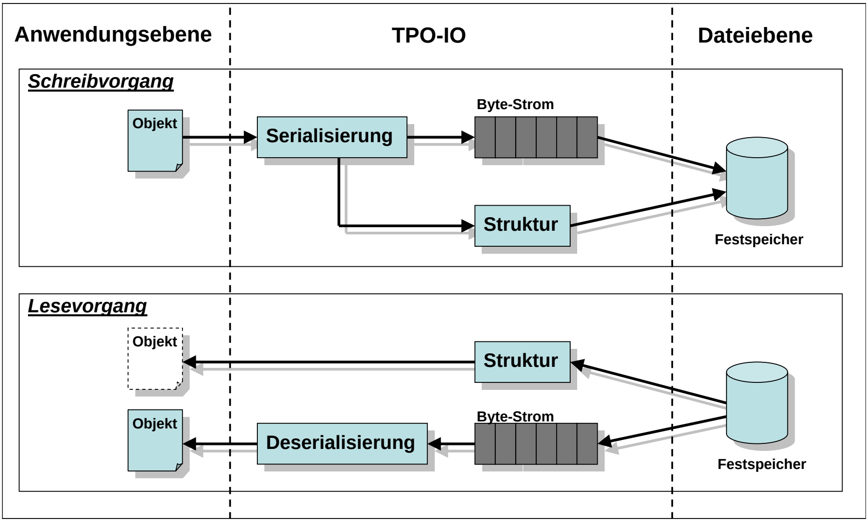


Abbildung 5.1: Schematischer Ablauf der Ein-/Ausgabe von objektorientierten Datenstrukturen. Die Kombinationen aus Serialisierungsmechanismus und Strukturdaten beim Speichern (oben) bzw. aus Deserialisierungsmechanismus und Strukturdaten beim Lesen (unten) ermöglichen eine effiziente Ein-/Ausgabe.

die Möglichkeit der Kommunikation von Standard-Datenstrukturen wie Feldern und Listen. Speziell in C++ sind dies die Datenstrukturen der Standard-Bibliothek STL, wie Vektoren und Listen. TPO-IO unterstützt dieses Konzept und stellt dem Benutzer entsprechende Schnittstellen zur Verfügung. Innerhalb von TPO-IO erfolgt dann eine Umsetzung des Konzepts mit Strukturinformationen auf die Realisierung von TPO++. Ein Iterator-Konzept vereinheitlicht dabei die Verwendung der Standard-Datenstrukturen, indem jeweils ein Start- und ein Endeiterator die Datenstruktur beschreiben. Das Konzept ermöglicht damit auch die Speicherung von Teilmengen einer gesamten Datenstruktur, die z.B. bei kollektiven Zugriffen häufig auftreten.

5.1.2 Speicherung von Strukturinformationen

Die parallele Ein-/Ausgabe objektorientierter Datenstrukturen erfordert einen zusätzlichen Aufwand für die Speicherung von Strukturinformationen. Um eine möglichst hohe Effizienz zu erreichen, ist es daher notwendig, bei der Entwicklung von TPO-IO geeignete Konzepte für diese Speicherung zu finden.

TPO-IO erreicht dies durch zwei Mechanismen: Zum einen durch die Minimierung der Anzahl zusätzlich zu speichernder Strukturdaten und zum anderen durch die Rekonstruktion der Struktur während des Deserialisierungsprozesses zur Laufzeit. Die zusätzlichen Informationen beschränken sich dabei auf das *gesamte* Objekt, z.B. dessen Länge und die Anzahl der in der Datei gespeicherten Objekte. Weil diese Informationen jedoch nicht ausreichend sind, um Objekte aller Speicherkategorien rekonstruieren zu können, wird der zweite Mechanismus benötigt. Dieser speichert Informationen über die *innere* Struktur des Objekts, also die Struktur aller innerhalb des Objekts verwendeten Attribute. Die Umsetzung dieses Mechanismus ist dabei Aufgabe des Benutzers, der innerhalb der Anwendung eine leere Objektstruktur generiert, die dann zur Laufzeit mit den gelesenen Daten gefüllt wird.

Das Konzept von TPO-IO nimmt somit eine Kategorisierung der Strukturinformationen vor und trennt darüber hinaus den Ort, an dem diese Daten gespeichert werden, von dem Ort der eigentlichen Daten ab. Hintergrund ist zum wiederholten Mal die Effizienz der Schnittstelle. Die Informationen der inneren Struktur werden zusammen mit den Objektdaten gespeichert und diesen in der Datei vorangestellt. Dadurch kann das Objekt zusammen mit den Strukturinformationen als Ganzes gelesen werden, und es müssen keine weiteren Lesevorgänge durchgeführt werden. Die Bündelung zu einem großen Datenzugriff steigert die Effizienz deutlich im Vergleich zu mehreren kleinen Zugriffen.

Bei den Informationen über das gesamte Objekt ist diese Lösung jedoch nicht realisierbar, da die Informationen verfügbar sein müssen, *bevor* ein Objekt überhaupt gelesen werden kann. Sind die Informationen und die Objektdaten in derselben Datei gespeichert, müssten zahlreiche Zugriffe auf die Datei erfolgen, da die Informationen über die gesamte Datei verteilt wären (siehe Abb. 5.2 oben). Zusätzlich müssten an einer vordefinierten Stelle in der Datei die Startpunkte aller Objekte abgespeichert sein, damit das System die Strukturinformation überhaupt finden kann. Dieser Ineffizienz begegnet TPO-IO durch eine Trennung der Informationen von den Objektdaten. Physikalisch werden die beiden Datenströme durch Verwendung einer separaten Metadatei getrennt, in welcher die Strukturinformationen abgelegt werden. Die Metadatei wird dann in einem einzigen Aufruf eingelesen (siehe Abb. 5.2 unten), und TPO-IO berechnet intern aus den Daten die Startpunkte der Objekte. Eine zusätzliche Speicherung dieser Daten wird damit hinfällig, wodurch die Effizienz der Schnittstelle erhöht wird.

Aufgrund der geringen Größe der Metadaten können die Dateien in vielen Fällen von den verwendeten Dateisystemen in einem Cache vorgehalten werden, so dass wiederholte Zugriffe auf die Daten sehr schnell durchgeführt werden können. Ein weiterer Vorteil der Trennung der Daten liegt darin, dass es keine Überlappung von Metadaten und Objektdaten geben kann, die z.B. durch eine Vergrößerung der Anzahl von Strukturinformationen in einer gemeinsamen Datei entstehen würde. Die

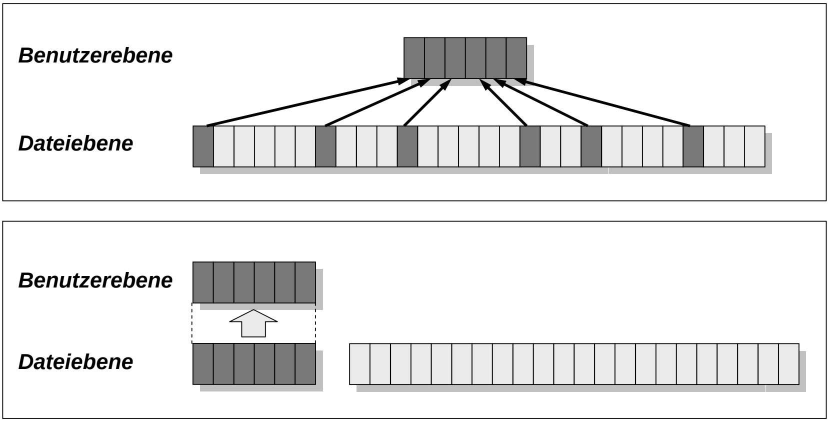


Abbildung 5.2: Die Trennung von Objekt- und Metadaten ermöglicht es, viele kleinere Zugriffe (oben) in einem einzigen größeren Zugriff zusammenzufassen (unten).

Größe einer separaten Metadatei kann hingegen ohne Auswirkungen auf die anderen Daten verändert werden. Gerade bei den im parallelen wissenschaftlichen Rechnen sehr häufig verwendeten kollektiven Zugriffen mehrerer Prozesse auf eine Datei ist es sehr wichtig, dass die einzelnen Schreib- und Lesebereiche der Prozesse in einer Datei einfach und damit schnell berechnet werden können. Die Berücksichtigung eines Headers in Form von Metadaten, die sich im Laufe einer Anwendung ändern oder ändern können, erhöht diesen Aufwand und hat daher negative Auswirkungen auf die Leistung der Schnittstelle.

Insgesamt führt damit die Kombination aus der Speicherung weniger Informationen und der Erweiterung der Anwendung in Verbindung mit der Trennung der Metadaten von den Objektdaten zu einer effizienten Lösung des Problems der Wiederherstellung von Objekten. Darüber hinaus kann im Falle trivialer Objektstrukturen die Effizienz noch gesteigert werden, da eine Erweiterung der Anwendung bei diesem Speichertyp nicht notwendig ist.

Es ist anzumerken, dass durch die beiden Mechanismen die geschriebenen Daten fest an die Anwendung gebunden werden und nicht von anderen Anwendungen verwendet werden können. Dies ist jedoch bei den bestehenden Bibliotheken HDF5 und pNetCDF nicht anders. Der Fokus der Arbeit lag entsprechend dem Umfeld des

Höchstleistungsrechnens auf einer möglichst effizienten Umsetzung und nicht auf der Entwicklung einer Schnittstelle, die den Anwendungen die Persistenz ganzer Klassenhierarchien (wie das in Java der Fall ist) oder der gesamten Anwendung selbst inklusive deren Zustandes (wie von Checkpointing-Systemen durchgeführt) ermöglicht. Die Flexibilität muss an dieser Stelle eindeutig hinter dem Aspekt der Leistung zurücktreten. Jedoch gibt es Möglichkeiten, die Bedienbarkeit ohne nennenswerten Leistungsverlust zu verbessern. Diese werden in Abschnitt 5.2 näher erläutert.

5.1.3 Vereinfachung der Schnittstelle

Der Einsatz von C++ als objektorientierter Programmiersprache ermöglicht eine deutliche Vereinfachung der Schnittstelle für den Anwender. Neben der Nutzung spezieller Techniken wie dem Überladen, der Parametrisierung von Klassen durch Templates und der Verwendung von Standardwerten bieten vor allem die Konventionen der STL Möglichkeiten zu einer einfacheren Schnittstelle für den Benutzer. So führt in TPO-IO die Unterstützung des Iteratorkonzepts der STL zur Vereinheitlichung der Schnittstelle. Unabhängig vom verwendeten Datentyp erwartet die Schnittstelle dabei vom Benutzer nur die Angabe von zwei Iteratoren des zu speichernden oder zu ladenden Speicherbereichs.

Neben diesen durch spezielle Techniken hervorgerufenen Vereinfachungen soll vor allem eine geeignete Abbildung der MPI-IO-Funktionen auf ein objektorientiertes Klassenmodell der Schnittstelle deren Anwendung im Vergleich zu anderen Bibliotheken deutlich vereinfachen. Die Konzeption von TPO-IO zur Abbildung und Strukturierung der Klassen wird im nächsten Abschnitt dargestellt.

Zur Verdeutlichung des in diesem Abschnitt sehr theoretisch beschriebenen Entwurfs der Schnittstelle finden sich in den korrespondierenden Unterabschnitten 6.1.2 und 6.1.3 des folgenden Kapitels einige Beispielimplementierungen der vorgestellten Konzepte.

5.2 Verbesserung der Bedienbarkeit

Die bereits in Abschnitt 2.5.3 dargestellen Quelltextbeispiele zeigen eines der Hauptprobleme von MPI-IO für den Benutzer. Unübersichtliche Datenstrukturen und umfangreiche Parameterlisten der Methodenaufrufe, kombiniert mit oft notwendigen zusätzlichen Berechnungen für Offsets in Dateien oder die Länge einer Datenstruktur führen zu einem nahezu unüberschaubaren und damit schlecht wartbaren Programm. Diese Eigenschaften wirken sich darüber hinaus sehr stark auf die Produktivität eines Anwenders aus und erhöhen somit drastisch die Entwicklungskosten eines Programms.

Ein weiteres Ziel bei der Konzeption von TPO-IO lag daher in einer deutlichen Verbesserung der Bedienbarkeit der Schnittstelle für den Benutzer. Neben der bereits in Abschnitt 5.1.3 dargestellten Vereinfachung der Schnittstelle durch das Design und der damit verbundenen Erhöhung der Transparenz für den Benutzer soll die Bedienbarkeit durch zusätzliche Maßnahmen weiter erhöht werden. Dies sind die in den folgenden Abschnitten dargestellten Bereiche der Automatisierung und der Nutzung von Metadaten. Abschnitt 5.2.3 versucht, die durch diese Maßnahmen zu erwartende Steigerung der Produktivität zu quantifizieren.

5.2.1 Automatisierung

Eine deutliche Verbesserung der Bedienbarkeit durch eine Automatisierung liegt in diesem Zusammenhang immer dann vor, wenn dem Benutzer umständliche und damit zeitraubende Berechnungen oder die Angabe redundanter Informationen abgenommen werden können. Dabei ist es wichtig, zwischen einer aus Sicht des Benutzers überflüssigen, optimierbaren oder notwendigen Berechnung oder Information zu unterscheiden. Das folgende Beispiel eines kollektiven Schreibvorgangs unter Angabe einer Datenpartitionierung soll dies demonstrieren:

```
1  // generieren der Partitionierung (1,1,0,0,0,0)
2  MPI_Type_contiguous(2, MPI_INT, &contig);
3  extent = 6 * sizeof(int);
4  MPI_Type_create_resized(contig, 0, extent, &filetype);
5  MPI_Type_commit(&filetype);
6
7  // oeffnen der Datei
8  MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_CREATE | MPI_MODE_RDWR,
9                MPI_INFO_NULL, &fh);
10
11 // setzen der Partitionierung
12 MPI_File_set_view(fh, 0, MPI_INT, filetype, "native", MPI_INFO_NULL);
13
14 // kollektives schreiben
15 MPI_File_write_all(fh, writebuf, bufcount, MPI_INT, &status);
16
17 // schliessen der Datei
18 MPI_File_close(&fh);
```

In diesem Zusammenhang für den Benutzer überflüssige Angaben von Informationen wären z.B. in Zeile 15 die Parameter sowohl der Länge des Puffers (`bufcount`) als auch des Datentyps (`MPI_INT`), denn beide lassen sich aus dem übergebenen Puffer `writebuf` durch die Schnittstelle ermitteln. Die an einer anderen Stelle ermittelten Werte dieser beiden Variablen bedeuten demnach für den Benutzer einen zusätzlichen Programmieraufwand. In TPO-IO werden sämtliche dieser in MPI-IO vorhandenen Parameter ermittelt und innerhalb der Schnittstelle berechnet. Diese Reduktion der Parameteranzahl führt im direkten Schluss zu einer Erhöhung der Produktivität.

Das Beispiel zeigt die für den Benutzer bei der Partitionierung von Daten sehr umständliche Vorgehensweise von MPI-IO. In den Zeilen $2-5$ wird in aufwändiger Art ein Schreibmuster der Gestalt 110000 generiert, wobei eine 1 den zu beschreibenden Bereich und eine 0 den zu überspringenden Bereich markiert. Das Konzept von TPO-IO optimiert durch Abstraktion der Datenpartitionierung in der Klasse `View` diese für den Benutzer sehr zeitintensive Erstellung einer Sicht. Dadurch entfallen zusätzlich in Zeile 12 die beiden Argumente Datentyp (`MPI_INT`) und Datenrepräsentation (`“native”`). Letzteres wird aufgrund der Sicherstellung der Datenportierbarkeit durch die Schnittstelle fest auf den Typ `external32` gesetzt. Die Implementierung des Beispiels in TPO-IO ist in Unterabschnitt 6.2 dargestellt.

Insgesamt wird durch beide Optimierungen die Übersichtlichkeit der Schnittstelle und damit die Bedienbarkeit deutlich gesteigert. Es wäre denkbar, dass dadurch eine Reduktion der durch den Benutzer potentiell programmierbaren Fehler erzielt wird. Die Stabilität der Schnittstelle würde zunehmen und die der darauf aufsetzenden Anwendungen folglich gesichert werden.

5.2.2 Nutzung von Metadaten

Damit eine Automatisierung von Berechnungen innerhalb der Schnittstelle überhaupt möglich ist, sind zusätzliche Informationen über die Datenstrukturen notwendig. Diese erhält die Schnittstelle aus den zur Verfügung stehenden Metadaten. An dieser Stelle wird noch einmal die Relevanz der kleinen Metadatenmengen und der Trennung der Metadaten von den eigentlichen Daten deutlich. Beide Eigenschaften führen zu einer sehr effizienten Umsetzung der Automatisierung, da sie der Schnittstelle erlauben diese Informationen im Hauptspeicher vorzuhalten. Mehrfache Zugriffe erfordern damit keinen erneuten Ladevorgang vom Festspeicher. Die Anforderungen an eine hohe Effizienz der Schnittstelle können damit weiterhin erfüllt werden.

5.2.3 Steigerung der Produktivität

Als Ergebnis der Bemühungen zur Verbesserung der Bedienbarkeit ergibt sich insgesamt eine Steigerung der Produktivität des Anwendungsentwicklers. Anstatt diese Produktivitätssteigerung mit umfangreichen Studien zu beweisen, soll hier ein anderes Maß verwendet werden: Erfahrungen haben gezeigt, dass eine geringere Menge an Quelltext meist zu einer Erhöhung der Produktivität des Entwicklers führt. Zur Ermittlung des Produktivitätsgewinns durch TPO-IO soll daher diese Kennziffer herangezogen werden. Es ist jedoch zu beachten, dass eine Steigerung der Produktivität durch die angegebenen Maßnahmen nur für den Teil, der die parallele Ein-/Ausgabe betrifft, erreicht werden kann, da die Auswirkungen auf die gesamte Anwendung vom Anteil der parallelen Ein-/Ausgabe an der gesamten Implementierung abhängig sind.

5.3 Portabilität und Effizienz

Um die Portabilität einer Schnittstelle zu gewährleisten, gibt es im Wesentlichen zwei Möglichkeiten. Zum einen kann die Schnittstelle auf jeder Architektur neu implementiert werden. Dies garantiert zwar die höchste Portabilität, verlangt aber aufgrund der Vielzahl paralleler Rechnerarchitekturen auch vom Entwickler den größten Aufwand. Im zweiten Fall wird eine weit verbreitete Standard-Bibliothek als Basis verwendet und die neue Schnittstelle darauf aufgesetzt. Dadurch wird das Problem der Sicherung der Portabilität auf die Standard-Bibliothek verlagert.

Zur Programmierung von Systemen mit verteiltem Arbeitsspeicher hat sich in den letzten Jahren der Standard MPI durchgesetzt. Da sich dieser Trend auch in Zukunft weiter fortsetzen wird, sieht die Konzeption von TPO-IO bezüglich einer Sicherstellung der Portabilität die Verwendung der Definition der Schnittstelle zur parallelen Ein-/Ausgabe (MPI-IO) des Standards MPI-2 als Basis vor. Die weitere Zielsetzung von TPO-IO lautet daher, diesen Standard in vollem Umfang zu unterstützen. Dies erfordert eine Abbildung von MPI-Konzepten auf eine objektorientierte Abstraktion, deren Konzeption im nächsten Abschnitt beschrieben wird.

Die Anforderungen an die Konzeption der Portierbarkeit von TPO-IO gehen jedoch noch weiter und sehen neben einer portablen Schnittstelle noch drei weitere Aspekte vor. Zum Ersten werden in MPI-IO keine Typinformationen der Datenstrukturen gespeichert, wodurch eine Wiederherstellung von Objekten unmöglich wird. In Abschnitt 5.1.2 wurde bereits die Konzeption von TPO-IO beschrieben, die das Problem durch Speicherung von Strukturinformationen löst. Zum Zweiten muss die Partitionierung der Daten zwischen mehreren Prozessen portabel implementiert werden, da ein Wechsel auf eine andere Architektur ansonsten zu einer falschen Verteilung der Daten führt. Die Mechanismen von TPO-IO zur Lösung dieses Problems werden in Abschnitt 5.4.3 beschrieben. Zum Dritten sollen auch die von einer Anwendung mit TPO-IO gespeicherten Daten ebenfalls portabel sein, so dass sie auf einer anderen Architektur wiederverwendet werden können. Auf dieses Problem der Datenportierung wird in Abschnitt 5.4.4 detailliert eingegangen. Damit werden die Schnittstelle und sämtliche damit gespeicherten Daten portabel und können ohne weiteres zusammen mit der Anwendung auf andere parallele Rechnerarchitekturen, die den Standard MPI-IO unterstützen, portiert werden.

Das Aufsetzen auf die bestehende Standard-Bibliothek MPI-IO hat natürlich Auswirkungen auf die Effizienz der Schnittstelle. Deren Leistung kann damit immer nur in Relation zu MPI-IO gesehen werden und erreicht im bestmöglichen Fall die Effizienz von MPI-IO. Die gesamte Konzeption der Anforderungen an die Schnittstelle stellte daher stets das Ziel einer effizienten Umsetzung in den Vordergrund.

5.4 Funktionalität

MPI-IO ist eine der im parallelen wissenschaftlichen Rechnen am häufigsten eingesetzen Schnittstellen. Daher orientieren sich andere Bibliotheken zur parallelen Ein-/Ausgabe an deren Funktionsumfang und ermöglichen so den Benutzern neben einer leichten Portierbarkeit der bestehenden Anwendung auch einen einfacheren Umstieg auf die neue Schnittstelle, da sie das bereits vorhandene Wissen wiederverwenden können. Aus diesem Grund orientiert sich auch TPO-IO an einer Konformität zu MPI-IO, verfolgt das Ziel der Abdeckung des Funktionsumfangs und verwendet die bestehende Terminologie und Semantik von MPI-IO. Dadurch bleibt TPO-IO voll abwärtskompatibel zu MPI-IO und ermöglicht es, auch einfache Datenstrukturen persistent zu machen.

Die Konzeption zur Abstraktion des Funktionsumfangs von MPI-IO auf ein objektorientiertes Klassenmodell soll aufgrund der Vielzahl von 69 Funktionen[1] diese neu organisieren. Eine Analyse der Struktur der Methoden ergibt, dass sich im Wesentlichen drei Hauptgruppen unterscheiden lassen: Methoden des Dateimanagements, Datenzugriffsmethoden und Methoden zur Partitionierung der Daten auf verschiedenen Prozessoren. Die Einteilung der Funktionen in diese drei Gruppen ergibt daher gleichzeitig die in Abbildung 5.3 dargestellte Klassenstruktur von TPO-IO.

Einen weiteren wichtigen Aspekt neben der Umsetzung des Funktionsumfangs stellt die Portierbarkeit der Daten dar. Darunter versteht man die Möglichkeit mit Hilfe der Schnittstelle einmal gespeicherte Daten einer Anwendung auf einer anderen Architektur mit derselben Anwendung wiederverwenden und weiterverarbeiten zu können.

In den folgenden Unterabschnitten werden die Konzepte dieser vier Aspekte nun detailliert dargestellt.

5.4.1 Methoden des Dateimanagements

In diese Kategorie fallen sämtliche Methoden, die zur Verwaltung von Dateien notwendig sind. Dies sind z.B. Methoden zum Öffnen und Schließen, Löschen, Setzen der Größe oder Festlegen der Zugriffsrechte von Dateien. TPO-IO bündelt diese in der für den Benutzer sichtbaren Klasse *File*.

Innerhalb der Klasse verwendet TPO-IO eine Datei von MPI-IO, welche die Verbindung zwischen TPO-IO und MPI-IO bildet. Sämtliche in TPO-IO durchgeführten Operationen werden zentral über diese MPI-Datei an das darunterliegende MPI-IO weitergeleitet. Dadurch wird es auch möglich, die bereits an anderer Stelle geforderte Abwärtskompatibilität zu MPI zu erhalten, so dass auch Anwendungen, die mit MPI-IO parallelisiert sind, nach Integration der neuen Schnittstelle ohne zusätzliche Implementierungsarbeiten funktionieren. Dies ermöglicht dem Benutzer eine

[1] Zum Vergleich: Der Standard MPI-1 besitzt insgesamt 129 Funktionen

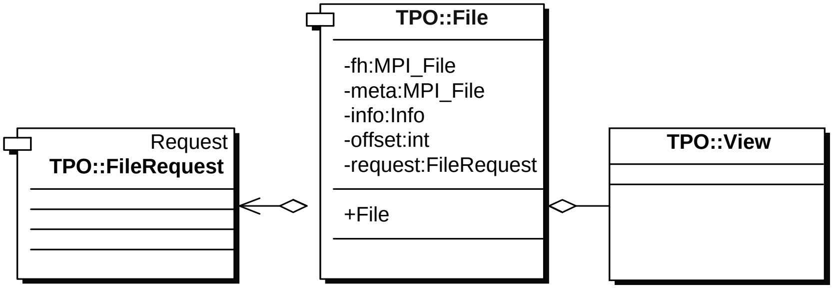


Abbildung 5.3: Die grundlegende Klassenstruktur von TPO-IO ordnet die MPI-Funktionen einer der drei Hauptkategorien Dateimanagement (*File*), Datenzugriff (*FileRequest*) oder Datenpartitionierung (*View*) zu.

schrittweise Anpassung der MPI-IO Implementierung an TPO-IO und schränkt so eine eventuell auftretende Fehlersuche stark ein.

Die Klasse `File` beinhaltet darüber hinaus die Verwaltung der benötigten Strukturinformationen durch Metadateien. Die Informationen dienen der internen Optimierung und Umsetzung effizienter Zugriffe sowie der Partitionierung der Daten zwischen mehreren Prozessoren (siehe Abschnitt 5.4.3). Dem Anwender bringen diese Informationen keinen Nutzen, so dass die Schnittstelle verborgen werden kann und die Übersichtlichkeit von TPO-IO nicht beeinträchtigt wird.

Eine weitere Besonderheit von MPI-IO die Möglichkeit, Informationen über die Ein-/Ausgabearchitektur, z.B. die Striping-Größe oder die Anzahl vorhandener Festplatten im verwendeten parallelen Dateisystem, MPI-IO bekannt zu machen. Innerhalb der Bibliothek können dann Datenzugriffe unter Verwendung dieser Informationen optimiert werden. TPO-IO realisiert diese Funktionalität in einem Objekt der Klasse *Info*. Da es sich um systemabhängige und dateiunabhängige Informationen handelt, wurde diese Klasse von der Klasse *File* entkoppelt. Eine Referenz auf ein *Info*-Objekt ermöglicht dann die Verwendung der Informationen innerhalb der Klasse *File*. Durch die Entkopplung wird es zusätzlich möglich, diese Informationen auch anderweitig zu nutzen. So könnten Erweiterungen des Info-Objekts um andere Systemparameter von der Bibliothek verwendet werden, um z.B. die Kommunikation optimieren zu können. Angaben über die Größe der im System verwendeten Kommunikationspuffer könnten die Effizienz der Übertragung deutlich erhöhen. Voraussetzung dafür ist natürlich die genaue Kenntnis des Anwenders über das verwendete System.

5.4.2 Datenzugriffsmethoden

Die objektorientierte Modellierung von TPO-IO sieht aus Gründen der Wiederverwendbarkeit und Transparenz eine Trennung von Datenstrom und Kommunikationsmodus vor. Die Konzeption zur Generierung des Datenstroms wurde bereits in Abschnitt 5.1 dargestellt. Dieser Abschnitt beschreibt nun die Konzeption der Art und Weise der Speicherung dieses Datenstroms. Die dabei eingesetzten Datenzugriffsmethoden bilden den umfangreichsten Teil des Funktionsumfangs von MPI-IO. Die insgesamt 30 Funktionen ergeben sich durch die Kombinationsmöglichkeiten der folgenden orthogonalen Kommunikationsmodi (siehe auch Abschnitt 2.5.3):

- **Richtung:** Hierbei unterscheidet man zwischen einem *Lese-* und einem *Schreibvorgang*.
- **Synchronisierung:** Im Gegensatz zur *synchronen (blockierenden)* Ein-/Ausgabe ermöglicht die *asynchrone (nicht-blockierende)* Ein-/Ausgabe eine zeitliche Überlappung von Berechnungen und Ein-/Ausgabe.
- **Koordination:** Bei einem *nicht-kollektiven* Zugriff bearbeitet ein Prozess die Datei exklusiv. Ein *kollektiver* Zugriff ermöglicht das gleichzeitige Bearbeiten einer Datei durch mehrere Prozesse.
- **Positionierung des Dateizeigers:** Innerhalb einer Datei kann der Dateizeiger durch Angabe eines *expliziten* Offsets durch alle Prozesse *gemeinsam* oder durch jeden Prozess *individuell* verändert werden.

