Fachhochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Parallele Systeme
Prof. Dr. Rudolf Berrendorf

Input/Output Characteristics of Scalable Parallel Applications

Eine Zusammenfassung

Autor:
Nilson Reyes und Jens Mahnke
Matrikel: 9002560, 9003153

Datum: 25.06.2005

Inhaltsverzeichnis

Inhaltsverzeichnis 2

1 Einführung (NR) 3

2 Grundlagen (NR) 4

2.1 Arten von Speicherzugriffen 4

2.2 Experimentelle Methodik 5

2.2.1 Vorgehen 5

2.2.2 Eingesetztes Testsystem 6

3 Problemstellungen (NR) 8

3.1 Electron Scattering ESCAT 8

3.2 Terrain Rendering RENDER 9

3.3 Hartree Fock HTF 9

4 Analyse 11

4.1 ESCAT (NR) 11

4.2 RENDER (JM) 13

4.3 HTF (JM) 15

4.4 Diskussion der Ergebnisse (JM) 18

5 Fazit (JM) 20

6 Literaturangaben 22

1 Einführung (NR)

Die vorliegende Ausarbeitung beschäftigt sich im Rahmen der Vorlesung „Parallele System“ von Herrn Prof. Rudolf Berrendorf mit der Zusammenfassung einer Arbeit von Crandall, Aydt, Chien und Reed vom Department of Computer Science von der Universität von Illinois mit dem Titel Input/Output Characteristics of Scalable Parallel Applications.

Die zusammengefasste Arbeit behandelt Muster von Speicherzugriffen in parallelen Systemen. Dieses Thema ist für die Leistung von parallelen Systemen von großer Bedeutung, da das Wachstum an Leistung der heutigen Prozessoren um ein vielfaches größer ist als das Wachstum der Leistung im Bereich von Speichern. Dies kommt daher, dass die Nutzer von Speicher eher an Kapazität statt an Leistung interessiert sind und deswegen in diesem Bereich kein vergleichbarer Aufwand betrieben wird, verglichen mit dem Forschungsaufwand bei der Prozessorleistung. Daher sind Speicherzugriffe im Vergleich mit Berechnungen sehr kostenintensiv und müssen so effizient wie möglich eingesetzt werden.

Die Autoren untersuchen in ihrer Arbeit die Speicherzugriffe drei verschiedener Programme auf parallelen Systemen. Dabei wird untersucht, ob die Zugriffe bestimmten Mustern folgen und damit vorhersagbar sind und wie mithilfe des beobachteten Verhaltens eine Verbesserung der Zugriffe erzielt werden kann. Im Zentrum steht aber weniger die Verbesserung, als die Analyse des Zugriffsverhaltens der einzelnen Programme.

Im Folgenden wird eine kurze Einführung in die Thematik und die Methodik der Autoren gegeben (siehe Kapitel 2) anschließend werden die Probleme vorgestellt, die von den einzelnen Programmen gelöst werden (siehe Kapitel 3) und danach werden die einzelnen Programme auf ihre Zugriffsmuster hin analysiert und es wird versucht Verbesserungsmöglichkeiten aufzuzeigen (siehe Kapitel 4). Zum Schluss werden die Ergebnisse diskutiert und in einem Fazit zusammengefasst (siehe Kapitel 4.4 und 5).

2 Grundlagen (NR)

Dieses Kapitel soll kurz eine Einführung in verschiedene Arten von Speicherzugriffen zur besseren Klassifizierung in der Analysephase geben und anschließend zeigen mit welchem Werkzeug auf welchem System unter Verwendung welcher Methodik die Autoren die Programme analysiert haben.

2.1 Arten von Speicherzugriffen

Folgende Arten von Speicherzugriffen werden nach dem Dokument grundsätzlich unterschieden:

- **Compulsory access**: Diese Art des Zugriffs entsteht durch das Auslesen von Initialisierungsdaten, dem Schreiben von Ausgaben des Programms, etc. Diese Art der Speicherzugriffe kann nicht eliminiert werden, da sie notwendig zum Funktionieren des jeweiligen Programms ist. Ohne Ein/Ausgaben ist schließlich keine Berechnung im Sinne einer Funktion möglich.
- **Checkpoints**: Checkpoint Daten sind Daten, die an bestimmten Stellen des Programms mit einer gewissen Regelmäßigkeit abgelegt werden, um später wieder an dieser Stelle mit den Berechnungen fortzufahren oder den Programmverlauf zu protokollieren. Unter Umständen werden die Daten verändert um zu erfahren welchen Einfluss diese Veränderung auf den Rest der Berechnung hat. Checkpoint Daten können auch als Backupdaten genutzt werden, wenn beispielsweise das Programm unerwartet abbricht. Nach Meinung der Autoren können solche Zugriffe von intelligenten Speichersystemen unterstützt werden, da diese Zugriffe einem gewissen Muster folgen müssen (periodisch).
- **Out-of-Core input/output**: Diese Speicherzugriffe entstehen immer dann, wenn auf den Sekundärspeicher zugegriffen werden muss, weil der primäre Speicher die benötigten Daten nicht enthält. Viele Problemstellungen besitzen Datenstrukturen, die zu groß sind für heutige Primär-

speicher. Der Speicher ist zwar zu vergrößern, allerdings ist dies ökonomisch nicht immer sinnvoll.

