
	<p style="text-align: center;">Parallelrechner Übung Prof. Dr.-Ing. W. Rehm</p> <hr/> <p>Praktische Übung 1: Einführung in die Programmierung mit MPI Bearbeiter: Torsten Mehlan, Timo Schneider Letzte Änderung: René Oertel, 08.05.2015</p>	
-----------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------

1 Ziele

- Einführung in die Programmierung mit MPI
- Übersetzen von MPI-Programmen
- Starten von MPI-Programmen
- Handhabung der MPI-Implementierung Open MPI

2 Einführung

Die kontinuierliche Steigerung der Rechenleistung erfolgt seit vielen Jahren durch eine Kombination aus verbesserter sequentieller Verarbeitungsgeschwindigkeit einzelner CPUs und steigender Anzahl parallel arbeitender Prozessoren. Der Grad der Parallelität nimmt dabei einen zunehmend größeren Anteil ein. Diese Entwicklung ist dadurch begründet, dass eine Vergrößerung der Taktfrequenz moderner CPUs energetische und thermische Probleme verursacht. Des Weiteren ist die hohe Komplexität von impliziten Mechanismen wie Tomasulo-Logiken, Branch-Prediction-Units oder sehr tiefen Pipelines schwer zu handhaben.

3 Das Message Passing Interface

Das Message Passing Interface (MPI) ist ein Standard, in dem die Semantik und Syntax von Funktionen und Datentypen zum expliziten Nachrichtenaustausch beschrieben wird. Dabei wird das Paradigma verfolgt, eine Berechnung explizit in parallele Unterrechnungen aufzuspalten, die sequentiell ausgeführt werden und Zwischenergebnisse durch explizite Nachrichten verbreiten. Die Details der Implementierung sind dagegen nicht spezifiziert und erlauben eine große Bandbreite an Optimierungen für spezielle Rechnerarchitekturen und Netzwerke.

3.1 Historie

In den frühen 80er Jahren existierte eine Vielzahl von Schnittstellen für Message-Passing in parallelen Anwendungen. Viele Hersteller wie SGI oder Cray implementierten und definierten proprietäre Umgebungen, die nur auf den Maschinen der jeweiligen Hersteller betrieben werden konnten, wodurch selbstverständlich die Portabilität der Applikationen nicht mehr gewährleistet war. Dies gilt als Grund für die damals nur schleppenden Bestrebungen rechenintensive Software zu parallelisieren.

Im Jahr 1992 wurde auf der Supercomputing-Konferenz in Minneapolis beschlossen einen Standard für eine Message-Passing Schnittstelle zu schaffen, den alle Hersteller implementieren konnten und der Anwendern eine reichhaltige Funktionalität zur Verfügung stellen sollte. Die Idee des MPI-Standards wurde von vielen Universitäten, kommerziellen Softwareentwicklern und nahezu allen großen Hardwareherstellern – unter anderem waren IBM, Cray, Intel und NEC beteiligt – mit viel Euphorie aufgenommen. Die Version 1.0 des MPI-Standards wurde 1994 verabschiedet und bot somit die Möglichkeit, mit der Entwicklung entsprechender Software zu beginnen. Unter [1] ist die vollständige Dokumentation des Standards abgelegt. Ein Jahr später gab es bereits erste Implementierungen.

Der MPI-Standard wurde 1998 um zusätzliche Funktionen erweitert. Die daraus resultierende Version 2.0 ([2]) umfasst sämtliche Funktionen der vergangenen Revisionen ohne deren Syntax oder Semantik zu verändern; allerdings dauerte es deutlich länger als 1994 bis eine vollständige Implementierung der Version 2.0 fertig gestellt wurde. Heutzutage stellen nahezu alle Supercomputer, die in der internationalen Rangliste TOP500 [3] vertreten sind, MPI in den Versionen 1.2

oder 2.0 zur Verfügung. Die populärsten Bestrebungen eine unter Open Source Lizenz stehende MPI-Implementierung zu entwerfen sind MPICH2 und Open MPI [4].

3.2 Funktionsumfang

Bei jeder Implementierung von MPI handelt es sich um eine *Bibliothek*, die gegen den Quellcode der parallelen Anwendung gelinkt werden muss. Alle MPI-Funktionen werden dabei durch den Linker aufgelöst und durch Referenzen in die entsprechende Bibliothek ersetzt. Es ist nicht vorgesehen, eine dynamisch gelinkte Version eines MPI-Programms zur Laufzeit an beliebige MPI-Bibliotheken zu binden, da der Standard ausschließlich das *Application Programmer Interface (API)* definiert, nicht jedoch das Application Binary Interface (ABI).