Darüber hinaus wird in TPO-IO zwischen einfachen Objekten und der Container-Datenstruktur der STL unterschieden, so dass insgesamt 60 Methoden unterstützt werden müssen.

Im Hinblick auf die Vereinfachung der Schnittstelle und eine höhere Transparenz für den Benutzer lag bei der Konzeption ein Ziel in der Reduktion der Menge der Methoden, ohne jedoch auf die Funktionalität verzichten zu müssen. Eine Analyse der Kommunikationsmodi ergab, dass lediglich bei der Positionierung des Dateizeigers die Methoden optimiert werden können. Durch eine Parametrisierung der Methode wird es möglich, die Kommunikationsmodi *explizit* und *individuell* zusammenzulegen. Eine weitere Parametrisierung ermöglicht die Optimierung der Unterscheidung zwischen Objekten und Containern. Der Benutzer übergibt dabei der Methode entweder ein Objekt oder zwei Iteratoren. Die Differenzierung der Datenstrukturen erfolgt dann innerhalb von TPO-IO. Diese beiden Optimierungen führen zu einer Reduktion der notwendigen Methoden auf die überschaubare Zahl von 20.

Die Datenzugriffsmethoden werden in der zentralen Schnittstelle `File` dem Benutzer zugänglich gemacht. Der eigentliche Datenzugriff erfolgt vor dem Benutzer verborgen und wird intern durch die Klasse `FileRequest` abgebildet. Sie stellt damit gleichzeitig die Schnittstelle zwischen dem in der Klasse `Message_data` generierten Datenstrom und den unterschiedlichen Kommunikationsmodi dar. Dadurch kann die Behandlung der Ein-/Ausgabe vereinheitlicht und vereinfacht werden.

5.4.3 Methoden der Datenpartitionierung

Die Umsetzung der Funktionalität von MPI-IO beinhaltet auch die Abbildung der wichtigen Funktionen zur Partitionierung von Daten zwischen mehreren Prozessen. Diese im wissenschaftlichen Rechnen sehr häufig verwendeten Methoden sind an die Anforderungen einer objektorientierten Schnittstelle anzupassen. Im Einzelnen bedeutet dies, dass neben der Bereitstellung entsprechender Schnittstellen vor allem die Granularität an die objektorientierten Datenstrukturen angepasst werden muss. Die Abbildung der Partitionierung wird damit von der in MPI-IO verwendeten Basisdatenstruktur *etype* in TPO-IO auf die Ebene ganzer Objekte angepasst. Damit wird es möglich, die Sicht der Prozesse auf einzelne Objekte zu erlauben oder zu verhindern. Unter Berücksichtigung der Prinzipien objektorientierter Softwareentwicklung ist es daher *nicht* zulässig, einzelne Attribute eines Objekts auszublenden oder nicht. Gleichzeitig bedeutet die Einhaltung dieser Anforderungen jedoch auch eine effiziente Umsetzung, da eine feingranulare Auflösung der Objekte in ihre Bestandteile den Zusatzaufwand durch die Schnittstelle deutlich erhöhen und damit die Leistung stark beeinträchtigen würde.

Innerhalb der Schnittstelle findet dann eine Abbildung der Objekte auf die Basisdatenstruktur von MPI-IO statt. Aus Gründen der Leistung und Portierbarkeit der Daten (siehe dazu Abschnitt 5.4.4) wird dabei als *etype* die auf Bytes basierende Datenstruktur `MPI_BYTE` verwendet. Insgesamt können dadurch die in MPI-IO oder dem darunterliegenden Dateisystem verwendeten Zugriffsoptimierungen verwendet werden, um die Effizienz der Schnittstelle zu erhalten.

Neben der Umsetzung der Datenstrukturen auf die Konzepte von MPI-IO müssen ebenso die Sichten der Prozesse auf die Daten konvertiert werden. TPO-IO realisiert dies durch Kombination von Serialisierungsmechanismus und den von MPI-IO zur Verfügung gestellten Methoden zur Generierung einer Sicht. Der Benutzer generiert eine auf Basis der Objekte definierte Sicht und die Schnittstelle setzt diese auf eine zu MPI konforme Sicht um. Dabei ist zu beachten, dass nicht alle Methoden von MPI zum Erstellen einer Sicht auf jedem System gleich implementiert sein müssen. In diesem Zusammenhang werden portable und nicht portable Methoden unterschieden: Eine Sicht, die mit portablen Methoden generiert wurde, kann von der Implementierung an die verwendete Architektur angepasst werden. Dabei werden die innerhalb der Sicht vorhandenen Datenstrukturen entsprechend skaliert, so dass die Partitionierung auch auf heterogenen Systemen ohne eine Neuimplementierung funktioniert. Aus diesem Grund werden in TPO-IO lediglich portable Methoden verwendet. Der dabei entstehende geringfügig höhere Implementierungsaufwand wird durch den Erhalt einer portablen Schnittstelle mehr als gerechtfertigt. Die Portierung einer Sicht ist unabhängig von einer Datenportierung, die in Abschnitt 5.4.4 erläutert wird.

Die objektorientierte Modellierung abstrahiert die Partitionierung der Daten in Anlehnung an die bereits in Abschnitt 2.5.3 dargestellten *file views* von MPI-IO durch die Klasse `TPO::View`. Im Gegensatz zu MPI-IO entkoppelt diese Klasse die Sicht auf die Daten eines Prozesses von der verwendeten Datei. In diesem Zusammenhang können bereits definierte Sichten an anderer Stelle im Programm ohne erneute Definition wiederverwendet werden. Die in MPI-IO notwendige Anmeldung der Sicht beim System entfällt damit.

Eine Partitionierung der Daten auf unterschiedliche Prozesse bildet die Basis der häufig verwendeten kollektiven Zugriffe. Das Konzept der Klasse `TPO::View` sieht daher eine einfache und transparente Darstellung der Sichten vor. Die für den Benutzer oft sehr komplexen Berechnungen z.B. der Offsets in einem kollektiven Zugriff und die Angabe vieler Parameter, z.B. Länge, Anzahl und Typ der Datenstruktur, die bei der Definition einer Sicht in MPI-IO notwendig sind, sollen daher in TPO-IO durch interne Berechnungen reduziert werden.

5.4.4 Datenportierung

Gerade im Bereich des wissenschaftlichen Rechnens ist es wichtig, durch Simulationen generierte und abgespeicherte Daten, wie z.B. Anfangsverteilungen von Teilchensimulationen, portabel einer Anwendung auf einer anderen Architektur zur Verfügung stellen zu können. Im einfachsten Fall bedeutet dies jedoch, nicht nur das Einlesen der Daten zu ermöglichen, sondern eben auch die durch die Daten repräsentierte Information wiederherstellen zu können.

Um eine Portierung von Daten zwischen heterogenen Systemen zu realisieren, sind drei allgemeine Aspekte zu beachten:

1. Übertragung der Daten zwischen Festspeicher und Arbeitsspeicher
2. Speicherung von Typinformationen der Datenstrukturen
3. Konvertierung zwischen maschinenabhängigen Darstellungen von Daten

Die Übertragung der Daten lässt sich mittels der bereits in Abschnitt 5.4.2 vorgestellten Datenzugriffsmethoden realisieren. Der zweite Punkt ist mit MPI-IO jedoch nicht direkt realisierbar. Dateien von MPI-IO enthalten neben den eigentlichen Daten keine zusätzlichen expliziten Informationen über die Anwendung bzw. die Datenstrukturen der Anwendung und sind damit in keiner Weise selbstbeschreibend.

Eine aufwändige Speicherung entsprechender Daten könnte dieses Problem nur unter erheblichen Auswirkungen auf die Leistung der Schnittstelle lösen. Das Konzept von TPO-IO sieht daher einen Mechanismus zur Lösung dieses Problems vor, der die notwendigen Informationen über die Struktur eines Objekts in zwei Teile zerlegt: Zum einen in Informationen über die Reihenfolge der Speicherung der Attribute, zum anderen in Informationen über die gesamte Größe des Objekts. Die Reihenfolge der Attribute wird in den Übertragungsmethoden der jeweiligen Klasse festgelegt. An dieser Stelle liegt es im Verantwortungsbereich des Benutzers, die korrekte Reihenfolge beim Schreib- und Lesevorgang zu programmieren. Die Informationen über die Größe bekommt die Schnittstelle aus den gespeicherten Metadaten. Dieser Mechanismus ermöglicht eine Reduktion der zu speichernden Daten, indem nur ein Teil der Informationen zum Festspeicher transferiert wird. Die restlichen Informationen werden vom Programm generiert. Neben dem Vorteil einer hohen Effizienz ist diese Lösung auch compilerunabhängig.

Damit generiert die Schnittstelle lediglich einen Bytestrom, der keine Typinformationen beinhaltet. Ein noch zu lösendes Problem liegt nun darin, die bei der Übertragung dieses Datenstroms auf eine andere Architektur auftretenden unterschiedlichen Darstellungen der Daten, wie z.B. die Reihenfolge von Bytes (engl. little/big endian) oder Fließkommaformate, in eine einheitliche Form und damit architekturunabhängig zu konvertieren. Dazu unterstützt MPI-IO mehrere Datendarstellungen, von denen sich eine für den Transfer von Daten zwischen heterogenen Systemen anbietet. Dieses sogenannte *external32* Format konvertiert dabei die Daten in eine für jede MPI-Implementierung lesbare Form. Ein Nachteil von MPI-IO ist jedoch, dass es Aufgabe des Benutzers ist, die Übereinstimmung der Darstellung von geschriebenen und anschließend gelesenen Daten zu gewährleisten. TPO-IO greift daher das Konzept des einheitlichen Formats auf und erweitert es um Mechanismen zur Sicherstellung dieser Übereinstimmung.

Insgesamt liegt durch dieses Konzept einerseits mehr Verantwortung beim Benutzer, andererseits wird ihm jedoch der kritische Bereich der Erhaltung der Portierbarkeit der Daten durch die Schnittstelle abgenommen.

5.5 Zusammenfassung

Ausgehend von den im Kapitel 4.3 genannten Anforderungen an eine objektorientierte Schnittstelle zur parallelen Ein-/Ausgabe wurde im vorliegenden Kapitel deren Konzeption dargestellt und erläutert.

Es konnte gezeigt werden, wie bei der Konzeption unter Einsatz effizienzsteigernder Techniken eine hohe Leistung der Schnittstelle erzielt wird. Die zur Speicherung aller objektorientierten Datenstrukturen notwendigen Strukturinformationen verursachen einerseits einen geringen Leistungsverlust, vereinfachen aber andererseits die Anwendungsentwicklung durch eine höhere Transparenz. Der bei prozeduralen Schnittstellen notwendige Aufbau eigener Datenstrukturen entfällt, wodurch das Fehlerrisiko verringert wird. Zahlreiche Optimierungen der Bedienbarkeit erlauben neben einer weiteren Vereinfachung der Schnittstelle auch eine Steigerung der Produktivität. Das Ziel der Portabilität konnte durch Aufsetzen auf den Standard MPI-2 erreicht werden. Die zu erwartende Effizienz der Schnittstelle wurde dadurch jedoch gleichzeitig begrenzt. Die Abbildung der Funktionalität von MPI-IO auf ein Objektmodell mit drei zentralen Klassen erhöht einerseits das Abstraktionsniveau der Schnittstelle und damit andererseits die Lesbarkeit von Anwendungen. Die Unterstützung der Portabilität von Daten erlaubt den architekturunabhängigen Einsatz der Schnittstelle und steigert deren Akzeptanz im wissenschaftlichen Rechnen. Der Einbezug des Benutzers in den Verantwortungsbereich garantiert dabei den Erhalt einer hohen Effizienz.

Insgesamt wurde damit in diesem Kapitel gezeigt, wie die Konzeption von TPO-IO außer zur Abstraktion und Vereinfachung einer objektorientierten Schnittstelle zur parallelen Ein-/Ausgabe gleichzeitig zu einer hohen Leistung der Schnittstelle führt.

In den nächsten beiden Kapiteln wird die Implementierung des vorgestellten Konzepts detailliert beschrieben und die Leistungsfähigkeit im Vergleich zu MPI-IO anhand von synthetischen Messungen und dem realen Einsatz der Schnittstelle in objektorientierten Anwendungen belegt.

6 Implementierung von TPO-IO

It is better to aim at perfection and miss it
than to aim at imperfection and hit it.
T. J. Watson Sr.

Die im letzten Kapitel vorgestellten Konzepte von TPO-IO müssen nun umgesetzt werden. Die folgenden Abschnitte beschreiben daher die Implementierung der Schnittstelle in der objektorientierten Sprache C++ und erklären die Zusammenhänge zu der bereits in Abschnitt 5.4 dargestellten grundlegenden Klassenstruktur von TPO-IO. Neben den dabei eingesetzten Techniken wird auch die praktische Verwendung der Schnittstelle für den Benutzer erläutert.

6.1 Objektorientierte parallele Ein-/Ausgabe

6.1.1 Übertragung objektorientierter Datenstrukturen

In Abschnitt 5.1 wurde die Konzeption der parallelen Ein-/Ausgabe beschrieben, die die Persistenz objektorientierter Datenstrukturen unter Verwendung des Serialisierungsmechanismus von TPO++ in Verbindung mit der Speicherung notwendiger Strukturinformationen zur Wiederherstellung der Objektstruktur ermöglicht. Der Serialisierungsmechanismus bildet Objekte auf die einfache Datenstruktur `MPI_BYTE` ab. Die Schnittstelle dazu wird dabei von der Klasse `Message_data` vorgegeben, die die beiden Methoden `insert` und `extract` implementiert. Mit Hilfe dieser Methoden bestimmt der Benutzer innerhalb einer Klasse die zu übertragenden Attribute (siehe Abschnitt 3.3). Dabei können entweder einzelne Objekte oder unter Angabe von zwei Iteratoren auch mehrere Objekte in den Übertragungspuffer eingefügt werden. Der Serialisierungsmechanismus von TPO++ generiert anschließend aus dem Objekt oder den Objekten einen linearen Strom einfacher Datenstrukturen.

Die Verwendung eines reinen Datenstroms ist für eine parallele Ein-/Ausgabe von Objekten jedoch nicht ausreichend. Zur Rekonstruktion ist die Verwendung zusätzlicher Informationen notwendig. Um beim Benutzer keinen weiteren Aufwand zu generieren, werden diese Informationen innerhalb von TPO-IO bestimmt. Dazu wurde der Serialisierungsmechanismus von TPO++ erweitert, um für die parallele Ein-/Ausgabe wichtige Informationen während der Serialisierung zu extrahieren und diese in eine separate Metadatei zu speichern. Wurde demnach ein Objekt zur Kommunikation bereits dem System bekannt gegeben, so kann der Benutzer dieses

Objekt auch unmittelbar auf einen Festspeicher übertragen. Einem Benutzer, der bereits TPO++ verwendet, generiert die Schnittstelle zur parallelen Ein-/Ausgabe also keinen zusätzlichen Implementierungsaufwand. Eine Ausnahme bildet die in Abschnitt 5.1 bereits dargestellte Konzeption zur Wiederherstellung von im Hauptspeicher dynamisch verteilten Objekten, deren Implementierung im folgenden Unterabschnitt erläutert wird.

6.1.2 Wiederherstellung objektorientierter Datenstrukturen

Eine erfolgreiche Wiederherstellung von Objekten kann nur unter Verwendung von Strukturinformationen erfolgen. Abschnitt 5.1 hat bereits das Konzept der Unterscheidung von Informationen, die die gesamte Struktur, und Informationen, die die innere Struktur eines Objekts betreffen, dargestellt.

Die Implementierung zur Speicherung von Informationen über die gesamte Struktur erfolgt innerhalb von TPO-IO in der Klasse `FileRequest`. Neben der Implementierung der einzelnen Übertagungsmodi, die in Abschnitt 6.4 näher erläutert werden, und der eigentlichen Speicherung der Daten findet in dieser Klasse das Laden und Speichern von Metainformationen statt, die zur Wiederherstellung der gesamten Struktur notwendig sind. Unmittelbar bevor ein Speichervorgang durchgeführt wird, werden die Informationen über das Objekt, wie z.B. dessen Länge oder die Anzahl der Objekte, aus dem Datenstrom des Serialisierungsmechanismus extrahiert und in der bereits angelegten Metadatei gespeichert. Der folgende Quelltext zeigt beispielhaft die Implementierung für die Länge eines Objekts:

```
1 int olength = (int)getMDS()->get_objectsize();
2 int error = MPI_File_write_at(meta, 0, &olength, sizeof(int),
3                               getMDS()->get_datatype(), getMPIStatus());
```

In den Zeilen 1 und 3 sieht man den Zugriff auf das serialisierte Objekt `getMDS()`, das mehrere Methoden zur Informationsgewinnung enthält. Die Implementierung von TPO-IO verwendet aus Effizienzgründen an dieser Stelle eine MPI-Datei. Dies ist möglich, da es sich bei den Metadaten um einfache Datenstrukturen handelt. Neben der Länge eines Objekts wird zusätzlich die Anzahl der Objekte gespeichert. Die Positionen der einzelnen Objekte innerhalb einer Datei werden von der Bibliothek berechnet. Dies ist effizienter als die Speicherung sämtlicher Positionen, da gerade der Zugriff auf kleine Datenmengen die Leistung deutlich beeinträchtigen würde.

Die Wiederherstellung der inneren Struktur eines Objekts findet analog unmittelbar vor dem Einlesen der Daten innerhalb der Klasse `FileRequest` statt. Die folgende Implementierung zeigt analog zum oben angegebenen Beispiel die Wiederherstellung der Länge eines Objekts:

```
int olength;
int error = MPI_File_read_at(meta, 0, &olength, sizeof(int),
                             getMDS()->get_datatype(), getMPIStatus());
```

Bei Objekten, die ein triviales oder konstantes Speicherlayout besitzen, ist diese Information für eine erfolgreiche Wiederherstellung völlig ausreichend. Der Grund liegt darin, dass die Instanziierung eines solchen Objekts durch den Benutzer bereits die notwendigen Speicherbereiche initialisiert und die Daten lediglich an diese Stelle im Arbeitsspeicher kopiert werden müssen. Objekte, die ein dynamisches Speicherlayout besitzen, wie z.B. eine Referenz auf ein weiteres Objekt, initialisieren zwar diese Referenz, nicht jedoch den für das referenzierte Objekt notwendigen Speicher. Die Implementierung zur Wiederherstellung solcher Objekte erfordert demnach zusätzlich zu den bereits gespeicherten Informationen weitere Informationen zur inneren Struktur. Eine mögliche Lösung wäre die Speicherung des gesamten Speicherlayouts eines Objekts in der Metadatei. Die Fülle an Informationen würde die Effizienz der Schnittstelle jedoch enorm beeinträchtigen. Aus diesem Grund ist es Aufgabe des Benutzers, diese Strukur zu rekonstruieren.

Der Zusatzaufwand für den Benutzer ist dabei jedoch relativ gering, da z.B. lediglich der Konstruktor der entsprechenden Klasse in der Weise zu modifizieren wäre, dass bei Instanziierung des Objekts automatisch auch eine Initialisierung der verwendeten Attribute vorgenommen würde. Handelt es sich jedoch um ein Attribut variabler Größe, scheitert dieser Ansatz. Eine bessere Lösung bietet die Erweiterung der Serialisierungsmethoden, wie der folgende Quelltext zeigt:

```
1  class Point {
2  public:
3      void serialize (Message_data& m) const {
4          const_cast<Point *>(this)->dimension=p.size();
5          m.insert(dimension);
6          m.insert(p.begin(), p.end());
7      }
8      void deserialize (Message_data& m) {
9          m.extract(dimension);
10         p.resize(dimension);
11         m.extract(p.begin(), p.end());
12     }
13     std::vector<double> p;
14
15 private:
16     int dimension;
17 };
```

In diesem Beipiel muss der Benutzer lediglich die bereits bekannten Funktionen `serialize()` und `deserialize()` um die Dimension des Vektors erweitern. Zeile 4 ermittelt die momentane Größe des Vektors, die in Zeile 5 zusammen mit den eigentlichen Daten automatisch abgespeichert wird. Während des Einlesevorgangs wird diese Information verwendet, um die Struktur des Vektors wiederherzustellen (Zeile 10). Zwar erhöht sich dadurch die Menge der zu speichernden Daten, doch erfolgt das Einlesen eines Objekts blockweise. Die Effizienz der Schnittstelle wäre daher nur gefährdet, wenn die Informationen an einer anderen Stelle als die zu speichernden Daten abgelegt werden müssten.

6.1.3 Verwendung der objektorientierten Schnittstelle

Die Übertragung eines einfachen Basisdatentyps erfordert die Angabe des Übertragungsmodus, der Daten selbst und der Stelle in der Datei, an welche die Daten übertragen werden sollen. Der Übertragungsmodus ergibt sich durch Aufruf der entsprechenden Methode. Eine genaue Darstellung sämtlicher Möglichkeiten findet sich im nächsten Abschnitt. Alle Methoden sind dabei generisch implementiert und erlauben die Übertragung beliebiger Datenstrukturen. Als Rückgabewert erhält der Benutzer ein Objekt vom Typ `Status`. Die Angabe von Vorgabewerten reduziert die Zahl der notwendigen Methoden und vereinfacht die Schnittstelle, da im Standardfall die Angabe der Parameter entfallen kann. Dadurch können beispielsweise die Übertragungsmodi *individueller Dateizeiger* und *expliziter Dateizeiger* zusammengefasst werden, denn Letzterer unterscheidet sich von Ersterem lediglich durch die Angabe eines zusätzlichen Parameters, der die Positionierung der Daten innerhalb einer Datei angibt.

Im Unterschied zu den einfachen Basisdatentypen erfordert die Übertragung von Datenstrukturen der STL anstelle des Datums die Angabe eines Start- und Ende-Iterators. Das Iteratorkonzept wird von allen Übertragungsmodi in TPO-IO unterstützt, so dass alle Schnittstellen von TPO-IO neben der Übertragung einzelner Objekte auch die Übertragung mehrerer Objekte anbieten.

Insgesamt lautet damit die Schnittstelle der blockierenden, nicht-kollektiven, expliziten bzw. individuellen Ein-/Ausgabe:

```
template <class T>
Status read(T& buf, int offset = 0);
template <class T>
Status write(const T& buf, const int offset = 0);
template <class Iterator>
Status read(Iterator& first, Iterator& last, int offset = 0);
template <class Iterator>
Status write(const Iterator& first, const Iterator& last, const int offset = 0);
```

In Analogie zu diesen Methoden ergeben sich die Schnittstellen der anderen Übertragungsmodi gemäß der aus MPI-IO bekannten Semantik, so dass dem Benutzer insgesamt 20 Methoden zur Verfügung stehen, mit denen alle 60 Funktionen von MPI-IO ausgedrückt werden können. Alle Methoden werden für den Benutzer durch die Klasse `File` abstrahiert.

Der folgende Quelltext zeigt beispielsweise die Implementierung einer blockierenden, nicht-kollektiven und expliziten Speicherung eines einfachen Basisdatentyps:

```
TPO::File fh;
TPO::Status status;
```

```
double d = 1.5;

// Oeffnen der Datei
fh.open(TPO::CommWorld, fname, TPO_MODE_RDWR, NULL);

// Schreiben des Datums
status = fh.write(d, 100);

// Schliessen der Datei
fh.close();
```

Ein wichtiges Ziel von TPO-IO liegt in der Unterstützung von Datenstrukturen der STL. Deren Übertragung gestaltet sich ebenso unkompliziert wie die Übertragung einfacher Objekte. Im folgenden Beispiel wird ein Vektor mit Double-Werten durch einen einzelnen Prozess, blockierend und unter Angabe einer expliziten Positionierung in eine Datei gespeichert:

```
TPO::File fh;
TPO::Status status;
std::vector<double> d (10, 20.0);

// Oeffnen der Datei
fh.open(TPO::CommWorld, fname, TPO_MODE_RDWR, NULL);

// Schreiben des Vektors
status = fh.write(d.begin(), d.end(), TPO::CommWorld.rank()*100);

// Schliessen der Datei
fh.close();
```

6.2 Verbesserung der Bedienbarkeit

6.2.1 Automatisierung

Die im Rahmen der Konzeption analysierten Möglichkeiten der Optimierung von Methoden in Form einer Automatisierung von Berechnungen oder der Reduktion von Übergabeparametern werden in TPO-IO unter Verwendung des internen Schreib- bzw. Lesepuffers umgesetzt. Beispielsweise erfolgt die Berechnung der Lokalität eines Objekts durch Multiplikation der Länge eines einzelnen Objekts mit der Anzahl der Objekte, die sich vor dem betreffenden Objekt befinden. Dies ermöglicht dem Benutzer die Angabe von Abständen in Einheiten von Objekten und erfordert keine komplizierten Umrechnungen in Einheiten von Bytes. Insbesondere die Partitionierung von Daten zwischen mehreren Prozessoren steigert die Komplexität dieser Berechnungen erheblich. Aus dem Puffer lassen sich darüber hinaus noch weitere Informationen, z.B. über den Daten- bzw. Objekttyp und die Objektlänge, extrahieren, so dass viele in MPI-IO notwendigen Parameter vom Benutzer in TPO-IO nicht angegeben werden müssen. Das Beispiel aus Abschnitt 5.2.1 reduziert sich damit zu folgendem Quelltext:

```
TPO::File fh;
TPO::View view;
TPO::Status status;

// Definition der Sicht auf die Daten
view.set(0, writebuf, ''11000'', TPO_INFO_NULL);

// Oeffnen der Datei
fh.open(TPO::CommWorld, fname, TPO_MODE_RDWR, TPO_INFO_NULL);

// Setzen der Sicht auf die Daten
fh.set_view(view);

// Kollektives schreiben
status = fh.write_all(writebuf);

// Schliessen der Datei
fh.close();
```

6.2.2 Nutzung von Metadaten

Die Verwendbarkeit des Lese- bzw. Schreibpuffers endet mit der Terminierung der Anwendung. Lassen sich die benötigten Informationen zur Laufzeit ohne Probleme aus diesem Puffer ermitteln, so muss bei einem erneuten Start der Anwendung auf die Metadaten zurückgegriffen werden, um auch in diesem Fall eine Automatisierung erfolgreich durchführen zu können. Sowohl die Speicherung als auch das Einlesen der Metadaten erfolgt innerhalb der Klasse der Zugriffsmethoden von TPO-IO (`TPO::FileRequest`). Damit sind die bereits im letzten Unterabschnitt beschriebenen Informationen bekannt und können herangezogen werden, um die genannten Mechanismen zur Automatisierung umzusetzen.

