Fachhochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Multi-Collective I/O

Ausnutzung von Mehr-Datei-Zugriffsmustern

Sahin Demir

und

Moritz Vieth

Seminararbeit zur Vorlesung

"Parallel Systems"

gehalten von

Prof. Dr. Rudolf Berrendorf

Somemrsemester 2005

Inhaltsverzeichnis

1 Einleitung

Diese Seminararbeit beschäftigt sich mit dem Paper [MCIO], das die Entwicklung eines I/O-Konzeptes erläutert, dass die Parallelität eines Systemes zur Steigerung des I/O-Durchsatzes nutzt.

In der Mikroprozessorleistung hat in den letzten Jahrzehnten die Optimierung der I/O Performance Wichtigekeit gewonnen. Für eine optimale I/O kann neben der optimierten I/O Plattform /Hardware auch eine Softwareoptimierung eine wichtige Rolle spielen.

In dieser Seminararbeit wird ein neues Konzept (Multi Collective I/O) vorgestellt, dass auf der üblichen Collective I/O aufbaut. Im Gegensatz zu Collective I/O erweitert Multicollective I/O (MCIO) das Collective I/O (CIO) mit der Optimierung des I/O Zugriffes zu gleichzeitig mehreren Arrays. Dabei stimmen sich in diesem Ansatz mehrere Prozessoren aufeinander ab um die I/O für jeden auszuführen und, somit im Gesamten die I/O Zeit zu verbessern.

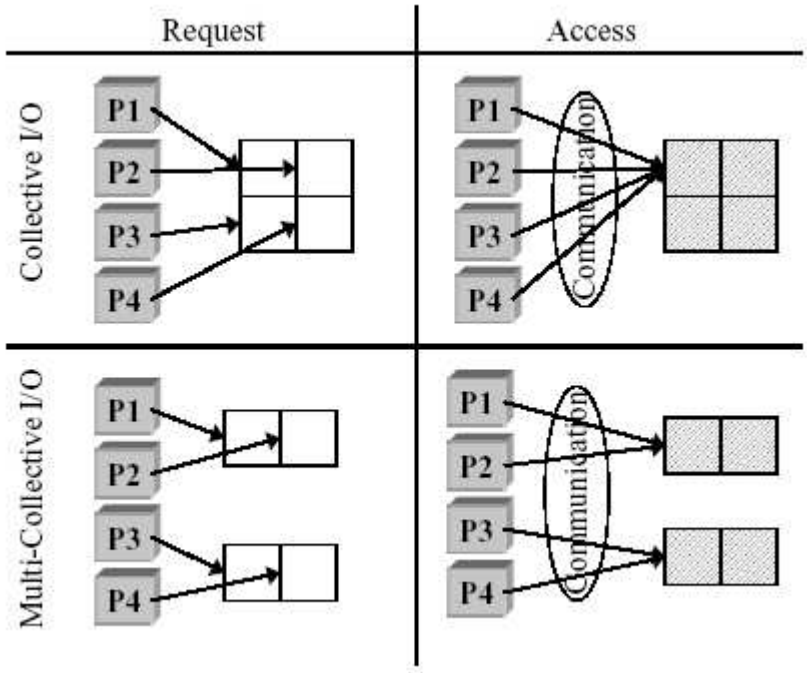

Abbildung 1: Unterschiede zwischen MCIO und traditionellem CIO [MCIO]

CIO greift auf einzelne Dateien von verschiedenen Prozessoren zu, die kombiniert eine einzelne und lange I/O-Anfrage bilden, um die Anfragezeit zu verbessern. Um die Effizienz zu steigern werden in MCIO Zugriffe aus verschiedenen Prozessoren auf mehrere Dateien kombiniert,.

Es wird gezeigt, dass das Ermitteln der optimalen MCIO Zugriffsmuster ein NP-vollständiges Problem ist und dafür zwei verschiedene Heuristiken (eine auf Sortierung und die andere auf

Abgleichung basierend) für das Zugriffsmuster-Erkennungsproblem (auch Zuordnungsproblem) vorgestellt werden. Beide Heuristiken sind in einer Laufzeitbibliothek implementiert und mit großen wissenschaftlichen Anwendungen getestet worden (die Effektivität ist unter Verwendung der beiden künstlichen Auslastungen evaluiert worden). Vorläufige Ergebnisse zeigen, dass MCIO Collective I/O um mehr als 87% übertrifft.

2 Verwandte Arbeiten

In Collective I/O werden einzelne I/O-Anfragen zu einer einzigen großen zusammengefasst und an das Speichersystem geschickt. Als Ergebnis wird die effektive I/O Bandbreite bedeutsam gesteigert. Obwohl jede CIO Technik für MCIO genutzt werden kann, wurde hier zwei-Phasen I/O genutzt, wie es in ROMIO eingesetzt wird. ROMIO ist eine portable MPI-IO-Lösung, die von Argonne National Laboratories [ROMIO] entwickelt wurde. ROMIO hat die verschiedenen MPI-Implementierungen der verschiedenen Hersteller (z.B. HP, SGI, NEC) und die zwei weit verbreiteten und frei erhältlichen portablen MPI Implementierungen MPICH und LAM in verschiedenen MPI-Bibliotheken zusammengefasst.

Cypher et al. [Cypher] untersuchten individuelle parallele wissenschaftliche Anwendungen, die zeitliche Muster in I/O-Zugriffen messen. Crandall et al. [Crandall] führt ein Analyse basierend auf Pablo [Bagrodia] auf drei wissenschaftliche Anwendungen. Nieuwejaar et al. [Nieuwejaar] charakterisiert eine Mischung von User Programmen auf Intel iPSC und CM-5. Alle diese Untersuchungen machen deutlich, dass die Verwendung von MCIO durch einziges Ausführen der untersuchten Anwendung auf mehrere Dateien zugegriffen wurde.

Weiterhin werden durch verschiedene parallele I/O APIs angeboten, die mit einem einzigen Aufruf auf mehrere Dateien zugreift. Zum Beispiel verwendet HPSS [Coyne] den Gedanken der „Dateimenge“ um eine Sammlung von Dateien zu definieren. Die Dateien in einer Dateimenge, können so manipuliert werden als ob sie eine einzelne Datei erzeugen. Auf die gleiche Weise benutzt SRB [Boru] „collections“ um eine Menge der Daten zu bestimmen.