Neben der reinen Bibliothek sind diverse Hilfsprogramme notwendig, die den Übersetzungsvorgang von Anwendungen erleichtern, den Start eines parallelen Jobs übernehmen oder verschiedene Verwaltungsaufgaben durchführen. Beispielsweise existiert üblicherweise ein Skript mit dem Namen `mpicc`, das den Aufruf des Compilers mit sinnvollen Flags, Definitionen und Linkeranweisungen kapselt. Eine nicht zu unterschätzende Komplexität wird durch die Interaktion mit sogenannten Batch-Systemen erzeugt. Üblicherweise werden Hochleistungsrechner den Anwendern über ein solches Batch-System zur Verfügung gestellt, das die einzelnen Rechenjobs in eine Warteschlange aufnimmt und fair auf die zur Verfügung stehenden Rechner verteilt. Daraus ergibt sich die Anforderung, dass eine MPI-Bibliothek mit diesen Systemen interagieren muss, um den korrekten Start von Jobs, die Umleitung von Ein- und Ausgabedaten sowie das Aufräumen nach Beendigung der Rechnung zu gewährleisten.

Dennoch bleibt der Kern einer jeden MPI-Implementierung die Bibliothek, in der die im Standard definierten Funktionen und Datentypen realisiert werden. Wir werfen nun einen Blick auf die Funktionsbereiche von MPI:

- *Punkt-zu-Punkt Kommunikation*
Diese Funktionen dienen zum Versenden und Empfangen von Nachrichten zu bzw. von genau einem anderen Prozess. Es existieren mehrere Sende- und Empfangsfunktionen, die sich im Detail unterscheiden. Darauf werden wir weiter unten zurückkommen.
- *Kollektive Kommunikation*
Dieser Bereich umfasst die Sende- und Empfangsroutinen, an denen beliebig viele Prozesse beteiligt sind. Es können – je nach Wunsch des Programmierers – maximal alle Prozesse der Anwendung daran teilnehmen (z. B. Broadcast).
- *Prozessgruppen und Kommunikationskontexte*
MPI stellt Funktionen zur Verfügung, mit denen man die Prozesse beliebig in Untergruppen organisieren kann. Eine solche Gruppierung dient als Ausgangsbasis zur Definition verschiedener Kommunikationsräume (*Communicator*), die beispielsweise den Wirkungsbereich kollektiver Kommunikation begrenzen.
- *Profiling Interface*
Dieser Teil des Standards ermöglicht es Entwicklern und Anwendern auf einfache Weise das Verhalten der Applikation zu studieren.
- *Prozessstopologien*
Hiermit können die logischen Kommunikationsbeziehungen der Anwendung (z. B. *2D-Torus*, *Butterfly-Netzwerk*, *3D-Mesh*) beschrieben werden. Dies kann zu diversen Optimierungen genutzt werden.
- *Definition von Datentypen*
Damit die MPI-Bibliothek den korrekten Speicherinhalt an andere Prozesse versendet und deren Datenformat mit dem des Senders in Einklang gebracht werden kann, müssen Eigenschaften von Datentypen definiert werden (z. B. Größe, Ausrichtung).

4 Das MPI-Paradigma

Grundsätzlich kann eine parallele MPI-Anwendungen aus unterschiedlichen Programmen bestehen, von denen auf jedem Prozessor eines zur Ausführung gelangt. Man könnte somit das Prinzip des *Multiple Program Multiple Data (MPMD)* verfolgen und jeder Ausführungseinheit eine spezielle Aufgabe zuweisen, die sich in einem speziellen Program manifestiert. In der Praxis kommen solche Anwendungsfälle jedoch nur sehr selten vor. Üblicherweise formuliert man den parallelen Programmcode im Stile des *Single Program Multiple Data*-Prinzips (SPMD), bei dem ein identisches Program durch jeden Prozessor ausgeführt wird. Die jeweils zu bearbeitenden Daten sind für jeden Prozess verschieden und müssen mit

MPI-Funktionen explizit zu anderen Prozessen übertragen werden. Sollen einzelne Prozesse der MPI-Anwendung spezifische Aufgaben übernehmen kann dies anhand einer eindeutigen Identifikationsnummer und bedingter Codeausführung auch im SPMD-Modell bewerkstelligen.