6.2.3 Implementierung von Standard-Views

Gerade im parallelen wissenschaftlichen Rechnen stellt die Verteilung der Daten auf mehrere Prozessoren eine notwendige Funktionalität dar. Die Art und Weise dieser Partitionierung läuft dabei häufig nach immer wiederkehrenden Mustern ab. Aus diesem Grund sind einige dieser Standardmuster in TPO-IO bereits vordefiniert. Mit Hilfe von Schlüsselwörtern kann der Benutzer dann die Art der Partitionierung auswählen. Eines der am meisten verwendeten Verteilungsmuster ist das *round-robin*-Verfahren. Dabei wird eine Liste von Objekten durchlaufen und eines nach dem anderen an die vorhandenen Prozessoren verteilt. Prozessor 1 erhält das 1. Objekt, Prozessor 2 das 2., usw. Der Benutzer kann dieses Standardverteilungsverfahren in TPO-IO folgendermaßen erzeugen:

```
TPO::View view;
view. set (0, writebuf , TPO_ROUND_ROBIN, TPO_INFO_NULL);
```

Anstelle einer binären Liste übergibt der Benutzer lediglich das Schlüsselwort `TPO_ROUND_ROBIN`. Innerhalb von TPO-IO wird unter Verwendung der Anzahl vorhandener Prozessoren dann die dazu passende Partitionierung generiert und gesetzt:

```
std :: vector<bool> mask;

// Generiere Vektor
for ( int i = 0; i < TPO::CommWorld.size(); i++)
    if ( i == TPO::CommWorld.rank()) mask.push_back(1)
    else mask.push_back(0);

// Setzen der Partitionierung
set ( offset , buf , mask, info );
```

Damit erzeugt jeder Prozessor abhängig vom jeweiligen Rang eine Sicht auf die Daten. Werden z.B. 4 Prozessoren verwendet, so lautet der Partitionierungsvektor des 2. Prozessors `(0,1,0,0)`. Übersteigt die Menge der Objekte die Anzahl der Prozessoren, wird jeder Vektor wiederholt durchlaufen, bis alle Objekte verteilt sind. Schließlich ist es dem Benutzer noch möglich, eigene Muster zu definieren und deren Verteilungsweise zu implementieren. Insgesamt erhöht die Modellierung der Klasse `View` damit die Wiederverwendbarkeit von TPO-IO und die Übersichtlichkeit der Anwendung.

6.3 Portabilität und Effizienz

Durch Aufsetzen von TPO-IO auf den Standard MPI-IO wird eine sehr hohe Portabilität sichergestellt, da die meisten parallelen Rechnerarchitekturen diesen Standard unterstützen. Aufgabe der Implementierung ist nun die Abstraktion und das Verbergen der prozeduralen Schnittstelle hinter einem objektorientierten Modell. Mit Hilfe eindeutiger Dateibezeichner (engl. file handles), die für den Benutzer nicht sichtbar sind, wird innerhalb von TPO-IO die Schnittstelle zu MPI-IO realisiert. Diese sind direkt in der Klasse `TPO::File` als Variable integriert und bilden damit das Bindeglied zwischen TPO-IO und MPI-IO. Sämtliche Methoden von TPO-IO können dann über dieses Bindeglied auf die Datei zugreifen. Der Vorteil liegt darin, dass die von MPI-IO vorgegebene rudimentäre Struktur eine stabile Grundlage bildet und daher in TPO-IO kein neues Dateimanagement entwickelt und implementiert werden muss. Im Rahmen der vorliegenden Arbeit wurde die Schnittstelle auf verschiedene paral-

Architektur	Betriebssystem	Compiler
Cray Opteron	SuSE SLES 8 (AMD64)	GNU C++ 3.3.1
Cray T3E	UnicosMK 2.0.5	GNU C++ 2.95.2
Hitachi SR8000	HI-UX/MPP 03-07	Kai CC 3.4
Kepler-Cluster	Linux 2.4.20	GNU C++ 3.2
NEC SX-6	Super-UX 14.1	C++/SX Version 1.0

Tabelle 6.1: Architekturen, auf die TPO-IO erfolgreich portiert werden konnte.

lele Rechnerarchitekturen portiert, die in Tabelle 6.1 unter Angabe des verwendeten Compilers der Zielarchitektur aufgelistet sind.

6.4 Funktionalität der Schnittstelle

6.4.1 Dateimanagement

Die Klasse `File` bildet die gesamte Schnittstelle für den Benutzer. Außer den zur parallelen Ein-/Ausgabe notwendigen Zugriffsmethoden stellt diese Klasse jedoch auch die Methoden zur Verwaltung von Dateien und zugehörigen Metadateien (siehe Abbildung 6.1). Die Kopplung der Metadateien an die eigentlichen Dateien erfolgt innerhalb der Methode zum Öffnen einer Datei:

```
int open(Communicator comm, char *filename, int amode, TPO::Info info){
    char metaname[255];
        this->info = info;
        strcpy(metaname, filename);
        strcat(metaname,".meta");
        MPI_File_open(MPI_COMM_WORLD, metaname, amode, info->getMPIInfo(), &meta);
        return MPI_File_open(MPI_COMM_WORLD, filename, amode, info->getMPIInfo, &fh);
}
```

Der innerhalb von TPO-IO verwendete eindeutige Bezeichner `meta` ermöglicht den direkten Zugriff auf die Metadatei. Die Metadatei erhält bei ihrer Erstellung zusätzlich zum Namen der Bezugsdatei die Endung „.meta“, damit sie in einer Verzeichnisstruktur eindeutig identifiziert und zugeordnet werden kann.

Die Klasse `Info` beinhaltet Systemparameter, die vom Benutzer mit Hilfe der Methoden `set_info` und `get_info` festgelegt bzw. abgefragt werden können. Jede Datei erhält beim Öffnen eine Referenz auf das entsprechende Objekt der Klasse `Info`, um diese Informationen stets innerhalb der Schnittstelle verwenden zu können. Eine Implementierung zur Optimierung von Zugriffen unter Verwendung dieser Informationen liegt nicht vor. Im Rahmen dieser Arbeit erfolgte lediglich die Abstraktion der Struktur `Info` von MPI-IO auf ein objektorientiertes Modell.

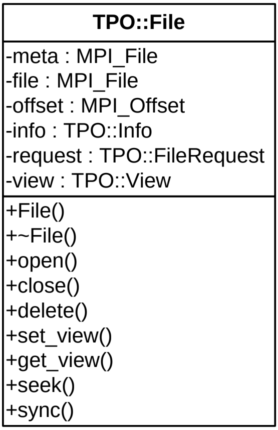


Abbildung 6.1: Die Klasse `File` bildet die Schnittstelle für den Anwender. Zur besseren Übersicht sind nur ausgewählte, für den Anwender sichtbare Methoden und keine Zugriffsmethoden dargestellt.

6.4.2 Datenzugriff

Die Umsetzung der Datenzugriffsmethoden hatte zum Ziel, den Übertragungsmodus von den zu übertragenden Daten zu entkoppeln. Grundlage dafür bildet die objektorientierte Modellierung der Klassen `Message_data` innerhalb von TPO++ und `FileRequest` von TPO-IO. Während die Klasse `Message_data` die zu übertragenden Daten in einem Übertragungspuffer kapselt, implementiert die Klasse `FileRequest` Methoden zur Kontrolle und Durchführung der Ein-/Ausgabe. Die unterschiedlichen Übertragungsmodi entsprechen dabei den von MPI-IO bekannten Zugriffsfunktionen, die sich nach Art der Synchronisierung, Richtung, Positionierung des Dateizeigers und Koordination des Zugriffs ergeben. Sie sind in Unterklassen der Klasse `FileRequest` implementiert und führen dem Modus entsprechende Verwaltungsaufgaben der Übertragungspuffer sowie die eigentliche Ein-/Ausgabe durch. Obwohl Anforderungen (engl. requests) in MPI-IO ursprünglich nur zur asynchronen Übertragung gedacht sind, werden in TPO-IO für sämtliche Modi Anforderungen eingesetzt. Dadurch folgt die Implementierung einem vollständig modularen Aufbau und stellt sich für den Anwender deutlich einfacher dar. Um trotzdem innerhalb der Schnittstelle synchrone Zugriffe realisieren zu können, wird ein asyn-

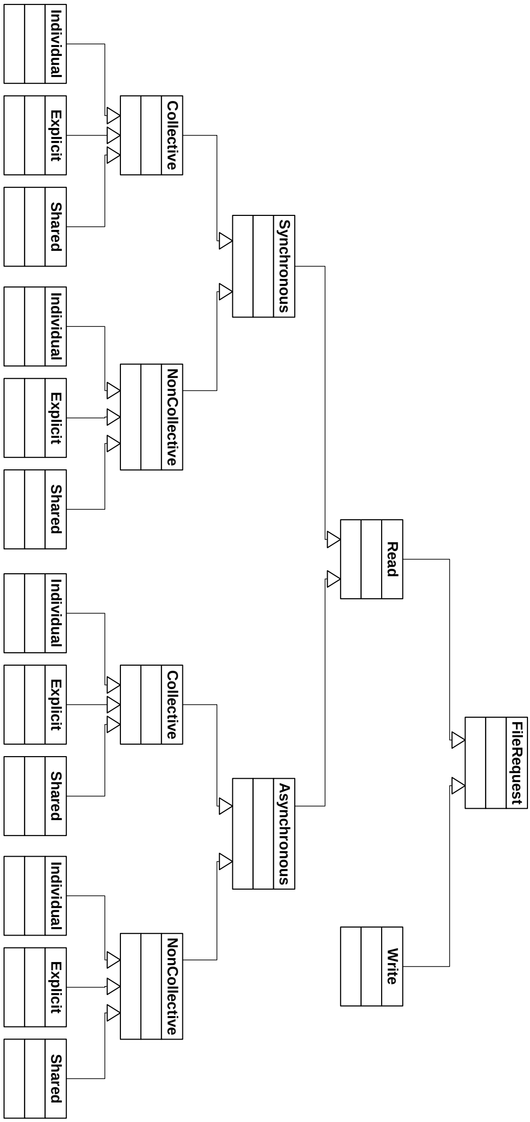


Abbildung 6.2: Objektorientierte Modellierung der unterschiedlichen Übertragungsmodi in TPO-IO. Zur besseren Übersicht sind die analogen Schreibmethoden nicht dargestellt.

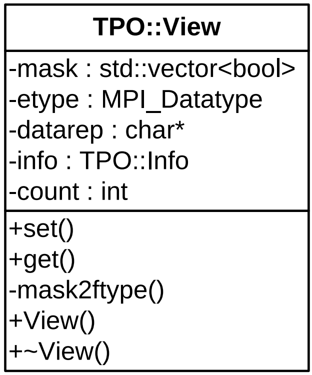


Abbildung 6.3: Die Klasse `View` zur Partitionierung von Daten

chroner Zugriff eingesetzt und bis zu dessen vollständig abgeschlossener Durchführung gewartet. Dies entspricht in der Summe exakt der Art und Weise einer synchronen Übertragung. Abbildung 6.2 zeigt die objektorientierte Modellierung sämtlicher Übertragungsmodi in TPO-IO.

Die Verwendung der Datenzugriffsmethoden ist für den Benutzer ausschließlich über die Klasse `File` möglich. In Abschnitt 6.1 wurde die dazu angebotene Schnittstelle bereits dargestellt. Innerhalb der Zugriffsmethoden der Klasse `File` erfolgt dann eine Umsetzung der aufgerufenen Methode auf die entsprechende Anforderung unter Verwendung aller notwendigen Parameter. An dieser Stelle liegt der Schlüssel zu der im nächsten Abschnitt vorgestellten Automatisierung: Die vom Benutzer an die Zugriffsmethode übergebenen Parameter werden um weitere, innerhalb der Schnittstelle generierte Daten ergänzt und erst dann an die Anforderung übergeben. Folglich können, bevor ein Ein-/Ausgabevorgang überhaupt durchgeführt wird, sowohl die zu übertragenden Datenströme als auch weitere Parameter, wie z.B. die Partitionierung der Daten, beliebig transformiert oder Metainformationen extrahiert und abgespeichert werden.

6.4.3 Datenpartitionierung

Die Ziele der Konzeption von TPO-IO zur Partitionierung von Daten lagen zum einen in der Erstellung einer geeigneten Schnittstelle und zum anderen in der Realisierung einer groberen Granularität. Um eine Trennung von Daten und Partitionierung zu erreichen, modelliert TPO-IO eine eigene Klasse `View` (siehe Abbildung 6.3). Diese stellt eine Maske für die Partitionierung dar und wird an die jeweils verwendeten objektorientierten Datenstrukturen angepasst. Dies ermöglicht dem Benutzer die Wiederverwendung einmal definierter Partitionsmuster. Innerhalb der Klasse `View` wird mittels eines Vektors der Standard-Bibliothek das gewählte Muster realisiert. Dieser beinhaltet Bool'sche Werte, die die Sicht auf ein Objekt entweder zulassen oder verweigern. Eine weitere Möglichkeit wäre, die Struktur `bitset` der STL zu wählen, die mit Hilfe von einzelnen Bits den gleichen Zweck durchaus effizienter erfüllen könnte. Das Problem liegt bei dieser Struktur jedoch darin, dass ihre Länge zur Übersetzungszeit bekannt sein muss. Eine Partitionierung der Daten hängt jedoch meist von der Anzahl der verwendeten Prozessoren ab, die bekanntlich unterschiedlich sein kann. Aufgrund der geringen Flexibilität dieser Struktur eignet sie sich daher nicht zur Partitionierung von Daten.

Die Umsetzung der Granularität von Datenstrukturen hin zu komplexen Objekten erfolgt in TPO-IO unter Verwendung des Serialisierungsmechanismus und mit Hilfe von Funktionen, die in MPI-IO zur Zusammensetzung von Datenstrukturen verwendet werden. Ersterer liefert den aus einem Objekt erstellten Datenstrom, während die Funktionen von MPI-IO genutzt werden, um die Partitionierung dem System bekannt zu machen. Die Umsetzung der Partitionierung auf Objektebene ist in der Methode `mask2ftype` implementiert:

```
MPI_Datatype mask2ftype(std::vector<bool> mask, MPI_Datatype etype = MPI_BYTE, int count){
        MPI_Datatype filetype, helptype;

        int on = 0;
        int i = mask.size();
        int *hdisp;
        hdisp = (int *)malloc(i);
        for(int k = 0; k < i; k++){
            if (mask[k] == 1){
                hdisp[on++] = k;
            }
        }
        int *disp;
        disp = (int *)malloc(on);
        for(int j = 0; j < on; j++){
            disp[j] = hdisp[j];
        }
        // Zusammensetzen des gesamten Views
        MPI_Type_create_indexed_block(on, 1, disp, etype, &helptype);
        int sz;
        MPI_Type_size(etype, &sz);
        int sz1 = (disp[on-1]+1)*sz;
        int holes = sz*(i-1-disp[on-1]);
        MPI_Type_hvector(count, 1, sz1+holes, helptype, & filetype);
```

```
25        MPI_Type_free(&etype);
26        MPI_Type_free(&helptype);
27        return filetype ;
28 }
```

Diese Methode generiert lediglich die Struktur der Partitionierung, die durch den Parameter `mask` angegeben wird, und enthält noch keine Daten. Erst im Anschluss an diese Funktion kann die in Zeile 27 zurückgegebene leere Struktur `filetype` mit einem Datenstrom gefüllt und damit verwendet werden.

Für den Benutzer gestaltet sich die Verwendung der Schnittstelle äußerst einfach. Mit Hilfe der Methode `set` der Klasse `View` können beliebige Partitionierungen durchgeführt werden. Dabei ist es auch möglich, anstelle der Angabe eines Vektors eine Zeichenkette zu übergeben, die das Muster der Partitionierung in Form von Nullen und Einsen enthält. Die Verwendung von Templates vereinfacht auch hier die Implementierung der Schnittstelle erheblich und erhöht die Transparenz von TPO-IO:

```
template < class T>
int set ( int disp , T& buf, std :: vector<bool> mask, int count,
            char *datarep = " external32 ", TPO::Info info );
```

Erst die Partitionierung von Daten auf mehrere Prozesse ermöglicht eine sinnvolle kollektive Ein-/Ausgabe. Jeder Prozess generiert dazu eine für seinen Datenbereich passende Sicht. Im Regelfall bedeutet dies das Aufteilen einer großen Datei auf mehrere Prozessoren. Nachdem TPO-IO aus Sicht des Benutzers eine Partitionierung auf Objektebene, intern jedoch eine Partitionierung mit Hilfe von Strukturen von MPI-IO realisiert, kann die kollektive Ein-/Ausgabe direkt auf den kollektiven Funktionen von MPI-IO aufgesetzt werden. Die Schnittstelle zur kollektiven Ein-/Ausgabe gleicht demnach der Schnittstelle zur nicht-kollektiven Ein-/Ausgabe und ist lediglich um die Endung `_all` erweitert. Damit entspricht die Schnittstelle der Semantik von MPI-IO. Das folgende Beispiel eines kollektiven Zugriffs unter Verwendung einer Partitionierung demonstriert die leichte Handhabung der Schnittstelle:

```
TPO::File fh;
TPO::Status status ;
TPO::View view;
Point p;
std :: vector<Point> vp(30);

// Definition der Sicht auf die Daten
view. set (0, p , ''11100'', TPO_INFO_NULL);

// Oeffnen der Datei
fh.open(TPO::CommWorld, fname, TPO_MODE_RDWR, TPO_INFO_NULL);

// Setzen der Sicht auf die Daten
fh.set_view(view);

```

```
16 // Kollektives Lesen
17 fh.read_all(vp.begin(), vp.end());
18
19 // Schliessen der Datei
20 fh.close();
```

Anstelle des sonst üblichen Vektors erfolgt in Zeile 8 die Übergabe einer Zeichenkette. Von der Unterstützung dieser Möglichkeit profitiert der Benutzer abermals, da es in diesem Fall deutlich einfacher ist, eine Zeichenkette zu erstellen als einen Vektor mit Bool'schen Werten zu generieren.

Die Implementierung der Klasse `View` bringt zahlreiche Vorteile mit sich. In erster Linie liegen diese in der einfachen und transparenten Darstellung der Muster zur Partitionierung. Darüber hinaus führt die Entkopplung von Objektdaten und Partitionierung zu einer robusten und modularen Schnittstelle. Weiterhin entfällt das in MPI-IO notwendige Anmelden einer Partitionierung am System, da dies von TPO-IO intern übernommen wird. Die Abbildung der Objekte auf Funktionen von MPI-IO ermöglicht TPO-IO die Nutzung von bereits in MPI-IO oder dem Dateisystem implementierten Optimierungen von Zugriffen. Schließlich bildet die Klasse `View` die Grundlage für den Einsatz kollektiver Zugriffsfunktionen. Der einzige Nachteil dieses Ansatzes liegt darin, dass eine Partitionierung lediglich für ganze Objekte und keine einzelnen Attribute durchgeführt werden kann. Das Aufbrechen objektorientierter Datenstrukturen würde jedoch ohnehin die Anforderungen an die Entwicklung einer objektorientierten Software verletzen.

6.4.4 Datenportierung

Ein abschließendes Ziel im Rahmen der Umsetzung der Funktionalität von MPI-IO liegt in der Portierbarkeit der gespeicherten Daten. MPI-IO kennt dazu bestimmte Darstellungsformen für Daten, von denen insbesondere das *external32*-Format zur Portierung von Daten zwischen unterschiedlichen Rechnerarchitekturen gedacht ist. Jedoch überprüft MPI-IO nicht, in welchem Format die vorliegende Datei gespeichert wurde. TPO-IO umgeht dieses Problem, indem sämtliche Objekte auf die Datenstruktur `MPI_BYTE` abgebildet werden und zur Speicherung das `external32`-Format verwendet wird. Im Rahmen einer objektorientierten Modellierung ist es möglich, dieses Format als Standard-Wert in allen betroffenen Methoden zu implementieren. Dies hat den Vorteil, dass die Daten standardmäßig vollständig portabel sind. Darüber hinaus kann TPO-IO durch Einsatz dieses Formats auf die Entwicklung eines eigenen Dateiformats verzichten und unterbindet daraus resultierende zeitintensive Datenkonvertierungen.

6.5 Zusammenfassung

Das vorliegende Kapitel hat gezeigt, wie die Konzeption von TPO-IO unter Verwendung von C++ und unter Einsatz damit verbundener Techniken erfolgreich umgesetzt werden konnte. Die Übertragung objektorientierter Datenstrukturen auf einen Festspeicher gelingt auf Basis der Analogie zur Kommunikation und mit Hilfe zusätzlich notwendiger Metainformationen, die aus Gründen der Effizienz in einer separaten Datei gespeichert werden. Die objektorientierte Modellierung der Schnittstelle bietet dem Benutzer eine hohe Transparenz und Wiederverwendbarkeit innerhalb der Anwendung. Mechanismen zur Automatisierung von Berechnungen innerhalb der Schnittstelle, die insbesondere die hohe Komplexität von Partitionierungsalgorithmen reduzieren, führen zu einer kompakten Darstellung der Schnittstelle. Schließlich führt der Einsatz von Vorgabeargumenten und vordefinierten Verteilungsmustern insgesamt zu einer verbesserten Bedienbarkeit und einfacheren Abbildung einer objektorientierten parallelen Ein-/Ausgabe. Die Portabilität der Schnittstelle wird durch Aufsetzen auf den weit verbreiteten Standard MPI-IO gesichert. Insgesamt wird der gesamte Funktionsumfang von MPI-IO unterstützt und an geeigneter Stelle optimiert. Um die Erreichung der in Abschnitt 4.3 aufgestellten Ziele zu belegen, folgen im nächsten Kapitel Darstellungen des praktischen Einsatzes der Schnittstelle und die Ergebnisse synthetischer Leistungsmessungen.

7 Ergebnisse und Anwendungen

Anything can be made measurable in a way
that is superior to not measuring it at all.
Tom Gilb

Dieses Kapitel verifiziert die in Abschnitt 4.3 aufgestellten Ziele durch die Darstellung des praktischen Einsatzes von TPO-IO. Die Abschnitte 7.1, 7.2 und 7.3 stellen Anwendungen vor, in denen die Schnittstelle eingesetzt wurde. Es handelt sich dabei um zwei Anwendungen aus dem Bereich der physikalischen Teilchensimulation und eine Anwendung aus dem Bereich der Bioinformatik. Der Einsatz von TPO-IO erfolgte in den ersten beiden Fällen in bereits bestehenden Implementierungen, die bereits über Ein-/Ausgabemechanismen verfügten. Die Integration von TPO-IO ermöglichte eine Analyse der Vorteile der *Objektorientierung*, z.B. der Vereinfachung der Ein-/Ausgabe, der Wiederverwendung und der Wartbarkeit. Die Verbesserung der *Bedienbarkeit* gegenüber prozeduralen Schnittstellen, die sich als durch einen besseren, parallelen Algorithmus und durch die objektorientierte Modellierung bedingter geringerer Aufwand für den Programmierer äußert, lässt sich anhand einer in C++ und einer in C programmierten Anwendung feststellen. Maßgeblich begünstigt wird diese Verbesserung durch eine einfachere Formulierung der Ein-/Ausgabe. Schließlich wurde TPO-IO in einer objektorientierten Anwendung bereits in der Entwicklungsphase eingesetzt. Dabei konnten Schnittstelle und Anwendung optimal aufeinander abgestimmt und die *Funktionalität* von TPO-IO in vollem Umfang ausgeschöpft werden. Die Überprüfung der *Effizienz* der Anwendungen erfolgte durch Messung ihrer Laufzeiten auf einer unterschiedlichen Zahl von Prozessoren. Die Ergebnisse vermitteln dadurch ein ungefähres Bild des Skalierungsverhaltens der Anwendung vor und nach Einsatz der Schnittstelle.

Ein weiteres Ziel der Arbeit lag in der Entwicklung einer portablen und möglichst effizienten Schnittstelle, um eine Akzeptanz im parallelen wissenschaftlichen Rechnen nicht nur aufgrund der vereinfachenden objektorientierten Methodik, sondern auch aufgrund der Leistung zu erreichen. In einer Anwendung stellt jedoch die Ein-/Ausgabe meist nur einen kleinen Teil der gesamten Berechnungen dar. Eine Erhöhung der Leistung dieses Teils wirkt sich damit nur gering auf die gesamte Leistung der Anwendung aus. Um generelle Abweichungen der Leistung zum Standard MPI-IO aufzudecken, wurden daher synthetische Leistungsmessungen auf vier unterschiedlichen Rechnerarchitekturen durchgeführt. Da sowohl TPO-IO als auch die in Kapitel 3.2 vorgestellten Schnittstellen auf MPI-IO aufsetzen, kann dabei folg-

Abbildung 7.1: 3D-Simulation der Dieseleinspritzung mit 1 Million Teilchen. Die Grafik zeigt den eingespritzten Dieselstrahl, der eine Dichtewelle voranschiebt und an dessen Spitze sich einzelne Tropfen ablösen.

lich maximal dessen Leistung erzielt werden. Abschnitt 7.4 stellt die gewonnenen Ergebnisse dar und untersucht das Zustandekommen der Leistungsunterschiede.

7.1 sph2000

7.1.1 Überblick

Smoothed Particle Hydrodynamics (SPH) als Methode zur Simulation hydrodynamischer Strömungen wird in der wissenschaftlichen Forschung seit langem vielseitig und erfolgreich eingesetzt. In den letzten Jahren sind, vor allem innerhalb der Astrophysik, verschiedene SPH-Anwendungen entstanden [85, 52]. Seit 1999 existiert auch eine Anwendung zur Simulation der Dieseleinspritzung [65]. Bisher sind die genannten SPH-Programme in Fortran oder C implementiert. Dies hat sich als gravierender Nachteil erwiesen, da unübersichtliche Strukturen schwierig zu erweitern sind und folglich nicht weitergepflegt werden. Stattdessen wurden für neue Anwendungen oft eigene spezialisierte Programme entwickelt. Auch die Konfiguration wird mit der Komplexität der Anwendung schwieriger. Durch eine Vorkonfiguration zur Übersetzungszeit wird der Test neuer Anwendungsfälle erschwert, da für jede Konfiguration eine entsprechende Anwendung erzeugt werden muss.

Der Einsatz objektorientierter Methodik mit Entwurfsmustern führt zu einer übersichtlichen Klassenbibliothek mit klar definierten und strukturierten Schnittstellen zwischen den Bibliothekselementen. Die Problemstellung wird dabei auf der Basis des Anwendervokabulars abstrahiert, d.h., die Klassen lassen sich mit den Aufgaben des Problembereichs identifizieren. Vor allem bei komplexen Anwendungen kann die objektorientierte Softwareentwicklung ihre Vorzüge ausspielen.

Mit den von Hüttemann geschaffenen Entwurfsmustern zum Einsatz in wissenschaftlichen Simulationen [42] wurde unter seiner Anleitung die in C++ geschriebene objektorientierte SPH-Teilchenbibliothek sph2000 erstellt [27, 38]. Die Parallelisierung basiert auf einer Gebietszerlegung und verwendet zur Kommunikation TPO++. Die Teilchenbibliothek bietet eine gut strukturierte Klassenhierarchie, deren Entwurfsziel neben Übersichtlichkeit und Wiederverwendbarkeit vor allem eine leichte Erweiterbarkeit war. Entwurfsmuster gruppieren die Klassen in Initialisierung, Kommunikation, Geometrie, Teilchen und physikalische Größen sowie die Berechnung der rechten Seite und Integration. Einheitliche Schnittstellen, die Austauschbarkeit einzelner Berechnungen und Entkopplung der verschiedenen Elemente spiegeln sich in den verwendeten Mustern wider. Die Implementierung benutzt die *Standard Template Library* (STL) von C++.

Die Gesamtstruktur einer SPH-Simulation lässt sich in folgende fünf Blöcke zusammenfassen:

1. *Initialisierung*
   Dabei werden die Simulationsdaten aus den Konfigurationsdateien eingelesen, die physikalischen Randbedingungen des Systems festgelegt und die numerischen Parameter der Methode initialisiert.

2. *Zeitintegration mit Fehlerkorrektur*
   Der in der Konfigurationsdatei angegebene Zeitintegrator wird verwendet, um die Integration der gewöhnlichen Differentialgleichung der SPH-Methode in der Zeit zu integrieren. Zur Festlegung der Genauigkeit der Simulation verwendet der Integrator eine dynamische Fehlerkorrektur.