3 Collective I/O

In vielen parallelen I/O-intensiven Anwendungen die auf große, multidimensionale und plattenspeicherresidente Datensätze zugreifen, benötigen sie größtenteils die Performance der I/O-Zugriffe, die Sicht der Daten in Dateien (Speichermuster) und Verteilung der Daten über Prozessoren (Zugriffsmuster). In den Fällen, wo Speicher- und Zugriffsmuster nicht zutreffen, lässt jeder Prozessor I/O Anfragen unabhängig ausführen. Collective I/O kann die Performance verbessern, wenn zuerst die Datensets gelesen werden und dann die Daten zwischen den Prozessoren (Nutzen des inter-prozessor Kommunikationsnetzwerkes) verteilt werden um um ein Zugriffsmuster zu erreichen,. Hierbei sollten in diesem Fall die totale Datenzugriffskosten berechnet als die Summe der I/O Kosten und Kommunikationskosten werden. Seitdem in I/O-intensiven Anwendungen die I/O-Kosten im allgemeinen Hauptkommunikationskosten sind, kann Collective I/O zu großen Einsparungen über die gesamte Ausführungszeit führen.

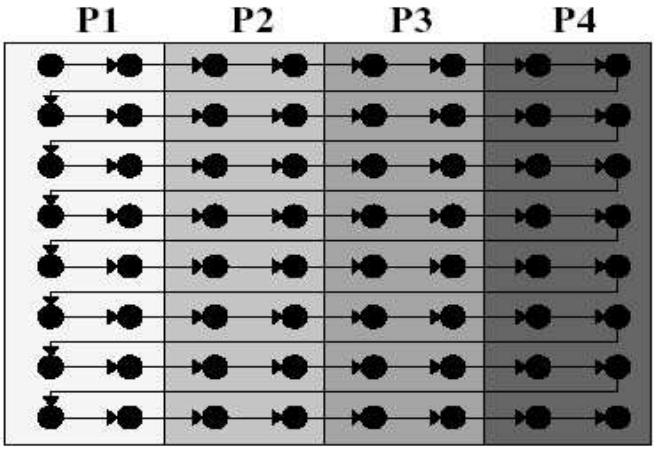

Abbildung 2: Ein Dateizugriff von 4 Prozessoren [MCIO]

Die Abbildung 2 zeigt eine Beispieldatei, die auf Muster zugreift (auf ein zweidimensionales Array). Die Kreise stellen die Datenelementen und Pfeile die Speichermuster der Elemente dar.

In diesem Fall sind die Speichermuster zeilenlastig und die Zugriffsmuster spaltenlastig. Wenn keine Collective-I/O-Techniken benutzt werden, macht jeder Prozessor an dem I/O Subsystem 8 kleine Anfragen (jeweils für zwei Arrayelemente) von insgesamt 24 kleinen Anfragen. Wahrscheinlich wird die Performance schwach sein, da die Anzahl der lesenden Elemente pro I/O Anfrage sehr klein sind. Collective I/O bündelt diese kleinen Anfragen und sendet größere Anfragen an das I/O-Subsystem, um die Performance zu verbessern. Wenn beispielsweise die zwei-Phasen I/O Technik angewandt wurde, greifen die Prozessoren auf die Daten, die eine Hauptzeile der Zugriffsmuster nutzen, welche mit den (Hauptzeilen) Speichermustern der Arrays (Abb. 3) kompatibel ist.

Jeder Prozessor liest soweit es geht mit einem einzigen I/O so viele aufeinander folgende Daten, dass sich die Anzahl der I/O Aufrufe verkleinern.

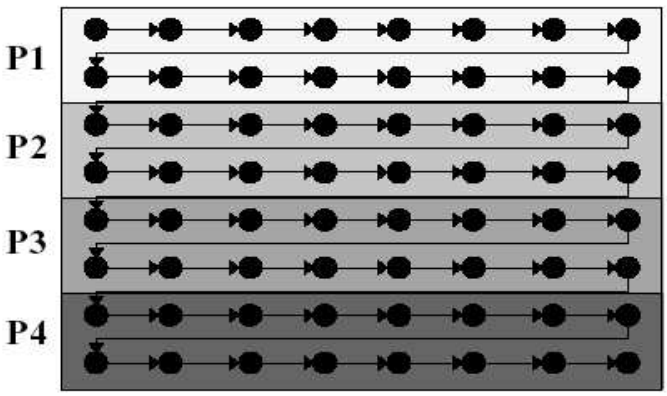

Abbildung 3: Dateizugriff mittels Collective I/O [MCIO]

Im weiteren Schritt beschäftigen sich die Prozessoren mit all-to-all inter-Prozess Kommunikation, so dass jeder Prozessor die ursprünglich angefragte Menge der Arrays empfängt (Ziel ist, dass jede Dateneinheit verteilt ist).

4 Multi-Collective I/O

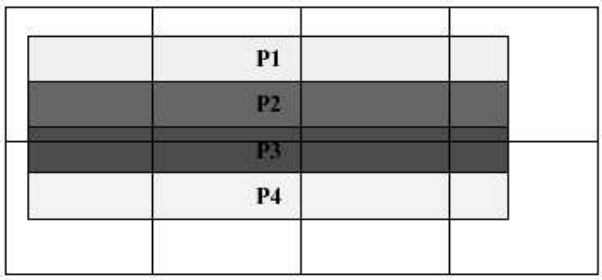

Abbildung 4: Zugriffsmuster mit 4 Prozessoren und 8 Dateien [MCIO]

In der Abbildung 5 gibt es vier Prozessoren und acht Dateien. Diese vier Prozessoren fragen verschiedene Teile von den acht Dateien an. Dabei können diese Dateien unabhängige Daten beinhalten. Das Problem besteht darin, die entsprechenden Datenmengen(-Teile) so schnell wie möglich zu lesen. Mit der bisherigen (‚naiven') Methode würden wir für jede der Dateien CIO benutzen. In Abbildung 4 würde das zum Beispiel 8 verschiedene I/O Aufrufen ergeben (jeder zwei oder drei Prozessoren involvierend). Während in dieser Methode die Anzahl der Dateien zunimmt, nimmt die Anzahl der I/O Aufrufe linear ohne Rücksicht auf die Anzahl der verfügbaren Prozessoren ab. Die Hauptidee hinter diesen Methoden ist mehrere Dateien zu mehreren Prozessoren zuzuordnen. Dadurch nimmt die I/O parallele Verfügbarkeit im System zu. Es entstehen dabei zwei wichtige Fragen, wie viele Dateien jedem Prozessor zugeteilt werden soll oder umgekehrt. Wie soll außerdem entscheiden werden, welcher Prozessor jeder Datei zugeordnet wird, wenn die Anzahl der Prozessoren festgelegt sind.

Im Folgenden wird auf die I/O Zugriffe konzentriert, wo die Anzahl der Dateien größer als die Anzahl der Prozessoren sind. Wobei, wenn die Anzahl der Dateien größer als die Anzahl der Prozessoren sind, versuchen wir die Dateien den Prozessoren zuzuordnen; Oder die Anzahl der Prozessoren größer als die Anzahl der Dateien sind, versuchen wir die Prozessoren den Dateien zuzuordnen.

