Transformation paralleler Programme zur verteilten Ausführung auf einem Rechnernetz unter DCE

Bernd Dreier
Universität Augsburg
Institut für Mathematik
86135 Augsburg
Telefon: 0821 598–2116
dreier@Uni-Augsburg.DE

Theo Ungerer
Universität Karlsruhe
Institut für Rechnerentwurf und Fehlertoleranz
76128 Karlsruhe
Telefon: 0721 608–6048
ungerer@informatik.Uni-Karlsruhe.DE

Zusammenfassung

Unser Ziel ist es, parallele Programme für Rechnernetze verfügbar zu machen. Insbesondere vielfädige (multithreaded) Programme, die für Multiprozessor-Workstations geschrieben sind, sollen ohne Neuprogrammierung auf ein Rechnernetz transferierbar sein. Als Zielsystem haben wir das verteilte Betriebssystem DCE gewählt, da DCE Threads unterstützt, die dem POSIX 1003.4a Draft genügen. DCE Threads können parallel auf verschiedenen Prozessoren desselben Rechners, jedoch nicht verteilt über verschiedene Rechner ausgeführt werden. DCE ermöglicht verteilte Programmierung ausschließlich über RPCs und bietet keinen globalen Adreßraum. Bei unserem Ansatz werden POSIX 1003.4a-konforme vielfädige Programme automatisch transformiert, so daß sie auf einem verteilten System unter DCE ablauffähig sind. Die Übersetzungsalgorithmen werden von einem Precompiler ausgeführt. Ein Laufzeitsystem, das auf DCE aufsetzt, implementiert einen globalen Adreßraum und verteilt die Threads auf die verschiedenen Rechner.

1. Einführung

Bekanntlich werden viele Workstation Cluster nur sehr unzureichend ausgenutzt und stehen im Prinzip zur Ausführung paralleler Programme zur Verfügung. Eine Voraussetzung dafür ist eine Softwareschicht, die es erlaubt, ein paralleles Programm mit möglichst geringen Programmänderungen auf dem verteilten System auszuführen. Derartige Systeme sind PVM [1], PARMACS [2], Linda [3], etc., die jedoch allesamt eigene Kommunikationsprimitive und eine eigene Sicht des verteilten Systems definieren.

Das Distributed Computing Environment (DCE) [4, 5] der Open Software Foundation ist ein verteiltes Betriebssystem, das die Probleme des Zugriffsschutzes und der gemeinsamen Dateiverwaltung in heterogenen Rechnernetzen löst. Parallele Programmierung ist durch POSIX 1003.4a-kompatible Threads und die verteilte Programmierung durch

Remote Procedure Calls möglich. Leider erfordert die Verwendung von RPCs in DCE ein umfangreiches Umprogrammieren schon vorhandener Software, die für eine parallele Ausführung unter einem vielfädigen Betriebssystem geschrieben wurde. Zudem besteht natürlich kein gemeinsamer Adreßraum für die verschiedenen RPCs, so daß bei der Umsetzung von parallelen Threads auf RPCs Änderungen der gesamten Programmstruktur notwendig sind.

Unser Ziel ist es, POSIX 1003.4a-kompatible Threads, wie sie in vielen vielfädige Betriebssystemen wie Solaris 2.3, OS/2 oder Windows/NT eingesetzt werden, nach geeigneten Transformationen auf den Rechnern eines Rechnernetzes unter DCE verteilt auszuführen.

Der nächste Abschnitt gibt eine Einführung in die wichtigsten DCE-Sprachkonstrukte für die parallele und verteilte Programmierung. Danach folgt im dritten Abschnitt eine Motivation für die Implementierung eines gemeinsamen Adreßraumes in DCE. Nach Einführung eines Beispielprogramms in Abschnitt 4 wird im Abschnitt 5 unser Lösungsansatz vorgestellt. Zum Schluß werden einige praktische Erfahrungen und mögliche Optimierungen aufgezeigt.

2. DCE

DCE benutzt als grundlegende Elemente das Client/Server-Modell [6], die DCE Remote Procedure Calls (RPCs), und DCE Threads. Client und Server sind abstrakte Begriffe, beispielsweise Programme, wobei Server Dienste für Client-Programme zur Verfügung stellen.