Mithilfe dieser Klassifizierungen wurden in der Vergangenheit bereits einige File Access Patterns entwickelt, die Autoren stellen sich allerdings die Frage, ob diese Patterns bedingt sind durch die Softwaresysteme auf denen sie beobachten wurden oder ob sie wie gewünscht direkt von den Applikationen erzeugt wurden. Aus diesem Grund sehen die Autoren einen Handlungsbedarf bei der Erstellung von aktuellen Access File Patterns.

2.2 Experimentelle Methodik

Dieses Kapitel erläutert zunächst das grundlegende wissenschaftliche Vorgehen der Autoren bei der Analyse und stellt auch das eingesetzte Werkzeug „Pablo“ kurz dar. Anschließend wird das Rechnersystem betrachtet auf dem die Tests gefahren wurden.

2.2.1 Vorgehen

Speicherzugriffe werden aus Programmen angestoßen und lösen als höchste Instanz einen Zugriff auf das Speichersystem aus. Ein Speichersystem hat die Aufgabe die Zahl der Speicherzugriffe zu minimieren und deren Effizienz zu maximieren. Aus diesem Grund haben die Autoren zum Ziel die Speicheraufrufe aus Applikationen zu protokollieren, um diese später auszuwerten und eventuell Verbesserungen vorzuschlagen.

Zur Durchführung dieser Aufgabe wird das Pablo Performance Environment eingesetzt. Pablo unterstützt folgende Aufgaben:

- Pablo zeichnet Speicherzugriffe mit benötigter Zeit und Ort der Daten auf. Die gesammelten Daten werden automatisch kompressiert. Pablo ist in der Lage festzuhalten wie lange eine Datei gelesen, geschrieben, gesucht, geöffnet oder geschlossen wurde und auf wie viele Bytes zugegriffen wurde.
- Pablo verfügt über eine graphische Anzeigefunktion der aufgezeichneten Ergebnisse zur Unterstützung bei der Analyse.

Mithilfe von Pablo untersuchten die Autoren die einzelnen Applikationen auf ihr Speicherverhalten. Dieses Speicherverhalten wird in Beziehung gesetzt mit der Aufgabe, die die Software zu diesem Zeitpunkt erfüllt.

2.2.2 Eingesetztes Testsystem

Die Tests wurden auf dem Intel Paragon XP/S an der Caltech Concurrent Supercomputing Facility durchgeführt. Das System besteht aus folgenden Komponenten:

- 512 computation nodes
- 16 I/O nodes
    - Raid-3 disk array mit 5 mal 1.2 GB/s Festplatten
- Betriebssystem: Intel OSF/1 1.2 mit PFS (Intel's Parallel File System)

PFS unterstützt folgende parallele Zugriffsmodi (auf diese Modi wird in den späteren Analysen referenziert):

- M_UNIX: jeder Knoten hat einen eigenen Pointer auf eine Datei.
- M_RECORD: wie oben, Zugang wird nach first come first serve geregelt und die Länge der Operationen ist festgelegt.
- M_ASYNC: wie M_UNIX, Zugang auf eine Datei ist nicht festgelegt, Atomizität von Operationen wird nicht garantiert.
- M_LOG: alle Knoten teilen einen gemeinsamen Zeiger auf eine Datei, Zugang wird nach first come first serve geregelt, die Länge der Operationen ist nicht festgelegt.
- M_SYNC: alle Knoten teilen einen gemeinsamen Zeiger auf eine Datei, der Zugang auf eine Datei wird der Knotennummer nach gewährt
- M_GLOBAL: alle Knoten haben einen gemeinsamen Zeiger und führen dieselben Operationen aus.

Die häufigste Variante ist M_UNIX, da damit die Parallelität am Besten ausgenutzt werden kann. Im Folgenden werden wir feststellen, dass kaum Modi benutzt werden, um von allen Knoten aus dieselbe Datei anzufragen. Es wird sich

zeigen, dass es effizienter ist, wenn nur ein Knoten die Datei liest und diese an die anderen Knoten per Netzwerk verteilt.

3 Problemstellungen (NR)

Im Folgenden werden die Probleme, die den Applikationen zugrunde liegen, vorgestellt. Es wird dabei eingegangen auf den grundsätzlichen Aufbau des Codes, der sich daraus ergibt.