4.1 Das Zuweisungsproblem

4.1.1 Definition des Zuweisungsproblems

Im Zuweisungsproblem fragen p Prozessoren Daten aus n verschiedenen Dateien an. Zur Einfachheit wird angenommen, dass $n > p$ ist. Jeder Datei ist ein Prozessor zuzuweisen, wenn Prozessoren auf ihren zugewiesenen Dateien zugreifen die Gesamtantwortzeit der Anfragen (gesamt der I/O und Kommunikationszeiten) minimiert wird. Wir berechnen die I/O-Zeit als die

Summe der Datenzugriffe eines Prozessors. Die Kommunikationszeit ist durch die Menge der Daten geschätzt, die von oder zu einem Prozessor übergeben worden sind. Für das Zuweisungsproblem soll die optimale Zuweisung der Prozessoren zu den Dateien gefunden werden, so dass $max_i \{\alpha \; x \; I/O_i + \beta \; x \; comm_i\}$ minimiert wird. α und β sind konstante Werte ,die die relativen Kosten der I/O und der Kommunikation im System abbilden. I/O_i und $comm_i$ sind die geschätzten I/O- und Kommunikationszeiten der Prozessoren i. Sie benutzen die folgenden Formeln:

- $I/O_i = \Sigma^{n}_{j=1} a_{i,j}$ (die Summe der Daten die durch Prozessor i zugegriffen wurden)
- $comm_i = \Sigma^{n}_{j=1} |r_{i,j} - a_{i,j}|$ (die Summe der Daten die durch Prozessoren i zugegriffen wurden abgezogen von der Summe der Daten die durch den Prozessor i angefragt wurden).

In den oben gezeigten Formeln entsprechen $r_{i,j}$ der Menge der Daten die vom Prozessor i aus der Datei j angefragt wurde. Und $a_{i,j}$ entspricht der Menge der Daten die von dem Prozessor i der Datei j zugegriffen wurde.

4.1.2 Komplexität des Zuweisungsproblems

In diesem Unterabschnitt wird gezeigt, dass (Zuweisungsproblem) die Optimierung der I/O Zeit (wenn $\alpha = 1$ und $\beta = 0$) ein NP-vollständiges Problem ist.

Es wird behauptet dass, für beliebige Anzahl von Prozessoren, Anzahl von Dateien und Anzahl der Dateigrößen, die optimale Zuweisung zu finden, um die I/O Zeit zu minimieren, ein NP-vollständiges Problem ist.

Genauer: wenn das Zuweisungsproblem in polynomialer Zeit gelöst werden kann, dann ist auch das Mehrprozessor Scheduling Problem [Gorey] in polynomialer Zeit lösbar. Multiprozessor Scheduling Probleme finden m zerlegte Partitionen endlicher tAufgabenmengen $A\ (A_1, A_2, \ldots, A_m)$, so dass

$$max_i \; \{\Sigma_{a \in A_i} \; length(a)\}$$

minimiert ist. Vorausgesetzt dass wir einen Solver für das Zuweisungsproblem haben, der die optimale Lösung in polynomialer Zeit findet. Dann kann ein Multiprozessor-Scheduling-Problem einfach die Länge der einzelnen Aufgaben zu einer dazugehörigen Dateigröße umwandeln und einen Zuweisungsproblem-Solver verwenden, um das Multiprozessor-Scheduling Problem zu lösen. Deswegen sollte man ein beliebiges Multiprozessor Scheduling Problem in polynomialer Zeit lösen können. Deshalb ist das Zuweisungsproblem ein NP-

vollständiges Problem, weil das Multiprozessor Scheduling Problem im strengen Sinne NP-vollständig für ein beliebiges *m* ist.

4.2 Prozessoranzahl

Obwohl das allgemeine Zuweisungsproblem für willkürliche Dateigrößen und für Dateien mit gleicher Größe, NP-Vollständig ist, kann es in polynomialer Zeit gelöst werden.

Es wird versucht, die Anzahl der Prozessoren zu finden, dass auf jede Datei zugreift. Um das herauszufinden, benutzt man das folgende einfache Modell für die I/O Zeit:

$$c_1 + c_2 * ((Datengröße)/(Prozessorzahl))$$

c_1 (entspricht den konstanten Kosten unabhängig von der Anfragengröße so wie die Suchzeit) und c_2 (entsprechend der Transferrate) sind systemabhängige Konstantenwerte. Mit der steigenden Prozessoranzahl wird in einer parallelen Umgebung die Menge der von jedem Prozessor gelesenen Daten reduziert.

Die gesamte Zeit für die Anfrage wird von dem Prozessor bestimmt für den die I/O Zeit Maximum ist. Wenn davon ausgehen wird, dass die zu lesende Datengröße(-menge) von jeder Datei gelesen wird und dass jeder Datei *k* Prozessoren zugewiesen werden, wird die Antwortzeit

$$m * c_1 + m * c_2 ((Datengröße)/(k)) \qquad \text{Gleichung 1}$$

sein. *m* ist die Anzahl der Dateien, dass dem langsamsten Prozessor zugewiesen ist. Wenn die Zuweisungen über die Prozessoren gleich verteilt wurde, dann ist

$$k = (m*p)/n$$

p ist die Anzahl der Prozessoren und *n* die Anzahl der Dateien. Nun kann Gleichung1 umgeschrieben werden als

$$m * c_1 + n/p * c_2 * Datengröße$$

Nachdem für jede Datei zum Schluss ein Prozessor zugewiesen ist, ist der kleinste Wert von *m (n/p).* Das bedeutet, dass nur ein Prozessor für jede Datei zugewiesen wird um die Antwortzeit jeder Datei zu minimieren. In der bisherigen (‚naiven') Strategie sind mehrere Prozessoren den Dateien zugewiesen, was die Antwortzeit verkleinert, in Anlehnung an unser Modell.

Wenn die Anzahl der Prozessoren größer als die Anzahl der Dateien ist, dann ist nur eine Datei jedem Prozessor zuzuweisen (umgekehrt ein Prozessors für jede Datei). Wenn die Dateienanzahl verschieden ist, dürfte das auf die Antwortzeit auswirken.