DCE RPCs ermöglichen eine verteilte Ausführung durch die Aktivierung von Prozeduren auf anderen Maschinen des Rechnernetzes. In Gegensatz zu Prozeduren, die nur lokal aufgerufen werden, können die Prozeduren, die durch entfernte Prozeduraufrufe aktiviert werden, nicht auf globale Variabeln zugreifen — es existiert somit *kein* gemeinsamer Adreßraum mit dem aufrufenden Programm. Ihre Ausführung geschieht jedoch wie bei lokalen Prozeduraufrufen synchron, d.h. das aufrufende Programm wartet auf die Beendigung des entfernten Prozeduraufrufs. Auf diese Weise unterstützen RPCs eine verteilte, aber keine parallele Ausführung. Mittels der DCE RPCs kann ein Client folglich die Dienste eines entfernten Server nutzen: Der Client aktiviert eine (entfernte) Prozedur, die auf dem Server-Rechner ausgeführt wird.

Neben der gesamten Kommunikation organisiert DCE die notwendigen Datenkonvertierungen selbständig. Nur die Schnittstelle zwischen Client und Server muß mittels der sogenannten Interface Definition Language (IDL) definiert werden. Ein Beispiel einer solchen IDL-Definition findet sich in Abschnitt 5.1.

Eine parallele Ausführung wird durch DCE Threads und deren Synchronisationsmethoden ermöglicht. Threads sind „leichtgewichtige“ Prozesse, wobei alle Threads eines

Client oder eines Server zu einem „schwergewichtigen“ Prozeß gehören und auf den selben Adreßraum zugreifen. Threads innerhalb eines Prozesses haben die globalen Daten, geöffnete Dateien und weitere Prozeß-Ressourcen gemeinsam. Threads von verschiedenen Prozessen, z. B. vom Client und vom Server, besitzen keine gemeinsame Daten. Threads eines Prozesses können parallel auf verschiedenen Prozessoren eines Rechners, jedoch nicht auf verschiedenen Rechnern ausgeführt werden. Um unter DCE eine parallele und verteilte Ausführung zu erreichen, müssen Threads und RPCs kombiniert werden: Threads werden (parallel) aktiviert und aus den Threads heraus RPCs aufgerufen.

Die Synchronisation von DCE Threads geschieht durch Mutexes und Bedingungsvariablen. Ein Mutex garantiert den gegenseitigen Ausschluß von Threads. Bedingungsvariablen ermöglichen es einem Thread, auf die Erfüllung einer Bedingung zu warten.

3. Implementierung eines gemeinsamen Adreßraumes in DCE

Bei lose gekoppelten Systemen — nachrichtengekoppelten Multiprozessoren oder Rechnernetzen — ist kein physikalisch gemeinsamer Speicher vorhanden. Dieser kann jedoch durch eine Softwareschicht, welche die Dienste des zugrundeliegenden nachrichtengekoppelten Kommunikationssystems verwendet, gegenüber den Anwenderprogrammen vorgespiegelt werden. Ein solches gemeinsames Speichermodell, angewandt auf lose gekoppelte Systeme wird als *Distributed Shared Memory* [7, 8, 9] bezeichnet. Durch die Einführung von Distributed Shared Memory können die Threads eines Anwenderprogramms auf den verteilten gemeinsamen Speicher wie auf einen lokalen Speicher zugreifen und auf verschiedenen Prozessoren oder Rechnern parallel ausgeführt werden. Daher kann Software, die für Uniprozessorssysteme oder für speichergekoppelte Multiprozessoren geschrieben ist, auch auf einem Rechnernetz ausgeführt werden.

DCE Threads und die zugehörigen Synchronisationprimitive sind — wie die Threads in den vielfädigen Betriebssystemen Solaris, OS/2, Windows/NT und vielen anderen — POSIX 1003.4a-konform. Durch Implementierung eines gemeinsamen Adreßraumes in DCE können POSIX 1003.4a-konforme vielfädige Anwenderprogramme, die ursprünglich für Multiprozessor-Workstations geschrieben wurden, ohne Neuprogrammierung auch auf mehreren Rechner eines Rechnernetzes ausgeführt werden.

Bei unserem Ansatz benutzen wir eine Softwareschicht, die auf DCE aufsetzt und es ermöglicht, mehrere Threads mit ihren Synchronisationsoperationen parallel auf einem Rechnernetz auszuführen. Diese Softwareschicht besteht aus einem Precompiler und einem Laufzeitsystem, die ein Umprogrammieren des ursprünglichen Programms unnötig machen.