3. *Initialisierung von Hilfsgrößen*
   Das SPH-Verfahren erfordert zur Integration die Einführung zusätzlicher Hilfsgrößen, z.B. der Stützstellen des Integrators, welche in jedem Integrationsschritt neu berechnet werden müssen.

4. *Lösung der partiellen Differentialgleichung*
   Die für das SPH-Verfahren notwendigen Terme der Differentialgleichung werden hier berechnet. Dies sind die Druckkräfte, der viskose Spannungstensor, die Volumenkräfte und die Beschleunigungen der Teilchen.

5. *Ein-/Ausgabe der Simulationsdaten*
   Die berechneten Simulationsdaten werden gespeichert, um einerseits die zeitliche Entwicklung eines Simulationslaufs nachvollziehen und diese andererseits mit anderen Läufen vergleichen zu können. Im Wesentlichen werden dabei die physikalischen Parameter der SPH-Teilchen gespeichert.

Die Teile 2, 3 und 4 sind in sph2000 weitestgehend parallelisiert und zeigen für sich gesehen eine gute Skalierung (siehe [42]). Die Leistungsmessungen der gesamten Anwendung zeigen jedoch, dass der sequentielle Ein-/Ausgabeteil aufgrund der Vielzahl von verwendeten Teilchen die Skalierung sehr stark beeinträchtigt [26]. Die neu entwickelte objektorientierte Schnittstelle zur Ein-/Ausgabe soll hier Abhilfe schaffen.

7.1.2 Parallele Ein-/Ausgabe

Aufgrund der Gitterfreiheit der SPH-Methode entspricht eine Parallelisierung der Methode auf Basis einer Gebietszerlegung faktisch einer Verteilung der SPH-Teilchen auf die einzelnen Prozessoren. Jeder Prozessor bearbeitet somit nur einen kleinen Teil aller zu simulierenden Teilchen. Um die Entwicklung der physikalischen Größen dieser Teilchen rekonstruieren zu können, wird in regelmäßigen Abständen die Teilchenverteilung auf einen Festspeicher übertragen.

Der in sph2000 anfänglich eingesetzte Algorithmus zur Ein-/Ausgabe kommuniziert dazu die Teilchen aller Prozessoren zu einem Master-Prozess, welcher die eingesammelten Teilchendaten in eine Datei speichert. Dadurch wird die Leistung sowohl durch die zusätzliche Kommunikation als auch durch die sequentielle Ein-/Ausgabe des Master-Prozesses deutlich beeinträchtigt.

Der Einsatz paralleler Ein-/Ausgabe verspricht hier ein deutliches Optimierungspotential. Durch eine Ein-/Ausgabe aller Prozessoren entfällt zum Ersten die Kommunikation der Teilchendaten zum Master-Prozess, zum Zweiten erfolgt dann die Ein-/Ausgabe der Daten parallel, wodurch die Daten im besten Fall mit der aggregierten Bandbreite des gesamten Systems übertragen werden können.

Die Anzahl der Strategien für die parallele Ein-/Ausgabe ergibt sich aus den Kombinationsmöglichkeiten aus synchroner bzw. asynchroner und kollektiver bzw. nichtkollektiver Ein-/Ausgabe. Gegenüber einem synchronen Zugriff hat ein asynchroner Zugriff den Vorteil, dass die Anwendung unmittelbar nach Aufruf einer Ein-/Ausgabeoperation weiterarbeiten kann. Der aus dieser Überlappung von Berechnung und Ein-/Ausgabe resultierende zeitliche Vorteil hängt jedoch entscheidend von Datenabhängigkeiten zwischen der Ein-/Ausgabe und den Folgeberechnungen ab. Bevor die Teilchendaten also durch anschließende Berechnungen verändert werden können, muss der Speichervorgang vollständig abgeschlossen sein. Da der Ablaufalgorithmus von sph2000 direkt im Anschluss an die Ein-/Ausgabe eine Integration der

Teilchen vorsieht, darf zur Sicherstellung der Konsistenz der Teilchendaten innerhalb eines Zeitschritts keine asynchrone Ein-/Ausgabe erfolgen.

Die Wahl der richtigen Strategie der Ein-/Ausgabe reduziert sich somit auf die Wahl zwischen einer synchronen, kollektiven oder synchronen, nicht-kollektiven Ein-/Ausgabe. Beide Ansätze werden in den folgenden Unterabschnitten kurz erläutert und diskutiert.

Nicht-kollektive Ein-/Ausgabe

Die Strategie der nicht-kollektiven Ein-/Ausgabe sieht vor, dass jeder Prozess seinen Teil der Teilchen zeitlich unabhängig von den anderen Prozessoren in eine eigene Datei speichert. Das hat den Vorteil, dass keine Zeitverluste durch eine Synchronisierung aller Prozessoren entstehen. Jedoch hat diese Strategie den Nachteil, dass die Daten der Teilchen auf N Dateien verteilt sind, wobei N die Anzahl der Prozessoren angibt. Zum einen wird es dadurch sehr schwierig und zeitaufwändig die verteilten Daten nach Abschluss der Simulation zusammenzuführen, zum anderen ist es nicht möglich, Daten einer Simulation die von N Prozessoren generiert wurden mit einer Anzahl M ($M \neq N$) Prozessoren weiterzuverarbeiten. Ein weiterer Nachteil der Strategie liegt in der Vielzahl von kleinen Ein-/Ausgabeanfragen der Prozessoren an die Ein-/Ausgabebibliothek und das Dateisystem. Der dadurch enstehende, systembedingte hohe Zusatzaufwand führt dazu, dass nur ein reduzierter Anteil der insgesamt vorhandenen Ein-/Ausgabebandbreite genutzt wird.

Kollektive Ein-/Ausgabe

Die Nachteile der nicht-kollektiven Strategie können durch die Verwendung eines kollektiven Algorithmus behoben werden. Eine synchrone, kollektive Ein-/Ausgabe bedeutet ein gleichzeitiges Speichern der Daten aller Prozessoren in eine gemeinsame Datei. Durch die Kollektivität werden die vielen kleinen Ein-/Ausgabeanfragen durch die Bibliothek synchronisiert, in eine einzelne Gesamtanfrage umgewandelt und an das Dateisystem weitergeleitet. Dadurch können Zugriffe optimiert und die Bandbreite bestmöglich ausgenutzt werden. Darüber hinaus liegt nach Abschluss des Speichervorgangs eine einzige, von der Prozessorzahl unabhängige Datei vor, welche wiederum von einer beliebigen Anzahl von Prozessoren weiterverarbeitet werden kann. Der geringfügige Zusatzaufwand dieser Strategie für die Synchronisierung der Prozessoren ist bei sph2000 vernachlässigbar, da im Anschluss an die Ein-/Ausgabe eine Lastbalancierung durchgeführt wird, und damit ohnehin ein Synchronisationspunkt vorliegt. Die Implementierung der parallelen Ein-/Ausgabe folgt daher dieser Strategie.

Implementierung

Die Implementierungsarbeiten der parallelen Ein-/Ausgabe beschränken sich aufgrund des gut modellierten Designs auf die Kommunikationsklasse *TpoCommunicator* und die Ein-/Ausgabeklasse *ParticleIO*.

In der Kommunikationsklasse wurde die Methode *collectParticles* angepasst. Diese sammelt die Teilchendaten aller Prozessoren ein und speichert sie durch indirekten Aufruf der Routine *saveDataFile* der Klasse *ParticleIO* ab. Der Einsatz einer parallelen Ein-/Ausgabe macht diese Kommunikation gänzlich überflüssig und reduziert den Quelltext um 52 Zeilen.

Die Ein-/Ausgabeklasse beinhaltet sämtliche Variablen und Routinen, die zur Ein- und Ausgabe der Teilchen von bzw. in Dateien notwendig sind. Der sequentielle Algorithmus differenziert zwischen Teilchendaten, die gespeichert werden sollen (Koordinaten, Impulse etc.), und solchen, die nicht gespeichert werden müssen (Stützstellen des Integrators). Diese Unterscheidung führt jedoch zu einem nicht unerheblichen zeitlichen Zusatzaufwand und ist in der parallelen Version daher nicht enthalten. Der Mehraufwand zur Speicherung der erhöhten Datenmenge wird durch die Umstellung des Dateiformats von ASCII-Format auf Binärformat mehr als kompensiert. Der folgende Quelltext zeigt die veränderte Methode *saveDataFile* der Klasse *ParticleIO*:

```
1  void ParticleIO::saveDataFile(const ParticleContainer& particles, string name){
2  
3      TPO::File fh;
4      TPO::View view(TPO_VIEW_CONTIGUOUS);
5  
6      fh.open(TPO::CommWorld, name, TPO_MODE_CREATE);
7      fh.setView(particles, view);
8  
9      fh.write_all(particles.begin(), particles.end());
10 
11     fh.close();
12 }
```

Der Übergabeparameter *ParticleContainer* in Zeile 1 stellt einen Datencontainer der STL dar und beinhaltet die jeweiligen Teilchen des Prozessors. Nach Öffnen der Datei (Zeile 6) und Setzen einer zusammenhängenden Partitionierung (Zeile 4 + 7) werden die Daten mit einem einzigen kollektiven Aufruf in Zeile 9 gespeichert. Dieser muss von allen Prozessoren gleichzeitig aufgerufen werden und stellt damit einen Synchronisationspunkt dar. Der Aufruf der Partitionierung in Zeile 7 zeigt einen deutlichen Nachteil von MPI-IO auf: Unmittelbar nach dem Öffnen einer Datei ist es zwingend notwendig die Sicht auf die Daten festzulegen. Dabei muss die zu lesende oder zu speichernde Datenstruktur bekannt sein. Zwar lässt sich mit Hilfe der Klasse `TPO::View` eine Entkopplung von Daten und Partitionierung vornehmen, doch führt die stringente Vorgabe von MPI-IO dazu, dass diese beim Öffnen einer Datei bereits wieder zusammengeführt werden. Der Übergabeparameter *particles* ist somit unvermeidbar und erhöht den Berechnungsaufwand innerhalb der Bibliothek. An dieser Stelle erkennt man deutlich die Grenzen einer objektorientierten Umsetzung der prozeduralen Schnittstelle MPI-IO.

Bevor jedoch die Teilchendaten auf diese einfache Weise gespeichert werden können, müssen sie mit TPO++ verschickbar gemacht werden. Dazu muss die Basisklasse der Teilchen *BaseParticle* um die beiden Methoden *serialize* und *deserialize* erweitert werden. Innerhalb der Methoden wird festgelegt, welche Parameter der Teilchen verschickt werden können. Das folgende Listing zeigt auszugsweise die erweiterte Klasse:

```
1  class BaseParticle
2  {
3
4  public:
5      void serialize (TPO::Message_data& msg) const{
6          msg.insert ( scalarQuantities .begin (), scalarQuantities .end ());
7          msg.insert ( vectorQuantities .begin (), vectorQuantities .end ());
8
9          msg.insert (id );
10         msg.insert ( fluid );
11     }
12
13     void deserialize (TPO::Message_data& msg){
14          scalarQuantities . resize ( BaseParticle :: scalarCount );
15         msg.extract ( scalarQuantities .begin (), scalarQuantities .end ());
16
17          vectorQuantities . resize ( BaseParticle :: vectorCount );
18         msg.extract ( vectorQuantities .begin (), vectorQuantities .end ());
19
20         msg.extract (id );
21         msg.extract ( fluid );
22     }
23
24 private :
25       vector<double> scalarQuantities ;
26       vector<Vector<double>> vectorQuantities ;
27 };
28 TPO_MARSHALL_DYNAMIC(BaseParticle);
```

An den Zeilen 25 und 26 erkennt man, dass ein SPH-Teilchen aus skalaren und vektoriellen Größen besteht. Diese werden unter Angabe von Start- und Ende-Iteratoren serialisiert bzw. deserialisiert. Das im Anschluss an die Klasse angegebene Makro *TPO_MARSHALL_DYNAMIC(BaseParticle)* meldet die Klasse bei TPO++ an und ermöglicht damit ihre Kommunikation und in der Folge auch ihre Persistenz.

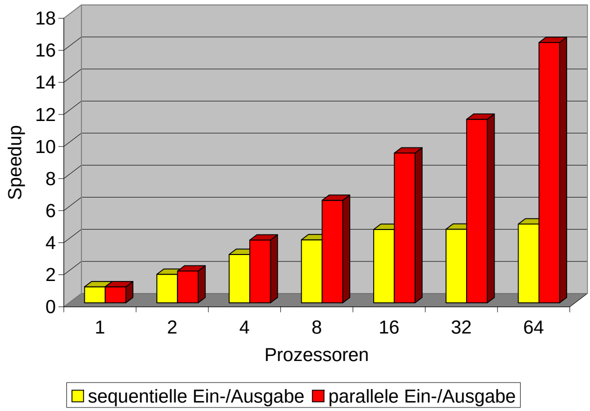


Abbildung 7.2: Vergleich der Speedups mit sequentieller und paralleler Ein-/-Ausgabe bei sph2000

7.1.3 Ergebnisse

Zur Demonstration des praktischen Einsatzes der Schnittstelle und um Leistungsmessungen durchführen zu können, wurde in sph2000 eine Beispielanwendung zur Einspritzung von Diesel in eine Verbrennungskammer implementiert. Aufgrund der guten Erweiterbarkeit eignet sich sph2000 gut zur Simulation dieser Phänomene, da neue physikalische Effekte sich leicht in die bestehende Anwendung implementieren lassen. Mittlerweile sind 5 Kernel-Funktionen zur Glättung der Teilchen, 6 Integratoren und mehr als 20 physikalischen Größen zur Berechnung der Zustands- und Bewegungsgleichungen der Diesel- und Luftteilchen enthalten.

Bereits während der Implementierung der parallelen Ein-/Ausgabe konnte festgestellt werden, dass das oberste Ziel der Entwicklung von TPO-IO erfüllt ist. Da eine objektorientierte Anwendung vorlag, sind die zu speichernden Listen von Partikeln bereits in objektorientierten Strukturen realisiert, die sowohl statische als auch dynamische Relationen enthalten. TPO-IO ermöglicht daher ohne weiteren Implementierungsaufwand eine direkte Übertragung dieser Objekte auf einen Festspeicher. Darüber hinaus werden in sph2000 zusätzlich verschachtelte Datenstrukturen der STL verwendet, die sich ebenfalls mit TPO-IO überaus einfach transferieren lassen.

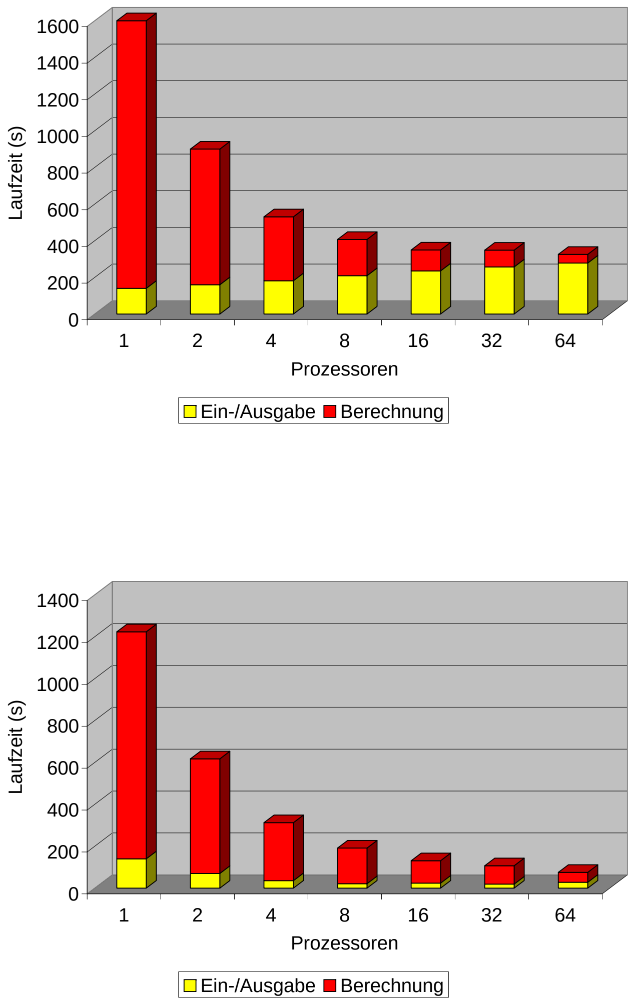


Abbildung 7.3: Laufzeitanteile für Ein-/Ausgabe und Berechnungen bei sph2000 unter Verwendung sequentieller (oben) und paralleler Ein-/Ausgabe (unten).

Weiterhin zeigen die Änderungen des Programms innerhalb der Methoden *collectParticle* und *ParticleIO*, dass der Quelltext gegenüber der sequentiellen Version um insgesamt 112 Zeilen reduziert werden konnte. Dies entspricht einer Verkürzung des Ein-/Ausgabeteils von ca. 90%. Gerade die hohe Funktionalität von TPO-IO - insbesondere die Unterstützung kollektiver Ein-/Ausgabe - trägt dazu in erheblichem Umfang bei. An dieser Stelle kann daher klar von einer Verbesserung der Bedienbarkeit gesprochen werden, die nur durch den Einsatz von TPO-IO und dessen zu STL konformen Schnittstellen möglich geworden ist. Neben der deutlich gestiegenen Übersichtlichkeit und Transparenz für den Benutzer erhöht die vereinfachte Formulierung der Ein-/Ausgabe schließlich auch die Wartbarkeit der Anwendung.

Im Anschluss an die erfolgreiche Implementierung des Modells wurden zahlreiche Simulationsläufe einer 3D-Simulation mit 1 Million Teilchen durchgeführt. Die Ergebnisse liefern wichtige Erkenntnisse über die zeitliche Entwicklung des Dieselstrahls und zeigen, dass es bereits nach kurzer Zeit zu einer Verbreiterung und schließlich einem Aufbrechen des Dieselstrahls und zu damit verbundenen Turbulenzen hinter der Strahlspitze kommt (s. Abb. 7.1).

Die Leistungsmessungen wurden auf dem bereits in Kapitel 2.6 vorgestellten Kepler-Cluster unter Verwendung der Pentium-Knoten mit jeweils einem Prozessor durchgeführt. Während der Simulation wurden nach jedem Zeitintegrationsschritt die Teilchendaten auf das parallele Dateisystem PVFS übertragen.

Die Ergebnisse der Implementierungen sowohl mit sequentieller als auch paralleler Ein-/Ausgabe sind in den Abbildungen 7.2 und 7.3 dargestellt und zeigen einen deutlichen Leistungsgewinn durch den Einsatz von TPO-IO. Während die Laufzeit der Ein-/Ausgabe im sequentiellen Fall aufgrund von erhöhtem Kommunikationsaufwand, der durch die steigende Prozessorzahl stetig zunimmt und dadurch die Skalierung der Anwendung bereits bei 16 Prozessoren begrenzt, kontinuierlich wächst, kann die parallele Version die Laufzeit für die Ein-/Ausgabe mit zunehmender Prozessorzahl reduzieren und skaliert damit auch noch bei 64 Prozessoren.

Bemerkenswert ist, dass selbst im direkten Vergleich der Laufzeiten mit nur einem Prozessor der Berechnungsaufwand der Version mit TPO-IO bereits um ca. 30% geringer ist als bei der sequentiellen Version. Dies liegt daran, dass der Quelltext zur Bestimmung der zu speichernden und der nicht zu speichernden Daten in der Version mit TPO-IO entfällt. Die reine Ein-/Ausgabeleistung ist davon nicht betroffen. Insgesamt zeigen die Ergebnisse damit, dass trotz des erforderlichen Mehraufwands durch die Schnittstelle der Einsatz von TPO-IO nicht nur die reine Ein-/Ausgabeleistung verbessern, sondern oft auch den Berechnungsaufwand, der aufgrund komplizierter Vorbereitungsmaßnahmen zur Durchführung paralleler Ein-/Ausgabe notwendig ist, reduzieren und damit die gesamte Anwendung optimieren kann. Trotz der im Verhältnis zum gesamten Quelltext relativ geringen Änderungen wirken sie sich überaus positiv auf die Effizienz der Anwendung aus.

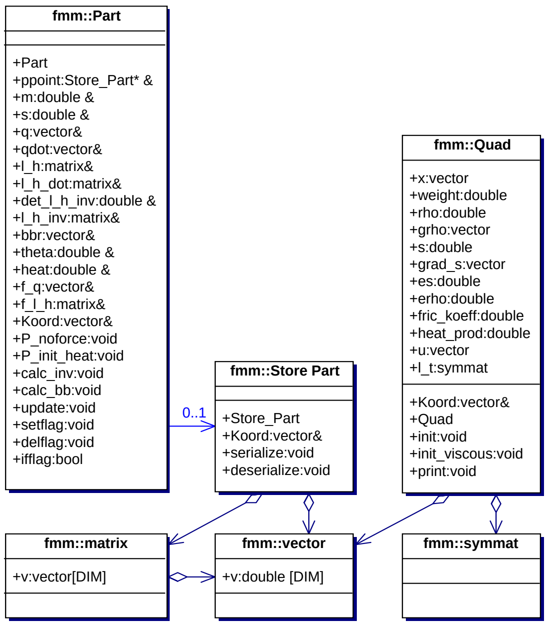


Abbildung 7.4: Zentrale Klassen- und Datenstrukturen bei FMM

7.2 Methode der finiten Massen

7.2.1 Überblick

Eine weitere Teilchensimulation die im Rahmen des Sonderforschungsbereich 382 entwickelt wurde, ist die am Lehrstuhl für Numerische Mathematik in Tübingen entwickelte Methode der finiten Massen (engl. *finite mass method*, kurz FMM) [28]. Es handelt sich im Gegensatz zu sph2000 um eine gitterfreie Methode, die der numerischen Behandlung von Problemen der Kontinuumsmechanik dient und in erster Linie zur Simulation kompressibler Strömungen verwendet wird. Die Methode ist ein Langrange'sches Verfahren und baut auf einer Diskretisierung der Masse auf. Sie unterscheidet sich dabei von anderen Ansätzen, wie z.B. den finiten Elementen oder den finiten Volumen, denen eine Diskretisierung des Raumes zugrunde liegt.

Die Masse wird in kleine Massenpakete zerlegt, von denen jedes einzelne endlich viele innere Freiheitsgrade, wie z.B. Translation, Größe und Rotation, besitzt. Unter dem Einfluss innerer und äußerer Kräfte und den Gesetzen der Thermodynamik gehorchend werden die Massenpakete bewegt. Die Freiheitsgrade passen sich dabei der lokalen Strömungsumgebung an.

Die Tatsache, dass sich die Teilchen überdecken und in komplizierter Weise miteinander in Wechselwirkung stehen, lässt relativ komplexe Programmstrukturen erwarten. Insbesondere scheint es notwendig, die Wechselwirkungspartner der Teilchen zu identifizieren und in geeigneter Form abzuspeichern. Bei genauerer Betrachtung stellt man jeoch fest, dass die Teilchen sich nur indirekt über globale Felder wie die Massendichte ρ oder das Geschwindigkeitsfeld $v = j/\rho$ beeinflussen und man daher ganz anders vorgehen kann.

Die wesentliche Idee besteht darin, die Quadraturpunkte völlig von den Teilchen zu separieren und entsprechende Datenstrukturen zu benutzen. Die erste dieser Datenstrukturen ist den Teilchen zugeordnet und enthält Informationen über deren fixe Masse, Position, Geschwindigkeit, Deformationsmatrix, Zeitableitung und die Kräfte, die auf die Teilchen wirken. Die zweite Datenstruktur steht direkt mit den Quadraturpunkten in Verbindung. Sie enthält zunächst natürlich deren Positionen und Gewichte, in ihr werden aber auch Feldgrößen wie Massendichte, Entropiedichte, Massenflussdichte oder deren Gradienten aufsummiert und abgespeichert. Abbildung 7.4 zeigt das entwickelte Klassendesign der verwendeten Datenstrukturen. Neben den zentralen Klassen für Teilchen (`Part`) und Quadraturpunkte (`Quad`) finden sich speziell für die Methode der finiten Massen optimierte Hilfsklassen für Vektoren, Matrizen und symmetrische Matrizen. Aus Leistungsgründen wurde die Klasse `Store_Part` eingeführt. Diese enthält die auf die wesentlichen Informationen reduzierten Daten eines Teilchens, die abgespeichert werden sollen, und verringert damit die zu speichernde Datenmenge.

Die Berechnung der auf die Teilchen wirkenden Kräfte zerfällt dann in drei Phasen. In der ersten Phase werden Feldgrößen wie die Massendichte in den Quadraturpunkten aufsummiert, d.h., es wird Information von den Teilchen auf die Quadraturpunkte übertragen. In der zweiten Phase wird nur Information manipuliert, die an die einzelnen Quadraturpunkte geheftet ist, z.B. die Ermittlung des Geschwindigkeitsfelds aus der Massendichte und der Massenflussdichte. In der abschließenden dritten Phase werden dann die auf die Teilchen wirkenden Kräfte aus den in den Quadraturpunkten vorhandenen Feldgrößen berechnet, d.h., die Information wird von den Quadraturpunkten zurück auf die Teilchen übertragen.

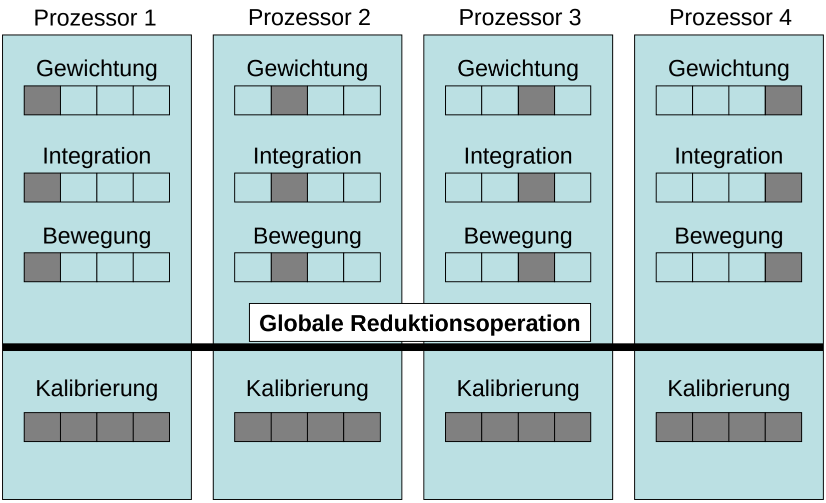


Abbildung 7.5: Ablaufschema des parallelisierten FMM-Algorithmus bei 4 Prozessoren

7.2.2 Parallele Ein-/Ausgabe

Die Implementierung der Anwendung besteht zu einem großen Teil aus prozeduralem C und zu einem kleinen Teil aus C++. Während der Arbeiten von Ritt [77] entstand eine auf TPO++ basierende parallele Anwendung. Dazu wurden die zur Kommunikation notwendigen Datenstrukturen unter Verwendung der bekannten Serialisierungsmechanismen verschickbar gemacht. Kerndatenstruktur ist dabei die Liste `Plist`, die sämtliche Teilchen und deren Informationen enthält.

Der Algorithmus der Parallelisierung orientiert sich dabei an den bereits genannten drei Phasen zuzüglich eines Kommunikationsschritts (siehe Abb. 7.5):

1. Nach Initialisierung und Verteilung der Teilchen durch einen Master-Prozess hält jeder Prozess die Teilchen redundant vor. Nun gewichtet jeder Prozess parallel die Teilchen auf einen Teil der Quadraturpunkte.

2. Jeder Prozessor führt parallel eine numerische Integration seiner lokalen Quadraturpunkte durch.

3. Mit den Ergebnissen der Integration lassen sich die auf die Teilchen wirkenden Kräfte berechnen.

4. Da jeder Prozessor nur einen Teil der Quadraturpunkte zur Bewegung der Teilchen verwendet hat, folgt ein globaler Kommunikationsschritt. Dieser kommuniziert und addiert die Teilinformationen in einem Schritt, so dass jeder Prozessor die aktualisierten Teilchendaten besitzt.