4.3 LP-Modell des Zuweisungsproblems

Es ist eine zwei dimensionale Matrix $R = [r_{i,j}]$ gegeben, so dass ein Eintrag $r_{i,j}$ in R die Menge der angefragten Daten vom Prozessor i aus der Datei j ergibt. In unserem ILP Formulierung möchten wir die Einträge der Matrix X = $[x_{i,j}]$ finden. Ein Eintrag $x_{i,j}$ besagt, ob der Prozessor i der Datei j ($x_{i,j}$ =1) zugewiesen oder nicht ($x_{i,j}$ = 0) zugewiesen ist. Die Prozessorauswahl für jede Datei wird nicht auf die I/O Zeit auswirken sondern auf die Kommunikationszeit. Im Folgenden entspricht n der Anzahl der Dateien auf die zugegriffen wird, und p der Prozessoranzahl. Es wird vorausgesetzt dass, $n > p$ ist.

$$\Sigma=_{i=1\ p}\Sigma_{j=1\ n}\ r_{i,j}\ x\ (1-x_{i,j})$$

Die Minimierung dieses Ausdrucks gibt uns die totale(gesamte) Menge der Daten an, die kommuniziert wurde. Wir werden die folgende Einschränkung für $x_{i,j}$ haben:

- $x_{i,j} \in \{0,1\}$

$x_{i,j}$ ist eine Entscheidungsvariable und sollte entweder zugeteilt oder nicht zugeteilt werden. Deswegen ist das LP Modell des Problem ein boolean oder null-eins (zero-one) Integer LP.

- $\Sigma^{p}_{i=1}\ x_{i,j} = 1$

Im vorherigen Abschnitt wurde rausgefunden, dass jeder Datei genau einem Prozessor zugeteilt wird. Der oben erwähnte Vergleich entspricht dieser Bedingung. Wenn die Anzahl der Prozessoren größer als die Anzahl der Dateien sind, bekommt diese Bedingung

- $\Sigma^{n}_{j=1}\ x_{i,j} = 1$

das Ergebnis, dass in jedem Prozessor nur einer Datei zugewiesen ist.

- $\Sigma^{n}_{j=1}\ x_{i,j} = n/p$

Diese Bedingung stellt sicher dass die Zuweisungen gleich verteilt sind. Das zeigt, dass die Anzahl der Dateien, die Prozessor i zugewiesen sind gleich n/p ist. Vorausgesetzt ohne Bezug auf die Datengröße die aus den Dateien gelesen wurde, die gleiche Anzahl der Dateien jedem Prozessor zugewiesen wird. Obwohl diese Annahme die Berechnung der Zuweisungsvariablen einfacher macht, dürfte es die Performance senken wenn die Varianz zwischen der gelesenen Datengröße groß ist. In diesem Fall kann man folgende Formel nutzen,

- $\Sigma^{n}_{j=1}(\Sigma^{p}_{k=1}\ r_{k,j})\ x\ x_{i,j} \leq P$

wo die gesamten zu lesenden Daten von jedem Prozessor annäherungsweise dasselbe sein sollte. Dabei entspricht P dem Durchschnitt der Datengröße auf die jeder Prozessor zugreifen

sollte. Wenn die Anzahl der Prozessoren die Anzahl der Dateien überschreitet, dann bekommt die Bedingung

- $\Sigma^{p}_{i=1} x_{i,j} = F_j$

F_j ist die Anzahl der Prozessoren, die der Datei j zugewiesen wird. Diese Anzahl kann durch Benutzen der Datenmenge die von der Datei j und von anderen Dateien gelesen wurden, leicht gefunden werden. Fj zu berechnen wird die folgende Formel benutzt:

- $F_k = p \text{ x } (\Sigma^{p}_{k=1} r_{k,j}) / (\Sigma^{p}_{i=1} \Sigma^{n}_{j=1} r_{i,j})$

p Prozessoren entsprechen der Menge der Datei j die verteilt wurden sind alle Daten zu lesen.

In diesem Model ist die Anzahl der Variablen gleich zu (*Anzahl der Prozessoren*) x (*Anzahl der Dateien*). Das oben erwähnten ZO-ILP Model hilft die Problemnatur zu verstehen. Aber es kann dieses Modell nicht in einer Laufzeitbibliothek benutzen um Prozessoren-Dateien Zuweisungen zu machen, da nämlich der schnellste LP-Solverextrem lange Laufzeit hat eine Lösung zu finden. Deswegen wurden zwei Heuristiken entwickelt das Problem zu lösen, die im nächsten Abschnitt vorgestellt werden.

5 Heuristiken für Multi-Collective I/O

Wie im vorherigen Kapitel beschrieben, ist das Zuweisungsproblem für Multi-Collective I/O NP-vollständig. Da (optimale) Lösungen hierfür eine ausgeprochen lange Laufzeit haben, liegt es nahe, diese anhand von Heuristiken anzunähern. Im Folgenden werden zwei solcher Heuristiken vorgestellt.

5.1 Greedy-Heuristik

Die Greedy-Heuristik basiert auf der Idee, die Zugriffe der Größe nach zu sortieren und dann den Prozessoren zuzuwiesen. Hierbei soll den Prozessoren jeweils die Datei zugewiesen werden, von der sie am meisten lesen müssen. Hierzu wird zunächst eine Liste erstellt, deren Einträge Tupel der Form (i, j, r_{ij}) sind. In diesen Tupeln steht i für einen Prozessor, j für eine Datei, und r_{ij} für die Datenmenge, die Prozessor i von Datei j lesen möchte. Nun wird diese Liste der Größe von r_{ij} nach sortiert.

Der Algorithmus für diese Heuristik lautet wie folgt (in Pseudocode):

```
GREEDY_ASSIGN (p, n, r_ij)
1       /* p is the number of processors, n is
2       the number of files, r_ij represents the
3       amount of data read by processor i from
4       file j. */
5       begin
6               for i=0 to p do
7                       for j=0 to n do
8                               if r_ij != 0 then
9                                       insert_entry(i, j, r_ij)
10                              end if
11                      end for
12              end for
13              sort_entries_according_to (last_field)
14              while (!list_empty) AND
15                              (assignments_made < n) do
16                      entry = list_top
17                      if entry.processor full OR
18                                      entry.file assigned then
19                              remove_entry (entry)
20                      else
21                              assign (entry.processor, entry.file)
22                              assignments_made++
23                      end if
24              end while
25              assign the remaining files to
26              remaining processors arbitrarily
27              return all_assignments
28      end.
```

Abbildung 5: Der Algorithmus für die Greedy-Heuristik [MCIO]

Im ersten Teil des Algorithmus (Zeile 6 bis 12) wird zunächst die (unsortierte) Liste erstellt und danach entsprechend der Anfragegrößen sortiert (Zeile 13). Im Folgenden wird versucht,

jeweils das erste Element der Liste zu nehmen. Wenn dem jeweiligen Prozessor sein Pensum an Dateien noch nicht zugewiesen wurde und die Datei noch keinem anderen Prozessor zugewiesen wurde, wird die jeweilige Datei dem Prozessor zugewiesen, anderfalls wird der Eintrag aus der Liste gelöscht. Das Pensum für jeden Prozessor ist die Anzahl der Dateien geteilt durch die Anzahl der Prozessoren, da jeder Prozessor gleich viele Dateien bearbeiten soll.