Im Gegensatz dazu müssen für DCE geschriebene verteilte Anwendungen in einem streng Client/Server-orientierten Entwurf vorliegen. Serverprogramme müssen erdacht und die

Threads so umgeändert werden, daß RPCs entstehen. Das führt zur Notwendigkeit einer zeitraubenden Umprogrammierung der ursprünglichen Software.

Da unsere Implementierung auf DCE aufsetzt, können alle Vorteile von DCE, wie die Gewährleistung des Zugriffsschutzes, die gemeinsame Dateihaltung und die Organisation der Kommunikation in heterogenen Netzen genutzt werden. Außerdem stellt DCE einen De-facto-Standard für verteilte Betriebssysteme dar.

4. Ein Beispiel

Das folgende vielfädige Beispielprogramm approximiert π durch die näherungsweise Berechnung des bestimmten Integrals über $f(x) = \frac{4}{(1+x^2)}$ zwischen 0 und 1 mittels der Rechteckregel.

```
#include "pi.h" /* includes standard headers and */
                /* contains prototype of eval    */

double total=0;
pthread_mutex_t total_mutex;
int number_workers, intervals;

void eval( int position )
{
  int           first, current, last;
  double        width, tmp, sum = 0;

  width         = 1.0 / (double) (number_workers * intervals);
  first         = position * intervals;
  last          = first + intervals;

  for ( current = first; current < last; current++ )
  {
    tmp = (0.5 + (double) current) * width;
    sum += width * (4.0 / (1.0 + tmp * tmp));
  }
  pthread_mutex_lock( &total_mutex );
  total = total + sum;
  pthread_mutex_unlock( &total_mutex );
}

int main(void)
{
  int            i;
  pthread_t      worker_threads[MAX_NR_OF_THREADS];
  pthread_addr_t status;

  pthread_mutex_init( &total_mutex, pthread_mutexattr_default );

  /* reading of parameters left out */
```

```
  for ( i=0; i<number_workers; i++ )
    pthread_create( &worker_threads[i], pthread_attr_default,
      (pthread_startroutine_t) eval, (pthread_addr_t) i );

  for ( i=0; i<number_workers; i++ )
    pthread_join( worker_threads[i], status );
}
```

5. Unser Lösungsansatz

Die Eingabe unseres Systems ist ein POSIX 1003.4a-konformes vielfädiges Programm in striktem ANSI-C. Ein Precompiler erzeugt daraus eine IDL-Datei mit den Startfunktionen der Threads, sowie DCE-konforme Client- und Serverprogramme. Die dynamische Verteilung der Server unter Berücksichtigung des Lastausgleichs wird durch sogenannte „Server-Server“ [10] ermöglicht, welche auf jedem zur Verfügung stehenden Rechner laufen. Die Server-Server bilden ein eigenständiges System, das eine Umsetzung des Master-Slave-Konzepts darstellt.[1] Eine Laufzeitbibliothek übernimmt die Kommunikation mit diesen. Die Details der dynamischen Verteilung und das Verfahren zum Lastausgleich werden hier nicht beschrieben. Der gemeinsame Speicher von Client und Server wird durch eine eigene Laufzeitbibliothek in Zusammenarbeit mit dem Precompiler simuliert.

5.1 Erzeugung der Ausgabeprogramme

Dieser Abschnitt gibt einen Überblick über die Transformation des vielfädigen Eingabeprogramms in eine verteilte Applikation. Da unser Ausgabeprogramm durch die vom Client ausgehende automatische Verteilung dem Master-Slave-Konzept entspricht, wollen wir im folgenden von Master- und Slaveprogrammen sprechen.

Der Hauptthread des Eingabeprogramms wird zum Masterprogramm transformiert. Die Startroutine eines Thread wird Bestandteil der Slaves, welche über mehrere Rechner verteilt sein können. Der Start eines Thread im Eingabeprogramm wird in zwei Aktionen des Master übersetzt: Ein Slave wird als DCE-Server auf einer möglichst wenig belasteten Maschine gestartet und anschließend wird ein RPC zu diesem veranlaßt. Master und Slave wird derselbe gemeinsame Adreßraum vorgespiegelt, den die Threads des Eingabeprogramms benutzten.