5 Das erste MPI-Programm

Wir wollen uns nun ein Hello-World-Programm unter Verwendung von MPI ansehen. Um dieses Beispiel nachvollziehen zu können, benötigt man einen Texteditor (z. B. vim), einen Compiler und Linker (z. B. die Werkzeuge von gcc), eine MPI-Bibliothek sowie einen Zugang zu einem Parallelrechner. In diesem Praktikum verwenden wir die Rechner des RA-Pools als Clustercomputer. Tatsächlich können auch größere Cluster aus herkömmlichen Komponenten von PCs zusammengebaut werden – diese Vorgehensweise wurde an der TU Chemnitz für den im Jahr 2000 in Betrieb genommenen CLiC gewählt ¹.

5.1 Der Programmcode

Der MPI-Standard definiert Sprachanbindungen für C und Fortran (MPI-2 definiert auch eine Anbindung für C++). Wir werden sämtliche Programme in C schreiben.

```
1 #include "mpi.h"
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 #define MASTER 0
6 #define TAG 1
7
8 int main(int argc, char **argv) {
9
10     MPI_Status status;
11     int myRank, nTasks, i, Buffer;
12
13     MPI_Init(&argc, &argv);
14     MPI_Comm_size(MPI_COMM_WORLD, &nTasks);
15     MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
16
17     if (myRank == MASTER) {
18         printf("Hi from Master, my Rank is %d\n", myRank);
19         for (i = 1; i < nTasks; i++) {
20             MPI_Recv(&Buffer, 1, MPI_INT, i, TAG, MPI_COMM_WORLD, &status);
21             printf("Received Message from Node %d\n", Buffer);
22         }
23     }
24     else MPI_Send(&myRank, 1, MPI_INT, MASTER, TAG, MPI_COMM_WORLD);
25
26     printf("Bye from %d\n", myRank);
27     MPI_Finalize();
28     return 0;
29 }
```

5.2 Erläuterungen zum MPI-Program

5.2.1 Der formale Rahmen

Die Anweisung `#include "mpi.h"` muss in jedem MPI-Programme stehen, da dort alle Makros, Datentypen und Funktionsprototypen definiert werden. Als erste MPI-Funktion muss immer `MPI_Init()` gerufen werden, welche die Kommandozeilenparameter als Argumente entgegennimmt. Den Abschluss des formalen Rahmens von MPI bildet die Funktion `MPI_Finalize()`, deren Aufgabe darin besteht alle Ressourcen der MPI-Bibliothek freizugeben. Nach dieser Funktion

¹Der CHiC (<http://www.tu-chemnitz.de/chic>) als Nachfolger des CLiC besteht zwar nicht mehr aus herkömmlichen PCs, dennoch kommen im Wesentlichen Standardkomponenten zum Einsatz

MPI-Datentyp	C-Datentyp
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_LONG_DOUBLE	long double
MPI_BYTE	—
MPI_PACKED	—

Abbildung 1: Zuordnung von MPI-Datentypen und C-Datentypen

darf kein weiterer Aufruf in die MPI-Bibliothek erfolgen, insbesondere ist eine erneute Initialisierung mit `MPI_Init()` verboten.

5.2.2 Zeile 14 und 15

Das MPI-Programm muss natürlich Informationen über konkrete Laufzeitparameter erfragen können. Die wichtigsten Attribute sind die lokale Identifikationsnummer – im Folgenden als *Rank* bezeichnet – und die Anzahl der gestarteten Prozesse. Ein sauber geschriebenes Programm muss sich an jede beliebige Anzahl von Prozessen anpassen, weswegen zu Beginn des Programms fast immer die Größe des parallelen Jobs und der Rank abgefragt werden. Sind n Prozesse gestartet worden, wird jedem von ihnen ein Rank r im Bereich $0 \leq r \leq n - 1$ zugewiesen. Der MPI-Standard definiert keine spezielle Abbildung von Prozessen zu Ranks, allerdings wird garantiert, dass jeder Rank eindeutig ist.

In den beiden Funktionen `MPI_Comm_rank()` und `MPI_Comm_size()` fällt die Verwendung des Parameters `MPI_COMM_WORLD` auf, mit dessen Hilfe ein Kommunikator spezifiziert wird. Im MPI-Standard werden Kommunikatoren als grundlegendes Konzept zur Definition abgeschlossener Kommunikationsräume definiert. Der Programmierer kann beliebige Prozesse in Gruppen zusammenfassen, die zur Bildung von Kommunikatoren dienen. Dadurch ist es möglich, die Kommunikation zwischen den Prozessen der Untergruppe von anderen Kommunikationsoperationen zu trennen. Als Konsequenz muss ein Prozess immer durch die Angabe eines Kommunikators und des dazugehörigen Ranks adressiert werden.