Neben einem sequentiellen Einlesen der Teilchendaten während der Initialisierung erfolgt nach einer frei wählbaren Menge von Integrationsschritten durch den Master-Prozess zusätzlich die sequentielle Sicherung der Teilchendaten auf Festplatte. Durch den Einsatz der parallelen Ein-/Ausgabe sollen diese sequentiellen Anteile vermieden werden, um die Skalierbarkeit der Anwendung deutlich zu verbessern.

Da sämtliche Teilchendaten in einer Datei gespeichert werden, bietet es sich - genau wie bei sph2000 - an, die Ein-/Ausgabe durch kollektive Zugriffe der beteiligten Prozessoren zu parallelisieren. Außer dass die Ein-/Ausgabe dadurch optimiert wird, entfällt damit auch die zu Beginn notwendige Verteilung der Daten durch einen Master-Prozess.

Die bisherige Implementierung der Ein-/Ausgabe findet durch die beiden Methoden `fmm::partread` und `fmm::partwrite` statt. Nachdem die Kommunikation der Teilchen bereits in TPO++ realisiert ist und die abzuspeichernden Daten auf die Teilchendaten begrenzt sind, müssen keine weiteren Klassen serialisiert werden. Die Implementierung der kollektiven Ein-/Ausgabe kann daher direkt in den beiden genannten Methoden vorgenommen werden und gestaltet sich für den Fall der Datenspeicherung folgendermaßen:

```
void fmm::partwrite(Plist& plist, string name)
{
    TPO::File fh;
    TPO::View view(TPO_VIEW_CONTIGUOUS);
    int plist_sz = plist.list_size();

    std::vector<InPart> outplist(plist_sz);

    for(int i=0; i < plist_sz; i++){
        if(plist[i]->ifflag(0)){ outplist[i].flag = 1.0;}
            else { outplist[i].flag = 0.0;}

        outplist[i].q = plist[i]->q();
        outplist[i].qdot = plist[i]->qdot();
        outplist[i].m = plist[i]->m();
        outplist[i].s = plist[i]->s();
        outplist[i].l_h = plist[i]->l_h();
        outplist[i].l_h_dot = plist[i]->l_h_dot();
    }

  fh.open(TPO::CommWorld, name, TPO_MODE_CREATE);
  fh.setView(outplist, view);

  fh.write_all(outplist.begin(), outplist.end());
```

```
25
26    fh.close();
27 }
```

Die Vorgehensweise folgt der Implementierung der kollektiven Ein-/Ausgabe von sph2000. In Zeile 7 wird der zu speichernde Teilchenvektor deklariert und in den Zeilen 9 bis 19 mit den zu speichernden Werten gefüllt. Nach Öffnen der Datei (Zeile 21) und Partitionieren der Daten auf die Prozessoren (Zeile 22) erfolgt in Zeile 24 ein kollektiver Schreibzugriff.

Zu Beginn einer Simulation erfolgt das Einlesen der Anfangsverteilung und die Verteilung der Teilchendaten auf die Prozessoren durch den Master-Prozess. Auch hier wurde die parallele Ein-/Ausgabe implementiert, so dass durch einen kollektiven Lesevorgang zu Beginn der Simulation die Ein-/Ausgabe parallelisiert wurde und die Kommunikation entfällt. Die Implementierung gestaltet sich dabei analog zu dem Schreibvorgang.

7.2.3 Ergebnisse

Im Gegensatz zu sph2000 wurden bei der Integration von TPO-IO die zu speichernden Datenstrukturen von FMM an die objektorientierte Schnittstelle angepasst. Zwar könnten aufgrund der Abwärtskompatibilität auch die vorhandenen C-Strukturen verwendet werden, doch hätte dies zur Folge, dass die Ein-/Ausgabe deutlich komplizierter formuliert werden müsste. Daher wurde die ursprüngliche Datenstruktur einer Liste von Partikeln in einen Container der STL umgewandelt. Im weiteren Verlauf der Implementierung konnte so von den objektorientierten Methoden von TPO-IO profitiert werden. Die Erhöhung der Granularität der C-Strukturen auf ein objektorientiertes Niveau verbessert zusätzlich die Bedienbarkeit und Wartbarkeit der Implementierung. Dies zeigt sich auch in der klaren Struktur des Programms bei kollektiven Zugriffen unter Verwendung einer Sicht, die die Funktionalität von TPO-IO völlig ausschöpfen.

Zur Überprüfung der Effizienz wurde als Beispielanwendung die Dynamik einer Gaskugel über 10.000 Integrationsschritte berechnet. Die Laufzeiten der Versionen mit sequentieller und paralleler Ein-/Ausgabe wurden auf dem Kepler-Cluster für unterschiedliche Prozessoren gemessen und verglichen.

Der Anteil der Ein-/Ausgabe an der gesamten Anwendung ist bei lediglich 400 verwendeten Teilchen und der damit verbundenen abzuspeichernden Datenmenge von 32KB zu gering, um eine wesentliche Leistungssteigerung erwarten zu können. Im Gegensatz zu den Ergebnissen der anderen Anwendungen wurde daher keine Differenzierung zwischen Berechnungs- und Ein-/Ausgabeaufwand vorgenommen.

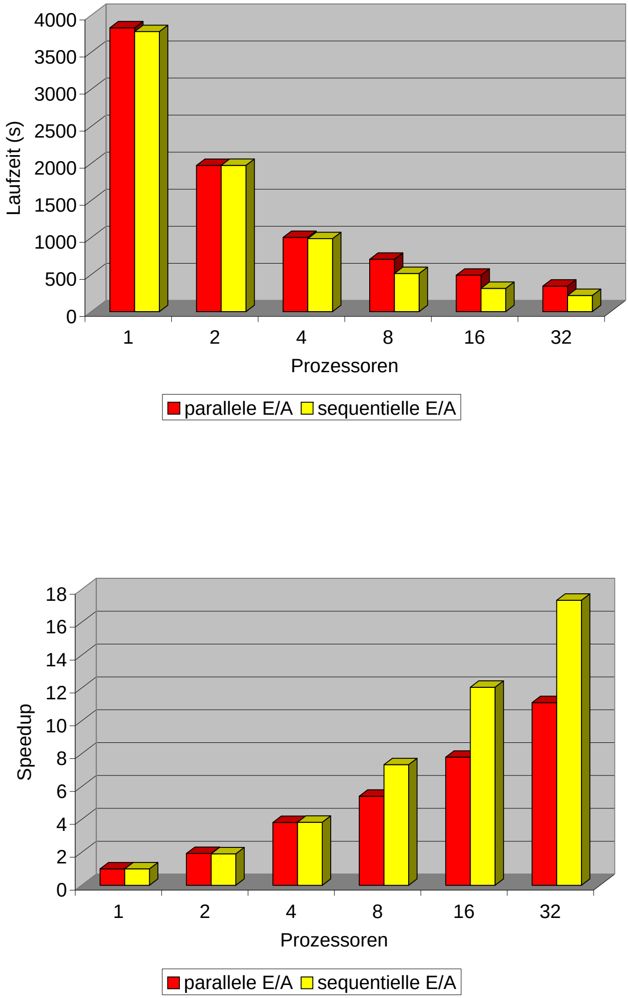


Abbildung 7.6: Vergleich von Laufzeit (oben) und Speedup (unten) der Implementierungen mit sequentieller und paralleler Ein-/Ausgabe bei FMM

In Abbildung 7.6 finden sich sowohl die Messungen der Gesamtzeit als auch die Speedups der beiden Implementierungen. Es ist zu erkennen, dass die Verwendung paralleler Ein-/Ausgabe zu einer leicht schlechteren Leistung führt. Der Zusatzaufwand durch Hinzunahme der Bibliothek kann durch die Parallelisierung nicht kompensiert werden. Dies liegt an dem bereits erwähnten ungünstigen Verhältnis zwischen Berechnungs- und Ein-/Ausgabeaufwand. Mit steigender Anzahl von Prozessoren nimmt die Leistung der parallelisierten Version zudem stetig ab, da sich die zu speichernde Datenmenge pro Prozessor weiter reduziert und der Zusatzaufwand durch die Bibliothek sich mehr und mehr bemerkbar macht. Zukünftige Anwendungen von FMM können daher von der parallelisierten Ein-/Ausgabe nur dann stark profitieren, wenn die Zahl der verwendeten Teilchen deutlich erhöht wird.

Es ist jedoch anzumerken, dass im Gegensatz zu sph2000 der globale Kommunikationsschritt der Teilchen aufgrund der Eigenart des Algorithmus von FMM trotz Verwendung kollektiver Ein-/Ausgabe nicht entfallen kann. Das Optimierungspotential von FMM wird daher auf die Parallelisierung der Ein-/Ausgabeoperationen selbst reduziert. Ein Vergleich der Ergebnisse mit den Messungen von sph2000 ist daher nicht möglich.

7.3 Gensequenzanalyse mit ParSeq

7.3.1 Überblick

Neben den beiden Teilchenmethoden sph2000 und FMM wurde mit ParSeq auch eine Anwendung aus dem Bereich der Bioinformatik auf die Parallelisierbarkeit der Ein-/Ausgabe hin untersucht. Im Gegensatz zu den Teilchenmethoden wurde TPO-IO bereits von Beginn an in die Anwendung integriert. ParSeq [84, 92, 73] entstand im Rahmen des Landesschwerpunktprogramms „Gensequenzanalyse auf Höchstleistungsrechnern“. Die Kooperation von Biologen, Informatikern und Bioinformatikern ermöglichte die Entwicklung eines Werkzeugs, welches vorgegebene Motive mit strukturellen und biochemischen Eigenschaften in DNA oder Proteinsequenzen sucht. Der Algorithmus stellt dabei eine Kombination aus der Suche nach vorgegebenen Strukturen, der Verifikation biochemischer Eigenschaften und einer approximativen Suche dar. Zu diesem Zweck verwendet ParSeq eine auf der Basis von regulären Ausdrücken erweiterte Eingabesprache. Der folgende Ausdruck stellt z.B. die Suche nach drei zusammenhängenden regulären Ausdrücken rex_1, rex_2 und rex_3 unter Berücksichtigung biochemischer Eigenschaften $func_1(a_1, ..., a_n)$ bis $func_k(b_1, ..., b_m)$ für den regulären Ausdruck rex_2 dar. Jede der Funktionen repräsentiert dabei genau eine biochemische Eigenschaft:

$$\texttt{<}rex_1\texttt{>@(<}rex_2\texttt{>)}/func_1(a_1, \ldots, a_n), \cdots, func_k(b_1, \ldots, b_m)\texttt{@<}rex_3\texttt{>}$$

Eine wichtige Funktionalität für die Biologen ist die Möglichkeit, ungenaue Suchen durchführen zu können. Da eine Variation im Strang einer DNA- oder Proteinsequenz an einer bestimmten Stelle in der Biologie sehr häufig auftritt, würden bei einer stringenten Suche diese Treffer nicht gefunden werden, obwohl es sich biologisch gesehen dabei um Treffer handelt. Daher ist die Möglichkeit einer approximativen Suche von hoher Bedeutung. Eine Modifikation der Funktionen zur Überprüfung biochemischer Eigenschaften ermöglicht es, diese Art von Suche als Spezialfall einer Motivsuche mit Einschränkungen zu generieren. Die Variationen der Suche werden mit Hilfe der Funktionen „edit distance" oder „hamming distance" an die Suchanfrage übergeben. Die Eingabeparameter der beiden Funktionen geben die Anzahl der erlaubten Fehler innerhalb eines Treffers an. Eine hohe Variation führt daher zu einer regelrechten Explosion des Ergebnisraumes und muss vom Benutzer mit höchster Sorgfalt eingesetzt werden.

Der Algorithmus zur Motivsuche lässt sich in drei Hauptschritte zerlegen. Als Erstes erfolgt eine Analyse der Eingabe des Benutzers, die eine Trennung von reinen regulären Ausdrücken und biochemischen Eigenschaften vornimmt. Anschließend werden, beginnend mit dem ersten Term von links, die Treffer unter Verwendung der implementierten Bibliothek für reguläre Ausdrücke in der Sequenzdatenbank gesucht. In einem letzten Schritt erfolgt die Überprüfung aller gefundenen Treffer bezüglich der vorgegebenen biochemischen Eigenschaften.

ParSeq unterstützt darüber hinaus eine inkrementelle Suche, wobei die Treffer einer vorhergehenden Suche als Basis einer neuen Suche verwendet werden. Dies ermöglicht dem Benutzer, sich an die gewünschten Sequenzbereiche Schritt für Schritt heranzutasten.

Unter Verwendung der Entwicklungsumgebung *eclipse* in der Version 3.0 wurde die Basisimplementierung von ParSeq in Java enwickelt. Sie bietet die Funktionalität einer lokalen Suche auf lokal gespeicherten Sequenzdaten unter Verwendung einer grafischen Oberfläche und der Berücksichtigung biochemischer Eigenschaften (siehe Abbildung 7.7).

Gerade im Bereich der Molekularbiologie werden durch modernste Methoden immer mehr Gene entschlüsselt. Dies führte in den letzten Jahren zu einer regelrechten Explosion der zur Verfügung stehenden Sequenzdaten. Zur Verwaltung dieser Daten stehen daher mittlerweile mehrere hundert Datenbanken im Internet zur Verfügung. Bekannte Beispiele sind unter anderem GenBank [11], EMBL [8] und DDBJ [89], von denen allein GenBank 4×10^{16} Nukleotidsequenzen beinhaltet.

Durch dieses exponentielle Wachstum steigt auch die Laufzeit für die Suche einer Sequenz in diesen Datenbanken stark an. Aus diesem Grund wurde für ParSeq eine parallele Version der Motivsuche auf Basis der bereits bestehenden sequentiellen Java-Version entwickelt und auf dem Kepler-Cluster implementiert. Durch die Entwicklung einer dreischichtigen Architektur (siehe Abb. 7.8) bestehend aus einem lokalen Klienten, der parallelen Anwendung und einem dazwischen gelegenen Proxy

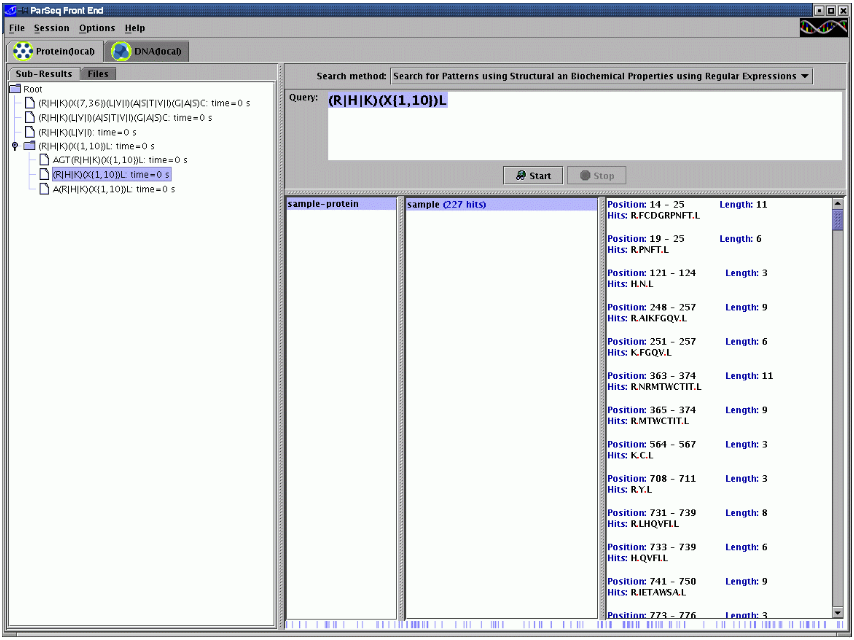


Abbildung 7.7: Die grafische Oberfläche von ParSeq

wird es dem Benutzer ermöglicht, von dem lokalen Rechner aus eine entfernte parallele Suche auf dem Cluster durchzuführen. Die Kommunikation zwischen den einzelnen Schichten erfolgt aus insgesamt drei Gründen über das Protokoll XML-RPC: Erstens ist XML-RPC ein sehr einfach anwendbares Protokoll zur Übertragung der in ParSeq verwendeten Datenstrukturen. Zweitens handelt es sich im Vergleich zu CORBA oder SOAP um ein sehr leichtgewichtiges, plattform- und sprachunabhängiges Protokoll. Drittens bietet es ein Mindestmaß an Sicherheit, da als Transportschicht HTTP verwendet wird, das bereits einfache Mechanismen zur Authentifizierung unterstützt.

7.3.2 Parallele Ein-/Ausgabe

Die Implementierungen von Klient und Proxy wurden in Java, die der parallelen Anwendung aus Leistungsgründen in C++ realisiert. Damit stellt XML-RPC zur Kommunikation zwischen den Schichten gleichzeitig die Schnittstelle zwischen Java und

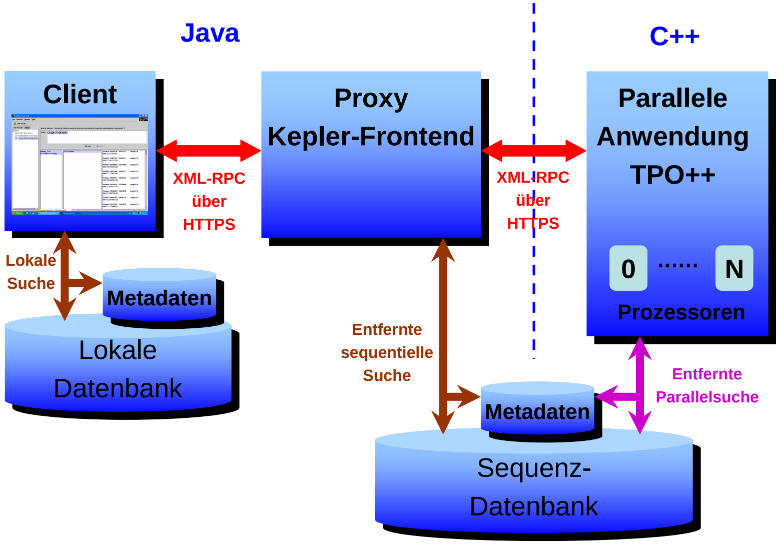


Abbildung 7.8: Die dreischichtige Architektur von ParSeq

C++ dar. Zur internen Kommunikation der parallelen Anwendung wurde die Bibliothek TPO++ eingesetzt.

Der Algorithmus der parallelen Anwendung folgt dem *Master/Worker*-Prinzip. Ein Prozessor (*Master*) dient als Schnittstelle zwischen paralleler Anwendung und Proxy und leitet die vom Klienten über den Proxy ankommenden Suchanfragen an alle beteiligten Prozessoren (*Worker*) der parallelen Anwendung in einem gekapselten `SearchRequest`-Objekt weiter. Dieses beinhaltet neben dem zu suchenden Ausdruck auch eine Liste der zu durchsuchenden Dateien.

Der parallele Algorithmus folgt einem datenparallelen Ansatz: Das vorgegebene Motiv wird von jedem *Worker* parallel auf einem Teil der gesamten Sequenzdaten gesucht. Die gefundenen Treffer werden anschließend an den *Master* geschickt, welcher redundante Treffer entfernt und das Ergebnis über den Proxy an den lokalen Klienten zurückleitet.

Wahl der Ein-/Ausgabestrategie

Aufgrund der stark datenorientierten Ausrichtung der Anwendung und des damit verbundenen hohen Anteils an Ein- und Ausgabeoperationen an der Gesamtrechenzeit wurde die Ein-/Ausgabe mit Hilfe von TPO-IO parallelisiert.

Die aktuelle Implementierung parallelisiert die durch das Einlesen der Sequenzdaten aus den zu durchsuchenden Dateien hervorgerufene Ein-/Ausgabe. Eine Datei kann dabei durchaus mehrere Sequenzen beinhalten, die durch zusätzliche Metainformationen über die Sequenz voneinander getrennt sind. Zusätzlich kann die Länge der einzelnen Sequenzen sehr stark variieren, so dass eine effiziente Strategie zur Lastverteilung der einzelnen Sequenzen auf die beteiligten Prozessoren entwickelt werden muss. Mehrere Möglichkeiten stehen zur Partitionierung der Daten auf unterschiedlichen Ebenen zur Verfügung:

1. *Dateiebene*
   Die Dateien werden der Reihe nach an die Prozessoren verteilt.

2. *Sequenzebene*
   Die Sequenzen aller Dateien werden auf die Prozessoren verteilt.

3. *Globale Partitionierung*
   Die gesamte Länge aller Sequenzen wird gleichmäßig auf alle Prozessoren verteilt.

Es ist klar, dass im ersten und zweiten Fall aufgrund unterschiedlicher Größen sowohl der Dateien als auch der einzelnen Sequenzen eine grobe Granularität vorliegen kann, die zu einer sehr unausgeglichenen Lastverteilung zwischen den Prozessoren führt. Daher fiel die Wahl auf die Implementierung der dritten Strategie. Die Länge aller zu durchsuchenden Dateien wird dabei ermittelt und diese werden auf mehrere Prozessoren in gleich großen Teilen verteilt. Sollten dabei einzelne Sequenzen geteilt werden müssen, wird an der Schnittstelle ein Überlappen der Sequenzen durch Einlesen zusätzlicher Daten generiert, um die Vollständigkeit der gefundenen Treffer der Motivsuche sicherzustellen. Neben der besten Aufteilung der Sequenzen hat diese Vorgehensweise darüber hinaus den Vorteil, dass die Zahl der Ein-/Ausgabeoperationen minimiert wird.

Implementierung

Der Einlesevorgang erfolgt kollektiv, um eine bestmögliche Effizienz der parallelen Ein-/Ausgabe zu erreichen. In einem abschließenden Prozess erfolgt die Extraktion der einzelnen Sequenzen aus dem gelesenen Puffer in einzelne Sequenzobjekte. Der folgende Quelltext zeigt die Implementierung der eben beschriebenen Funktion `readSequenceFile`:

```
vector<Sequence*> Sequence::readSequenceFile( string path ) {

  TPO::File fh;
  TPO::View view(TPO_VIEW_CONTIGUOUS);
  std :: vector<char> buf;
```

```
6
7     fh.open(TPO::CommWorld, path, TPO_MODE_RDONLY);
8
9     buf.resize((int)(fh.getSize / TPO::CommWorld.size()) + 1);
10
11    // partition file
12    fh.setView(buf, view);
13
14    // collective read
15    fh.read_all(buf.begin(), buf.end());
16
17    fh.close();
18
19    // return extracted vector of sequences
20    return extract(buf);
21 }
```

Der Quelltext macht auch in diesem Beispiel die Vorteile von TPO-IO deutlich: Eine transparente Darstellung der Zugriffe und der Partitionierung und insgesamt wenig Programmzeilen führen zu einem übersichtlichen und produktiven Quelltext. Der Ablauf der Funktion folgt daher analog der Vorgehensweise der bereits vorgestellen Anwendungen.

Die Sequenzdaten liegen in dem weit verbreiteten Standardformat FASTA vor, einem ASCII-Textformat, das neben den reinen Sequenzdaten noch zusätzliche Metainformationen über die Sequenz beinhaltet. Aus diesem Grund erfolgt der Zugriff auf die Daten mit einem reinen Bytevektor (Zeile 5). Die Hilfsfunktion `extract()` in Zeile 20 bereitet die Daten dann auf und generiert daraus einen Vektor mit entsprechenden `Sequence`-Objekten.

Der Vorteil einer objektorientierten Bibliothek kann daher in diesem Beispiel nicht voll ausgespielt werden, da die zu lesenden Datenstrukturen trivialer Natur sind und somit auch von MPI gelesen werden könnten. Andererseits beweist diese Anwendung jedoch, dass die Bibliothek eben auch bei diesen Datenstrukturen einwandfrei funktioniert. Durch Konvertieren der Sequenzdaten in ein binäres, von TPO-IO generiertes Dateiformat könnte die Leistung der Ein-/Ausgabe sicherlich deutlich gesteigert werden. Ob der Aufwand jedoch lohnt, ist abzuwägen, da sämtliche Datenbanken das FASTA-Format verwenden und jede Aktualisierung eine Neukonvertierung nach sich ziehen würde.

Ein deutlicher Vorteil gegenüber MPI findet sich jedoch in Zeile 9: Die Größe des zu lesenden und damit zu verarbeitenden Puffers kann jeder Prozessor individuell festlegen. In MPI jedoch ist dies bei kollektiven Zugriffen absolut verboten. Durch eine optimale Anpassung der Größe könnten mit TPO-IO z.B. Leistungsunterschiede einzelner Prozessoren in heterogenen Architekturen ausgeglichen werden.

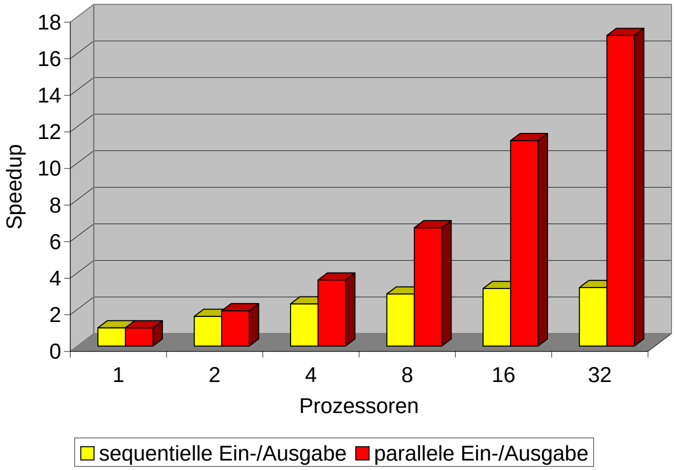


Abbildung 7.9: Speedupvergleich von sequentieller und paralleler Ein-/Ausgabe bei ParSeq

7.3.3 Ergebnisse

Da TPO-IO bereits in der Entwicklungphase eingesetzt wurde, konnte die Klassenstruktur von ParSeq in idealer Weise an die Anforderungen der parallelen Ein-/-Ausgabe angepasst werden. Die zu speichernden Strukturen wurden in eigenen Klassen unter Verwendung von Datenstrukturen der STL gekapselt und können damit direkt von TPO-IO verarbeitet werden. Der Einsatz einer prozeduralen Schnittstelle wäre dann nur durch Aufbrechen der Objekte in einfachere Strukturen und einen damit verbundenen erheblichen Mehraufwand realisierbar. Neben erhöhten Entwicklungskosten wären die Folgen eine schlechtere Bedienbarkeit und Wartbarkeit der Anwendung. Durch Verwendung von TPO-IO hingegen konnte der Implementierungsaufwand auf ein Minimum reduziert und gleichzeitig das Potential von TPO-IO in Bezug auf Bedienbarkeit, Transparenz, Wartbarkeit und Funktionalität maximiert werden.

Die Leistungsmessungen der Anwendung wurden auf dem Kepler-Cluster unter folgendem Szenario durchgeführt: Gesucht wurde das Motiv $CC(A|T)\{6\}GG$, ein

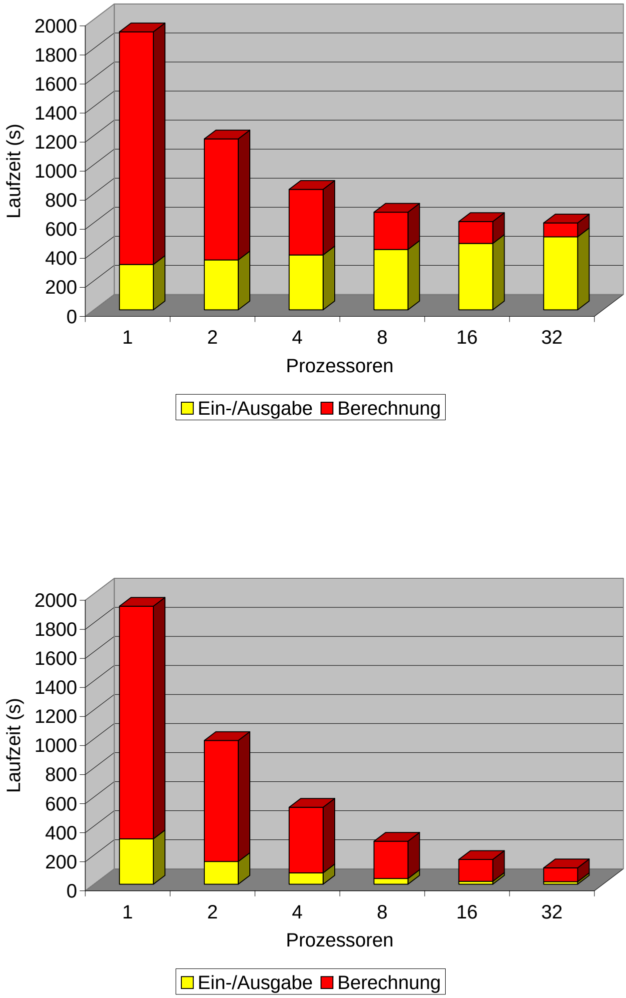


Abbildung 7.10: Laufzeitanteile für Ein-/Ausgabe und Berechnungen bei ParSeq unter Verwendung sequentieller (oben) und paralleler Ein-/Ausgabe (unten).