Der Precompiler erzeugt aus den Funktionsprototypen der Thread-Startroutinen des Eingabeprogramms automatisch eine entsprechende IDL-Definition der Schnittstelle zwischen Master und Slave. Hardware-abhängig implementierte C Datentypen werden dabei in die Grunddatentypen des DCE übersetzt. Außerdem erhält die Startfunktion

[1] DCE selbst bietet eine solche Möglichkeit bisher nicht. Die Notwendigkeit dieser Funktionaliät wurde aber offenbar auch von der OSF erkannt, denn mit dem Erscheinen von DCE 1.1 (Ende 1994) wird sie von DCE selbst zur Verfügung gestellt.

eval() des Beispielprogramms einen Parameter des Typs handle_t, der den jeweiligen Empfänger des RPC spezifiziert.

```
/* IDL definition */
[uuid(006489e4-feb8-1d71-a31d-02608c2f76eb),version(1.0)]
interface piapprox
{
void eval(
  [in] handle_t binding,
  [in] long position );
}
```

Das Masterprogramm geht aus dem Eingabeprogramm hervor, indem man die Startroutinen und deren Deklarationen herausnimmt. Der zusätzliche Aufruf der Funktion sm_init() dient der Initialisierung der Bibliothek für den gemeinsamen Adreßraum. pthread_create()-Aufrufe werden durch Bibliotheksaufrufe von rthread_create() mit zwei zusätzlichen Parametern ersetzt, die den Namen des Slaveprogramms, sowie Platz für dessen DCE-Adresse enthalten. rthread_create() initiiert den Start eines Slave auf einem der zur Verfügung stehenden Rechner. Anschließend führt ein lokaler Thread den RPC der Funktion eval() des Slave aus. sm_init() und alle mit rthread_ beginnenden Funktionen sind Teil unserer Bibliothek zur Implementierung des gemeinsamen Adreßraums.

```
/* master program */
#include "smsim.h" /* definitions for shared memory simulation */
#include "piapprox.h" /* generated automatically by IDL compiler */
#include "pi.h" /* modified header of the threaded program,
                   prototype of eval removed */

/* definition of the global variables as idl_* types */
idl_long_float total=0;
pthread_mutex_t total_mutex;
idl_long_int number_workers, intervals;

int main(void)
{
  int            i;
  pthread_t      worker_threads[MAX_NR_OF_THREADS];
  pthread_addr_t status;

  pthread_mutex_init( &total_mutex, pthread_mutexattr_default );

  /* reading of parameters left out */

  sm_init(); /* initialize shared memory */

  for ( i = 0; i < number_workers; i++ )
    rthread_create( &worker_threads[i], &binding[i], _slave_program,
      pthread_attr_default, (pthread_startroutine_t) eval, (pthread_addr_t)i )
```

```
  for ( i=0; i<number_workers; i++ )
    pthread_join( worker_threads[i], status );

  sm_cleanup(); /* cleanup of DCE information */
}
```

Das Slaveprogramm entsteht durch das Einfügen der Startfunktion(en) in einen festen Programmrahmen, der das Setup eines DCE Server implementiert. Die Struktur des Programmrahmens bleibt für alle möglichen Slaveprogramme unverändert. Der Programmrahmen mit den DCE-spezifischen Teilen ist hier nicht abgedruckt. Die Funktion `eval()` wird durch lokale Kopien der globalen Variablen (außer Synchronisationsvariablen) erweitert.

```
/* slave program */
#include "smsim.h" /* definitions for shared memory simulation */
#include "piapprox.h" /* generated automatically by IDL compiler */
#include <dce/idlbase.h> /* standard DCE includes */
#include <dce/rpc.h>
#include "pi.h" /* modified header without prototype of eval */

void eval( /* [in] */ handle_t binding, /* additional parameter */
           /* [in] */ idl_long_int position )
{
  int           first, current, last;
  double        width, tmp, sum = 0;

  /* following variables have to be inserted */
  idl_long_float total;
  pthread_mutex_t total_mutex;
  idl_long_int number_workers, intervals;

  /* reading global variables for subsequent use */
  read_global( NUMBER_WORKERS, &number_workers, IDL_LONG_INT );
  read_global( INTERVALS, &intervals, IDL_LONG_INT );

  width         = 1.0 / (double) (number_workers * intervals);
  first         = position * intervals;
  last          = first + intervals;

  for ( current = first; current < last; current++ )
  {
    tmp = (0.5 + (double) current) * width;
    sum += width * (4.0 / (1.0 + tmp * tmp));
  }

  /* first lock remote mutex, then read global data */
  rthread_mutex_lock( TOTAL_MUTEX );
  read_global( TOTAL, &total, IDL_LONG_FLOAT );
  total = total + sum;
  write_global( TOTAL, &total, IDL_LONG_FLOAT );
  rthread_mutex_unlock( TOTAL_MUTEX );
}
```