Natürlich kann jeder Prozess der Anwendung Mitglied in beliebig vielen Kommunikatoren werden. Die Konstante `MPI_COMM_WORLD` fungiert dabei als initialer Kommunikationsraum, an dem alle Prozesse partizipieren.

5.2.3 Zeile 20 bis 24

Das Beispielprogramm verwendet die bereits in Abschnitt 3.2 erwähnte Punkt-zu-Punkt Kommunikation, um eine Nachricht von allen Prozessen zu einem Master-Prozess zu senden. Anhand des Ranks wird bestimmt, welcher Prozess als Master fungiert und auf die eingehenden Daten wartet. Insgesamt enthält der MPI-Standard 8 Sende- und 2 Empfangsfunktionen, von denen hier die Basisvarianten zur Anwendung kommen.

Jegliche Kommunikation in MPI basiert auf Datentypen. Die Bibliothek kann dadurch diverse Typüberprüfungen vornehmen und die Korrektheit der Datenübertragung in vielen Fällen absichern. Anders als Schnittstellen, die auf niedrigem semantischen Niveau operieren (z. B. POSIX Sockets), wird dem Programmierer die Arbeit abgenommen sämtliche Daten in Byte-Ströme zu konvertieren. Der MPI-Standard definiert Basisdatentypen, die den entsprechenden Datentypen der verwendeten Sprache eindeutig zugeordnet sind, indem eine Übereinstimmung über das Speicherabbild der Daten besteht. In Tabelle 1 sind die Zuordnungen zwischen den Datentypen für die Sprache C aufgelistet.

Die MPI-Datentypen `MPI_BYTE` und `MPI_PACKED` besitzen in keiner Programmiersprache einen entsprechenden Typ. Vielmehr werden sie benutzt wenn der Programmierer die Typüberprüfung der MPI-Bibliothek abschalten möchte – beispielsweise um reine Byte-Ströme zu versenden. Momentan sollen uns allerdings nur die herkömmlichen Typen interessieren.

Die in Tabelle 1 dargestellten Datentypen werden von jeder MPI-Bibliothek definiert und können ohne weiteres im Quellcode genutzt werden. Darüber hinaus gibt es die Möglichkeit aus diesen Basistypen zusammengesetzte Datentypen zu bilden, die Strukturen, Felder oder Submatrizen repräsentieren. Jeder beliebige Typ eines MPI-Programms kann über diesen Mechanismus abgebildet werden.

Wie bereits angesprochen wird der Empfänger- bzw. der Absender einer Nachricht in den Funktionen durch ein Wertepaar aus Rank und Kommunikator spezifiziert. In der Funktion `MPI_Recv()` kann anstelle des Ranks des Absenders auch die Konstante `MPI_ANY_SOURCE` übergeben werden, um passende Daten von einer beliebigen Quelle zu empfangen. Des Weiteren dient ein *Tag* zur Identifikation einzelner Nachrichten. Der Empfang von Daten kann in diesem Fall nur zustande kommen, wenn die Tags der Sende- und Empfangsfunktion übereinstimmen. Die Konstante `MPI_ANY_TAG` dient als Platzhalter für alle möglichen Tags.

5.3 Eingabe und Ausgabe

Üblicherweise werden die meisten Daten eines MPI-Programms in Dateien geschrieben. Die Standardausgabe wird dagegen bestenfalls für Fehlermeldungen, Debugging und Zeitmessungen benutzt. Obwohl es bei den meisten Implementierungen von MPI möglich ist, einem Prozess der Anwendung Daten über die Standardeingabe zuzuführen, wird davon verständlicherweise nur sehr selten Gebrauch gemacht. Vielmehr werden Einstellungen über Kommandozeilenparameter vorgenommen und größere Mengen von Eingabedaten aus Dateien gelesen.

In diesem Versuch wird die MPI-Implementierung Open MPI zum Einsatz gelangen. Bei dieser Variante wird – wie bei den meisten anderen MPI-Bibliotheken auch – die Standardausgabe auf die Konsole umgeleitet, von der die Anwendung gestartet wurde. Deswegen können wir die Ausgabe jedes Prozesses auf der lokalen Konsole beobachten, auch wenn er auf einer anderen Maschine zur Ausführung gelangt.