Die Schleife, in der dies geschieht, wird solang durchlaufen, bis entweder n Zuweisungen gemacht wurden (also jede Datei einem Prozessor zugewiesen wurde) oder die Liste mit den Anfragen leer ist (da Einträge aus der Liste gelöscht wurden). Wurden noch nicht sämtliche Dateien einem Prozessor zugewiesen (dies kann geschehen, wenn Einträge gelöscht wurden, weil der zugehörige Prozessor schon voll belegt war), werden diese Dateien willkürlich auf Prozessoren verteilt, die noch freie Kapazitäten haben. Solche Prozessoren müssen vorhanden sein, da jede Datei einem Prozessor zugewiesen weren muss. Sind also noch Dateien nicht zugewiesen, müssen auch dementsprechend Prozessoren existieren, die nicht voll belegt sind.

Die Funktionsweise des Algorithmus wird im folgenden Beispiel verdeutlicht. Abbildung 6 zeigt eine Verteilung von I/O-Anfragen von verschiedenen Prozessoren. Hierbei wird angegeben, welche Datenmenge der jeweilige Prozessor von einer Datei lesen will. In Abbildung 7 ist die nach dem ersten Teil des Algorithmus geordnete Liste zu finden (die Mengen sind hier der Übersicht halber weggelassen).

Processor Number	File Numbers							
	1	2	3	4	5	6	7	8
1	10 MB	12.8 MB	12.8 MB	4.4 MB	0 MB	0 MB	0 MB	0 MB
2	10 MB	12.8 MB	12.8 MB	4.4 MB	0 MB	0 MB	0 MB	0 MB
3	2.5 MB	3.2 MB	3.2 MB	1.1 MB	7.5 MB	9.6 MB	9.6 MB	3.3 MB
4	0 MB	0 MB	0 MB	0 MB	10 MB	12.8 MB	12.8 MB	4.4 MB

Abbildung 6: Verteilung von I/O-Anfragen [MCIO]

Access List
$P1 \rightarrow F2, P1 \rightarrow F3, P2 \rightarrow F2, P2 \rightarrow F3, P4 \rightarrow F6, P4 \rightarrow F7, P1 \rightarrow F1, P2 \rightarrow F1, P4 \rightarrow F5, P3 \rightarrow F6, P3 \rightarrow F7,$ $P3 \rightarrow F5, P1 \rightarrow F4, P2 \rightarrow F4, P4 \rightarrow F8, P3 \rightarrow F8, P3 \rightarrow F2, P3 \rightarrow F3, P3 \rightarrow F1, P3 \rightarrow F4$

Abbildung 7: Sortierte Liste der Anfragen [MCIO]

Aufgrund dieser Reihenfolge kommt der Algorithmus auf die im folgenden Bild gezeigten Zuweisungen:

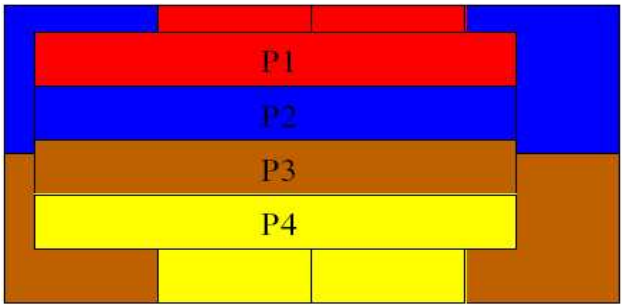

Abbildung 8: Prozessor/Dateizuweisungen aufgrund der Greedy-Heuristik [MCIO]

Prozessor 1 bearbeitet Datei 2 und 3, Prozessor 2 bearbeitet Datei 1 und 4, Datei 5 und 8 werden Prozessor 3 zugewiesen, und Prozessor 4 bearbeitet Datei 6 und 7.

5.2 Maximal Matching Heuristik

Eine weitere Möglichkeit, eine optimale Lösung anzunähern, ist die Maximal Matching Heuristik. Hier werden die I/O-Anfragen nicht als Liste, sondern als Graph repräsentiert. Für diesen Graphen lässt sich das Zuweisungsproblem anhand eines Maximal Matching Solvers lösen, der in diversen Bibliotheken zu finden ist. Die Entwickler von Multi-Collective I/O haben sich für den Solver entschieden, der in dem Netflow Solver Paket enthalten ist [Netflow].

Für einen Maximal Matching Solver wird ein bipartiter gewichteter Graph benötigt. Bipartite Graphen sind solche Graphen, die sich in zwei disjunkte Knotenmengen aufteilen lassen, wobei ausschließlich Kanten zwischen den beiden Partitionen existieren. Sei $G = (V_1 \cup V, E)$, dann gilt für jedes $<i, j> \in E$, dass $i \in V_1$ und $j \in V_2$. Nimmt man nun die Prozessoren und Dateien als Knoten, und erzeugt für jede I/O-Anfrage von Prozessor i auf Datei j eine Kante <i,j>, so ist diese Vorraussetzung gegeben.

Es sei an dieser Stelle angemerkt, dass der von den Entwicklern verwendete Solver nicht genau das Zuweisungsproblem löst, sondern das sogenannte Maximalflußproblem. Hier wird versucht, in einem bipartiten Graphen zu jedem Knoten aus V_1 eineindeutig einen (adjazenten) Knoten aus V_2 zu finden, so dass die Summe der Gewichte der Kanten, die diese Verbindungen darstellen, maximal ist. Aus der Vorraussetzung, dass die Zuweisung von Knoten eineindeutig sein muss, ergibt sich, dass die Anzahl der Knoten in V_1 der Anzahl derer in V_2 entsprechen muss. Anderfalls kann den "überschüssigen" Knoten der größeren Menge kein korrespondierender Knoten zugewisen werden. Da aber davon ausgegangen werden kann, dass in parallelen Anwendungen die Zahl der Prozessoren und Dateien stark voneinander abweicht, entsteht an dieser Stelle ein Problem. Dieses kann aber dadurch gelöst werden, dass die Knoten

der kleineren Menge repliziert werden und zwar F_j mal (wie sich F_j errechnet, wird in Kapitel 4.3 genauer erläutert). So kann es vorkommen, dass ein Prozessor mehrere Dateien bearbeiten muss bzw. dass eine Datei von mehreren Prozessoren bearbeitet wird.

Die Konstruktion dieses Graphen wird in Abbildung 9 erläutert (die Kantengewichte werden hier der Übersichtlichkeit halber nicht aufgeführt). Als Beispielanfragenverteilung wird dieselbe wie im Beispiel zur Greedy-Heuristik (Abbildung 6) zu Grunde gelegt.