5.2 Synchronisation

Alle Synchronisationsoperationen auf Mutexes oder Bedingungsvariablen, die im Eingabeprogramm vorkommen, werden auch zur Synchronisation der Ausgabeprogramme verwendet. Die Synchronisationsoperationen im Slave werden durch „nested“ RPCs zum Master implementiert. Der Master spaltet während der Initialisierung einen „Listenthread“ ab, der alle eingehenden Synchronisationsaufträge, sowie Lese- oder Schreiboperationen auf globalen Variablen in Empfang nimmt. Der Start dieses Listenthread wird implizit durch den Aufruf der Funktion `sm_init()` durchgeführt. `sm_init()` richtet weiterhin die Verbindung der Slaves zum Listenthread ein und initialisiert die Laufzeitbibliothek des Server-Server Systems. Die Implementierung von `sm_init()` kann hier wegen des beschränkten Platzes leider nicht abgedruckt werden.

Synchronisationsoperationen in den Slaves werden in RPCs zum Listenthread des Master übersetzt. Im Beispielprogramm wird der `pthread_mutex_lock()`-Aufruf durch ein `rthread_mutex_lock()` ersetzt. Dadurch wird im Master durch den Listenthread ein eigener Thread gestartet. Dieser führt ein lokales `pthread_mutex_lock()` aus. Für alle anderen Synchronisationsoperationen, auch für Operationen auf Bedingungsvariablen, wird dieses Verfahren genauso angewandt. Für die Korrektheit dieses Vorgehens ist die gegebene Synchronität der RPC von größter Wichtigkeit!

5.3 Implementierung des gemeinsamen Adreßraums

Globale Daten des Eingabeprogramms werden globale Daten des Master. Jeder Zugriff der Slaves auf globale Daten wird in einen RPC zum Listenthread des Master übersetzt. Der Precompiler fügt in die Startroutinen im Slave lokale Kopien aller globalen Variablen (mit Ausnahme der Synchronisationsvariablen) ein. Jedes Vorkommen einer globalen Variablen im Slave als R-Value wird durch einen vorhergehenden RPC `read_global()` erweitert, wodurch der aktuelle Wert der globalen Variablen vom Master auf die Kopie des Slave übertragen wird. An den Zugriff auf eine globale Variable als L-Value wird ein `write_global()` angehängt, der Wert der lokalen Kopie überschreibt die globale Variable im Master.

Die Variablen `total, total_mutex, intervals`, und `number_workers` sind globale Variablen unseres Beispielprogrammes. Außer der Synchronisationsvariablen `total_mutex` wird für jede dieser Variablen eine lokale Kopie im Slave angelegt. Da `total` in der Anweisung `total = total + sum` des Eingabeprogramms sowohl als R-Value, als auch als L-Value vorkommt, wird diese Anweisung in die Sequenz

```
read_global( TOTAL, &total, IDL_LONG_FLOAT );
total+=sum;
write_global( TOTAL, &total, IDL_LONG_FLOAT );
```

im Slaveprogramm übersetzt. `total` bezieht sich dabei auf die lokale Variable `total` des Slave. Die Konstante `TOTAL` dient dabei der Identifizierung der richtigen Variable durch den Listenthread des Master.

6. Praktische Erfahrungen und Ausblick

Die Software zur dynamischen Verteilung der Slaves, sowie die Bibliotheken zur Implementierung des gemeinsamen Adreßraums sind fertiggestellt und bereits seit einiger Zeit im Einsatz. Die Transformationsschritte des Precompilers sind definiert, wobei wir bis jetzt einer Einschränkung bei der Verwendung von Zeigern als globale Daten unterliegen. Auf dieser Grundlage haben wir einige Beispielprogramme handübersetzt, um Erfahrungen zu gewinnen. Die Realisierung des gemeinsamen Adreßraums als Schicht über DCE garantiert die Kompatibilität zu allen möglichen DCE Plattformen. Wie stark sich diese Lösung auf die Performance auswirkt, ist Gegenstand unserer weiteren Arbeit.

Um die Leistung unseres Systems einzuschätzen, müssen wir die Performance unseres Ausgabeprogramms mit der einer neuprogrammierten, verteilten DCE Applikation messen. Gleichzeitig darf man aber den Aufwand der Neuprogrammierung und das Fehlen des Master-Slave-Konzepts für die DCE-Applikation (d.h. die Server der Applikation müssen auf der jeweiligen Maschine *manuell* gestartet werden) nicht unterschätzen.