3.1 Electron Scattering ESCAT

Mit Electron Scattering wird das Streuungsverhalten von Elektronen bei Kollisionen von Molekülen bei kleiner Energie (<5eV) bezeichnet. Die Lösung dieses Problems ist wichtig im Zusammenhang mit Luft/Raumfahrt-Technologie und anderen Hightech Forschungsgebieten. Das Problem wird gelöst mithilfe der Schwinger Multichannel Methode. Da die Lösung des Problems nicht für unser Ziel, nämlich das Untersuchen von Speicherzugriffen, von Belang ist und außerdem das Verständnis der Lösung den Rahmen dieser Zusammenfassung verlassen würde, wird an dieser Stelle nur die Arbeitsweise des Algorithmus untersucht der letztendlich in Quellcode formuliert später auf seine Speicherzugriffsmuster untersucht wird.

Das Programm ist in C, Fortran und Assembler geschrieben. Es besteht aus vier Ausführungsschritten:

1. Im ersten Schritt wird mit einem *compulsary read* (siehe Kapitel 2.1) die Problemdefinition in Form einiger Matrizen geladen.
2. Im nächsten Schritt berechnet jeder Knoten eine gewisse Anzahl von Integralen. Dieser Schritt wird parallel von allen Knoten ausgeführt und ist rechenintensiv. Nach der Berechnung eines jeden Integrals wird die Lösung auf den Speicher geschrieben (synchron zwischen allen Knoten)
3. In der nächsten Phase werden aus den Integralen lineare Gleichungen ermittelt.
4. In Phase 4 werden die in 3. ermittelten Gleichungen auf dem Speicher abgelegt.

3.2 Terrain Rendering RENDER

Bei RENDER geht es darum aus NASA fly-by Daten und Daten über die Höhenunterschiede auf Planeten dreidimensionale Bilddaten dieser Planeten zu erzeugen. Beim hintereinander Zeigen dieser Daten kann ein virtueller Rundflug auf einem Planeten simuliert werden. Daten für solche Rundflüge sind zum Beispiel für Mars und Venus verfügbar.

Der Code, der diesen Rundflug berechnet arbeitet in etwa wie folgt:

- Ein einzelner Knoten steuert den gesamten Rendering Prozess und beginnt mit dem Lesen der Initialisierungsdaten.
- Für jeden Frame wird ein read-render-write Zyklus ausgeführt. Dazu liest der Knoten eins die zu rendernden Daten, verteilt diese an die einzelnen parallelen Knoten und erhält die gerenderten Bilder zurück. Diese Bilder legt er dann auf dem Speicher ab.

3.3 Hartree Fock HTF

Bei HTF wird versucht chemische Reaktionen ohne Experiment vorherzusagen und dessen Eigenschaften zu bestimmen. Diese Vorgehensweise bietet sich an, um Experimente zu simulieren, die zu teuer oder zu gefährlich sind, um tatsächlich durchgeführt zu werden. HTF ist eine Untergruppe der sogenannten „ab initio“ Methoden der theoretischen Chemie. Die theoretische Chemie beschäftigt sich im Gegensatz zur experimentellen Chemie mit der Berechnung von Reaktionen, statt die Experimente tatsächlich durchzuführen. Auch hier ist ein tieferes Verständnis von HTF für das Ziel dieser Ausarbeitung nicht notwendig, es wird stattdessen auf die Arbeitsweise des Algorithmus eingegangen, der HTF löst:

Als Ausgangsbasis werden im ersten Schritt Daten geladen, die Informationen, wie die Geometrie der involvierten Atome, enthalten. Als nächstes werden Integrale berechnet, um die molekulare Dichte zu approximieren. Daraus wird dann eine so genannte Fock Matrix gebildet. Zum Schluss wird die SCF (self consistent field) Methode angewandt, bis die Dichten der Moleküle ausreichend genau sind. Das zugehörige parallele Programm (FORTRAN) besteht aus drei Bestandteilen:

- Psetup: liest Initialisierungsdaten, transformiert diese und schreibt das Ergebnis in den Speicher.
- Pargos: Benutzt die Ergebnisse von Psetup und berechnet Integrale, die dann anschließend wieder auf das Speichermedium geschrieben werden.
- Pscf: Führt die SCF Methode aus, indem die Integrale von Pargos mehrfach gelesen werden, da sie zu groß für den Speicher sind und löst letztendlich die SCF Gleichungen.

4 Analyse

Die vorgestellten Programme werden nun auf ihre Speicherzugriffe hin analysiert.