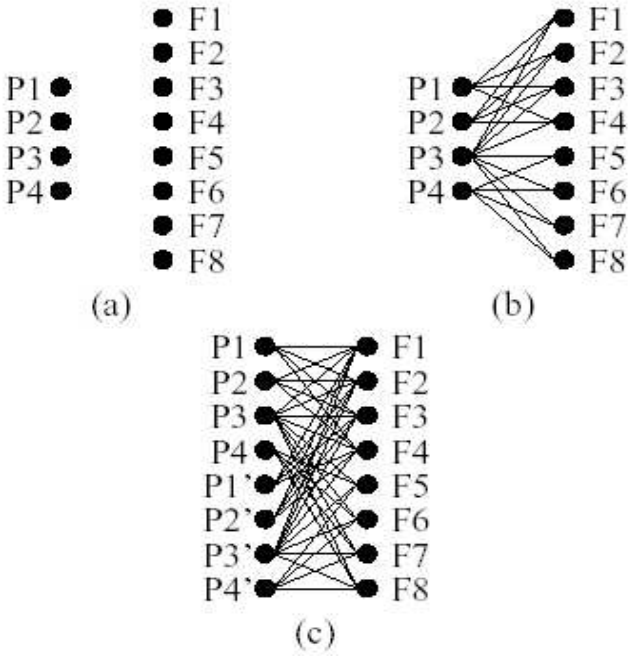

Abbildung 9: Modellierung des Zuweisungsproblems als Graph [MCIO]

Bild (a) zeigt die aus den Prozessoren und Dateien erzeugten Knoten. In Bild (b) wurden den Anfragen entsprechend Kanten hinzugefügt. Hier ist die Menge der Prozessoren noch wesentlich kleiner als die der Dateien. Der aus der Replikation entstehende, endgültige Graph ist in Bild (c) zu sehen. Hier ist auch deutlich zu erkennen, dass es sich um einen bipartiten Graph handelt.

Dieser so entstandene Graph wird dem Solver übergeben, der nun, wie erläutert, die Kantenmenge ermittelt, deren Summe der Kantengewichte maximal ist. Diese Kanten werden nun als Zuweisungen betrachtet, da sie Prozessoren und Dateien verbinden. Die aus dem Beispiel entehende Verteilung ist in Abbildung 10 zu sehen:

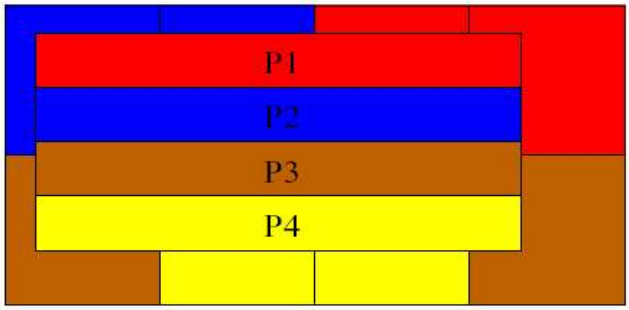

Abbildung 10: Prozessor/Dateizuweisungen nach der Maximal Matching-Heuristik [MCIO]

Vergleicht man dies mit der aus der Greedy-Heuristik entstandenen Verteilung (Abbildung 8), fällt auf, dass die Ergebnisse sehr ähnlich sind. Inwiefern sich der Unterschied bemerkbar macht, wird im folgenden Kapitel erläutert.

6 Multi-Collective I/O in der Anwendung

In diesem Kapitel werden die in den vorherigen Kapiteln beschriebenen Techniken zur Anwendung gebracht und mit den bisherigen Methoden (sprich: 'naives' CIO) verglichen. Hierzu haben die Entwickler sowohl künstlich erzeugte Zugriffsmuster verwendet, als auch die Bibliotheken ine eine wissenschaftliche, parallele Anwendung eingebunden.

Getestet wurde auf einer IBM SP-2 in den Argonne National Laboratories [Argonne], einer der führenden Forschungseinrichtungen im Bereich der Parallelen Systeme und Urheber von ROMIO, einem Multi-Collective I/O sehr ähnlichem CIO-System (vgl. Kapitel 2). Die Charakteristika der IBM SP-2 können der folgenden Tabelle entnommen werden:

Number of Processors	128 (120 compute nodes, 8 I/O nodes)
Processor Type	Compute Nodes: RS/6000 Model 370, I/O Nodes: RS/6000 Model 970
Clock Rate	332 MHz
L1 Cache	32 KB split, 2-way set-associative
L2 Cache	256 KB unified, 2-way set-associative
Memory Capacity	128 MB per compute node, 256 MB per I/O node
Network	100 Mbs Ethernet, 155 Mbs ATM and 800 Mbs HiPPI
Disk Space	9 GB per I/O node
Operating System	AIX 4.2.1
Parallel File System	PIOFS

Abbildung 11: Systemcharakteristika der IBM SP-2 [MCIO]

Als MPI-Distribution wurde bei beiden Anwendungen die MPI-2 Bibliothek [MPI-2] verwendet. Da MCIO ähnlich wie ROMIO arbeitet, sollte eine Portierung in andere MPI-Distributionen allerdings ohne großen Aufwand möglich sein. Die Aufrufe von MCIO ähneln denen von MPI-IO [MPI-IO] sehr stark. Zum Beispiel sieht ein *read_all*-Aufruf in MCIO wie folgt aus:

```
int MCIO_File_read_all (MPI_File *fh, int filecount, void **buf, int *count, MPI_Datatype *datatype, MPI_Status *status)
```

Der entsprechende MPI_IO-Aufruf lautet:

```
int MPI_File_read_all (MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status status)
```

Wie sofort auffällt, unterscheiden sich die Aufrufe lediglich darin, dass bei MCIO Felder anstatt 'einzelner' Werte übergeben werden (da ja von mehreren Dateien gelesen werden soll), und dass zusätzlich in der Variable `filecount` angegeben wird, von wievielen Dateien gelesen werden soll. Somit bündelt ein MCIO-Aufruf mehrere MPI-IO-Aufrufe, nämlich genau so viele, wie Dateien, von denen gelesen werden soll.

6.1 Künstlich erzeugte Zugriffsmuster

Für die künstlich erzeugten Zugriffsmuster wurden zwei Arten von Tests durchgeführt. In beiden Tests enthielten die Dateien Fließkommazahlen-Arrays der Größe 1024 x 2048 Byte. Der Unterschied in den Tests bestand in der Art, in der die Prozessoren auf die Daten zugreifen möchten. Im ersten Fall waren die Dateien zeilenweise verteilt, im zweiten Fall spaltenweise, d.h. jeder Prozessor greift auf konsekutive Reihen bzw. Spalten des jeweiligen Arrays zu. Die Tests wurden mit verschiedenen Mengen an Prozessoren und Dateien durchgeführt.