Übereinstimmungsmotiv des *Serum Response Element (SRE)*, auch bekannt als *CArG*-Box. Die Suche wurde auf dem kompletten menschlichen Genom durchgeführt. Dieses besteht aus insgesamt 24 Sequenzen mit einer Gesamtgröße von 3121 MB. Die Sequenzen liegen in dem parallelen Dateisystem PVFS und sind damit für alle Prozessoren erreichbar. Die Messungen wurden auf dem neueren Teil des Kepler-Clusters mit 1, 2, 4, 8, 16 und 32 AMD-Prozessoren durchgeführt. Der Proxy wurde auf einem der Frontend-Rechner gestartet und empfing von einem entfernten Rechner die Suchanfrage.

Die Abbildungen 7.9 und 7.10 zeigen die gemessenen Speedups und Laufzeiten des Testszenarios. Um das Skalierungsverhalten genau ermitteln zu können, enthalten die Messungen ausschließlich die Laufzeiten der parallelen Anwendung und keine zusätzlichen Kommunikationszeiten zwischen anderen Schichten der Anwendung. Es ist anzumerken, dass die Ergebnisse der sequentiellen Ein-/Ausgabe auf theoretischen Berechnungen basieren, die von einer Verteilung der Sequenzdaten durch den *Master*-Prozess an die anderen Prozessoren über das vorhandene Myrinet-Netzwerk mit 115 MB/s ausgehen. Zusätzlich erhöht sich der Zeitaufwand für die Ein-/Ausgabe des Master-Prozesses, welcher die gesamten Sequenzen mit maximal 11 MB/s einlesen könnte. Mit zunehmender Anzahl der Prozessoren ergibt sich daher ein höherer Kommunikationsaufwand während der Ein-/Ausgabe, der die Skalierung der Anwendung stark beeinträchtigt und zu einem maximalen Speedup von 5 bei 32 Prozessoren führt. Im Gegensatz dazu kann man der Abbildung 7.9 deutlich entnehmen, dass die Parallelisierung der Ein-/Ausgabe sehr gut skaliert und bei 32 Prozessoren einen Speedup von 17 und damit eine relativ hohe Effizienz von ca. 50% erreicht. Aufgrund des datenparallelen Ansatzes und der effizienten Ein-/Ausgabe eignet sich daher ParSeq in idealer Weise für den Einsatz von TPO-IO.

7.4 Synthetische Leistungsmessungen

Beim realen Einsatz der objektorientierten Schnittstelle in Anwendungen ist der durch sie entstehende Zusatzaufwand nur schwer aus den Messdaten zu extrahieren ist. Neben anwendungsspezifischen Parametern, wie z.B. dem Verhältnis von Berechnungsaufwand zu Ein-/Ausgabeaufwand oder dem Parallelisierungsalgorithmus, gehen dabei auch architekturspezifische Parameter, wie z.B. der verwendete C++-Compiler oder die Leistung der Hardware, in die Messung mit ein. Mit Hilfe von synthetischen Leistungsmessungen wird nun versucht, die Auswirkungen der anwendungsspezifischen Parameter zu minimieren und zu kontrollieren. Auf die architekturspezifischen Parameter kann hingegen in der Regel kein Einfluss genommen werden.

Die Portabilität der Schnittstelle ist durch den Umstand gewährleistet, dass auf die weit verbreitete und standardisierte Bibliothek MPI-IO aufgesetzt wird. Diese bildet demnach auch gleichzeitig den Vergleichsindex der Leistungsmessungen. Ein Ziel bei der Entwicklung war es, sich dieser Leistung bestmöglich anzunähern. Um die Portabilität der Schnittstelle zu demonstrieren, wurden die synthetischen Leistungsmessungen auf unterschiedlichen Rechnerarchitekturen durchgeführt. Zum einen auf den zwei Cluster-Architekturen *Kepler-Cluster* und *Cray-Opteron-Cluster*, bei denen es sich um Rechner mit verteiltem Hauptspeicher handelt, zum anderen wurden als aktuelle hybride Architekturen die beiden 8-fach-SMP (engl. symmetric multiprocessing) Höchstleistungsarchitekturen Hitachi *SR 8000-F1* und *NEC SX-6* in die Messungen mit aufgenommen.

Damit die Ergebnisse untereinander vergleichbar sind, wurden dieselben Messanwendungen auf allen Architekturen durchgeführt. Dabei handelt es sich sowohl um einfache als auch um kollektive Lese- bzw. Schreibtests. Erstere zeichnen sich dadurch aus, dass ein Prozessor sowohl Lese- als auch Schreibvorgänge in eine eigene Datei durchführt. Bei einem kollektiven Zugriff hingegen werden von einer festen Anzahl mehrerer Prozessoren Daten in dieselbe Datei transferiert. Durch die Kombinationsmöglichkeiten ergeben sich somit insgesamt vier unterschiedliche Messanwendungen.

Die einfachen Lese- und Schreibtests sollen dabei den direkten Zusatzaufwand durch die Schnittstelle zeigen, ohne dass weitere interne Berechnungen berücksichtigt werden müssen, die im Falle kollektiver Zugriffe notwendig sind. Hierbei wurde im Falle von TPO-IO ein einfacher Vektor mit dem Standard-Datentyp `char` und bei MPI-IO ein Array derselben Länge mit der in MPI intern verwendeten Datenstruktur `MPI_BYTE` gefüllt. Die Messung komplexerer Datenstrukturen wie Objekte ist zu Vergleichszwecken nicht möglich, da die Umsetzung auf Datenstrukturen mit MPI-IO entweder nicht möglich oder so umfangreich wäre, dass die Messergebnisse nicht miteinander vergleichbar wären.

Das Ziel der kollektiven Tests hingegen ist es, den durch die Schnittstelle insgesamt hervorgerufenen Zusatzaufwand zu bestimmen. Dazu wurde in TPO-IO ein Vektor von Integerwerten und in MPI-IO ein Array von `MPI_INT`-Werten verwendet. Anschließend wurde die Sicht auf die Daten so gewählt, dass jeder Prozessor der Reihe nach einen Integerwert nach dem anderen zugewiesen bekommt (round-robin-Algorithmus), bis alle Daten verteilt sind. Dieser Test misst also neben dem Zusatzaufwand für das Lesen und Schreiben von Metainformationen auch den Aufwand, der von der Schnittstelle durch automatisch vorgenommene Berechnungen der Offsets in einer Datei, das automatische Setzen einer Sicht auf die Daten und die dazugehörige Kommunikation zwischen den Prozessoren generiert wird.

Die erzielte Transferrate ergibt sich dann aus dem Quotienten der übertragenen Datenmenge und der dafür benötigten Zeit. Zu beachten ist, dass abhängig von der Architektur entweder die aggregierte Transferrate oder die des einzelnen Rechenknotens gemessen wurde. Liegt der zu erwartende Flaschenhals im Bereich des Dateisystems, wurde die aggregierte Leistung gemessen. Ist jedoch der einzelne Rechenknoten der Engpass wurden die einzelnen Transferraten verglichen. Die zu übertragende

Datenmenge pro Prozessor wurde dabei in Zweierpotenzen festgelegt, beginnend mit der Übertragung von 1 Byte. Die Obergrenze wurde so gewählt, dass der Sättigungsbereich der Architektur vollständig erreicht wird. In den gemessenen Fällen war dies spätestens bei einer Datenmenge von 512 MB pro Prozessor gegeben.

Alle Ergebnisse der Transferraten sind im doppelt logarithmischen Maßstab angegeben, um den hohen Wertebereich der Messungen anschaulich darstellen zu können. Zusätzlich zu den Messergebnissen ist immer das von der Architektur vorgegebene theoretische Maximum als Obergrenze angegeben, das sich aus einer Mischung von empirischen Befunden und maximaler Hardwareleistung zusammensetzt. Damit lassen sich die Messungen in gute Relation zueinander setzen und die absolute Leistung der Implementierungen besser beurteilen.

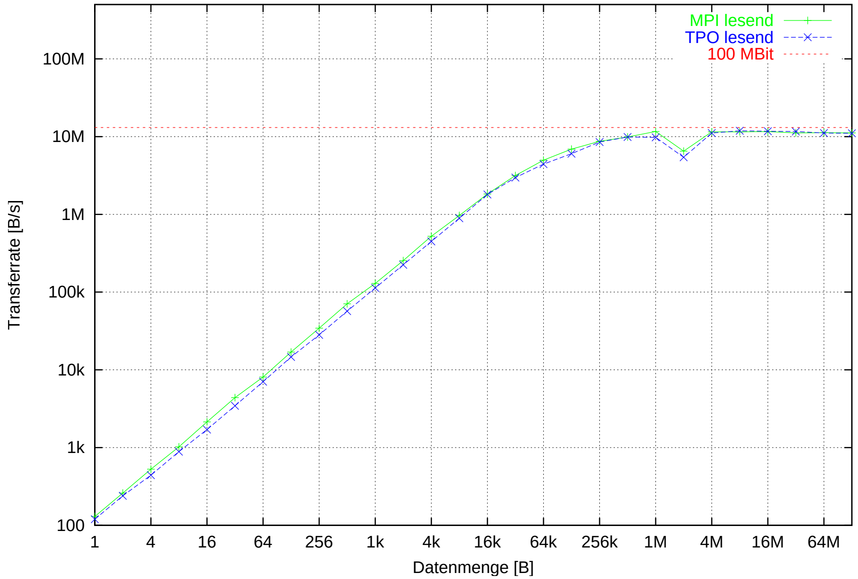
7.4.1 Kepler-Cluster

Die Messungen auf dem Kepler-Cluster wurden durch Portierung von TPO-IO auf die vorhandene MPICH-Implementierung der Version 1.2.4 in Kombination mit der ROMIO-Implementierung 1.2.5.1 durchgeführt [72]. Der gemessene Datentransfer erfolgte dabei von den einzelnen Prozessoren in das auf den 32 AMD-Knoten installierte parallele Dateisystem PVFS der Version 1.5.8.

Die theoretische Gesamtleistung des Systems ergibt sich aus den 32 Knoten mit Festplatte, die über Fast-Ethernet eine aggregierte Bandbreite von theoretisch 400 MB/s erreichen könnten. Die zur Vernetzung der Knoten verwendeten Switche begrenzen diese jedoch auf empirisch ermittelte 180 MB/s. Sowohl bei den einfachen als auch bei den kollektiven Testläufen mit 16 Prozessoren bildet die Fast-Ethernet-Karte mit einer Bandbreite von 100 MBit/s den limitierenden Faktor. Aus diesem Grund ist in den folgenden Ergebnisgrafiken diese als Systemgrenze angegeben.

Abbildung 7.11 zeigt die Messungen der nicht kollektiven Ein-/Ausgabe eines einzelnen Prozessors sowohl für Lese- als auch für Schreibvorgänge. Betrachtet man die Leseleistung, so stellt man eine nahezu identische Leistung der beiden Bibliotheken fest. Dies liegt daran, dass einerseits der Mehraufwand durch die Schnittstelle bei einfachen Datentypen sehr gering ist, und andererseits die zur Wiederherstellung der Objekte benötigten Metadaten noch im Cache verfügbar sind. Ab einer Datenmenge von 512 KB konvergieren beide Implementierungen gegen die Systemgrenze von 100 MBit/s. Im Vergleich dazu zeigt die Schreibleistung einen geringfügigen Leistungsunterschied, der durch den zusätzlichen Schreibaufwand für die Metadaten hervorgerufen wird. Aufgrund des insgesamt sehr schnellen Dateisystems fällt dieser Unterschied jedoch sehr gering aus. Darüber hinaus lässt sich feststellen, dass bereits ab einer Datenmenge von 64 KB die Implementierungen gegen das Hardwarelimit konvergieren.

Die Messungen zur kollektiven Lese- und Schreibleistung entstanden mit 16 Knoten unter Verwendung von jeweils einem Prozessor und die Ergebnisse sind in Ab-



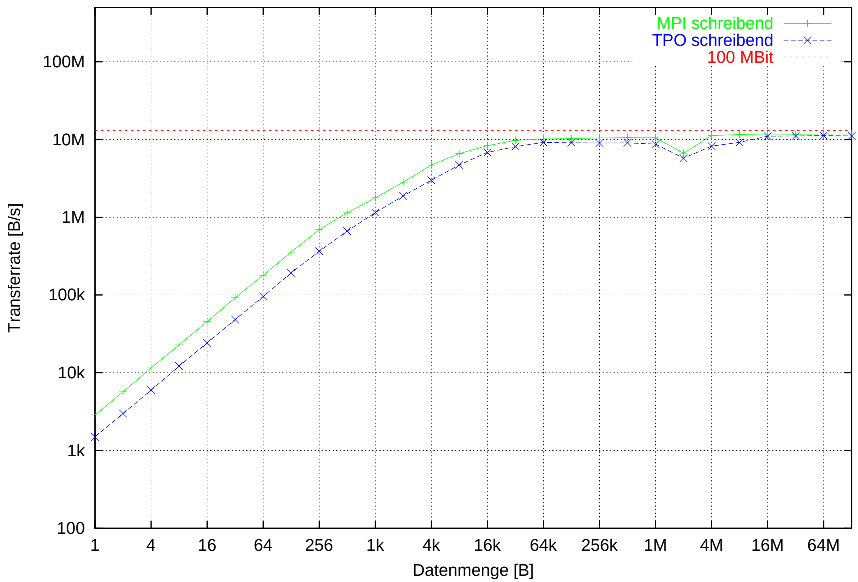


Abbildung 7.11: Vergleich der einfachen E/A-Leistung von TPO-IO und MPI auf dem Kepler-Cluster. Obere Grafik: Leseleistung. Untere Grafik: Schreibleistung.

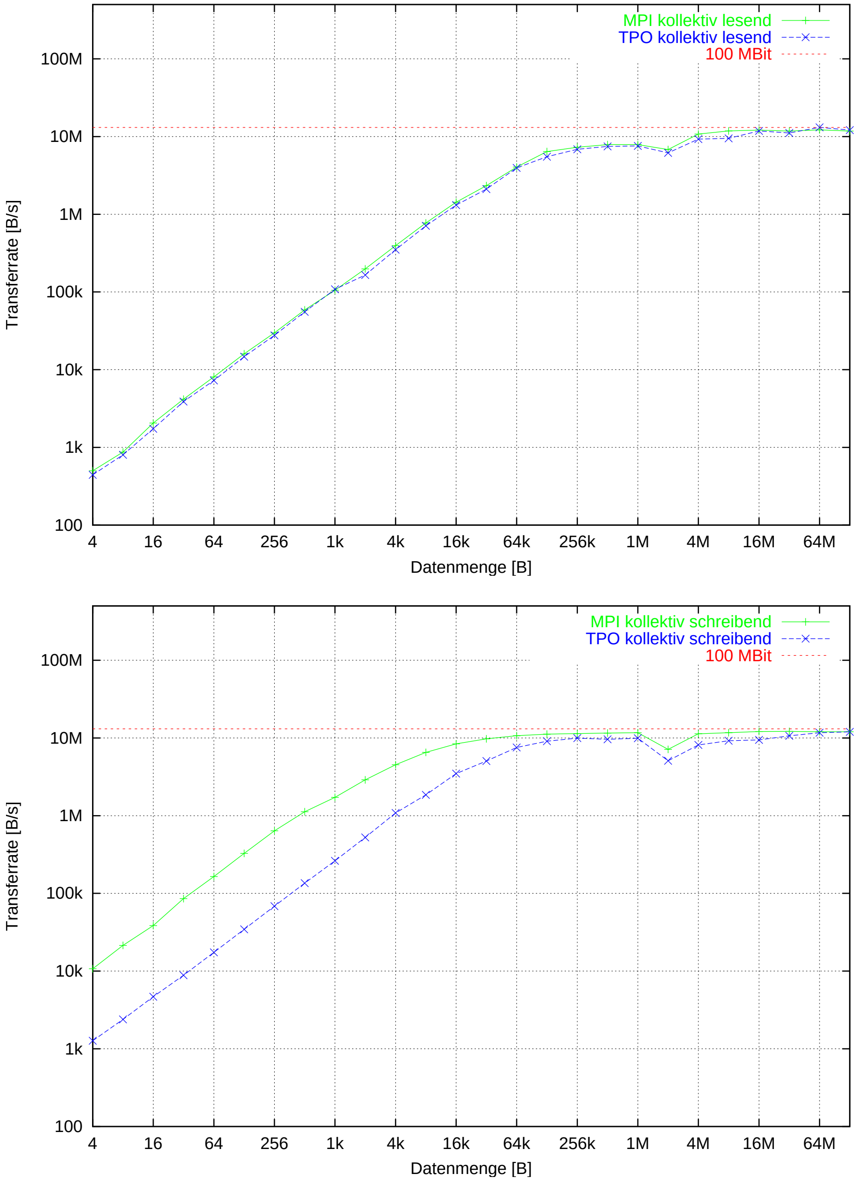


Abbildung 7.12: Vergleich der kollektiven E/A-Leistung von TPO-IO und MPI auf dem Kepler-Cluster. Obere Grafik: Leseleistung. Untere Grafik: Schreibleistung.

bildung 7.12 dargestellt. Die durchschnittliche Leseleistung pro Knoten erreicht auch hier aus den bereits oben angegebenen Gründen bei beiden Implementierungen identische Werte. Die durchschnittliche Schreibleistung zeigt bei kleinen Datenmengen jedoch einen enormen Unterschied. Dies ist einerseits auf das im System verwendete Fast-Ethernet-Netzwerk zurückzuführen, das mit einer Bandbreite von 100 MBit/s gerade bei den hier verwendeten kollektiven Zugriffen den Mehraufwand von TPO-IO deutlich sichtbar macht, andererseits auch auf die mit 1.5 GFlop/s sehr geringe CPU-Leistung. Interne Berechnungen, z.B. der Offsets, oder die Generierung der Sicht wirken sich daher stärker negativ auf die Gesamtleistung aus als bei schnelleren Prozessoren.

Der in allen vier Fällen auftretende Leistungsrückgang der Implementierungen bei einer Datenmenge von 2 MB pro Prozessor ist systembedingt und wird durch Umschaltung des Übertragungsprotokolls der verwendeten Netzwerkkarte hervorgerufen.

7.4.2 Cray-Opteron-Cluster

Die grundlegende Architektur des Opteron-Clusters von Cray ist der des Kepler-Clusters sehr ähnlich. Die linuxbasierten Knoten kommunizieren ebenfalls über Myrinet. Bei der Ein-/Ausgabearchitektur unterscheiden sich jedoch beide Systeme deutlich. Während der Kepler-Cluster über ein paralleles Dateisystem mit 32 Knoten verfügt, müssen sich die insgesamt 125 Knoten des Opteron-Clusters 2 Knoten für die Ein-/Ausgabe teilen. Hinzu kommt, dass beide Knoten über jeweils ein eigenes, voneinander getrenntes Dateisystem verfügen und somit die Daten der Messanwendungen nur auf einen dieser Knoten verteilt werden können. Der Datentransfer erfolgt dabei nicht über das schnelle Myrinet, sondern über das etwas langsamere GigaBit-Ethernet. Dadurch ergibt sich ein Flaschenhals auf Seiten der Ein-/Ausgabeknoten, der eine theoretische aggregierte Gesamtleistung von lediglich 1 GBit/s ermöglicht. Diese Systemgrenze ist in den folgenden Ergebnisgrafiken zur Erleichterung der Interpretation zusätzlich eingezeichnet.

In Abbildung 7.13 sind die Messergebnisse der einfachen Ein-/Ausgabe eines einzelnen Prozessors sowohl für Lese- als auch für Schreibvorgänge abgetragen. Sie zeigen den bereits bei Kepler dargestellten Verlauf und lassen sich durch die dort genannten Gründe erklären. Der leichte Rückgang der Schreibleistung bei 512 KB ergibt sich durch die vorhandene MPI-IO-Implementierung, deren Schreibpuffer eine Größe von 512 KB hat. Eine Überschreitung dieser Grenze führt demnach zu einem zusätzlichen Aufwand und einer damit verbundenen schlechteren Transferrate. Insgesamt konvergieren auch bei dieser Architektur ab einer Datenmenge von 512 KB beide Implementierungen gegen die Systemgrenze von 1 GBit/s.

Zur Messung der Ergebnisse der kollektiven Lese- und Schreibleistung wurden 32 Knoten mit jeweils einem Prozessor verwendet. Abbildung 7.14 stellt die gewonne-

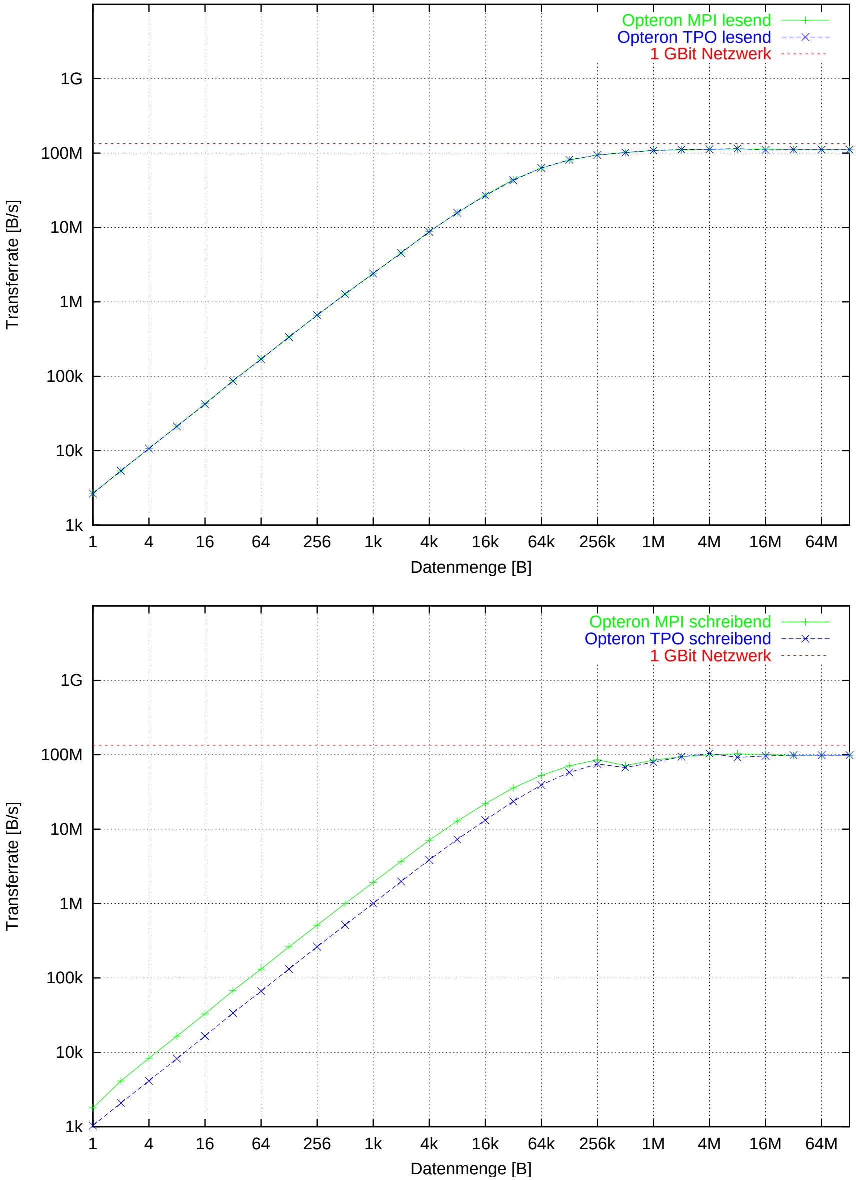


Abbildung 7.13: Vergleich der einfachen E/A-Leistung von TPO-IO und MPI auf dem Cray-Cluster. Obere Grafik: Leseleistung. Untere Grafik: Schreibleistung.

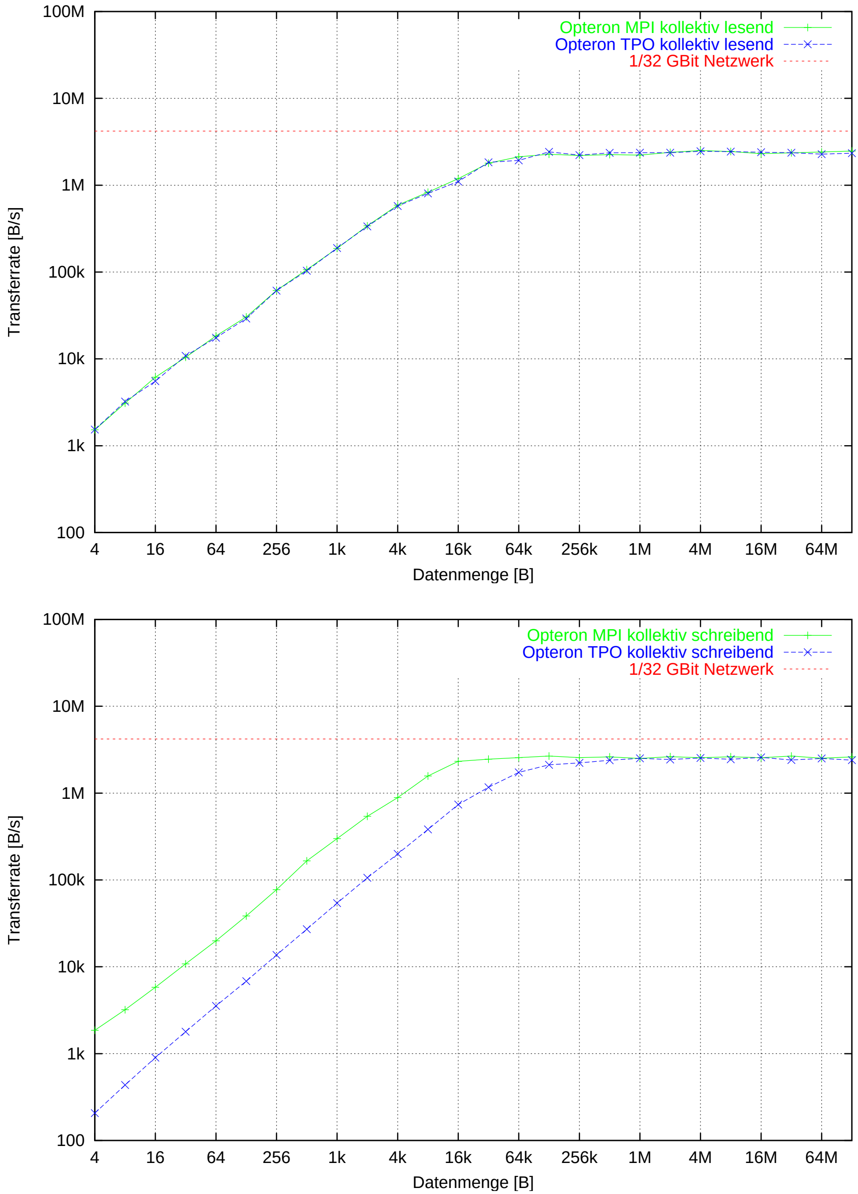


Abbildung 7.14: Vergleich der kollektiven E/A-Leistung von TPO-IO und MPI auf dem Cray-Cluster. Obere Grafik: Leseleistung. Untere Grafik: Schreibleistung.

nen Resultate grafisch dar. Im Gegensatz zum Kepler-Cluster handelt es sich hierbei um die durchschnittliche Leistung eines einzelnen Prozessors. Aus diesem Grund liegt das Systemlimit in der Grafik nicht bei 1 GBit/s sonder bei $1/32$ GBit/s. Der etwas unruhige Verlauf der Ergebnisse resultiert aus dem deutlich überlasteten Ein-/Ausgabeknoten. Die durchschnittliche Leseleistung pro Knoten erreicht auch hier aus den bereits oben angegebenen Gründen bei beiden Implementierungen identische Werte. Die durchschnittliche Schreibleistung von TPO-IO kommt bei dieser Architektur jedoch etwas besser an MPI-IO heran als bei Kepler, da zur internen Berechnung nun AMD Athlons mit 4 GFlop/s und für die Kommunikation ein GBit-Ethernet zum Einsatz kommen.