6 Alternative Punkt-zu-Punkt-Funktionen

Neben den bereits in Abschnitt 5.2.3 diskutierten Sende- und Empfangsfunktionen gibt es alternative Möglichkeiten eine bestimmte Semantik des Nachrichtentransfers zu erzwingen. Prinzipiell ist es denkbar, dass die MPI-Bibliothek sämtliche Daten in einen Zwischenpuffer verschiebt und von dort aus über das Netzwerk sendet. Andererseits könnte die Nachricht direkt aus dem spezifizierten Speicherbereich übertragen werden, ohne zusätzliche Speichertransfers zu initiieren. Die Semantik des herkömmlichen `MPI_Send()` erlaubt beide Vorgehensweisen gleichermaßen.

Wenn der Nutzer eine spezielle Semantik erzwingen möchte, stehen die alternativen Funktionen `MPI_Bsend()` und `MPI_Ssend()` zur Verfügung. Das „B“ in der ersten Funktion deutet an, dass die Nachricht in jedem Fall in einen Zwischenspeicher verschoben wird, bevor die MPI-Bibliothek den Netzwerktransfer startet. Das zusätzliche „S“ in der zweiten Funktion soll hingegen darauf hinweisen, dass dieser Sendevorgang eine synchronisierende Wirkung besitzt. Im Unterschied zu den anderen Sendefunktionen kann der Aufrufer sicher gehen, dass der Empfänger die zugehörige Empfangsfunktion betreten hat, wenn die Funktion zurückkehrt.

7 Benutzung von Open MPI

7.1 Open MPI im RA-Pool

Die (Open) MPI Laufzeitumgebung benötigt Informationen darüber, auf welchen Systemen die MPI-Anwendung wie ausgeführt werden soll. Diese Informationen kann man über verschiedene Wege als Kommandozeilenparameter angeben. Eine Alternative stellen die verschiedenen Batch-Systeme wie OpenPBS, SGE oder SLURM dar. **SLURM**² ist auf allen Poolrechnern installiert und kann für die faire Zuordnung von Prozessen für Berechnungen und Tests verwendet werden. Mit dem Werkzeug `sinfo` kann man sich die vorhandenen Ressourcen anzeigen lassen. Mit `salloc` Ressourcen für die interaktive Verwendung allokiert bzw. `srun/sbatch` für die direkte Ausführung von (parallelen) Applikationen auf verschiedenen Rechnern. Das Kommando `squeue` dient zur Anzeige laufender Jobs. Eine weitere Voraussetzung ist gewöhnlich die Verwendung eines gemeinsamen Dateisystems, welches bei allen beteiligten Servern vorhanden sein muss, damit die MPI-Anwendung auch dort gestartet werden kann.

Nun können wir unser Eingangsbeispiel übersetzen und ausführen:

²<http://slurm.schedmd.com/>

```
# Verzeichnis im NFS anlegen
mkdir -p /home/pool/shared/2015/${USER}
cd /home/pool/shared/2015/${USER}
which mpicc # Prüfen ob das "richtige" Open MPI gefunden wird!
mpicc pr_u1_a1.c -o pr_u1_a1
salloc -n4 -N2 # 4 Tasks auf 2 Knoten, interaktiv
mpirun ./pr_u1_a1
exit # Ressourcen wieder freigeben
```

7.2 Alternative: Eigene Open MPI Installation

Da man bei der Nutzung von Clustern oder anderen Rechnersystemen oftmals vor dem Problem steht, dass nicht alle benötigten Softwarepakete in den gewünschten Versionen vom Administrator bereits vorinstalliert sind, ist man häufig gezwungen diese Bibliotheken in sein Homeverzeichnis zu installieren. Zunächst sollten wir also Open MPI in unser Homeverzeichnis installieren. Hierzu könnte wie folgt vorgegangen werden, wobei Besonderheiten von AFS usw. beachtet wurden und die gewählten Pfade in anderen Umgebungen entsprechend angepasst werden können:

```
cd /tmp
wget http://www.open-mpi.org/software/ompi/v1.8/downloads/openmpi-1.8.5.tar.bz2
tar -xjf openmpi-1.8.5.tar.bz2
mkdir openmpi-1.8.5_tmp
cd openmpi-1.8.5_tmp
../openmpi-1.8.5/configure --prefix=${HOME}/PUBLIC/bin/openmpi-1.8.5
make -j$(grep -c ^proc /proc/cpuinfo)
make install
```