Die Entwickler haben für die Tests nicht nur die Greedy- und die Maximal Matching-Heuristik, sondern auch das in Kapitel XY vorgestellte LP-Modell evaluiert. Da letzteres wesentlich mehr Zeit benötigt, um die Zuweisungen zu machen, legt dies nahe, dass die Zuweisungen vorher berechnet wurden bzw. die Zeitmessung lediglich für die reine I/O- und Kommunikationszeit getätigt wurde. Dies ist von den Entwicklern jedoch nicht weiter kommentiert.

Es ist herauszuheben, dass alle drei Methoden für die hier angewandeten Anfragemengen exakt die gleichen Zuweisungen machten, was darauf zurückzuführen ist, dass alle Anfragen die selbe Größe hatten. Daher wird bei den Ergebnissen nicht unterschieden, mit Hilfe welcher Methode sie letztendlich erzeugt wurden.

Allegemein ist festzustellen, dass sämtliche Methoden sich lediglich in der Berechnungs- und Kommunikationszeit, also dem Overhead, unterscheiden, nicht jedoch in der Zeit, die für den eigentlichen I/O-Vorgang benötigt wird. Dies liegt daran, dass die benötigte Zeit für den I/O-Vorgang ausschließlich von der Anfragemenge abhängt. Die von jeder Datei gelesene (bzw. auf sie geschriebene) Datenmenge bleibt immer gleich; durch die Zuweisungen wird lediglich festgelegt, welcher Prozessor diese Daten liest bzw. schreibt. Im Folgenden sollen daher die Vor- und Nachteile der einzelnen Methoden diskutiert werden.

Da der Berechnungs-Overhead für das LP-Modell völlig inakzeptabel ist, sollte es nie verwendet werden. Die Greedy-Heuristik kommt wesentlich schneller als die Maximal Matching-Heuristik zu einem Zuweisungsergebnis, während die Zuweisungen der Maximal Matching-Heuristik immer eine bessere Kommunikationszeit nach sich ziehen, vorrausgesetzt, die Anfragen unterscheiden sich stark (da sonst die Greedy-Heuristik zur selben Zuweisungsverteilung kommt). Daher ist es anzuraten, in Fällen, in denen die Anfragen gleich groß oder sehr ähnlich sind, die Greedy-Heuristik zu verwenden, in allen anderen Fällen ist die Maximal Matching-Heuristik zu bevorzugen.

Die Ergebnisse für die Tests sind in den Abbildungen 12 und 13zu finden:

Number of Files	Number of Processors		
	4	8	16
4	49.97	51.22	49.97
8	52.28	78.33	77.41
16	53.42	80.12	86.67
32	54.33	80.52	87.18

Abbildung 12: Ergebnisse für zeilenlastige Zugriffsmuster [MCIO]

Number of Files	Number of Processors		
	4	8	16
4	41.26	55.76	49.04
8	43.90	76.49	79.26
16	45.21	77.59	80.28
32	45.84	78.07	80.82

Abbildung 13: Ergebnisse für spaltenlastige Zugriffsmuster [MCIO]

Die Abbildungen stellen die Verbesserung zur 'naiven' CIO in Prozent dar. Wie leicht zu erkennen ist, erzielten die Test mit zeilenweisem Zugriff leicht bessere Ergebnisse. Es ist anzumerken, dass sich die Tests lediglich auf Lesevorgänge beziehen. Zu Schreibvorgängen wurde keine Aussage gemacht, es ist jedoch anzunehmen, dass ähnliche Ergebnisse erzielt würden.

Die Performance von MCIO ist, wie zu erkennen, deutlich besser im Vergleich zu naivem CIO, wobei festzustellen ist, dass bei Anstieg der Anzahl von sowohl Prozessoren als auch Dateien die Performance auch steigt. Bei je vier Prozessoren und Dateien beträgt der Performance-Zuwachs etwa 50 Prozent für zeilenweisen und etwa 41 Prozent für spaltenweisen Zugriff; bei 16 Prozessoren und 32 Dateien sogar etwa 87 beziehungsweise 81 Prozent. Allerdings fällt auf, dass, sobald die Anzahl der Prozessoren die Anzahl der Dateien (stark) übersteigt, die Performancesteigerung stagniert oder sogar leicht einbricht. Dies ist darin zu ergründen, dass so mehrere Prozessoren jeweils auf die selbe Datei zugreifen müssen, was einen zusätzlichen Synchronisations-Overhead mit sich bringt.

6.2 Einsatz in einer realen Anwendung

Die Anwendung, mit der MCIO zusätzlich zu den künstlichen Zugriffsmustern getestet wurde, ist *astro3d* [Astro3d]. Astro3d ist eine in Java geschriebene parallele Anwendung aus der Astrophysik, die in VRML-Objekte einliest und darstellt. Hierbei werden sechs verschiedene Werte aus einer Datei gelesen. Dies würde normalerweise bedeuten, dass sechs verschiedene MPI-IO-Aufrufe getätugt werden müssen. Mittels MCIO lässt sich dies in einem einzigem Aufruf zusammenfassen. Die Ergebnisse dieser Änderung sind Abbildung 14 zu entnehmen:

	4 processors	8 processors
Collective I/O	3.33	3.51
MCIO	2.04	1.30

Abbildung 14: Ergebnisse für Tests mit einer realen Anwendung [MCIO]

7 Zusammenfassung

Die starke Zunahme von parallelen Systemen sowie Anwendungen für diese stellen I/O-System vor völlig neue Herausforderungen. Während I/O schon immer den größten Flaschenhals in der Datenverarbeitung dargestellt hat, kommt dies bei parallelen Anwendungen noch stärker zum Tragen. Ein sehr erfolgreicher Ansatz zur Verbesserung der Performance ist Collective I/O, bei dem I/O-Anfragen von mehreren Prozessoren gesammelt und gebündelt gelesen werden, um Overhead zu vermeiden. Das hier vorgestellte Multi-Collective I/O treibt diesen Ansatz noch weiter, indem es die Anfragen nicht auf einem Prozessor, sondern auf mehreren Prozessoren bündelt, um so einen noch höheren Durchsatz zu erlangen.

Durch die Tatsache, dass die Bündelung der Anfragen auf mehrere Prozessoren verteilt wird, und diese Verteilung möglichst gleichmäßig sein soll, ensteht das sogenannte "Zuweisungsproblem". Hier wird versucht, ein optimales Muster für die Verteilung zu finden. Die Berechnung dieser Verteilung ist NP-vollständig. Aus diesem Grund wurden zwei Heuristiken entwickelt, die eine optimale Lösung annähern.