7.4.3 NEC SX-6

Mit der NEC SX-6 wurde auch eine Vektorarchitektur in die Leistungsmessungen mit aufgenommen. Übersetzt wurde die TPO-IO-Implementierung mit dem Gnu-C++-Compiler der Version 2.96 unter Verwendung der speziell für diese Architektur entwickelten MPI/SX-Bibliothek. Diese basiert auf der bekannten MPICH-Implementierung und bindet die Ein-/Ausgabe über ROMIO der Version 1.0.2 ein. Insgesamt stehen 4 Dateisysteme zur Verfügung, die jeweils einzeln angesteuert werden können. Das globale Dateisystem GFS verteilt die Daten dabei auf die vorhandenen Festplatten.

Die theoretische Gesamtleistung des Systems ergibt sich aus dem in jedem Knoten vorhandenen Fibre-Channel-System. Dieses besteht aus 4 Kanälen mit einer Bandbreite von jeweils 2 GBit/s und ergibt somit eine maximale Transferrate von insgesamt 1 GB/s, welche in den folgenden Ergebnisgrafiken als Systemlimit eingezeichnet wurde.

Die Messergebnisse der nicht-kollektiven Ein-/Ausgabe eines einzelnen Prozessors sowohl für Lese- als auch Schreibvorgänge finden sich in Abbildung 7.15. Sowohl die Lese- als auch die Schreibleistung zeigen hier ein ähnliches Verhalten wie bei den beiden vorhergehenden Architekturen, das wie dort bereits angegeben begründet werden kann. Die insgesamt höhere Ein-/Ausgabeleistung des Systems führt jedoch dazu, dass erst ab einer Dateigröße von 1 MB eine Sättigung erreicht wird.

Abbildung 7.16 stellt die Ergebnisse der kollektiven Leistungsmessungen dar. Aufgrund der insgesamt nur 6 vorhandenen Knoten wurden die kollektiven Messungen auf nur 2 Knoten und unter Verwendung aller 8 Prozessoren eines Knotens durchgeführt. Die Abbildung zeigt die aggregierte Bandbreite der verwendeten 16 Prozessoren. Im Unterschied zu den anderen Architekturen ist hingegen bei der kollektiven Schreibleistung nur eine geringe Differenz zwischen MPI-IO und TPO-IO zu erkennen. Dies resultiert außer aus der hohen Leistung einer einzelnen CPU auch aus dem sehr schnellen Verbindungsnetzwerk der SX-6. Der Zusatzaufwand der internen Berechnungen fällt daher nur sehr gering aus.

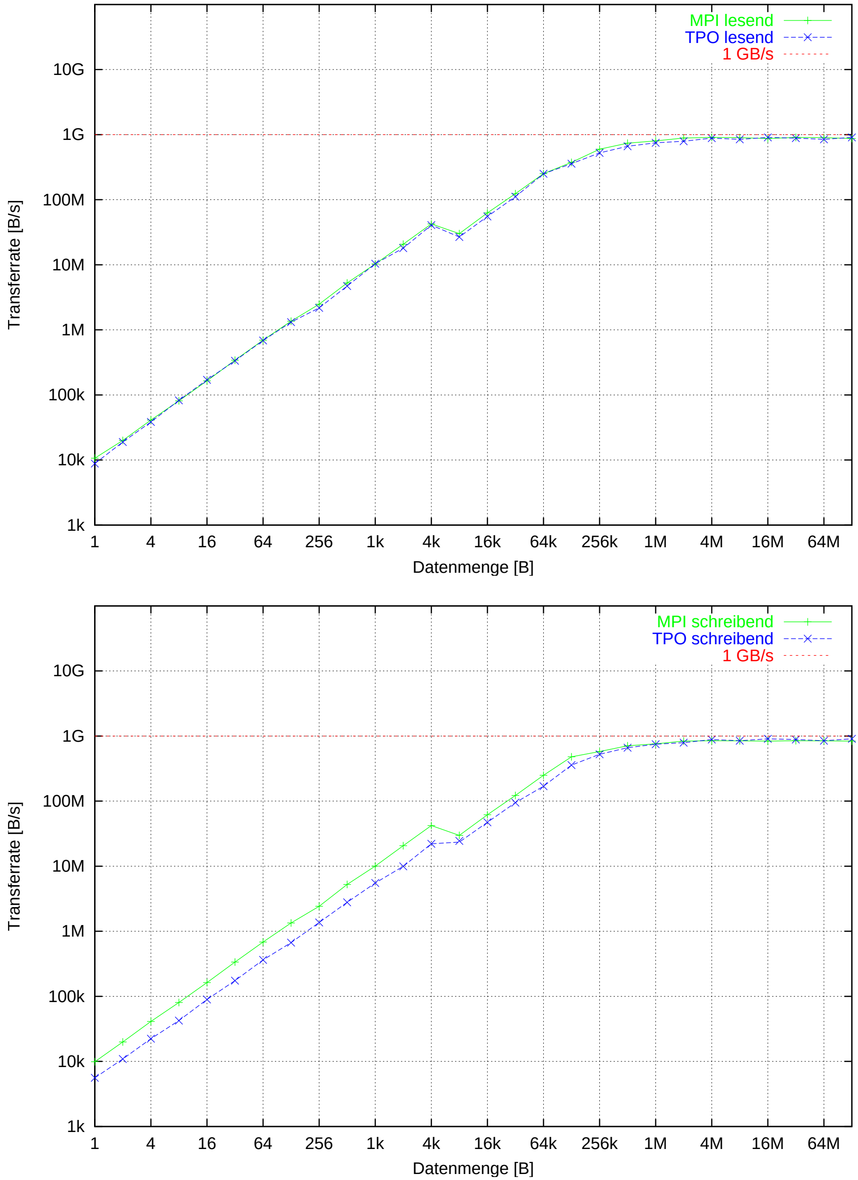


Abbildung 7.15: Vergleich der einfachen E/A-Leistung der NEC SX-6. Obere Grafik: Leseleistung. Untere Grafik: Schreibleistung.

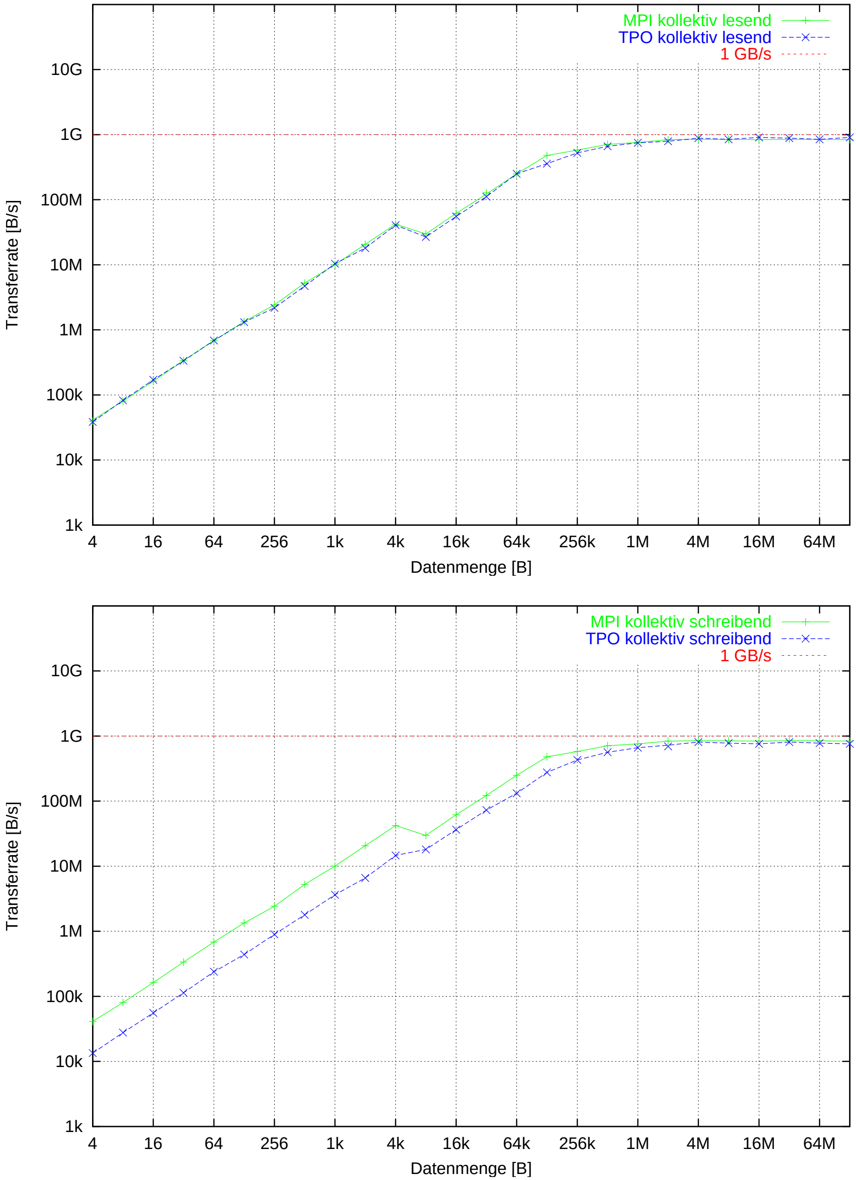


Abbildung 7.16: Vergleich der kollektiven E/A-Leistung von TPO-IO und MPI auf der NEC SX-6. Obere Grafik: Leseleistung. Untere Grafik: Schreibleistung.

Der in allen vier Fällen auftretende Leistungsrückgang der Implementierungen bei einer Datenmenge von 8 KB ergibt sich aus der Verwendung des globalen Dateisystems GFS. Dieses setzt auf dem bekannten Dateisystem NFS auf, dessen Übertragungsprotokoll bei einer Datenmenge von 8 KB auf einen anderen Mechanismus umschaltet.

7.4.4 Hitachi SR 8000

Neben der NEC SX-6 wurde mit der Hitachi eine weitere 8-fach-SMP-Architektur in die Leistungsmessungen mit einbezogen. Im Gegensatz zur SX-6 erfolgen bei dieser Architektur jedoch Ein-/Ausgabe und Kommunikation der Knoten über dasselbe Netzwerk. Ein globales Dateisystem partitioniert die Daten dabei auf insgesamt 8 Knoten. Die Kommunikationsleistung eines Knotens liegt bei 20 MB/s und ergibt daher für die Ein-/Ausgabe eine theoretische Leistung von 160 MB/s. Nachdem die Bisektions-Bandbreite des Kreusschienenverteilers um den Faktor 30 schneller ist als die Bandbreite zum Dateisystem, stellt das Dateisystem den Flaschenhals der Architektur dar. Die nun folgenden Ergebnisse der Leistungsmessungen wurden daher mit der aggregierten Bandbreite des Dateisystems verglichen.

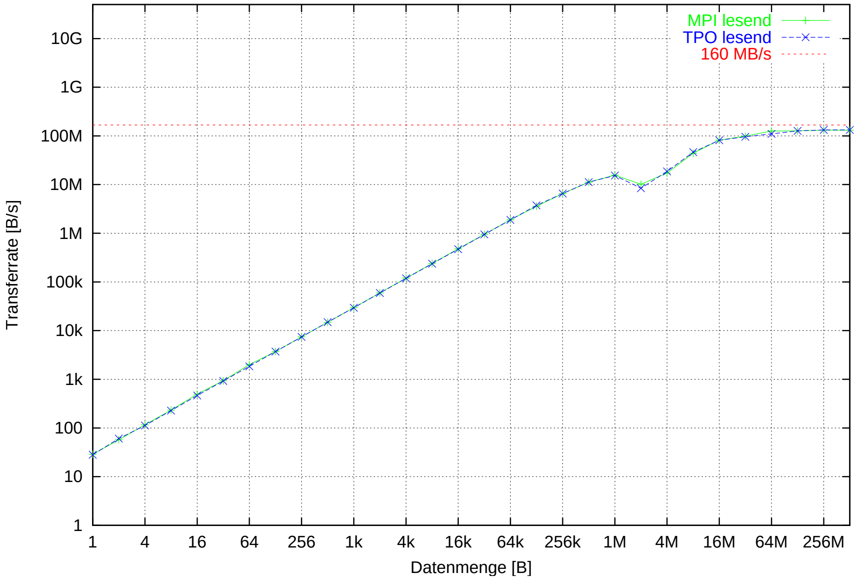
Betrachtet man die Ergebnisse des einfachen Tests in Abbildung 7.17 stellt man einen deutlichen Leistungsrückgang bei einer Datenmenge von 2 MB fest. Dieser ist bedingt durch den Datenpuffer des Dateisystems, der eine Größe von 2 MB besitzt. Die Bandbreite konvergiert ab einer Datenmenge von 1 MB gegen das Systemlimit.

Die Messergebnisse des kollektiven Tests sind in Abbildung 7.18 dargestellt und zeigen die aggregierte Gesamtleistung beider Implementierungen. Die kollektiven Messungen erfolgten unter Verwendung von 32 Knoten mit jeweils einem Prozessor. Der bereits aus den Ergebnissen des einfachen Tests bekannte Leistungsrückgang ist hier bei 64 KB gegeben, da sich im kollektiven Fall die an das Dateisystem insgesamt übertragene Datenmenge auf $64 * 32 = 2048$ KB beläuft.

Nachdem die Hitachi sowohl leistungsschwächere Prozessoren als auch ein langsameres Dateisystem besitzt als die NEC SX-6, fallen die Unterschiede zwischen TPO-IO und MPI-IO deutlicher aus und decken sich daher auch erst ab einer Dateigröße von 64 MB.

7.5 Zusammenfassung

In den Anwendungsfällen sph2000 und FMM konnte gezeigt werden, dass die Implementierung der parallelen Ein-/Ausgabe in eine bereits bestehende Anwendung bei einem entsprechend guten Design sehr leicht und schnell durchgeführt werden kann. Der bei diesen Anwendungen ursprünglich implementierte Master-Worker-Algorithmus ist im wissenschaftlichen Rechnen weit verbreitet und ergibt somit für den Einsatz der entwickelten Schnittstelle ein hohes Anwendungspotential.



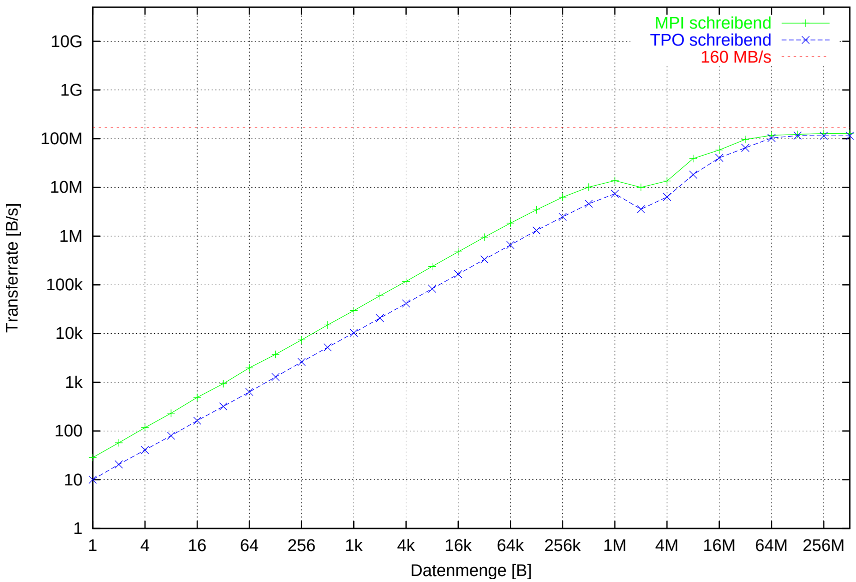


Abbildung 7.17: Vergleich der einfachen E/A-Leistung von TPO-IO und MPI auf der Hitachi SR 8000. Obere Grafik: Leseleistung. Untere Grafik: Schreibleistung.

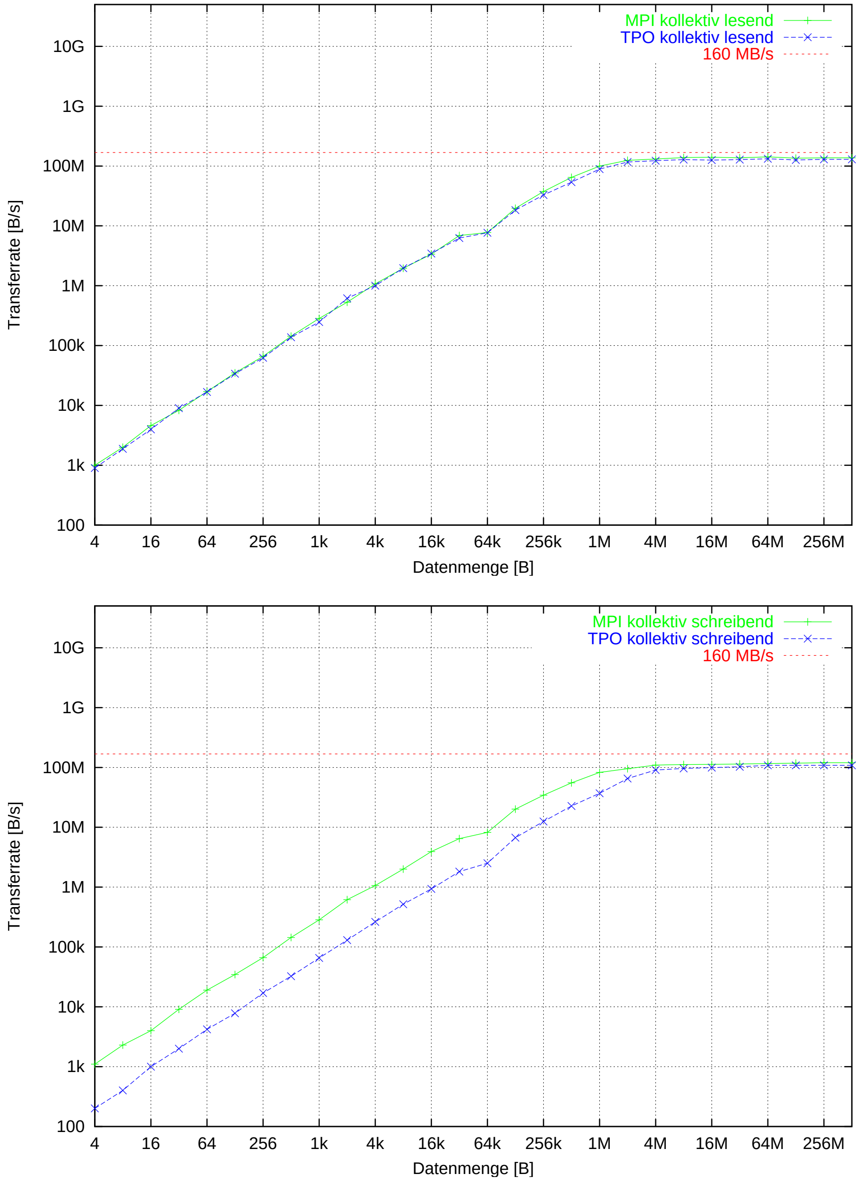


Abbildung 7.18: Vergleich der kollektiven E/A-Leistung von TPO-IO und MPI auf der Hitachi SR 8000. Obere Grafik: Leseleistung. Untere Grafik: Schreibleistung.

Idealerweise findet die objektorientierte Schnittstelle bereits in der Analyse- und Entwurfsphase einer Neuentwicklung Berücksichtigung, wie dies bei ParSeq der Fall ist. Dadurch kann eine Qualität der Anwendung erreicht werden, die anderenfalls durch Anpassungen bereits bestehender Datenstrukturen nie hätte erzielt werden können.

Die synthetischen Messungen auf den vier verschiedenen Rechnerarchitekturen zeigen im allgemeinen einen nur geringen Leistungsunterschied zwischen der objektorientierten Schnittstelle und reinem MPI-IO. In allen gemessenen Fällen konnte bereits ab einer relativ geringen Transfergröße dieselbe Bandbreite erreicht werden. Die Leistungsunterschiede sind neben der objektorientierten Abstraktion auf die bibliotheksinternen Berechnungen, wie z.B. die Ermittlung der Offsets, und dem Schreiben bzw. Lesen von Metainformationen zurückzuführen. Aufgrund ihrer geringen Menge werden diese jedoch meistens vom Dateisystem zwischengespeichert und reduzieren somit nur marginal die Leistung der Schnittstelle. Abhängig von der Prozessorleistung, der Leistung des Netzwerks und des Dateisystems ergeben sich entweder größere oder kleinere Unterschiede zwischen der Transferrate der objektorientierten Schnittstelle und MPI-IO. Die Vielfältigkeit der untersuchten Rechnerarchitekturen erlaubt es, die gewonnenen Ergebnisse und Erkenntnisse auf andere Architekturen zu übertragen und damit als nahezu allgemein gültig erklären zu können.

Damit konnte in diesem Kapitel gezeigt werden, dass mit der Entwicklung von TPO-IO die gesteckten Ziele erreicht werden konntn. Neben der Möglichkeit sämtliche objektorientierte Datenstrukturen auf einen Festspeicher zu übertragen, führte in allen Anwendungsfällen der Einsatz von TPO-IO zu einer deutlichen Vereinfachung der Parallelisierung, ohne dass die Funktionalität gegenüber MPI reduziert werden müsste. Der Zugewinn einer effizienten Parallelisierung der Ein-/Ausgabe überwog darüber hinaus fast immer gegenüber der Aufwandserhöhung, die durch Einbinden von TPO-IO hervorgerufen wird. Die Höhe der Leistungssteigerunghängt dabei erwartungsgemäß entscheidend von dem Verhältnis zwischen Berechnungs- und Ein-/Ausgabeaufwand ab. Schließlich ergeben die synthetischen Leistungsmessungen, dass die Vorteile durch die objektorientierte Schnittstelle TPO-IO gegenüber dem Nachteil eines geringen Leistungsverlustes deutlich überwiegen.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Die vorliegende Arbeit stellt die Konzeption und Implementierung einer objektorientierten Schnittstelle zur parallelen Ein-/Ausgabe dar, die mit Hilfe von standardisierten prozeduralen Ein-/Ausgabesystemen nicht erzielt werden kann. Die semantische Lücke zwischen objektorientiertem Design und der existierenden Softwareentwicklung auf Höchstleistungsrechnern wurde geschlossen, indem eine Implementierung entwickelt wurde, die auf einer prozeduralen Schnittstelle aufsetzt und diese hinter einem objektorientierten Modell verbirgt. Eine klare methodische Strukturierung von Design und Automatisierungsmechanismen trägt deutlich zur Steigerung der Transparenz bei. Moderne Techniken der Programmiersprache C++, wie z.B. Template-Mechanismen, führen zu einer effizienten Umsetzung der Objekte auf einfachere Datenstrukturen. In insgesamt drei Anwendungen aus teilweise unterschiedlichen Anwendungsgebieten wurde die Schnittstelle zur Parallelisierung der Ein-/Ausgabe bereits erfolgreich implementiert. Dabei wurde deutlich, wie unter Verwendung einer objektorientierten Schnittstelle die Formulierung der Ein-/Ausgabe einer Anwendung vereinfacht und gleichzeitig eine Leistungssteigerung erzielt werden konnte. Darüber hinaus konnte die hohe Leistung der Schnittstelle, die den prozeduralen Einsätzen in nichts nachsteht, in zahlreichen synthetischen Messungen bestätigt werden. Dies ebnet den Weg für einen praktischen Einsatz der Schnittstelle im Höchstleistungsrechnen.

8.2 Ausblick

Die im Rahmen dieser Arbeit vorgestellten Konzepte von TPO-IO können in vielfältiger Weise weiterentwickelt werden. Ein aktueller Trend im wissenschaftlichen Rechnen liegt in der Nutzung vieler verteilter Rechner, die z.B. über das Internet zusammengeschlossen werden (*Grid-Computing*). Zur Einrichtung einer solchen Grid-Architektur existieren zahlreiche frei zugängliche Softwarepakete, wie z.B. das *Globus Toolkit* [82]. Die Verbindung von TPO-IO und MPI-IO wird durch eine sehr schlanke Schnittstelle realisiert, die eindeutige Dateibezeichner und einfache Byteströme verwendet. Eine Möglichkeit der Weiterentwicklung von TPO-IO liegt daher in dessen Portierung auf ein bereits existierendes System, wie z.B. Globus, um eine parallele Ein-/Ausgabe auch in einem verteilten Rechnerverbund zu ermöglichen.

Die Schwierigkeit liegt dabei in erster Linie in der Synchronisierung der Datenzugriffe bei Einsatz kollektiver Methoden und der dabei nötigen Sicherstellung der Konsistenz. Ein aktuell initiiertes Forschungsprogramm, getragen von vier Bioinformatik- und Informatik-Abteilungen der Universität Tübingen und der Abteilung Bioinformatik des Max-Planck-Instituts für Entwicklungsbiologie in Tübingen, verfolgt dieses Ziel. Die Partner verfügen neben dem Kepler-Cluster über vier weitere, sehr leistungsfähige Cluster mit jeweils mindestens 32 Prozessoren, die mittels einheitlicher Grid-Software für die Bioinformatik-Anwender nutzbar gemacht werden sollen.

Ein weiterer Forschungsschwerpunkt könnte sich auf die Lösung der Problematik beziehen, dass die von TPO-IO gespeicherten Daten nur von derselben Anwendung wieder eingelesen werden können. Dies stellt insbesondere bei der Weiterverarbeitung der Daten, z.B. zum Zwecke der Visualisierung, ein Problem dar, da diese bisweilen sehr zeitaufwändige Konvertierungsarbeiten erfordert. Die Ergänzung der Strukturinformationen um geeignete Annotationen, die die gespeicherten Daten beschreiben, könnte dieses Probem lösen. In diesem Zusammenhang wäre auch eine Anbindung von TPO-IO an eines der Dateiformate netCDF oder HDF5 vorstellbar, die bereits selbstbeschreibende Datenstrukturen verwenden. Die Umsetzung der objektorientierten Datenstrukturen von TPO-IO auf die genannten Formate wäre jedoch nur unter extremen Leistungsverlusten möglich und sollte daher sinnvollerweise nur optional angeboten werden.

