FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

GIO: Ein System für parallele Ein-/Ausgabe auf Rechnerverbundsystemen

Karsten Scholtyssik

FZJ-ZAM-IB-2002-03

März 2002

(letzte Änderung: 21.03.2002)

Preprint: Proceedings zum 6. Workshop „Parallele Systeme und Algorithmen“, PASA 2002, Karlsruhe, 11.4.2002

GIO: Ein System für parallele Ein-/Ausgabe auf Rechnerverbundsystemen

Karsten Scholtyssik , Forschungszentrum Jülich, Zentralinstitut für Angewandte Mathematik

Inhalt:

Dieser Artikel beschreibt den Entwurf und die Implementierung des Systems GIO (Global I/O), das als Zusatz zum System Global Arrays (GA) innerhalb eines Rechnerverbundsystems globale Dateien implementiert. Die Implementierung basiert auf einer Bibliothek namens ARMCI (Advanced Remote Memory Copy Interface), die auch als Basis für GA dient. Um eine zu starke Verflechtung der verschiedenen Bibliotheken zu vermeiden, wurde ARMCI um sog. Remote User Callbacks bereichert, die als Grundlage für beliebige Erweiterungen dienen können. Darauf aufbauend wurde ein System entwickelt, dass es dem Benutzer erlaubt, gemeinsame Dateien unter Ausnutzung lokaler Massenspeicher anzulegen.

1. Einleitung

Portabilität, Effizienz und einfache Benutzbarkeit sind die Kriterien, die für die Auswahl des Programmiermodells für parallele, skalierbare Anwendungen wichtig sind. Während das message-passing Modell weit verbreitet und aufgrund existierender Standards gut portierbar ist, ist das shared-memory Modell deutlich einfacher zu verwenden und vereinfacht die Erstellung neuer oder die Parallelisierung sequenzieller Programme. Leider ist es nur schwer portierbar und erlaubt wenig Kontrolle über die Kosten für den Datenaustausch zwischen Prozessoren.

Hinzu kommt, dass moderne Hochleistungsrechner heute zumeist aus gekoppelten SMP (Symmetric Multiprocessor) Knoten zusammengesetzt sind, die oft keinen globalen Speicherzugriff ermöglichen.

Das System Global Arrays (GA) [4] versucht eine optimale Nutzung solcher Systeme zu erreichen, indem es die Vorteile beider Programmierparadigmen kombiniert. Es unterstützt einen sog. globalen Adressraum, d.h. eine parallele Anwendung kann globale Speichersegmente anlegen, auf die von allen ihren Prozessen aus zugegriffen werden kann, ohne explizite Kommunikationsaufrufe durchführen zu müssen. Dabei kann der Benutzer festlegen, wie die globalen Daten auf die einzelnen Prozesse zu verteilen sind, um so eine maximale Datenlokalität zu erreichen.

GA ist auf einer Vielzahl von Systemen verfügbar und hat insbesondere im Bereich der theoretischen Chemie eine große Verbreitung gefunden. So wird es z.B. von dem Quantenchemiepaket MOLPRO [8] zur Parallelisierung der Algorithmen verwendet. Da solche komplexen Programme oft sehr große Datenmengen verarbeiten, die nicht vollständig in den Hauptspeicher eines Knotens passen, sind sie auf eine effiziente Ein-/Ausgabe, z.B. zur Realisierung von „out of core solvern", angewiesen. Zwar existieren zu GA Erweiterungen, die den Zugriff auf Dateien ermöglichen, diese benötigen aber eine gemeinsame Sicht auf ein Dateisystem, die aber i.d.R. auf Rechnerverbundsystemen nicht oder nur unzulänglich implementiert sind.

Dieser Artikel beschreibt GIO (Global I/O), einen Zusatz zu GA, der diese Lücke schließt. GIO erlaubt dem Anwender Dateien anzulegen, auf die alle Prozesse einer parallelen Anwendung transparent zugreifen können. Die Implementierung von GIO weist bewusst eine große konzeptionelle Nähe zum Programmiermodell von GA auf, damit der Benutzer die ihm aus der Speicherverwaltung bekannten Konzepte leicht auf Dateien übertragen kann. So muss sich der Benutzer nur mit bekannten Eigenschaften auseinandersetzen und kann das neue System ohne große Einarbeitungszeit effizient in seiner Applikation verwenden.