4.1 ESCAT (NR)

Der in Kapitel 3.1 beschriebene Code, um ESCAT zu berechnen, wurde exemplarisch auf 128 Knoten ausgeführt. Die Eingabegrößen wurden so gewählt, dass die Programmdauer nicht länger als 1,75 Stunden betrug. Auf einem „echten“ System würden 500 Knoten um die 20 Stunden zur Berechnung benötigen. Die vier beschriebenen Phasen wurden wie folgt ausgeführt:

In Phase 1 werden die Daten von einem Knoten (Knoten null) gelesen und an alle anderen per broadcast über das Netzwerk verteilt. In der zweiten Phase berechnen und schreiben (M_UNIX) die Knoten mehrfach Integrale. Die Größe dieser Ergebnisse beträgt jeweils 2K (von je einem Integral). Danach werden die Daten wieder von allen Knoten gelesen (M_RECORD), weitere Berechnungen durchgeführt und wieder an Knoten null gesendet, der die Endergebnisse auf den Speicher schreibt.

Operation	Operation Count	Volume (Bytes)	Node Time (Seconds)	Percentage I/O Time
All I/O	26,418	60,983,136	38,788.95	100.00
Read	560	34,226,048	81.19	0.21
Write	13,330	26,757,088	16,268.50	41.94
Seek	12,034	-	20,884.11	53.84
Open	262	-	1179.06	3.04
Close	262	-	376.06	0.97

Tabelle 1: Anzahl, Größe und Dauer der I/O Operationen in ESCAT, aus [Crandall]

Operation	Operation Size			
	< 4 KB	< 64 KB	< 256 KB	≥ 256 KB
Read	297	3	260	0
Write	13,330	0	0	0

Tabelle 2: Schreib/Lese Größen für ESCAT, aus [Crandall]

Tabelle 1 zeigt ein kontroverses Bild: Obwohl die meisten Daten gelesen werden, verbraucht der Lesevorgang selber nur 0.21% der gesamten Zeit. Das Schreiben hingegen verbraucht 41.94% der I/O Zeit. In Tabelle 2 wird die Erklärung für diese ungewöhnliche Verteilung gegeben. Währende die Schreibzugriffe zwischen kleinen und großen Dateigrößen variieren, werden nur ausnahmslos kleine Daten geschrieben (2K). Dies verursacht natürlich eine hohe Ineffizienz. Weiterhin fällt auf, dass das System viel Zeit damit verbringt Daten zu suchen (seek – 53.84%). Die Autoren sind daher der Meinung, dass noch weitere Lese/Schreibmodi benötigt werden, die es erlauben selbstgeschriebene Daten schneller wieder zu finden. Dies wäre hier von Vorteil, da die Knoten nur die Daten lesen, die sie zuvor berechnet haben, das System (PFS) aber keinen Vorteil daraus gewinnen kann.

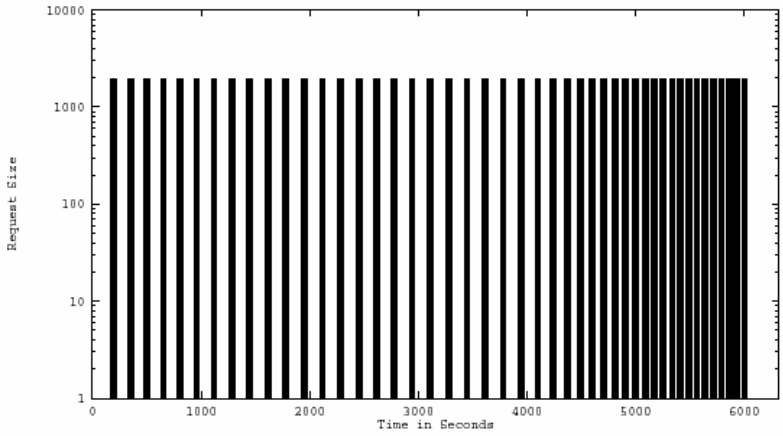

Abbildung 1: Schreibverhalten von ESCAT, aus [Crandall]

Abbildung 1 verdeutlicht noch einmal die Schwäche des Codes: Es werden immer wieder kleine Daten (2k) in den Speicher geschrieben. Die Autoren schlagen vor, die Schreibaufrufe zu sammeln und in einem Aufruf auf den Speicher zu schreiben. Dies könnte die Schreibkosten erheblich senken und das Verhalten in Abb.1 drastisch reduzieren.

Ein weiterer interessanter Punkt ist, dass die Programmierer es vorzogen die Initialisierungsdaten von einem Knoten lesen zu lassen und anschließend an die anderen Knoten per Netzwerk zu versenden, statt von jedem Knoten die

Daten auslesen zu lassen (beispielsweise mit einem gemeinsamen Speicher). Die eingesetzte Variante ist tatsächlich schneller und zeigt noch einmal, dass das Speichersystem in vielen Fällen der Flaschenhals für die Applikation sein kann.