Bei vielen parallelen Anwendungen handelt es sich um sehr umfangreiche Simulationen, die eine lange Laufzeit in Anspruch nehmen. Um das Risiko eines Datenverlustes durch Systemfehler zu verringern, existieren so genannte Checkpointing-Systeme. Diese speichern den Gesamtzustand einer Anwendung im Laufe ihrer Ausführung in regelmäßigen Abständen auf einen Festspeicher. Das System sammelt dabei genügend Informationen, um eine nahtlose Wiederaufnahme der Anwendung an diesem Speicherpunkt zu ermöglichen. Nach einem Systemfehler kann dann die Anwendung direkt fortgesetzt werden und muss nicht von vorne gestartet werden. TPO-IO unterstützt bereits Mechanismen zur Speicherung von Strukturinformationen und legt damit den Grundstein zur Ausweitung auf ein anwendungsseitiges Checkpointing-System. Eine wesentliche Rolle bei dieser Entwicklung spielt die Effizienz. Bei den zusätzlichen Zugriffen handelt es sich jedoch um Schreibvorgänge, so dass ein möglicher Ansatz in der Überlappung von Ein-/Ausgabevorgängen und Berechnungen liegen könnte. Die dazu notwendigen asynchronen Zugriffe werden von TPO-IO bereits unterstützt.

Einen weiteren Ansatzpunkt bildet die in TPO-IO realisierte Restriktion objektorientierter Softwareentwicklung, die lediglich den Zugriff auf die gesamte Datenstruktur ermöglicht. Ein Zugriff auf einzelne Variablen (z.B. eine Ortskoordinate) ist damit nicht möglich. Sowohl während einer Simulation als auch zur Weiterverarbeitung der Daten, z.B. zum Zwecke der Visualisierung, ist ein solcher Zugriff jedoch oft notwendig. Die im Grunde überflüssig übertragenen Daten belasten die Bandbreite eines Systems und führen damit zu einem Effizienzverlust der Anwendung, der umso höher ist, je kleiner der Anteil der benötigten Variable an der gesamten Datenstruktur ist. Durch Erweiterung der bereits eingesetzten Metadaten könnte diese Aufgabe durch TPO-IO realisiert werden, ohne die Konzepte der objektorientierten Programmierung zu verwerfen. Der durch den Mehraufwand der Datenspeicherung entstehende Effizienzverlust könnte durch asynchrone Zugriffe auf die Metadaten kompensiert werden.

Die sehr einfache Formulierung der Ein-/Ausgabe unter Verwendung objektorientierter Datenstrukturen und die flexible Definition von Sichten auf Daten ermöglicht insgesamt den Einsatz von TPO-IO in zahlreichen wissenschaftlichen Disziplinen. Dadurch entstanden in den letzten Jahren viele Kooperationen zwischen dem Teilprojekt C6, in dessen Rahmen die vorliegende Arbeit entstand, und anderen Teilprojekten, deren Anwendungen eine Parallelisierung der Ein-/Ausgabe erforderten. Ein weiteres Einsatzgebiet für TPO-IO könnten auch Objektdatenbanken (engl. object-oriented database management system, OODBMS) sein. Diese speichern im Gegensatz zu relationalen Datenbanken die Informationen eines Datensatzes zentral, innerhalb eines Objekts. Die Ein-/Ausgabe der Objekte könnte unter Verwendung von TPO-IO parallelisiert werden, da die dazu notwendigen Schnittstellen bereits vorhanden sind. Dadurch könnte die Leistung der Datenbank gesteigert und die Latenz einer Anfrage auf ein Minimum reduziert werden.

A Schnittstelle zur parallelen Ein-/Ausgabe

```
/// FILEMANAGEMENT
/// Chapter 7.2 File Manipulation in MPI−2 Reference
int open(Communicator comm, char *filename, int amode, MPI_Info info );
int close ();
int delete (char * filename , MPI_Info info;
int set_size ( int size );
int preallocate ( int size );
int get_size ();
Group getGroup();
int get_amode(int *amode);

/// FILE VIEW
/// Chapter 7.3 File View in MPI−2 Reference
int set_view(TPO::View view);
TPO::View get_view();

/// FILEACCESS
/// Chapter 7.4 Data Access in MPI−2 Reference
/// non−collective
/// − local pointer / explicit offset
template < class T>
Status read(T& buf, int offset = 0);
template < class T>
Status write (const T& buf, const int offset = 0);
template < class Iterator >
Status read( Iterator & first , Iterator & last , int offset = 0);
template < class Iterator >
Status write (const Iterator & first , const Iterator & last , const int offset = 0);
template < class T>
Status iread (T& buf, int offset = 0);
template < class Iterator >
Status iwrite (const T& buf, const int offset = 0);
template < class T>
Status iread ( Iterator first , Iterator last , int offset = 0);
template < class Iterator >
Status iwrite (const Iterator first , const Iterator last , const int offset = 0);

/// − shared pointer
template < class T>
Status read_shared (T& buf);
template < class T>
Status write_shared (const T& buf);
template < class Iterator >
Status read_shared ( Iterator first , Iterator last );
```

```
template < class Iterator >
Status write_shared (const Iterator first , const Iterator last );
template < class T>
Status iread_shared (T& buf);
template < class T>
Status iwrite_shared (const T& buf);
template < class Iterator >
Status iread_shared ( Iterator first , Iterator last );
template < class Iterator >
Status iwrite_shared (const Iterator first , const Iterator last );

/// collective
/// — local pointer / explicit offset
template < class T>
Status read_all (T& buf, int offset = 0);
template < class T>
Status write_all (const T& buf, const int offset = 0);
template < class Iterator >
Status read_all ( Iterator & first , Iterator & last , int offset = 0);
template < class Iterator >
Status write_all (const Iterator & first , const Iterator & last , const int offset = 0);
template < class T>
Status read_all_begin (T& buf, int offset = 0);
template < class T>
Status write_all_begin (const T& buf, const int offset = 0);
template < class Iterator >
Status read_all_begin ( Iterator first , Iterator last , int offset = 0);
template < class Iterator >
Status write_all_begin (const Iterator first , const Iterator last , const int offset = 0);
template < class T>
Status read_all_end (T& buf, int offset = 0);
template < class T>
Status write_all_end (const T& buf, const int offset = 0);
template < class Iterator >
Status read_all_end ( Iterator first , Iterator last , int offset = 0);
template < class Iterator >
Status write_all_end (const Iterator first , const Iterator last , const int offset = 0);

/// — shared pointer
template < class T>
Status read_ordered (T& buf);
template < class T>
Status write_ordered (const T& buf);
template < class Iterator >
Status read_ordered ( Iterator first , Iterator last );
template < class Iterator >
Status write_ordered (const Iterator first , const Iterator last );
template < class T>
Status read_ordered_begin (T& buf);
template < class T>
Status write_ordered_begin (const T& buf);
```

```
template < class Iterator >
Status read_ordered_begin( Iterator first , Iterator last );
template < class Iterator >
Status write_ordered_begin(const Iterator first , const Iterator last );
template < class T>
Status read_ordered_end(T& buf);
template < class T>
Status write_ordered_end(const T& buf);
template < class Iterator >
Status read_ordered_end( Iterator first , Iterator last );
template < class Iterator >
Status write_ordered_end(const Iterator first , const Iterator last );

/// Chapter 7.5 File Interoperability in MPI−2 Reference
int seek(int off , int whence);
int get_position ();
int get_byte_offset ( int off );
int get_type_extent (MPI_Datatype datatype , int ∗ extent );

/// Chapter 7.6 Consistency and Semantics in MPI−2 Reference
int set_atomicity ( int flag );
int get_atomicity ( int ∗ flag );
int sync();
}
```

Literaturverzeichnis

Alle Quellenangaben im Internet sind auf dem Stand Dezember 2005. Soweit bekannt ist das Datum der letzten Aktualisierung in eckigen Klammern angefügt.

[1] 10 Gigabit Ethernet Alliance: *10 Gigabit Ethernet Technology Overview White Paper, Revision 2, Draft A*, 2002. Online im Internet: http://www.ethernetalliance.org/technology/white_papers/10gea_overview.pdf [Stand April 2002].

[2] Alexandrescu, A.: *Modern C++ Design: Generic Programming and Design Patterns Applied*. AW C++ in Depth Series. Addison Wesley, Januar 2001.

[3] Alliance Systems Technology: *The Move to PCI Express in Next Generation Systems, White Paper*. Online im Internet: http://www.alliancesystems.com/Documents/PCI_Express_Whitepaper.pdf .

[4] Almasi, G.S. und A. Gottlieb: *Highly parallel computing*. The Benjamin/-Cummings series in computer science and engineering. Benjamin/Cummings California, 1989.

[5] AMD: *HyperTransport Technology-Based System Architecture*, 2002. Online im Internet: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/AMD_HyperTransport_Technology-Based_System_Architecture.pdf [Stand Mai 2002].

[6] Amdahl, G.M.: *Validity of the single processor approach to achieving large scale computing capabilities*. In: *AFIPS Proc. of the SJCC*, Band 30, Seiten 483–485, 1967.

[7] Anderson, T.E., M.D. Dahlin, J.M Neefe, D.A. Patterson, D.S. Roselli und R.Y. Wang: *Serverless network file system*. In: *ACM Transactions on Computer Systems*, Band 14(1), Seiten 41–79, 1996.

[8] Baker, W., A. van den Broek, E. Camon, P. Hingamp, P. Sterk, G. Stoesser und M.A. Tuli: *The EMBL Nucleotide Sequence Database*. In: *Nucleic Acids Research*, Band 28 (1), Seiten 19–23, 2000.

[9] BARKES, J., M.R. BARRIOS, F. COUGARD, P.G. CRUMLEY, D. MARTIN, H. REDDY und T. THITAYANUN: *GPFS: A Parallel File System*. In: *IBM International Technical Support Organization*, 1998.

[10] BARRETT, B.W., J.M. SQUYRES und A. LUMSDAINE: *Implementation of Open MPI on Red Storm*. Technischer Bericht LA-UR-05-8307, Los Alamos National Laboratory, 2005.

[11] BENSON, D., I. KARSCH-MIZRACHI, D.J. LIPMAN, J. OSTELL, B.A. RAPP und D.L. WHEELER: *GenBank*. In: *Nucleic Acids Research*, Band 28 (1), Seiten 15–18, 2000.

[12] BERNHOLDT, D.E., B.A. ALLAN, R. ARMSTRONG, F. BERTRAND, K. CHIU, T.L. DAHLGREN, K. DAMEVSKI, W.R. ELWASIF, T.G.W. EPPERLY, M. GOVINDARAJU, D.S. KATZ, J.A. KOHL, M. KRISHNAN, G. KUMFERT, J.W. LARSON, S. LEFANTZI, M.J. LEWIS, A.D. MALONY, L.C. MCINNES, J. NIEPLOCHA, B. NORRIS, S.G. PARKER, J. RAY, S. SHENDE, T.L. WINDUS und S. ZHOU: *A Component Architecture for High-Performance Scientific Computing*. Intl. J. High-Perf. Computing Appl., 2005.

[13] BODEN, N.J., D. COHEN, R.E. FELDERMAN, A.E. KULAWIK, C.L. SEITZ, J.N. SEIZOVIC und W. SU: *Myrinet: A Gigabit-per-Second Local Area Network*. In: *IEEE Micro*, Band 15(1), Seiten 29–36, 1995.

[14] BOOCH, G.: *Objektorientierte Analyse und Design*. Addison-Wesley, 1995.

[15] BOOCH, G., J. RUMBAUGH und I. JACOBSON: *Unified Modeling Language, Version 1.0*. Rational Software Corporation, Santa Clara, CA, 1997.

[16] CARNS, P.H., W.B. LIGON III, R.B. ROSS und R. THAKUR: *PVFS: A Parallel File System for Linux Clusters*. In: *Proceedings of the 4th Annual Linux Showcase and Conference*, Seiten 317–327, 2000.

[17] CHENG, H. und C. KING: *File declustering for efficient parallel I/O on networks of workstations*. In: *Cluster computing*, Seiten 121–132, Commack, NY, USA, 2001.

[18] CORBETT, P.F. und D.G. FEITELSON: *The Vesta parallel file system*. In: *ACM Transactions on Computer Systems*, Band 14(3), Seiten 225–264, 1996.

[19] CORBETT, P.F., J. PROST, C. DEMETRIOU, G. GIBSON, E. RIEDEL, J. ZELENKA, Y. CHEN, E. FELTEN, K. LI, J. HARTMAN, L. PETERSON, B. BERSHAD, A. WOLMAN und R. AYDT: *Proposal for a common parallel file system programming interface version 1.0*. The official SIO Low-Level API standard,

1996. Online im Internet: http://www.pdl.cs.cmu.edu/SIO/ [Stand November 2004].

[20] CORMEN, T.H. und D. KOTZ: *Integrating Theory and Practice in Parallel File Systems*. In: *Proceedings of the 1993 DAGS/PC Symposium*, Seiten 64–74, Hanover, NH, 1993.

[21] ENGESSER, H.: *Duden der Informatik*. Dudenverlag, 1988.

[22] ETNUS LLC.: *TotalView*. Online im Internet: http://www.etnus.com/TotalView/index.html [Stand 2005].

[23] FEITELSON, D.G., P.F. CORBETT, S.J. BAYLOR und Y. HSU: *Parallel I/O Subsystems in Massively Parallel Supercomputers*. In: JIN, H., T. CORTES und R. BUYYA (Herausgeber): *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, Seiten 389–407, New York, NY, 2001.

[24] FLYNN, M.J.: *Some computer organizations and their effectiveness*. In: *IEEE Trans. on Computers*, Band C-21, Seiten 948–960, September 1972.

[25] GABRIEL, E., G.E. FAGG, G.BOSILCA, T. ANGSKUN, J.J. DONGARRA, J.M. SQUYRES, V. SAHAY, P. KAMBADUR, B. BARRETT, A. LUMSDAINE, R.H. CASTAIN, D.J. DANIEL, R.L. GRAHAM und T.S. WOODALL: *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*. In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Seiten 97–104, 2004.

[26] GANZENMÜLLER, S., S. PINKENBURG und W. ROSENSTIEL: *SPH2000: A Parallel Object-Oriented Framework for Particle Simulations with SPH*. In: *Proc. of the 11th International Euro-Par Conference*, Seiten 1275–1284, 2005.

[27] GANZENMÜLLER, S.: *Analyse und Design einer objektorientierten SPH-Bibliothek mit Entwurfsmustern unter dem Aspekt der Parallelisierung*. Diplomarbeit, Universität Tübingen, Technische Informatik, 2000.

[28] GAUGER, C., P. LEINEN und H. YSERENTANT: *The finite mass method*. In: *SIAM J. Numer. Anal.*, Band 37, Seiten 1768–1799, 2000.

[29] GIBSON, G.A. und R. VAN METER: *Network attached storage architecture*. Communications of the Association for Computing Machinery, 43(11):37–45, November 2000.

[30] GIGABIT ETHERNET ALLIANCE: *Gigabit Ethernet*, 1997. Online im Internet: http://www.ethernetalliance.org/technology/white_papers/gea_baset.pdf [Stand 1997].

[31] GOULD, E. und M. XINU: *The network file system implemented on 4.3 BSD*. In: *USENIX Association Summer Conference Proceedings*, Seiten 294–298, 1986.

[32] GROPP, W., E. LUSK und R. THAKUR: *Using MPI-2: Advanced Features of the Message-Passing Interface*. The MIT Press, Cambridge, MA, 1999.

[33] HARKIN, R.L.: *Tiger Shark – A scalable file system for multimedia*. In: *IBM Journal of Research and Development*, Band 42(2), Seiten 185–197, 1998.

[34] HÖCHSTLEISTUNGSRECHENZENTRUM BAYERN (HLRB): *Hitachi SR8000-F1/168 website*, 2004. Online im Internet: http://www.lrz-muenchen.de/services/compute/hlrb/ [Stand September 2005].

[35] HÖCHSTLEISTUNGSRECHENZENTRUM STUTTGART (HLRS)): *CRAY-Opteron-Cluster Documentation*, 2005. Online im Internet: http://www.hlrs.de/hw-access/platforms/strider/user_doc.pdf [Stand April 2005].

[36] HÖCHSTLEISTUNGSRECHENZENTRUM STUTTGART (HLRS): *NEC SX-6 Cluster Documentation*, 2005. Online im Internet: http://www.hlrs.de/hw-access/platforms/sx6/user_doc.pdf [Stand Dezember 2005].

[37] HENNESSY, J.L., D.A. PATTERSON, D. GOLDBERG und K. ASANOVIC: *Computer architecture: A quantitative approach*. Morgan Kauffmann Publishers Inc., Dritte Auflage, Juni 2002.

[38] HEUSER, F.: *Objektorientierte Implementierung der SPH-Methode für Dieseleinspritzung mit Entwurfsmustern*. Diplomarbeit, Universität Tübingen, Technische Informatik, 2000.

[39] HIGH PERFORMANCE FORTRAN FORUM: *High Performance Fortran Language Specification, version 1.0*. Technischer Bericht CRPC-TR92225, Rice University, Houston, TX, 1993.

[40] HIGH PERFORMANCE FORTRAN FORUM: *High Performance Fortran Language Specification, version 2.0*. Technischer Bericht, Rice University, Houston, TX, 1997.

[41] HOWARD, J.H.: *An overview of the Andrew File System*. In: *USENIX Association Winter Conference Proceedings*, Seiten 213–216, 1988.

[42] HÜTTEMANN, S.: *Untersuchungen über objektorientierte Design-Patterns für massiv-parallele Teilchensimulationsverfahren anhand von Smoothed Particle Hydrodynamics*. Doktorarbeit, Technische Informatik, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Dezember 2001.

[43] HUGHES-JONES, R., P. CLARKE und S. DALLISON: *Performance of 1 and 10 Gigabit Ethernet cards with server quality motherboards*. Future Gener. Comput. Syst., 21(4):469–488, 2005.

[44] HYPERTRANSPORT TECHNOLOGY CONSORTIUM: *HyperTransport I/O Link Specification Revision 2.00b*, 2005. Online im Internet: http://www.hypertransport.org/docucontrol/HTC20031217-0036-0009.pdf [Stand April 2005].

[45] INFINIBAND TRADE ASSOCIATION: *InfiniBand Architecture Specification Volume 1, Release 1.1*, 2002. Online im Internet: http://www.infinibandta.org/specs .

[46] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS: *Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API)*. Technischer Bericht, IEEE Standard for Information Technology , New York, 1996.

[47] INTEL: *Trace Analyzer and Collector Version 6.0*. Online im Internet: http://www.intel.com/cd/software/products/asmo-na/eng/cluster/tanalyzer/index.htm [Stand 2005].

[48] INTEL CORPORATION: *Paragon System User's Guide*. Technischer Bericht, Intel SSD, Beaverton, OR, 1995.

[49] JACOBSON, I., M. CHRISTERSON, P. JONSSON und G. ÖVERGAARD: *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley Publishing Company, Wokingham, England, 1992.

[50] KITAGAWA, K., S. TAGAYA, Y. HAGIHARA und Y. KANOH: *A hardware overview of SX-6 and SX-7 supercomputer*. NEC Research & Development Journal, 44(1):2–7, 2003. Online im Internet: http://www.nec.co.jp/techrep/en/r_and_d/r03/r03-no1/rd02.pdf [Stand Januar 2003].

[51] KOTZ, D. und C.S. ELLIS: *Practical prefetching techniques for parallel file systems*. In: *PDIS '91: Proceedings of the first international conference on Parallel and distributed information systems*, Seiten 182–189, Los Alamitos, CA, USA, 1991.

[52] KUNZE, S.: *Numerische Simulationen von Akkretionsscheiben in Kataklysmischen Variablen mit Smoothed Particle Hydrodynamics*. Doktorarbeit, Theoretische Astrophysik, Universität Tübingen, 2000.

[53] LAZOWSKA, E.D., J. ZAHORJAN, D.R. CHERITON und W. ZWAENEPOEL: *File access performance of diskless workstations*. ACM Trans. Comput. Syst., 4(3):238–268, 1986.

[54] LI, J., W. LIAO, A. CHOUDHARY, R. ROSS, R. THAKUR, W. GROPP, R. LATHAM, A. SIEGEL, B. GALLAGHER und M. ZINGALE: *Parallel netCDF: A High-Performance Scientific I/O Interface*. In: *Proceedings of SC2003: High Performance Networking and Computing*, 2003.

[55] LI, J., W. LIAO, A.N. CHOUDHARY und V.E. TAYLOR: *I/O Analysis and Optimization for an AMR Cosmology Application*. In: *IEEE International Conference on Cluster Computing (CLUSTER 2002)*, Seiten 119–126, 2002.

[56] MACHE, J.: *An Assessment of Gigabit Ethernet as Cluster Interconnect*. In: *IWCC '99: Proceedings of the 1st IEEE Computer Society International Workshop on Cluster Computing*, Seite 36, 1999.

[57] MAY, J.M.: *Parallel I/O for High Performance Computing*. Morgan Kaufmann Publishers, 2001.

[58] MESSAGE PASSING INTERFACE FORUM: *MPI: A message passing interface standard*. Technical Report UT-CS-94-230, Computer Science Department, University of Tennessee, Knoxville, TN, Mai 1994.

[59] MESSAGE PASSING INTERFACE FORUM: *MPI-2: Extensions to the Message Passing Interface*, Juli 1997. Online im Internet: `http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html` [Stand September 2001].

[60] MEYERS, S.: *Effective STL: 50 Specific Ways to Improve the Use of the Standard Template Library*. Professional Computing Series. Addison Wesley, Second Auflage, August 2001.

[61] THE NATIONAL CENTER FOR SUPERCOMPUTING APPLICATIONS: *HDF4 Home Page*. Online im Internet: `http://hdf.ncsa.uiuc.edu/hdf4.html` [Stand März 2005].

[62] THE NATIONAL CENTER FOR SUPERCOMPUTING APPLICATIONS: *HDF5 Home Page*. Online im Internet: `http://hdf.ncsa.uiuc.edu/HDF5/` [Stand November 2005].

[63] OESTEREICH, B.: *Objektorientierte Softwareentwicklung. Analyse und Design mit der Unified Modeling Language*. 4. Auflage, Oldenbourg, 1998.

[64] OLDHAM, J.D.: *POOMA: A C++ Toolkit for High-Performance Parallel Scientific Computing*, 2002. Online im Internet: http://www.codesourcery.com/public/pooma/manual.pdf [Stand Mai 2003].

[65] OTT, F.: *Weiterentwicklung und Untersuchung von Smoothed Particle Hydrodynamics im Hinblick auf den Zerfall von Dieselfreistrahlen in Luft*. Doktorarbeit, Theoretische Astrophysik, Universität Tübingen, 1999.

[66] PATTERSON, D.A., G. GIBSON und R.H. KATZ: *A case for redundant arrays of inexpensive disks (RAID)*. In: *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, Seiten 109–116, 1988.

[67] PC CLUSTER CONSORTIUM: *SCore cluster system software*. Online im Internet: http://www.pccluster.org [Stand Dezember 2002].

[68] PCI-SIG: *PCI-X Addendum to the PCI Local Bus Specification Revision 1.0*, 1999. Online im Internet: http://www.pcisig.com/specifications/pcix_20/pci_x .

[69] PCI-SIG: *PCI Express Base Specification, Revision 1.0*, 2002. Online im Internet: http://www.pcisig.com/specifications/pciexpress .

[70] PFISTER, G.F.: *An Introduction to the InfiniBand Architecture*. In: JIN, H., T. CORTES und R. BUYYA (Herausgeber): *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, Seiten 617–632, New York, NY, 2001.

[71] PINKENBURG, S., M. RITT und W. ROSENSTIEL: *Parallelization of an Object-Oriented Particle-in-Cell Simulation*. In: *Workshop Parallel Object-Oriented Scientific Computing, OOPSLA*, 2001.

[72] PINKENBURG, S. und W. ROSENSTIEL: *Parallel I/O in an Object-Oriented Message-Passing Library*. In: *Proc. of the 11th European PVM/MPI Users' Group Meeting*, Seiten 251–258, 2004.

[73] QIN, J., S. PINKENBURG und W. ROSENSTIEL: *Parallel Motif Search Using ParSeq*. In: *Proc. of the International Conference on Parallel and Distributed Computing and Networks (PDCN) 2005*, 2005.

[74] RABENSEIFNER, R. und A.E. KONIGES: *Effective File-I/O Bandwidth Benchmark*. In: *Proceedings of Euro-Par 2000 – Parallel Processing*, Seiten 1273–1283, 2000.

[75] REW, R. und G. DAVIS: *Data Management: NetCDF: an Interface for Scientific Data Access*. In: *IEEE Comput. Graph. Appl.*, Band 10(4), Seiten 76–82, 1990.

[76] REW, R., G. DAVIS und S. EMMERSON: *NetCDF User's Guide: An Interface for Data Access, Version 2.3*, 1993.

[77] RITT, M.: *Eine objektorientierte Kommunikationsbibliothek zur parallelen Programmierung - TPO++*. Doktorarbeit, Universität Tübingen, 2003.

[78] ROSENSTIEL, W.: *Rechnerarchitektur II*. Vorlesung zur Rechnerarchitektur. Universität Tübingen.

[79] ROSS, R., D. NURMI, A. CHENG und M. ZINGALE: *A case study in application I/O on Linux clusters*. In: *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Seiten 11–11, 2001.

[80] RUMBAUGH, J.: *OMT - The functional model*. In: *Journal of Object-Oriented Programming*, Band 8(1), Seiten 10–14, 1995.

[81] SACHS, M. und A. VARMA: *Fibre channel and related standards*. IEEE Communications Magazine, 34(8):40–50, 1996.

[82] SANDHOLM, T. und J. GAWOR: *Globus Toolkit 3 Core A Grid Service Container Framework*. 2003. Online im Internet: `http://www-unix.globus.org/toolkit/3.0/ogsa/docs/gt3_core.pdf` [Stand Juli 2003].

[83] SAPHIR, W., L.A. TANNER und B. TRAVERSAT: *Job Management Requirements for NAS Parallel Systems and Clusters*. In: *JSSPP*, Seiten 319–336, 1995.

[84] SCHMOLLINGER, M., I. FISCHER, C. NERZ, S. PINKENBURG, F. GÖTZ, M. KAUFMANN, K.-J. LANGE, R. REUTER, W. ROSENSTIEL und A. ZELL: *ParSeq: Searching Motifs with Structural and Biochemical Properties*. In: *Journal of Bioinformatics*, Band 20(9), Seiten 1459–1461, 2004.

[85] SPEITH, R.: *Untersuchung von Smoothed Particle Hydrodynamics anhand astrophysikalischer Beispiele*. Doktorarbeit, Theoretische Astrophysik, Universität Tübingen, 1998.

[86] SQUYRES, J.M. und A. LUMSDAINE: *A Component Architecture for LAM/MPI*. In: *Proceedings, 10th European PVM/MPI Users' Group Meeting*, Nummer 2840 in *Lecture Notes in Computer Science*, Seiten 379–387, Venice, Italy, 2003.

[87] SWEENEY, A., D. DOUCETTE, W. HU, C. ANDERSON, M. NISHIMOTO und G. PECK: *Scalability in the XFS file system*. In: *Proceedings of the 1996 USENIX Technical Conference*, Seiten 1–14, 1996.

[88] TANENBAUM, A.S.: *Modern operating systems*. Prentice-Hall Inc., 1992.

[89] TATENO, Y., S. MIYAZAKI, M. OTA, H. SUGAWARA und T. GOJOBORI: *DNA Data Bank of Japan (DDBJ) in Collaboration with Mass Sequencing Teams*. In: *Nucleic Acids Research*, Band 28 (1), Seiten 24–26, 2000.

[90] THAKUR, R., W. GROPP und E. LUSK: *An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces*. In: *Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computation*, Seiten 180–187. Argonne National Lab., 1996.

[91] THAKUR, R., W. GROPP und E. LUSK: *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation*. In: *Technical Memorandum ANL/MCS-TM-234*, Band (28)1, Seiten 82–105. Mathematics and Computer Science Division, Argonne National Laboratory, 1998.

[92] UNIVERSITÄT TÜBINGEN: *ParSEQ Homepage*. Online im Internet: http://www-ti.informatik.uni-tuebingen.de/parseq/index.html [Stand Februar 2004].

[93] UNIVERSITÄT TÜBINGEN: *Kepler cluster website*, 2001. Online im Internet: http://kepler.sfb382-zdv.uni-tuebingen.de [Stand Juni 2001].