GIO ist dabei unabhängig von GA implementiert, nutzt allerdings gemeinsam mit GA die Bibliothek ARMCI (Advanced Remote Memory Copy Interface) [5], die zu diesem Zweck erweitert wurde.

Der Rest des Artikels gliedert sich wie folgt:

Abschnitt 2 beschreibt die Bibliothek ARMCI, ihre Implementierung und die für diese Arbeit wichtigen zugrunde liegenden Konzepte. In Abschnitt 3 wird auf die im Rahmen dieser Arbeit durchgeführten Erweiterungen von ARMCI eingegangen. Abschnitt 4 erläutert die Schnittstelle und Implementierungsdetails von GIO. Erste Leistungsdaten werden in Abschnitt 5 präsentiert. Der Bezug zu ähnlichen Systemen wird in Abschnitt 6 hergestellt. Eine Zusammenfassung erfolgt in Abschnitt 7 .

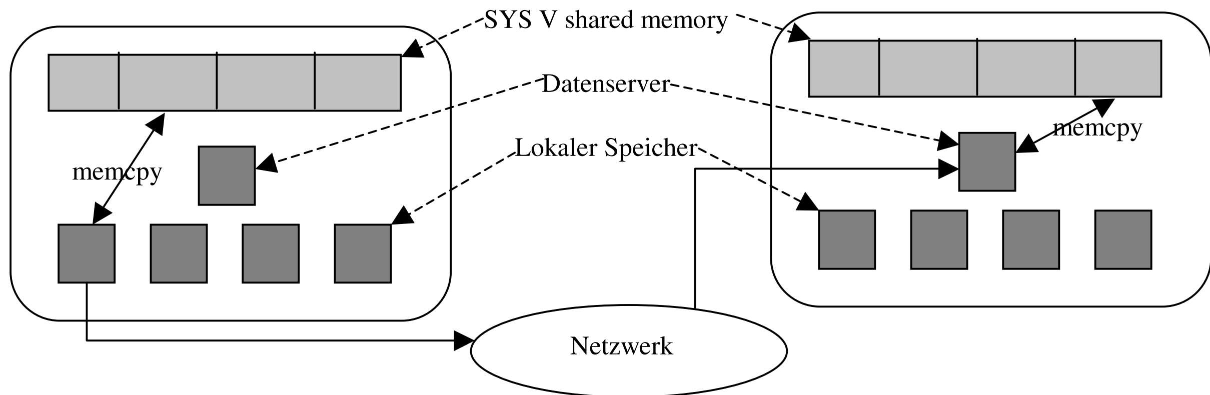

ABBILDUNG 1. Ablauf einer PUT Operation in ARMCI

2. Die ARMCI Bibliothek

Bevor auf die durchgeführten Erweiterungen von ARMCI eingegangen wird, soll hier kurz auf die grundlegende Struktur der Bibliothek eingegangen werden, um so ein besseres Verständnis für die interne Struktur der Basis von GIO zu bekommen.
Das Ziel bei der Entwicklung von ARMCI war die Unterstützung von Operationen auf entferntem Speicher, wie sie in Systemen mit globalem Adressraum, wie GA [4], aber auch in Compilerlaufzeitsystemen, wie der PRC Adlib [5] benötigt werden. Die Bibliothek bietet Kopier-, Akkumulations- und Synchronisationsoperationen auf entferntem Speicher an. Es wurde besonderer Wert auf Portierbarkeit und Kompatibilität mit existierenden Bibliotheken zum Nachrichtenaustausch, wie etwa MPI oder PVM, gelegt.
Im Vergleich mit existierenden ähnlichen Systemen, wie etwa Crays SHMEM [1], hebt sich ARMCI besonders durch die Unterstützung von nicht kontinuierlichen Datentransfers, wie sie in vielen technisch-wissenschaftlichen Anwendungen (z.B. für Ausschnitte aus mehrdimensionalen, dicht oder dünn besetzten Matrizen, Scatter/Gather Operationen) benötigt werden, ab. ARMCI bietet dazu spezielle Schnittstellen für den Zugriff auf nicht kontinuierliche Daten an.
Um Verbünde aus SMP-Knoten optimal nutzen zu können, verwendet ARMCI auf solchen Systemen ein hybrides Kommunikationsmodell. Beim Start der verteilten Applikation stellt die Bibliothek fest, welche Prozesse auf welchem Knoten laufen. Prozesse auf demselben Knoten werden dann zu sog. Clustern zusammengefasst. Innerhalb eines Clusters findet die Kommunikation ausschließlich über den gemeinsamen Speicher statt, was mittels System V shared-memory Segmenten realisiert ist. Es werden also keinerlei Nachrichten innerhalb eines Clusters ausgetauscht. Zwischen den Clustern wird hingegen mittels Nachrichtenaustausch kommuniziert.