4.2 RENDER (JM)

Der in Kapitel 3.2 vorgestellte Algorithmus zur Visualisierung von NASA Fly-by Daten wurde zur Untersuchung des I/O Verhaltens mit einem kompletten Set an Produktionsdaten durchgeführt. Dafür wurden die Daten eines Vorbeiflugs der Viking Sonde am Mars genutzt, wobei aber zur Laufzeitlimitierung auf eine Beschränkung der Frames zurückgegriffen wurde. Bei der Berechnung erzeugt der RENDER Code periodische die Frames, die an nahezu gleichen Zeitintervallen und mit gleicher Größe gespeichert werden. Die Initialisierung in der Testumgebung mit 128 Knoten benötigte acht Minuten und gab nach der Berechnung 100 Frames aus, wobei die produktive Laufzeit cirka 30 Minuten benötigt und dabei 5000 oder mehr Frames erzeugt werden. In der produktiven Umgebung werden die Daten nicht wie beim Test mit M_UNIX auf einer Festplatte abgelegt, sondern auf einem HiPPi Frame Buffer gespeichert.

Die dominanten I/O Operationen beschränken sich auf den Initialen Lesevorgang, in dem cirka ein Gigabyte Daten von einem Knoten gelesen werden (M_UNIX) und an die anderen Knoten per Broadcast versendet werden. In der Berechnungsphase beschränken sich die I/O Zugriffe auf das Lesen von Konfigurationsdateien mit Ansichts-Koordinaten. Als Ergebnis der Berechnung schreiben die Knoten jeweils eine große Datei mit dem berechneten Frame in den HiPPi Frame Buffer. Bei der Durchführung der Tests wurden die Frames über das M_UNIX auf eine Festplatte geschrieben.

Operation	Operation Count	Volume (Bytes)	Time (Seconds)	Percentage I/O Time
All I/O	1504	979,162,982	164.75	100.00
Read	121	8457	0.17	.10%
AsynchRead	436	880,849,125	4.60	2.79
I/O Wait	436	-	88.44	53.68
Write	300	98,305,400	31.76	19.28
Seek	4	0	.13	0.08
Open	106	-	32.78	19.90
Close	101	-	6.87	4.17

Tabelle 3 Anzahl, Größe und Dauer der I/O Operationen in RENDER, aus [Crandall]

Operation	Operation Size			
	< 4 KB	< 64 KB	< 256 KB	≥ 256 KB
Read	121	0	0	436
Write	200	0	0	100

Tabelle 4 Schreib/Lesegrößen bei RENDER, aus [Crandall]

Die in Tabelle 3 und 4 angegebenen Messwerte entsprechen der während der Laufzeit von 470 Sekunden gesamten I/O Zugriffe des RENDER Codes. Die gemessenen Lesezugriffe dominieren beim Volumen mit cirka 90% im Gegensatz zu den Schreibzugriffen, wobei dafür 2,9% der Zeit im Gegensatz zu 20% bei den Schreibzugriffen benötigt wird. Der größte Zeitanteil wird aber mit dem Warten auf I/O Operationen verbraucht, cirka 54%. Die jeweiligen Größen einer I/O Operation sind in der Tabelle 4 festgehalten und sind bi-modal, dass heißt entweder kleiner als 4 KB oder größer 256 KB. Das Ergebnis des Zugriffsmusters müssen die beiden in Kapitel 3.2 beschriebenen Phasen unterschieden werden.

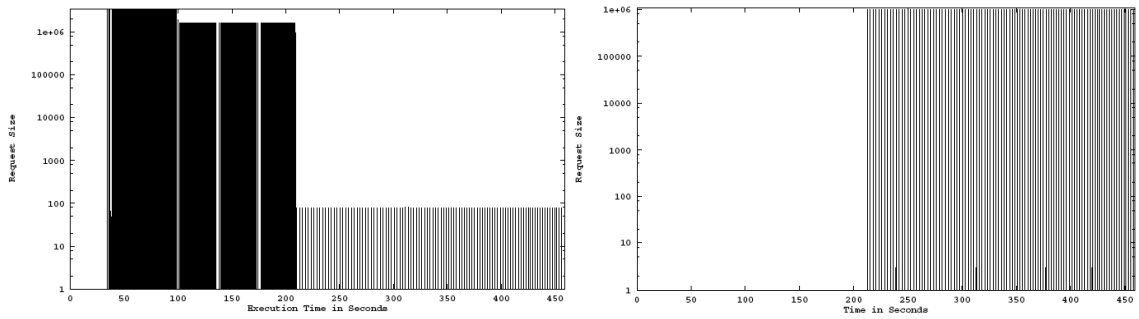

Abbildung 2: Lese und Schreibzugriffe über die Zeit von RENDER, aus [Crandall]

Die Initialisierungsphase ist nach 210 Sekunden beendet und besteht aus vier großen Blöcken, die gelesen und danach an die Knoten verteilt werden, vgl. Abb. 2 linkes Diagramm. Nach der Initialisierung treten nur noch die schon anfangs erwähnten kleinen Zugriffe auf die Koordinatendateien auf. Bei den Schreibzugriffen werden nach der Initialisierung die Frames (jeweils 1 MB groß) auf die Festplatte geschrieben, siehe Abb. 2 rechtes Diagramm. Somit entspricht das Zugriffsmuster einer klassischen Wissenschaftlichen Anwendung: ein großer initialer Lesevorgang gefolgt von der Ausgabe der Ergebnisse.

Durch die Weiterentwicklung der Genauigkeit von Sensoren bei der NASA wird der initiale Lesevorgang immer wichtiger, da die Sensoren immer größere Mengen an Daten sammeln, die dann in die Berechnung der Frames eingehen müssen. Weiterhin werden mit dieser Weiterentwicklung auch die Schreibzugriffe kritischer, da die aktuelle Ausgabe Bilder der Größe 640*480 mit 24 Bit Farbtiefe bei besseren Sensorendaten auch erhöht wird. Somit benötigt der RENDER Code höhere I/O Leistung zur Erzeugung hoch auflösender Frames wie z.B. 3000*2000 Bildpunkte.

Bei der Entwicklung des RENDER Codes wurde auch der Einsatz von M_UNIX als paralleler I/O Zugriff für die Berechnungsknoten untersucht, was aber keine Verbesserung der Leistung mit sich brachte. Ein anderer Ansatz, wobei jeder Knoten eigene Inputdaten erhält wurde ebenfalls untersucht und mit dem Ergebnis, dass zusätzliches Preprocessing benötigt wird (teuer) und die Daten dann an bestimmte Knoten gebunden sind, verworfen. Abschließend kann festgehalten werden, dass das Schreiben der Ergebnisse in einen HiPPi Frame Buffer (streaming I/O) als eine gute Idee bezeichnet werden kann, aber noch keine Beachtung bei den skalierbaren parallelen Systemen erreicht hat.

4.3 HTF (JM)

Der in Kapitel 3.3 beschriebene Algorithmus, wurde für die Messungen mit einer beschränkten Menge von 16 Atomen ausgeführt. Daraus resultierte eine Laufzeit für die jeweilige Phase von 127 Sekunden für „psetup“, 1173 Sekunden für „pargos“ und 1008 Sekunden für „pscf“ auf 128 Knoten. Von der gesamten Laufzeit wurden nur ca. 20% für I/O Operationen verbraucht. Während der Initialisierung wurden nur kleine Datenmengen gelesen und transformiert wieder

geschrieben, siehe Tabelle 5 im oberen Drittel. Dahingegen werden in der zweiten Phase die transformierten Daten geöffnet, aber die Schreiboperationen sind bei N gelesenen Daten N². Das liegt daran, dass aus einer Fock-Matrix der Größe N O(N²) Integrale für ein Elektron bzw. O(N^4) Integrale für zwei Elektronen erzeugt werden, so dass die Datenmenge während der Verarbeitung extrem ansteigt, siehe Tabelle 5 im mittleren Block. Die zweite Phase ist also sehr schreibintensiv, wohingegen die dritte Phase sehr leseintensiv ist, da für die SCT Berechnungen alle Knoten wiederholt die Integraldateien lesen. Somit ist die gelesene Datenmenge auch sehr groß, siehe Tabelle 5 unteres Drittel.

Operation	Operation Count	Volume (Bytes)	Node Time (Seconds)	Percentage I/O Time
HTF Initialization				
All I/O	832	7,267,422	55.23	100.00
Read	371	3,522,497	15.34	27.77
Write	452	3,744,872	5.50	9.96
Seek	2	53	0.43	0.78
Open	4	-	31.49	57.02
Close	3	-	2.47	4.47
HTF Integral Calculation				
All I/O	17,854	698,992,502	6,398.03	100.00
Read	145	34,393	0.47	0.00
Write	8,535	698,958,109	1996.4	31.20
Seek	130	0	0.14	0.00
Open	130	-	4056.60	63.40
Close	129	-	11.43	0.18
Lsize	128	-	15.27	0.24
Forflush	8,657	-	317.72	4.98
HTF Self-Consistent Field Calculation				
All I/O	52832	4,205,483,650	32,800.99	100.00
Read	51499	4,201,634,304	32,263.20	98.36
Write	207	3,849,268	5.88	0.02
Seek	813	3,495,198,798	1.67	0.00
Open	157	-	518.74	1.58
Close	156	-	11.50	0.04

Tabelle 5: Anzahl, Größe und Dauer der I/O Operationen in HTF, aus [Crandall]