Die erste Heuristik, Greedy-Heuristik genannt, versucht durch einen - wie der Name schon sagt – Greedy-Ansatz die Dateien den Prozessoren zuzuweisen. Hierbei wird die Menge der Anfragen sortiert und nacheinander den Prozessoren zugewiesen. Die zweite Methode ist die Maximal Matching-Heuristik. Hier wird das Zuweisungsproblem in die Graphentheorie übertragen, indem aus den Prozessoren, Dateien und Anfragen ein bipartiter Graph konstruiert wird und mit spezielle Bibliotheken für diesen das sogenannte Maximalfluß-Problem gelöst wird. Die Lösung dessen kann als Zuweisungsmenge interpretiert werden.

In Test wurde festgestellt, dass die Greedy-Heuristik schneller eine Lösung findet, die Lösungen der Maximal Matching-Heuristik in Bezug auf den Kommunikationsoverhead aber besser sind (vorrausgesetzt, es besteht eine große Diversität in der Anfragemenge). Unabhängig davon, welche Heuristik verwendet wurde, ergaben die Tests, dass Multi-Collective I/O im Vergleich zu "naivem" Collective I/O zwichen 40 und 87 Prozent bessere Performance erreicht, abhängig von Anzahl der Prozessoren und Dateien. Hierbei steigt die Performance mit der Anzahl der Dateien und Prozessoren.

8 Fazit

Aufgrund der Testergebnisse lässt sich sagen, dass MCIO augenscheinlich eine signifikante Verbesserung zu 'naivem' CIO darstellt. Allerdings war nicht in Erfahrung zu bringen, ob sich MCIO aufgrund dieser Ergebnisse durchgesetzt hat oder zumindest in einige MPI-Distributionen eingebunden wurde. Die im Paper erwähnte Laufzeitbibliothek ist nicht zum Download verfügbar, auch finden sich neben [MCIO] keine weiteren Publikationen zu dem Thema.

Zudem planten die Autoren, die Effektivität von Multi-Collective I/O in einem Sub-Datei-basierten System zu testen. Des weitern war geplant, ein MCIO integrierendes Compiler-Framework zu entwickeln, um entsprechende Entwickler die Details des Dateisystems und des Zugriffes darauf zu ersparen. Allerdings war auch hier nicht in Erfahrung zu bringen, in wie weit diese Ansätze weiter verfolgt wurden.

Auch wenn die Testergebnisse sehr vielversprechend waren und die Funktionsweise von MCIO sehr genau erläutert wurde, blieben trotzdem noch einige Fragen offen. So wurde nicht darauf eingegangen, wie MCIO letztendlich auf das zugrundeliegende Dateisystem zugreift. Aufgrund der Nähe zu ROMIO [ROMIO] ist zu vermuten, dass auch hier für den konkreten Zugriff ADIO [ADIO] verwendet wurde. Allerdings könnte auch ein direkter Zugriff implementiert worden sein, was eine schlechte Portierbarkeit zur Folge hätte. Dies könnte die fehlende Verbreitung erklären.

In den Tests fehlen leider auch Ergebnisse für Schreibzugriffe, es wurden lediglich Leseoperationen untersucht bzw. präsentiert.

Des weiteren wurden bei den Tests lediglich bis zu 16 der 128 möglichen Prozessoren verwendet. Somit lässt sich leider keine faktenbasierte Aussage zur Skalierbarkeit der Anwendung machen. Allerdings steht aufgrund der Tatsache, dass sich die Performance leicht verschlechterte, sobald die Anzahl der Prozessoren signifikant größer war als die der Dateien (Faktor 4), zu vermuten, dass die Leistung bei einer weiteren Steigerung der Prozessorenanzahl weiter eingebrochen wäre. Dies ist, wie bereits in Kapitel 6.1 erwähnt, in der nötigen Synchronisation bei gleichzeitigem Zugriff auf eine Datei zu begründen.

Trotz der o.g. Einschränkungen steht zu vermuten, dass MCIO für kleine oder I/O-lastige parallele Anwendungen signifikante Verbesserungen der Performance bewirken kann.

Literaturverzeichnis

[ADIO] Thakur, R. et. al.: *An Abstract-Device Interface for Implementing Portable Parallel-I/O Intefaces.* In: Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation, Argonne, IL, 1996

[Argonne] Argonne National Laboratories, http://www.anl.gov

[Astro3d] Malagoli, A.: *ASTRO3D – 2.0.*The ASCI Flash Center, Chicago, IL, September 1998

[Bagrodia] Bagrodia, R., Chien, A., Hsu, Y., Reed, D.: *Input/ouput: Instrumentation, Characterization, Modeling and Management Policy.* Technical Report CCSF-41, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994

[Boru] Baru, S.: *Storage Ressource Broke (SRB) Reference Manual.* Enabling Technologies Group, San Diego Supercomputer Center, La Jolla, Ca, August 1997

[Coyne] Coyne, R. A., Hulen, H., Watson, R.: *The high performance storage system.* In: Proceedings of Supercomputing '93, Portland, OR, November 1993

[Crandall] Crandall, P., Aydt, R., Chien, A., Reed, D.: *Input/output characteristics of scalable parallel applications.* In: Proceedings of Supercomputing '95, Dezember 1995

[Cypher] Cypher, R., Ho, A., Konstantinidou, S., Messina, P.: *Architectural Requirements of Parallel Scientific Applications with Explicit Communication.* In: Proceedings of the 20th International Symposium an Computer Architecture, 1993, Seiten 2 - 13

[Garey] Garey, M. R., Johnson, D. S.: *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979

[MCIO] Memik, G., Kandemir, M., Choudhary, A.: *Exploiting Inter-File Access Patterns Using Multi-Collective I/O.* In: Proceedings of the FAST '02 Conference on File and Storage Technologies, Monterey, CA, 2002, Seiten 245 - 258

[MPI-2] Message Passing Interface Forum (Hrsg.): *MPI-2: Extension to the message passing interface.* Technical Report, University of Tennessee, Juli 1997

[MPI-IO] Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, B., Prost, J., Snir, M., Traversat, B., Wong, P.: *Overview of the MPI-IP parallel I/O interface.* In: Proceedings of the Third Workshop on I/O in Parallel and Distributed Systems, IPPS '95, Santa Barbara, CA, April 1995

[Netflow] Johnson, D., McGeoch, C. (Hrsg.): *Network Flows and Matching.* American Mathematical Society, Providence, RI, 1993

[Nieuwejaar] Nieuwejaar, N., Kotz, D., Purakayastha, A., Ellis, C., Best, M.: *File-Access Characteristics of Parallel Scientific Workloads.* In: IEEE Transactions on Parallel and Distributed Systems, Oktober 1996 (Band 7, Nr. 10), Seiten 1075 – 1089

[ROMIO] Thakur, R., Lusk, E., Gropp, W.: *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation.* Technical Memorandum ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratories, IL, Juli 1998

Abbildungsverzeichnis