Neben dem Leistungsgewinn durch Ausnutzung der gegenüber dem Netzwerk viel größeren Speicherbandbreite wird auch die Kommunikation zwischen den Clustern vereinfacht: Um entfernten Prozessen den Zugriff auf cluster-lokalen Speicher zu ermöglichen, reicht es aus, einen Prozess pro Cluster für die Kommunikation abzustellen. Es ist nicht nötig, dass jeder Prozess mit jedem anderen kommuniziert. Stattdessen existiert in jedem Cluster ein sog. Datenserver, der die Kommunikation mit den Prozessen der anderen Cluster abwickelt. Dieser Datenserver ist auf zwei verschiedene Arten implementiert. Auf Systemen, die eine pthread-Bibliothek zur Verfügung stellen, besteht der Datenserver aus einem thread innerhalb eines Prozesses eines Clusters, der in einer einfachen Nachrichtenschleife arbeitet. Auf Systemen, die diese Funktionalität nicht bieten, wird stattdessen ein schwergewichtiger Prozess verwendet, der mittels `fork()` beim Start der Applikation erzeugt wird. Da dieser eigenständige Prozess während des Programmlaufs mit Konfigurationsinformationen (z.B. über die shared memory Segmente) versorgt werden muss, ist in einer solchen Konfiguration eine gewisse lokale Kommunikation zwischen Datenserver und Clusterprozessen nicht zu vermeiden.
Um die Vorgänge innerhalb von ARMCI etwas zu veranschaulichen, ist in **Abbildung 1** beispielhaft der Ablauf einer PUT Operation dargestellt. In diesem Beispiel sind zwei Cluster mit je vier Mitgliedern und einem Datenserver dargestellt. Einer der Prozesse führt eine ARMCI-Put Operation aus, die Daten aus dem lokalen Speicher in den globalen Adressraum überträgt. Ein Teil der Daten wird lokal mittels `memcpy` in den gemeinsamen Speicher des lokalen Clusters übertragen, der Rest der Daten über das Netzwerk zum Datenserver des anderen Clusters gesendet und von diesem in den gemeinsamen Hauptspeicher seines Clusters übertragen.

3. Remote User Callbacks in ARMCI

Um eine zu starke Verflechtung von ARMCI und der zu implementierenden globalen Ein-/Ausgabe zu vermeiden, wurde GIO nicht direkt in ARMCI integriert. Stattdessen erfolgte zunächst der Entwurf einer Zwischenschicht, die von beliebigen Erweiterungen zur Kommunikation zwischen verschiedenen Clustern genutzt werden kann, ohne sich um Kommunikationsdetails kümmern zu müssen. Sie bietet die Möglichkeit, durch Versenden einer Nachricht im Datenserver eines beliebigen Clusters, benutzerdefinierte Funktionen ausführen zu können. Diese Funktionalität wird als RUC (Remote User Callback) bezeichnet und ähnelt stark dem RPC (Remote Procedure Call) Mechanismus, ohne jedoch dessen Mächtigkeit und Vielfalt zu erreichen.