Nachdem nun Open MPI installiert ist (der Installationsvorgang nimmt einige Minuten in Anspruch), müssen wir dafür sorgen, dass diese Installation auch standardmäßig benutzt wird. Hierzu editieren Sie die Datei `~/.bashrc` und fügen folgende Zeilen an:

```
export PATH=${HOME}/PUBLIC/bin/openmpi-1.8.5/bin:$PATH
export LD_LIBRARY_PATH=${HOME}/PUBLIC/bin/openmpi-1.8.5/lib:$LD_LIBRARY_PATH
```

Open MPI verwendet standardmäßig `ssh` um Prozesse auf entfernten Rechnern zu starten. Um nicht beim Starten von MPI-Prozessen mehrfach unser Passwort eingeben zu müssen, kann man den Public-Key-Authentifizierungsmechanismus von `ssh` verwenden. Dieser wird wie folgt konfiguriert:

```
ssh-keygen # default Einstellungen verwenden (immer Return drücken)
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
echo "StrictHostKeyChecking=no" >> ~/.ssh/config
```

Bevor wir ein paralleles Programm auf mehreren Rechnern zur Ausführung bringen können müssen wir Open MPI mitteilen, welche Rechner dafür zur Verfügung stehen. Dies geschieht im einfachsten Fall über ein sogenanntes *machinefile*. Hierzu tragen Sie einfach die IP-Adressen oder die Rechnernamen aller nutzbaren Rechner in eine Textdatei ein, die Datei könnte bei uns im RA-Pool also so aussehen:

```
homer
bart
marge
lisa
maggie
rod
patty
selma
milhouse
monty
lenny
waylon
```

8 Aufgaben

8.1 Kontrollfragen

1. Was unterscheidet SPMD- von MPMD-Applikationen?
2. Welche Gründe gibt es dafür zu fordern, dass vor dem Ruf von `MPI_Finalize()` sämtliche MPI-Kommunikation des Programms abgeschlossen ist?
3. Erläutern Sie den Unterschied zwischen einem „Application Programming Interface“ (API) und einem „Application Binary Interface“ (ABI)!

8.2 Praktische Aufgaben

1. Tippen Sie das Hello-World-Programm aus Abschnitt 5.1 ab und bringen Sie dann das übersetzte Programm auf 4 Rechnern und 8 Rechnern zur Ausführung.
2. Die Anzahl der Iterationen soll über die Standardeingabe zur Laufzeit des Programms gelesen werden. Ändern Sie das Hello-World-Programm, so dass die Ausgaben der Prozesse in einer Schleife wiederholt werden. Testen Sie eine Variante, bei der nur ein Prozess von der Standardeingabe liest und eine Variante, bei der alle Prozesse von der Standardeingabe lesen.
3. Die Funktion `MPI_Send()` kann sich entweder wie `MPI_Ssend()` oder wie `MPI_Bsend()` verhalten. Schreiben Sie ein Programm, welches das Verhalten geeignet feststellt und führen Sie es aus.
4. Schreiben Sie ein Programm, das die Ausführungszeit der Funktionen `MPI_Send()` und `MPI_Ssend()` für verschiedene Nachrichtengrößen misst. Was stellen Sie fest?

9 Befehlsreferenz

Funktion	Beschreibung
<code>MPI_Init (int *argc, char ***argv)</code>	Initialisierung der Bibliothek; Die Parameter der <code>main</code> -Funktion werden übergeben
<code>MPI_Finalize ()</code>	Beenden aller MPI-Funktionen
<code>MPI_Send (void* buf, int cnt, MPI_Datatype dt, int dst, int tag, MPI_Comm com)</code>	Sendet eine Nachricht an einen entfernten Prozess. Übergeben werden eine Speicheradresse, Datentypinformationen und Adressinformationen.
<code>MPI_Recv (void* buf, int cnt, MPI_Datatype dt, int src, int tag, MPI_Comm com, MPI_Status *status)</code>	Empfängt eine Nachricht von einem entfernten Prozess. Übergeben werden eine Speicheradresse, Datentypinformationen, Adressinformationen und ein Nachrichtenstatus mit zusätzlicher Information.
<code>MPI_Bsend (void* buf, int cnt, MPI_Datatype dt, int dst, int tag, MPI_Comm com)</code>	Sendet eine Nachricht an einen entfernten Prozess. