Seminararbeit

HighLevelBuffering

Oliver Schwaneberg, Oliver Zilken

25. Juni 2005

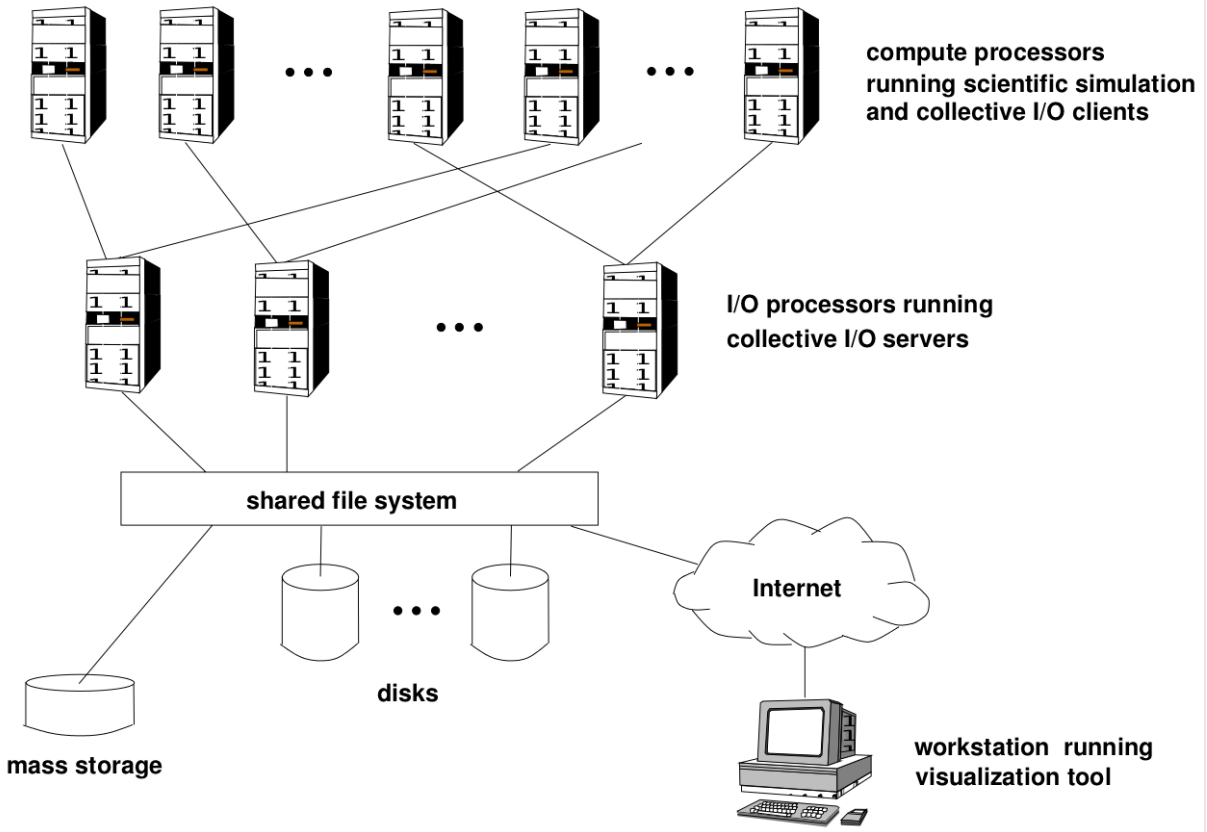
Inhaltsverzeichnis

1 Einleitung

In den letzten Jahrzehnten sind, durch den Einsatz von Supercomputern und großangelegten Rechnerclustern, ungeahnte Möglichkeiten entstanden. Das Feld der wissenschaftlichen Simulation von physikalischen oder chemischen Vorgängen war und ist dabei eine der Hauptkanwendungsgebiete. Die hier durcheführten Simulationen beinhalten das Berechnen von sehr großen Gleichungen und können tage- oder wochenlang Berechungszeit in Anspruch nehmen. Beispiele hierfür sind Simulationen in der Automobilindustrie, in der Raumfahrt oder in der Pharmazeutischen Industrie. Bei solchen Simulationen entstehen typischerweise riesige Datenmengen, die periodisch gesichert werden müssen, um sie später korrekt auswerten zu können(auch *Snapshots* genannt). Diese Ergebnisdaten können heutzutage leicht bis zu mehreren Terabyte pro Tag groß sein. Diese Datenmengen stellen ein zentrales Problem bei parallelen Berechungen dar. Zwar konnte in den letzten Jahrzehnten die Berechungsgeschwindigkeit von Parallelrechnern exponentiell gesteigert werden. Jedoch ist die benötigte Zeit zum Lesen und Schreiben von Daten auf die Speichermedien (im folgenden IO genannt) nur in einem sehr viel geringerem Maße gestiegen. Dies führt zu einer immer weiter auseinanderklaffenden Schere zwischen Berechungs- und IO-Geschwindigkeit. Diese Schere bildet einen bedeutenden Bottleneck, weil, während ein Parallelrechner mit IO beschäftigt ist, er keine Berechungen durchführen kann. Es wird heutzutage mit verschiedenen Mitteln versucht, diesen Bottleneck so gering wie möglich zu halten. Ein Ansatz dazu, der im Laufe dieser Seminarreihe schon vorgestellt wurde, ist Collective IO. Hiermit wird versucht, viele unkoordinierte und kleine Schreibzugriffe zu größeren zusammen zu fassen; Dadurch kann dann die Ausführungszeit einzelnen IO-Phasen verkürzt werden.

Nun soll ein weiterer Versuch zur Beseitigung dieses Bottlenecks vorgestellt werden: Das High Level Buffering (Auch: Active Buffering). Dieses dem Programmierer transparente Schema erlaubt der verwendeten IO -library, die IO-Zeit des Programms entscheidend zu optimieren. Hierzu wird jedoch nicht einfach die die Schreib-/Lesegeschwindigkeit bei IO erhöht. Es wird durch viel mehr versucht, die entstehenden Kosten und Latenzen durch eine vielschichtige Puffer-Hiererchie zu verbergen.

Die anfallenden IO-Kosten lassen sich nicht reduzieren. Die Idee des High Level Buffering ist es, diese IO-Kosten zu kaschieren. Dazu wird versucht, die Berechnungen und die IO-Transaktionen weitgehend zu parallelisieren.
Es gibt viele mögliche Implementationen eines solchen Systems.



Der hier behandelte Ansatz ist ein Client-Server-Modell, obwohl auch ein Thread-basiertes System implementiert wurde. Hierbei erstellen die berechnenden Prozessoren (=Clients) in regelmäßigen Abständen Teilergebnisse (=Snapshots), die sofort zu den Servern übertragen werden. Die Aufgabe der Server ist die Datenmigration und die Ausgabe, wobei die Ausgabe (bzw. Präsentation von Teilergebnissen) zur Laufzeit somit ohne Unterbrechung der Clients geschieht. Ziel ist es, dass die Snapshots sofort in schnelle Buffer übertragen werden können, damit die berechnenden Prozessoren sofort wieder anfangen können zu arbeiten während die Server die Datenmigration vornehmen. Die Übertragung der Teilergebnisse in größeren Snapshots soll den Overhead, der durch viele kleine Schreiboperationen entstehen würde, minimieren. Außerdem ermöglicht die Regelmäßigkeit der anfallenden Daten eine Optimierung der Datenmigration, da die Server wissen zu welchem Zeitpunkt sie wie viel unbelegten Buffer verfügbar haben müssen. Bei Systemen, die keine Datenmigration benötigen, hat sich eine andere Variante als sinnvoll erwiesen. Bei parallelen Systemen, die über große SMP-Nodes verfügen, kann je ein Prozessor pro Node als I/O Prozessor verwendet werden, während die übrigen als berechnende Prozessoren agieren. Da die IO Operationen ohne Migration wenig CPU-lastig sind und lediglich durch die Performance des verteilten Dateisystems, also dessen Speichermedien und Netzwerk, beschränkt sind, ist eine CPU pro Node ausreichend. Die Erzeugung der Snapshots geschieht hierbei synchron, da dieses Ereignis durch einen kollektiven *write*

call ausgelöst wird. Die berechnenden Prozessoren führen eine Zwischenspeicherung der Snapshots im lokalen Buffer (lokales Dateisystem und Arbeitsspeicher) durch. Wenn die lokalen Buffer überlaufen, werden die verbleibenden Daten zum IO Prozessor gesendet, welcher für das Ausschreiben der Daten zuständig ist.

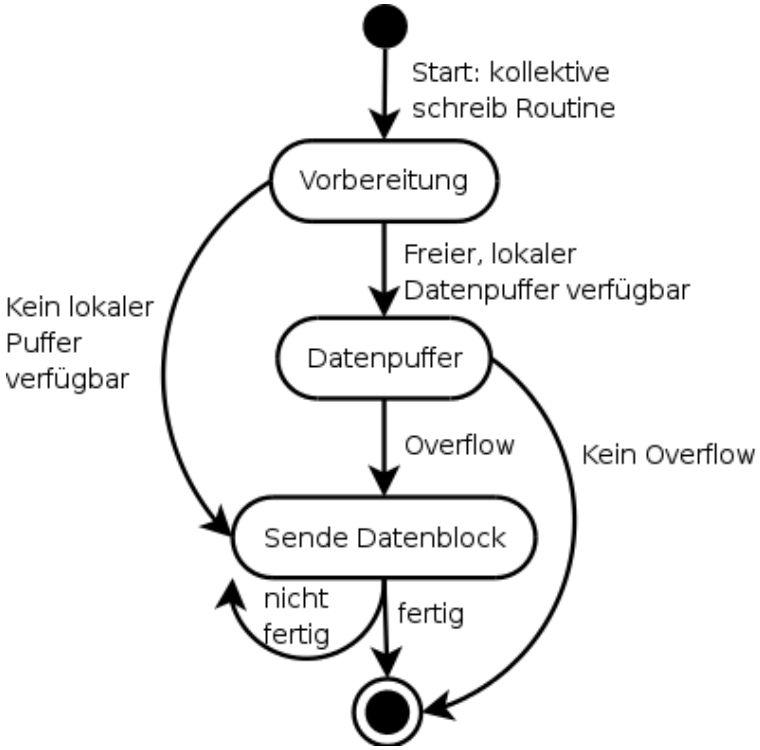
2 Active Buffering

2.1 Architektur des Active Buffering

2.1.1 Buffer Hierarchie

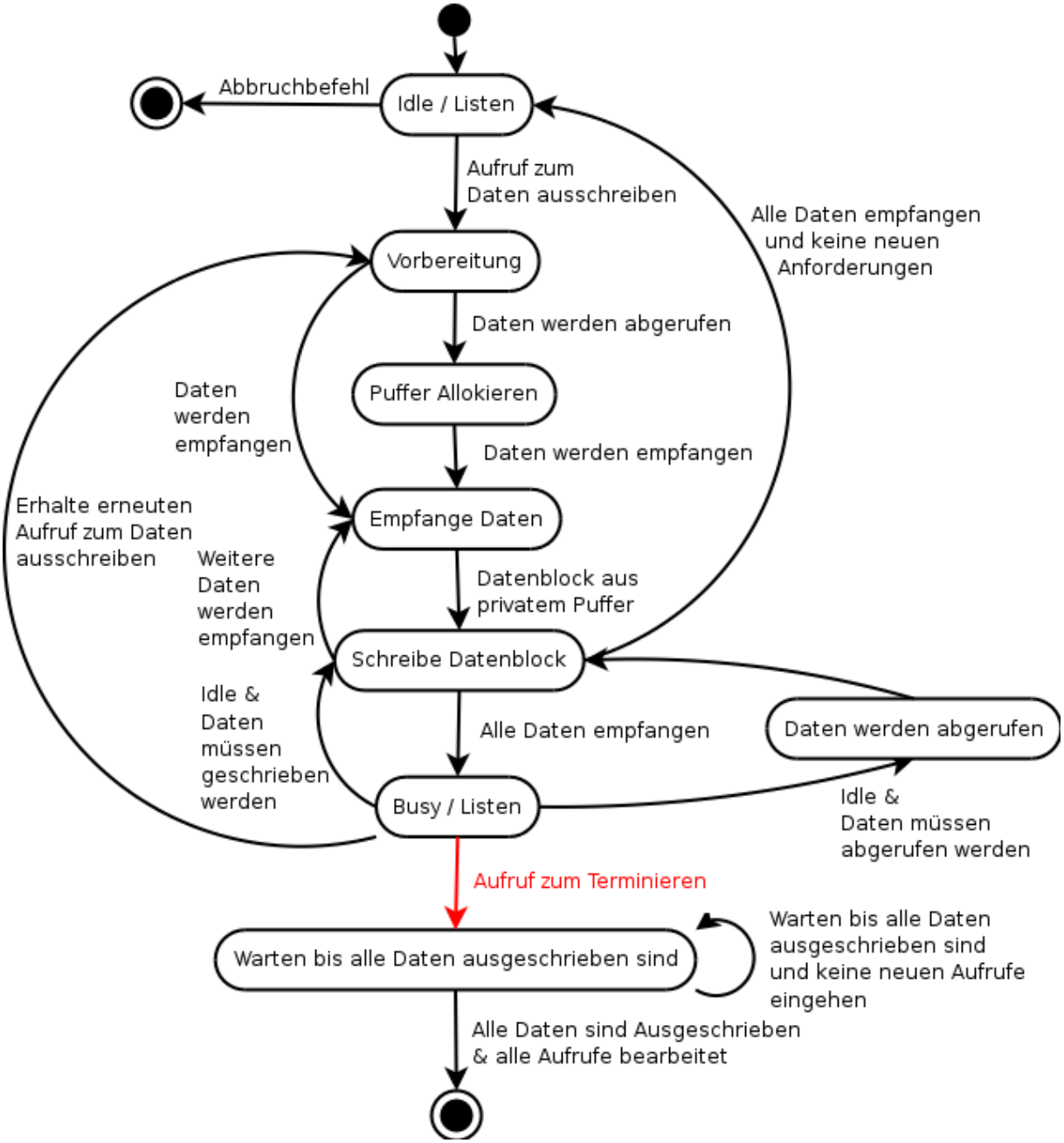
In der Vollversion des HighLevelBuffering verwenden Server und Clients ihre lokalen Speicher, um die Daten zu puffern. Die Server können zu diesem Zweck einen Großteil ihres Speichers verwenden. Die Clients hingegen müssen sich auf freien Speicher beschränken, der nicht von der Applikation benötigt wird. Die Applikation sollte in der Lage sein, anzugeben, wie viel Speicher sie maximal benötigt. So kann die maximale Puffergröße auf dem Client bereits im Voraus festgelegt werden. Allerdings ist auch eine dynamische Regelung zur Laufzeit möglich.
Wenn die Kapazität des lokalen Puffers des Clients erschöpft ist, werden die überschüssigen Daten direkt zu den Servern geschickt. Dies geschieht durch MPI-Nachrichten. Wenn die Puffer der Server selbst nicht ausreichend sein sollten, so müssen diese ihre Puffer schnellst möglich ausschreiben.
Solange die lokalen Puffer der Clients ausreichend sind, sind die Applikation lediglich für die Zeitspanne des lokalen Kopierens blockiert. Die lokalen Puffer arbeiten vollkommen parallel und skallieren mit der Anzahl der Clients, daher bieten sie i.d.R. den höchsten Daten-Durchsatz. Die sichtbaren I/O-Kosten sind also sehr gering. Wenn die lokalen Puffer nicht ausreichen, wird die Applikation für die Zeitspanne des lokalen Kopierens und der Datenübertragung zum Server blockiert sein. Der Anteil der sichtbaren I/O-Kosten steigt somit. Hierbei hat die Struktur der Client-Server-Kommunikation sowie das Clients-pro-Server-Verhältnis den entscheidenden Einfluss auf die Performance. Zu diesen sichtbaren I/O-Kosten kann die Zeit für das endgültige Ausschreiben der Daten hinzukommen, wenn die Puffer der Server ebenfalls gesättigt sind. In diesem Fall bringt Active Buffering keinen Vorteil. Daher ist es notwendig, die Größe der Snapshots sowie deren Intervall, die Kapazitäten der verschiedenen Puffer und das Verhältnis von Clients zu Servern gut anzupassen. Die Server können zu diesem Zweck Informationen über den Ablauf der Berechnungen sammeln. Auch eine selbstständige Justierung zur Laufzeit ist möglich.

2.1.2 Client-State-Machine



Das Zustandsdiagramm beschreibt die Logik der Clients. Hierbei werden Teilergebnisse entgegen genommen und in den lokalen Puffer übertragen. Dies geschieht solange bis entweder die Zeit für das Übertragen eines Snapshots zu den Servern gekommen ist oder die lokalen Puffer überlaufen. Im zweiten Fall werden die Daten, die nicht in den lokalen Puffern gehalten werden können, sofort an die Server übergeben.

2.1.3 Server-State-Machine



Das Zustandsdiagramm beschreibt die Logik der Server. Zunächst ist der Server unausgelastet und wartet auf Befehle, dieser Zustand heißt Idle / Listen. Wenn nun der Befehl zum Ausschreiben eines Snapshots folgt wird er entweder die Snapshots von den Clients abrufen oder warten bis diese eintreffen. Dies ist hänt von der Konfiguration des Systems ab (Polling oder Interrupt). Wenn ein Snapshot vollständig übertragen wurde kann er in das verteilte Dateisystem ausgeschrieben werden. Solange die Migration der Daten andauert befindet sich der Server im "Busy / IdleSZustand. Das bedeutet, dass

er parallel zur Datenmigration weitere Daten von den Clients entgegen nimmt bzw. die Daten weiterhin abruft.
Wenn ein Server den Befehl zum terminieren erhält, wird er in jedem Fall alle gepufferten Daten ausschreiben bevor er terminiert.

2.1.4 Puffer Allokation und Deallokation

Zur Allokation wird die SHMEM Bibliothek für einseitige Kommunikation verwendet. SHMEM ist für viele Plattformen verfügbar und bietet die schnellste Inter-Prozessor-Kommunikation für große Datenpakete. MPI-2 bietet ebenfalls eine vergleichbare Schnittstelle, die jedoch noch nicht auf allen Plattformen unterstützt wird.
Durch ein *src_buf = shmalloc(size)*-Kommando können alle Prozessoren dazu aufgerufen werden, einen Speicherbereich gleichzeitig zu allokieren. Die Größe des Speicherbereichs (*size*) ist bei allen Prozessoren identisch. Der dadurch allokierte Speicher wird als *symm Buffer* bezeichnet.
Jeder Client wird nun seine Ausgabedaten in seinen lokalen *symm Buffer* übertragen. Ein Server kann anschließend seinen eigenen Buffer mit dem Inhalt des Buffers eines beliebigen Clients füllen. Dazu muss er das Kommando-*shmem type get(dest_buf, src_buf, size, i)* ausführen, wobei *i* der MPI-World-Rank des jeweiligen Clients ist. *type* ist dabei der Typ der gepufferten Daten.
Da alle Clients gleich viel Speicher allokieren, ist es möglich das die Puffer der einzelnen Clients nicht immer voll belegt sind. Die Server erhalten daher Metadaten von den Clients, damit bekannt ist wie viele Nutzdaten sich im Puffer befinden. Es empfiehlt sich eine ausbalancierte Verteilung der Daten auf alle Clients, was bei den meisten wissenschaftlichen Anwendungen kein Problem darstellt. Wenn alle Daten verarbeitet sind wird der *symm Buffer* auf allen Prozessoren gleichzeitig freigegeben. Dazu wird der Befehl *shfree(src_buf)* an alle Knoten gesendet. Außerdem verfügt jeder Server über einen Pool von weiteren privaten Puffern, die entkoppelt von den anderen Prozessoren verwaltet werden. Ein serverseitiger *symm Buffer* kann zwar für die Daten verschiedener Clients wiederverwendet werden, aber die Server benötigen weitere Puffer um Overflow-Daten der Clients empfangen zu können.
Die Server allokieren jedoch nur soviel Speicher wie maximal verwendet werden muss. Nicht mehr benötigte Puffer werden sofort frei gegeben.

2.2 Vergleich Active Buffering mit anderen I/O-Schemata

2.2.1 Active Buffering vs System-level-buffering

Praktisch alle Dateisysteme, ob lokal oder entfernt, werden vom Betriebssystem aus gepuffert. Diese Pufferung alleine ist nicht ausreichend, da die verwendeten Algorithmen kein Wissen über die jeweiligen Anwendungen besitzen und sich nur geringfügig durch den Benutzer konfigurieren lassen. Somit sind diese Puffer nicht auf ankommende Daten-Peaks vorbereitet. Daher reicht der System-level-buffer nicht aus um die I/O-Kosten in solchen Situationen zu verbergen.

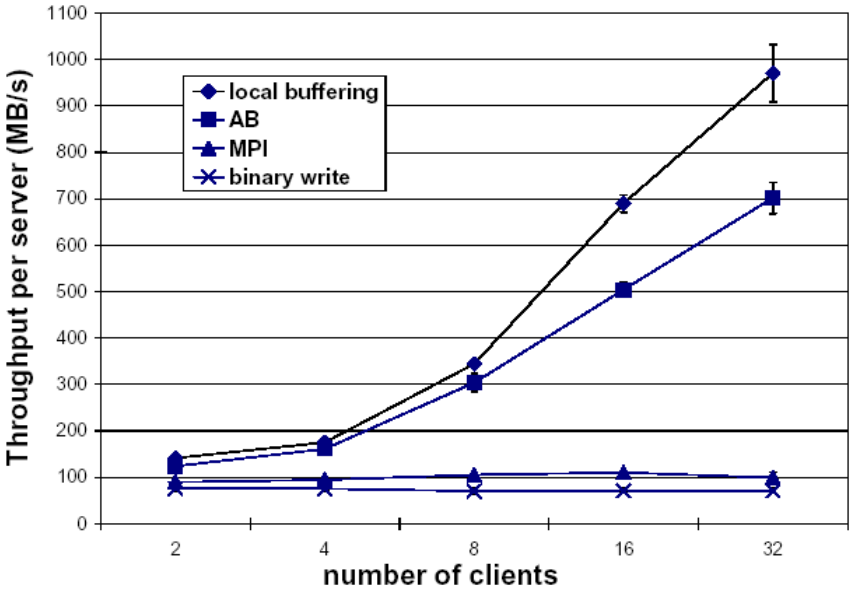
2.2.2 Active Buffering vs trad. asynchronem I/O

Beide Ansätze werden verwendet um I/O und Berechnungen zu parallelisieren. Das High-Level-Buffering operiert auf einer höheren Abstraktionsstufe, es soll eine bessere Portierbarkeit ermöglichen und soll einfacher vom Benutzer zu verwenden sein.
Dies wird erreicht, da die High-Level-Buffering-Bibliothek ein einfaches I/O-Interface, welches für den Entwickler transparent arbeitet, bereit stellt. So muss der Entwickler nach einem Write-Call nicht überprüfen ob seine Daten tatsächlich ausgeschrieben wurden. Statt dessen kann er seinen Puffer direkt nach einem Aufruf wiederverwenden.
Außerdem ist High-Level-Buffering ein plattformunabhängiges Konzept. Somit lässt sich eine High-Level-Buffering-Bibliothek schnell und einfach auf allen Plattformen implementiert oder portieren, da sie auf einer hohen logischen Schicht des Systems operiert.
High-Level-Buffering kann die I/O-Operationen und die Berechnungen auch dann parallelisieren, wenn das zu Grunde liegende System keine asynchrone Datenhaltung unterstützt. Außerdem bietet es bessere Funktionen zur Optimierung von kollektiven I/O-Operationen.

3 Active Buffering in der Praxis

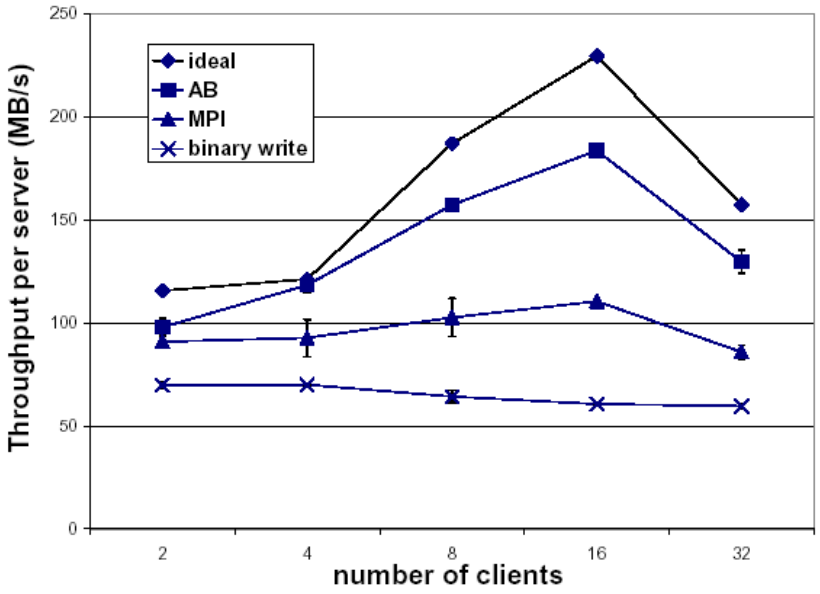
In diesem Kapitel soll gezeigt werden, wie sich der Einsatz von Active Buffering auf die Performance von parallelen Berechnungen auswirkt. Die hier ausgefühten Benchmarks wurden auf zwei verschiedenen Parallelrechnern ausgeführt: Der erste ist ein SGI Origin 2000 System. Dies ist ein Shared-memory-Rechner mit 256 MIPS R10000 Prozessoren, die jeweils mit 250 MHz getaktet sind. Er hat insgesamt 128GB Hauptspeicher und 450GB Festplattenspeicher mit einem XFS Dateisystem. Der zweite Rechner ist der ÄSCII Frost; ein IBM System.Er enthält 86 POWER3 375 MHz 16-wege-SMP-Nodes. Der Hauptspeicher des Systems bezträgt insgesamt 1088GB, der Festplattenspeicher betheht aus einer 20TB großen GPFS-Partition.

Die Benchmarks bestehen größtenteils darin, große Dateien verschiedenster Formate auf das verteilte Dateisystem zu schreiben. Dies waren unter anderen Dateien im Binär-Format und im HDF4-Format. Diese Dateitypen sind in der Praxis dafür bekannt, am schnellsten bzw. am langsamsten schreibbar zu sein.



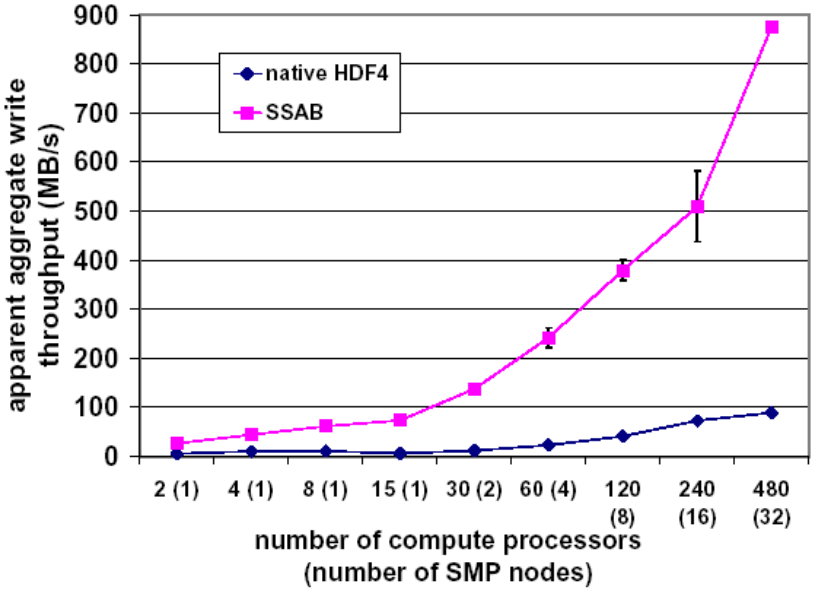
In dieser Abbildung ist der Datendurchsatz beim Schreiben von verschiedenen Dateiformaten auf dem SGI Rechner zu sehen. Es wurde ein volles Active Buffering mit 256MB privatem und 64MB Symm-Buffer verwendet. Das Verhältnis von Clients und IO-Server betrug hier 16 1. Hier sieht man, dass die Performance mit Active Buffering (Die ÄB¨-Linie) sehr hoch ist. Die Performancesteigerung in diesem Beispiel beträgt in etwa einen Faktor von 8 zum Binär- und HDF-Format. Es ist zu erkennen, dass die erzielbare Perfomrance mit Active Buffering zu etwa 70% an die von lokalem IO heranreicht. Hierbei ist jedoch zu beachten, dass die Performance von Active Buffering langsamer mit der Anzahl der Clients skaliert als die lokale Variante.

In der nächsten Abbildung wird gezeigt, wie sich ein Overflow bei der Speicherung zu großer Dateien auf die Performance auswirkt:



Hierbei wurde die Größe des Symm-Buffer auf 64MB reduziert. Dann wurde versucht, pro Client 92MB an Daten auszugeben. Da der IO-Server 256MB an privatem Buffer besitzt, kann er den Overflow von bis zu 16 Clients gleichzeitig aufnehmen. Es ist ersichtlich, dass in diesem Falle (mit 32 Clients) die Performance absinkt. Obwohl die Performance in diesem Falle um den Faktor 3 niedrige ist als im ersten Beispiel, so liegt Active Buffering mit ca. 80% noch näher an der idealen Performance.

In der nächsten Abbildung ist nun die Performance von Active Buffering in einer realen Applikation zu sehen. GENx ist ein Programm zur Simulation von Raketenstarts. Es ist mit einer parallelen Panda-Bibliothek ausgestattet, die Active Buffering für periodischen Output in einem HDF4-Format benutzt. In diesem Falle wurde der oben beschriebene IBM-Frost Rechner verwendet.



In dieser Abbildung ist der Datendurchsatz bei der Ausführung einer Raketensimulationssoftware (GENx) auf dem IBM Frost Rechnen zu sehen. Hier wurde nur Serverseitiges Active Buffering benutzt. 15 Nodes dienten als Clients, einer als IO-Server. Die HDF-Linie bedeutet, dass hier die snapshots nur lokal für jeden Prozessor in nativem HDF4 gespeichert wurde. COllective IO wurde nicht verwendet. Mit dieser ursprünglichen Variante verglichen, liefert Serverseitiges Active Buffering eine entscheidend höhere Performance und eine bessere Skalierbarkeit. Die enstehenden IO-Kosten werden versteckt, so daß die Applikation keine längere Laufzeit als ohne active Buffering braucht.

4 Fazit

.

Im Laufe dieser Seminararbeit wurde das *Active Buffering* vorgestellt, eine mehrschichtige Buffer-Architektur, die es parallelen IO-Bibliotheken erlaubt, periodisch auftretende IO-Kosten zu verbergen. Auf diese Art können diese Bibliotheken unbenutzte Prozessorzeit dazu verwenden, IO durchzuführen und so die dem User sichtbare mit IO verbrachte Zeit beträchtlich zu reduzieren.
Active Buffering ist auch deshalb sehr nützlich, weil es keine Limitierung an den freiem Speicherplatz oder der Prozessorauslastung hat, und ohne große Veränderungen am Applikations-spezifischen Quellcode auskommt.
Active Buffering wird mittlerweile schon in einigen weit verbreiteten IO-Bibliotheken eingesetzt, die in vorhergehenden Vorträgen dieser Seminarreihe vorgestellt wurden (ROMIO, Panda).
Als wichtige Zukunft von Active Buffering sei hier auch noch der Bereich der Visualisierung genannt. Active Buffering könnte hier, durch das Verstecken der Kosten zum Input von Visualisierungsdaten, zu einem Durchbruch im Bereich der Real-Time Visualisierung führen. Dieser Anwendungsbereich befindet sich momentan im Fokus der Entwicklungstätigkeiten des Active Bufferings.

Literatur

[1] Xiaosong Ma,Jonghyun Lee, Marianne Winslett: *High-level Buffering for hiding periodic Output Cost in scientific Simulations.* University of Illinois, USA 2004.