Die Einschränkung, dass RUCs nur im Datenserver eines Clusters ausgeführt werden können, ist eine direkte Folge der Architektur von ARMCI. Da ein Prozess nur mit dem Datenserver eines Clusters kommunizieren kann, ist es offensichtlich auch nicht möglich, einen RUC Aufruf in einem anderen als einem Datenserverprozess auszulösen. Jede Erweiterung, die RUCs nutzen möchte, muss dies berücksichtigen.

Beim Entwurf der RUC Schnittstelle wurde besonders Wert darauf gelegt, sie so einfach wie möglich zu gestalten. Im Wesentlichen wurden daher nur drei neue Funktionen implementiert.

```
ARMCI_Callback_handle_t
ARMCI_Callback_register(RUC_t   func);
```

dient dazu, einen RUC beim System zu registrieren. Hierbei handelt es sich um eine kollektive Funktion, die einen eindeutigen Bezeichner für den neuen RUC zurückgibt.

```
int ARMCI_Callback_send(
Armci_Callback_handle_t h,
int proc, void *header, int header_size,
void *data, int data_size, void *result,
int exp_res_size, int *res_size);
```

dient dazu, die durch `h` spezifizierte Funktion im zu `proc` gehörenden Datenserver auszuführen. Dabei werden die in `header` und `data` spezifizierten Daten zum Zielprozess transportiert und dort beim Empfang als Parameter an die RUC Funktion übergeben. Diese kann in `*result` ein Ergebnis zurückliefern, das dann an den aufrufenden Prozess zurückgesendet wird. `ARMCI_Callback_send()` arbeitet grundsätzlich asynchron. Daher ist es nötig, mit dem von der Funktion zurückgelieferten Handle eine Wartefunktion aufzurufen, bevor die im Aufruf angegebenen Puffer wiederverwendet werden dürfen. `ARMCI_Callback_wait(int rqid)` blockiert, bis die in `rqid` spezifizierte Operation abgeschlossen ist.

Dieser Ansatz vereinigt zwei Vorteile in sich. Zum Einen war er ohne aufwändige Änderungen in ARMCI zu integrieren und kann von beliebigen Erweiterungen des Datenservers verwendet werden. Zum Anderen ist er durch seine Simplizität für den Entwickler einer Erweiterung leicht zu verstehen und zu benutzen, ohne dass auf Details der Netzwerkkommunikation geachtet werden muss.

4. Die Global I/O Bibliothek

Angeregt von den Anforderungen der Entwickler des Pakets MOLPRO, wurde die Schnittstelle der neu zu implementierenden Ein-/Ausgabebibliothek entwickelt. Es sollte dabei die Möglichkeit bestehen, sehr große Dateien effizient nutzen und über verschiedene Festplatten auf unterschiedlichen Knoten verteilen zu können. Für den Benutzer soll dabei kein Unterschied zwischen lokalen und entfernten Platten bestehen und auch die Möglichkeit bestehen, innerhalb eines SMP-Knotens verschiedene Festplatten zu nutzen, um so eine Leistungssteigerung zu erzielen. Des weiteren soll der Benutzer in der Lage sein, die Verteilung der Daten auf die einzelnen Prozesse zu beeinflussen, falls das gewünscht ist.

Um semantisch unterschiedliche Daten in einer Datei zusammenfassen zu können, und um effizient auf diese zugreifen zu können, besteht eine GIO Datei aus beliebig vielen Records. Diese können unterschiedlich groß und auf verschiedene Weise auf die teilnehmenden Prozesse verteilt sein. Z.Zt. ist nur eine geblockte Verteilung möglich, wobei jeder Prozess maximal einen Block pro Record verwaltet. Dies ist ein einfacher und für den Anwender leicht zu verstehender Ansatz.

Der Zugriff auf den Inhalt eines Records erfolgt über die Angabe eines Recordhandles und eines Offsets, wobei auf die Verteilung der Daten keine Rücksicht genommen werden muss. Die Bibliothek wickelt evtl. notwendige Kommunikation transparent ab.

Die Schnittstelle der Bibliothek gliedert sich in zwei Teile: Kollektive und einseitige Operationen. Zu ersteren zählen hauptsächlich das Anlegen und Zerstören von Dateien und Records. Alle anderen Operationen, bei denen die wichtigsten die Lese- und Schreiboperationen sind, können von jedem Prozess ohne Mitarbeit anderer aufgerufen werden. Exemplarisch sollen im Folgenden einige Funktionen dargestellt werden.

1. `int GIO_Open(IN char *path_name, OUT int *file_handle)`
   Diese kollektive Operation dient dem Anlegen einer Datei. Jeder Prozess kann einen lokalen Pfad angeben, in dem die diesem Prozess gehörenden Daten abgespeichert werden sollen. Dieser muss auch für alle anderen Prozesse des Clusters gültig sein.

2. `int GIO_Create_record_irreg(`
   `IN int file_handle,`

```
OUT int *record_handle,
IN int *segs, IN int num_segs);
```

Zum Anlegen eines neuen Records müssen alle Prozesse diese Funktion aufrufen. In `segs` wird die Verteilung der Daten spezifiziert. `segs[i]` gibt die Größe des Blocks für Prozess `i` an. Jeder Prozess kann somit maximal einen Block erhalten, die Gesamtlänge des Records ergibt sich aus der Summe der Einzelblöcke. Die Syntax und Semantik dieser Funktion lehnt sich stark an die GA Funktion NGA_Create_irreg() an, was die Nähe der beiden Konzepte unterstreichen soll.

3.
```
int GIO_Read_asyn(
IN int file_handle,
IN int record_handle, IN int lo, IN
int hi, OUT char *buf,
IN int stride);
```

Asynchrone, einseitige Operation zum Lesen aus einer Datei. Um nicht kontinuierliche Daten einfach verarbeiten zu können, werden `n=hi-lo+1` Bytes von `buf[stride*(i-1)], i=1..n` ab Offset `lo` in den angegebenen Record der Datei geschrieben. Die Funktion kehrt sofort zurück. Bevor der Puffer wiederverwendet werden kann, muss eine Wartefunktion aufgerufen werden.

Weitere Funktionen dienen dem Schreiben in eine Datei und dem Abfragen von Verteilungsinformationen, die den entsprechenden Funktionen von GA entsprechen. Zusätzlich existiert eine Akkumulationsfunktion, die ähnlich NGA_Acc(), Daten aus einem Speicherbereich und einer Datei akkumuliert.

4.1. Implementierung

Wie bereits erwähnt, verwendet die Implementierung von GIO den RUC Mechanismus von ARMCI, der extra für diesen Zweck entwickelt und implementiert wurde. Des weiteren wird das native Dateisystem verwendet, um die Daten persistent abzulegen.

Dabei ergibt sich das Problem, dass RUCs nur im Datenserver eines Clusters, nicht aber in beliebigen Prozessen der Applikation ausgeführt werden können. Da jeder Prozess beim Aufruf von `GIO_Open()` einen anderen Pfad angeben darf, besitzt jeder der Prozesse eine eigene, lokale Datei, in der die dem Prozess zugehörigen Daten gespeichert werden. Diese lokalen Dateien müssen von allen anderen Prozessen innerhalb des Clusters aus zugreifbar sein. Nur so kann sichergestellt werden, dass der Datenserver eines Clusters Anforderungen aus anderen Clustern korrekt bearbeiten kann. Gleichzeitig ist durch diese Bedingung aber die Möglichkeit geschaffen worden, analog zur Verwaltung des globalen Adressraums, innerhalb eines Clusters beim Zugriff auf die globalen Daten auf Kommunikation zu verzichten. Jeder Prozess kann auf die von ihm benötigten Daten direkt zugreifen, sofern diese von einem Prozess im selben Cluster verwaltet werden.

Während der Initialisierung von GIO, die automatisch beim ersten Aufruf von `GIO_Open()` durchgeführt wird, registriert die Bibliothek fünf verschiedene RUCs, die dem Öffnen, vor allem aber dem Lesen und Schreiben von Dateien in fremden Clustern dienen. Um den lokalen Datenserver darüber zu informieren, dass eine neue Datei geöffnet wurde und aus welchen lokalen Einzeldateien sie besteht, existiert ein spezieller RUC, der allerdings nur dann zum Einsatz kommt, falls der Datenserver als eigener Prozess realisiert ist.

Desweiteren wird während der kollektiven Operationen auf die Kommunikationsschicht von ARMCI zurückgegriffen, um Daten zwischen einzelnen Prozessen auszutauschen. Dies ist leider nicht zu vermeiden, da RUCs ja nur im Datenserver ausgeführt werden können.

Beim Anlegen einer neuen Datei finden die folgenden Vorgänge statt:

1. Erzeuge einen eindeutigen lokalen Dateinamen, basierend auf dem übergebenen Pfad und lege eine Datei mit diesem Namen an.
2. Teile diesen Dateinamen allen anderen Prozessen mit und empfange die Namen der anderen Prozesse. Dazu werden die Funktionen zum Nachrichtenaustausch in ARMCI, insbesondere ein Broadcast, verwendet.
3. Falls ich der Prozess mit der kleinsten ID im Cluster bin und ein Serverprozess verwendet wird: Teile alle Dateinamen dem Datenserver mit. Dazu wird der oben erwähnte RUC verwendet.

Beim Zugriff auf einen Record besteht kein Unterschied zwischen lesenden und schreibenden Operationen. Zur Vereinfachung werden daher in den folgenden Erläuterungen nur Leseoperationen beschrieben.

Fordert der Benutzer mittels `GIO_Read()` einen Teil der Daten eines Records an, so muss zunächst festgestellt werden, welche Prozesse die benötigten Daten besitzen. Mittels dieser Information ist es dann möglich auszurechnen, wo in der lokalen Datei des Besitzers die Daten abgespeichert sind. Dazu benötigt jeder Prozess die vollständige Information darüber, aus wievielen Records die globale Datei besteht und wie diese verteilt sind. Aus diesem Grund ist das Anlegen eines neuen Records eine kollektive Operation. Der Ablauf eines Zugriffs auf eine globale Datei läuft also folgendermaßen ab:

1. Berechne eine Liste, die die Verteilung der benötigten Daten wiedergibt. Diese enthält pro

Eintrag: physikalischer Offset, Länge des Blocks und Nummer des Prozesses, der die benötigten Daten besitzt.

2. Für alle Elemente der Liste:
   Falls die lokale Datei des Pozesses zugreifbar ist, führe einen lokalen Zugriff aus.
   Falls nicht, kontaktiere mittels `ARMCI_Callback_send()` den zuständigen Datenserver. Gib eine requestid zurück, die die ausgeführten Operationen beschreibt.

Da RUC-Aufrufe asynchron sind und viele Betriebssysteme auch asynchrone Ein-/Ausgabeoperationen unterstützen, ist so eine Überlappung der Teilzugriffe möglich, was die Leistung des Systems verbessern sollte.

Der hybride Aufbau von GIO folgt dem physikalischen Aufbau eines Rechnerverbundsystems und verspricht daher eine gute Ausnutzung solcher Architekturen. Zusätzlich ist er für den Anwender leicht zu verstehen, ermöglicht so die effiziente Nutzung und verringert die Anzahl potenzieller Fehlerquellen. Desweiteren ist es mittels der RUCs möglich, den Implementierungsaufwand gering zu halten, was wiederum in einer höheren Leistungsfähigkeit resultiert.

5. Leistung des Systems

In diesem Kapitel sollen einige grundlegende Messwerte vorgestellt werden. Alle Messungen wurden auf dem am ZAM installierten PC Cluster unter Linux durchgeführt.

Um die Ergebnisse bewerten zu können, wurden neben der Leistung von GIO auch Vergleichswerte für NFS und das lokal verwendete ext2 Dateisystem ermittelt. Dabei wurden für Zugriffe auf NFS und ext2 die Systemaufrufen read/write verwendet. Da auch GIO im Endeffekt das lokale Dateisystem, in diesem Fall also ext2, verwendet, ist offensichtlich, dass GIO keine besseren Bandbreiten erreichen kann als ext2. Das Ziel sollte es also sein, den durch das GIO System erzeugten Overhead möglichst gering zu halten und beim Zugriff auf entfernte Dateien das Netzwerk optimal zu nutzen. Um festzustellen, ob tatsächlich das Netzwerk einen limitierenden Faktor darstellt, wurde GIO zunächst zur Verwendung des TCP Protokolls über ein fast-ethernet Netzwerk konfiguriert. Zum Vergleich wurden dann Messwerte unter Verwendung einer TCP Implementierung auf Myrinet ermittelt. Die Messreihen sind in **Abbildung 2** mit tcp/f.e. für fast-ethernet und tcp/m. für tcp über Myrinet gekennzeichnet.

Der verwendete Benchmark legt eine 1MB große Datei an, die zunächst beschrieben und danach gelesen wird. Um die reine Dateisystemleistung zu ermitteln, wird eine weitere, sehr große Datei verwendet, um den Dateisystemcache vor Lese- oder Schreibzugriffen zu löschen.

Abbildung 2 zeigt die beim Zugriff auf diese Datei erreichten Bandbreiten für die verschiedenen Dateisysteme/Netzwerke.

Es zeigt sich, dass der von GIO induzierte Overhead vernachlässigbar gering ist. GIO erreicht eine Bandbreite, die nur im Bereich von 1% geringer ist als ein direkter Zugriff auf das ext2 Dateisystem. Der Vergleich der unterschiedlichen Netzwerke zeigt, dass offensichtlich das Netzwerk einen limitierenden Faktor darstellt. Mit knapp 10 MB/s nutzt GIO beim Schreiben das fast-ethernet nahezu optimal aus. Dass ein schnelleres Netzwerk die Leistung weiter verbessert, zeigen die Leistungen unter Verwendung von Myrinet. Hier steigt die Schreibbandbreite auf über 23 MB/s, was nahe an der oberen Grenze der TCP Implementierung liegt. Bei direkter Verwendung von Myrinet, also bei Verzicht auf den Umweg über TCP, ließe sich diese Leistung weiter verbessern.

Diese Untersuchung zeigt, dass GIO in der Lage ist, die gesteckten Ziele zu erreichen. Da GIO die Verteilung einer Datei auf mehrere Festplatten erlaubt, lässt sich die Leistung einer I/O intensiven Applikation durch Ausnutzung von Lokalität deutlich verbessern.

6. Ähnliche Arbeiten

Im Bereich der parallelen Ein-/Ausgabe wurden in den letzten Jahren viele verschiedene Systeme vorgestellt. Zumeist handelt es sich dabei um parallele Dateisysteme, wie z.B. PVFS [2]. Diese Systeme sind ins Betriebssystem integriert und unterscheiden sich daher substanziell vom hier vorgestellten Ansatz.

MPI-IO, als Bestandteil des MPI-2 Standards [3], legt nur eine zu implementierende Schnittstelle fest.

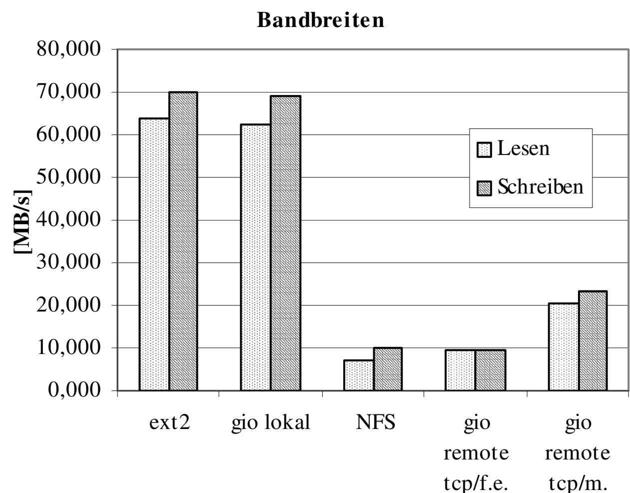

ABBILDUNG 2. Bandbreiten beim Zugriff auf 1MB Daten

Keinesfalls ist definiert, wie intern der gemeinsame Zugriff auf eine Datei zu realisieren ist. Daher stützen sich die meisten bisher existierenden Implementierungen auf existierende parallele oder verteilte Dateisysteme. Es ist durchaus möglich, GIO als Basis für eine MPI-IO Implementierung zu verwenden.

Das im System RIVER [7] eingebettete I/O Modell bietet ebenfalls globale Dateien auf Clustern an, legt seinen Fokus aber auf die Replikation von Daten, was ja bei GIO explizit unerwünscht ist, da es dem Benutzer keine Kontrolle über die Datenverteilung erlaubt.

7. Zusammenfassung und Ausblick

In diesem Artikel wurde ein Zusatz zu Global Arrays (GA) vorgestellt, der ein globales Ein-/Ausgabe Modell auf Rechnerverbundsystemen implementiert. Er ermöglicht die Verwendung von global zugreifbaren Dateien unter Ausnutzung lokaler Massenspeichermedien. Beim Entwurf dieser GIO genannten Bibliothek wurde besonders auf Portierbarkeit, aber auch auf einfache Verwendbarkeit geachtet. Der Benutzer soll in der Lage sein, das zugrundeliegende System schnell zu verstehen, um so eine effiziente Lösung seines Problems erreichen zu können.

GIO baut auf der Bibliothek ARMCI auf, die im Rahmen dieser Arbeit erweitert wurde, um die Implementierung von GIO zu vereinfachen, ohne zu tief in bestehenden Code eingreifen zu müssen.

Für die Zukunft ist zunächst eine detaillierte Leistungsuntersuchung des Systems geplant, um so einerseits Hinweise an Benutzer geben zu können, andererseits das System selber verbessern zu können. So ist z.B. die Integration einer eigenen Cache-Schicht vorstellbar, um derart die Netzwerkkommunikation auf ein Minimum verringern zu können. Dafür ist allerdings zu untersuchen, inwiefern dies dem Gedanken der benutzerspezifizierten Datenverteilung entgegenläuft, bzw. zu den Benutzer überraschenden Effekten führt.

8. Literatur

[1] R. Barriuso, Allan Knies: *SHMEM Users Guide*. CRAY Research, SN-2516, 1994

[2] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur: *PVFS: A Parallel File System For Linux Clusters.* Proc. 4th Annual Linux Showcase and Conference, Atlanta, GA, October 2000, pp. 317-327

[3] MPI-Forum: *MPI-2: Extensions to the Message-Passing Interface.* 1995-97, http://www.mpi-forum.org/docs/ mpi-20-html/mpi2-report.html

[4] J. Nieplocha, R. J. Harrison, R. J. Littlefield: *Global Arrays: A shared memory programming model for distributed memory computers.* In Proc. of the ACM/IEEE international conference on supercomputing 1994, Washington D.C., 14-18 November 1994, pp. 340-349, IEEE Computer Society Press 1994

[5] J. Nieplocha, B. Carpenter: *ARMCI: A Portable Remote Memory Copy Library for distributed Array libraries and* Compiler *Run-time Systems.* In Proc. RTSSP IPPS/SPDP'99, San Juan, 12-16 April 1999

[6] Parallel Compiler Runtime Consortium: *Common Runtime Support for High-Performance Parallel Languages.* In Proc. of the ACM/IEEE international conference on supercomputing 1993, Portland, OR, 15-19 November 1993, pp. 752-757, IEEE Computer Society Press 1993

[7] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, Kathy Yelick: *Cluster I/O with River: Making the Fast Case Common.* Proc. Sixth Workshop on Input/Output in Parallel and Distributed Systems

[8] Werner, H.-J., Knowles P. J.: *MOLPRO: User's Manual Version 2000.1*, Dec. 1999, http://www.tc.bham.ac.uk/ molpro/current/molpro_manual