Die bisherigen Beispielprogramme wurden den definierten Transformationen entsprechend handübersetzt. Die π-Approximation zeigt einen sehr guten Speed-Up von 3.22 auf vier `IBM RS/6000` Workstations über 400 000 000 Intervalle. Eine neuprogrammierte DCE-Applikation erzielte einen Speed-Up von 3.38. Um ähnlich gute Ergebnisse auch für schlecht-konditionierte Probleme mit vielen Zugriffen auf globale Daten, wie z.B. die Matrixmultiplikation, zu erreichen, gibt es viele interessante Möglichkeiten das bisher geschaffene System zu optimieren:

- Bislang wird außer dem vielfädigen Eingabeprogramm kein zusätzliches Wissen über die Daten bekanntgegeben. Die Einteilung der Daten in verschiedene Klassen, wie die Kennzeichnung von read-only Daten, würde es erlauben, diese Daten nur einmal beim Start des Slave zu diesem zu kopieren.

- Um die korrekte Datenkonvertierung zu gewährleisten, wird immer auf einen einzelnen Grunddatentyp zugegriffen. Ohne diese Bedingung zu verletzen ist es aber möglich, ganze Arrays zu übertragen. Gerade bei read-only Daten ist dies wegen der unnötigen Synchronisation leicht zu realisieren. Die Implementierung dieser und der obengenannten Optimierung würde den Datenverkehr bei der Matrixmultiplikation bereits auf das Minimum einer neuprogrammierten DCE-Applikation reduzieren.

- Im bisherigen Ausgabeprogramm wird versucht, durch das sofortige Lesen bzw. Schreiben vor bzw. nach jedem Zugriff auf globale Daten die sequentielle Konsistenz [7, 11] dieser Daten zu garantieren. Da in den vielfädigen Eingabeprogrammen die Synchronisation beim Zugriff auf gemeinsame Daten vorausgesetzt wird, können wir zur Verwendung eines zu diesem Verfahren passenden Konsistenzmodell übergehen. Es handelt sich dabei um die schwächere Entry Konsistenz [12]. Konsistenz von gemeinsamen Daten wird dabei nur nach der Anwendung eines

Synchronisationsprimitivs gewährleistet. Für unsere Ausgabeprogramme würde dies bedeuten, geschützte Daten nur einmal nach einem `pthread_mutex_lock()` zu lesen und nur einmal vor dem korrespondierenden `pthread_mutex_unlock()` zu schreiben. Um dies zu realisieren müßte dem Precompiler die Bindung von Daten an Mutexes bekannt gegeben werden. Da diese aber nur im Wissen des Programmierers existiert, ist hier die explizite Angabe der Bindungen von Daten an Mutexes notwendig.

Die weitere Arbeit des Projekts konzentriert sich vor allem auf die Anwendung und Auswertung obengenannter Optimierungen, sowie die Implementierung des Precompilers.

Literatur

[1] V.S. Sunderam, A. Geist, J. Dongarra, and R. Mancheck. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20:531–546, April 1994.

[2] R. Calkin, R. Hempel, H.-C. Hoppe, and P. Wypior. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20:615–632, April 1994.

[3] N. Carriero, D. Gelernter, T.G. Mattson, and A.H. Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20:633–655, April 1994.

[4] Harold W. Lockhart Jr. *OSF DCE Guide to Developing Distributed Applications.* McGraw-Hill, Inc., 1994.

[5] John Shirley. *Guide to Writing DCE Applications.* O'Reilly & Associates, 103 Morris Street, Suite A, Sebastopol CA 95472, June 1992.

[6] O. Spaniol, C. Popien, and B. Meyer. *Dienste und Dienstvermittlung.* Thomson Verlag, Bonn, 1994.

[7] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[8] Michael Stumm and Songnian Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, pages 54–64, May 1990.

[9] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, pages 52–60, August 1991.

[10] Bernd Dreier und Markus Zahn. Entwicklung einer verteilten Programmierumgebung für das DCE. Diplomarbeit, Universität Augsburg, Oktober 1993.

[11] Kai Hwang. *Advanced Computer Architecture.* McGraw-Hill, New York, 1993.

[12] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. *IEEE Compcon*, pages 528–537, 1993.