Operation	Operation Size			
HTF Initialization				
	< 4 KB	< 64 KB	< 256 KB	≥ 256 KB
Read	151	220	0	0
Write	218	234	0	0
HTF Integral Calculation				
Read	143	2	0	0
Write	2	1	8,532	0
HTF Self-Consistent Field Calculation				
Read	165	109	51225	0
Write	43	158	6	0

Tabelle 6: Schreib/Lese-Größen für HTF, aus [Crandall]

Wenn nun die Größen der einzelnen I/O Operationen der jeweiligen Phasen betrachtet werden, siehe Tabelle 6, kann man folgende Aufstellung machen:

- Phase 1: kleine Lese- und Schreibzugriffe, kleiner 64 KB.
- Phase 2: überwiegend größere Schreibzugriffe, kleiner 256 KB.
- Phase 3: überwiegend größere Lesezugriffe, kleiner als 256 KB.

Bei der Durchführung mit den Testdaten stellte sich heraus, dass jeder Knoten cirka 5 MBytes an Daten während der Integralberechnung in eigene Dateien schreibt und während der abschließenden Berechnungsphase werden diese Daten dann von den Knoten wieder eingelesen.

Die Messungen ergaben, dass der HTF Code erhebliche Anforderungen an das I/O Verhalten von Parallelen Systemen stellt, wobei es sich beim gemessenen Problem um ein sehr kleines handelt. Laut den Entwicklern der Software soll der Algorithmus auch für größere Probleme genutzt werden, weshalb auch die drei Phase Aufteilung genutzt wird, da so der Berechnungsaufwand durch die Wiederbenutzung der schon transformierten Daten erheblich gesenkt wird. Leider wachsen die Anforderungen an das I/O System mit $O(N^4)$, was mit den damals aktuellen Systemen nicht ausführbar war. Daher wurden durch die jeweilige Neuberechnung der Integrale die Berechnungskosten wieder gesteigert, aber dadurch die I/O Kosten wieder gesenkt. Zur Berechnung der Integrale wer-

den an jedem Node 5-10 MByte/s für I/O Operationen benötigt, was dazu führt, dass jeder Prozessor über eine eigene Festplatte bzw. Festplattenarray verfügen muss. Weiterhin bedeutet dies, dass die Leistung der Festplatten bei steigender Prozessorleistung im gleichen Maß anwachsen muss.

Die HTF Anwendung erfordert also hohe Speicherkapazität und hohen Durchsatz und benutzt dabei einfache Zugriffsmuster.

4.4 Diskussion der Ergebnisse (JM)

Die augenscheinlichste Beobachtung der Messungen zeigt, dass die untersuchten Anwendungen Anforderungen an das I/O Verhalten der parallelen Systeme stellen, die die I/O Leistung damals aktueller skalierbarer paralleler Systeme bei weitem übersteigt. Weiterhin sind die I/O Zugriffsmuster und I/O Anforderungen der wissenschaftlichen Anwendungen komplizierter als einfache Stereotypen. So verfügen alle untersuchten Anwendungen über eine breite Palette an Lese/Schreibzugriffsmixturen und verschiedene Lese/Schreibgrößen. Daher müssen skalierbare parallele I/O Systeme die verschiedenen Zugriffsmuster unterstützen, damit die wissenschaftlichen Anwendungen zufrieden stellend ausgeführt werden können.

Da die damals aktuelle I/O Systeme I/O Zugriffe mit großen Datenmengen und hoher Bandbreite favorisieren, ergeben sich für die Programmierer zwei Konsequenzen für Anwendungen mit kleinen Datenmengen:

1. die kleinen I/O Zugriffe werden „manuell“ gesammelt und dann zusammen auf die Datenträger geschrieben bzw. gelesen.
2. die I/O Zugriffe werden vom Dateisystem über Caching bzw. Prefetching transparent auf das jeweilige Zugriffsmuster der Anwendung abgebildet.

Da aber nicht einmal die beschränkte Auswahl an wissenschaftlichen Anwendung eine Charakterisierung der I/O Größen bzw. Muster zulässt, müssen die I/O Systeme sich ändernden Anforderungen, auch während dem Ablauf einer Anwendung, anpassen um sie bedienen zu können. Daher müssen die Entwickler der wissenschaftlichen Anwendung bei der Entwicklung von neuen Dateisystemen mit involviert werden, da nur sie über ausreichendes Wissen über I/O Anforderungen aktueller Anwendungen verfügen.

Bei den Untersuchungen stellte sich eine Gemeinsamkeit der drei Anwendungen heraus, so lesen bzw. schreiben sie die Dateien immer als ein ganzes und dies meist von einem einzelnen Knoten aus. Dies zeigt, dass die Entwickler der Anwendungen die Kontrolle über die Daten auf der Festplatte behalten wollen, da so eine höhere Leistung erreicht wird oder, dass die I/O Operationen unabhängig vom parallelen Dateisystem sind, damit die Anwendung portierbar bleibt (z.B. ein Knoten liest die Daten und macht einen Broadcast mit den Daten).

Bei der Entwicklung neuer Dateisysteme muss also nicht nur die Anwendung portierbar sein, sondern auch Optimierungen von einem Dateisystem auf das andere auch portiert werden. Diese Anforderung ist natürlich einfach aufgestellt, aber die Implementierung ist wohl eine der großen Herausforderungen bei den parallelen Systemen.

Eine weitere Gemeinsamkeit der untersuchten Anwendungen ist, dass Schreibzugriffe auf Dateien direkt auf den Festplatten stattfanden. Problematisch ist dieses Verhalten vor allem beim Schreiben kleiner Dateien (z.B. Logfiles, temporäre Dateien). So ergibt sich für die Caching Strategie zukünftiger paralleler Dateisysteme die Anforderung, dass die erreichbare Bandbreite der I/O Systeme maximiert werden sollte und nicht die Menge an Daten die geschrieben wird zu reduzieren. So könnten z.B. kleine Schreibzugriffe gesammelt werden und dann gemeinsam auf dem Datenträger gespeichert werden.

Da es sich bei der Untersuchung um eine kleine Menge von Anwendungen auf einem bestimmten Hardwaresystem handelt, geben sie zwar Indizien für die damals aktuellen Probleme beim parallelen I/O, sind aber nicht repräsentativ für alle parallelen Anwendungen. Ein weiteres Problem ist die Frage, ob bei der Entwicklung einer Anwendung das Wissen um die verwendete Hardwarearchitektur den Programmierer bei seinen Optimierungstätigkeiten beeinflusst hat. Aus diesem Grund haben sich Crandall, Aydt, Chien und Reed entschlossen ihre Untersuchungen auch auf alternativen Hardwareplattformen durchzuführen.

5 Fazit (JM)

Das in diesem Dokument behandelte Dokument „“Input/Output Characteristcs of Scalable Parallel Applications“ von Crandall et al. stellt zuerst die Methodologie zur Untersuchung der Zugriffsverhalten von skalierbaren parallelen Anwendungen vor. Im Anschluss wurden diese Methode auf drei verschiedenen wissenschaftliche Applikationen angewendet. Dabei stellte sich heraus, dass allein bei den untersuchten Anwendungen eine breite Palette von verschiedenen Zugriffsmustern, sowohl räumlich, als auch zeitlich, beobachtet wurde. Auch bei den benutzten Schreib- bzw. Lesegrößen konnten keine eindeutigen Muster erkannt werden. Die Beobachtungen ergaben, dass die meisten Zugriffe sequentiell erfolgten, wobei der zeitliche Abstand nicht vorhersehbar und unregelmäßig war. Bei der Größe der Anfragen kommen feste Größen öfters vor, aber insgesamt gesehen kommen von sehr kleinen bis sehr großen Anfragen in den Anwendungen alle Ausprägungen vor.

Aus diesem Grund muss beim Design paralleler Dateisysteme darauf geachtet werden, dass keine festen Methoden für die Zugriffe benutzt werden, sondern diese an die Anforderungen der jeweiligen Anwendungen angepasst werden bzw. werden können. Bei der automatischen Anpassung von Zugriffsmethoden an die Anwendung ist die Identifizierung des richtigen Zugriffsmusters eine sehr schwierige Aufgabe. Aus diesem Grund setzen die Autoren zur Identifizierung von Zugriffsmuster mit anderen Anwendungen fort.

Zusammenfassend kann festgehalten werden, dass ineffiziente und unreife I/O Subsysteme als der bedeutendste Flaschenhals skalierbarer paralleler Systeme sind. Ein Grund dafür ist, dass die Entwickler nur unzureichende Daten über das allgemeine Zugriffsverhalten wissenschaftlicher Anwendungen haben und daher auf Hochrechnung von Beobachtungen traditioneller Vektorcomputer zurückgegriffen wurde. Anstatt die Zugriffstechniken basierend auf nicht mehr aktuellen Supercomputertechniken zu entwickeln, sollten Methoden wie das Caching/Prefetching für kleine und sequentielle I/O Zugriffe und das Streaming der Daten direkt in die Anwendung für große und unregelmäßige I/O Zugriffe genutzt werden. Die Autoren haben angefangen generelle, adaptive prefetching

Methoden zu entwickeln, die die Zugriffsmethoden automatisiert erlernen und vorhersagen können.

6 Literaturangaben

[Crandall] Crandall, P.E.; Aydt, R. A.; Chien, A. A.; Reed, D. A.: Input/Output Characteristics of Scalable Parallel Applications. Department of Computer Science, University of Illinois.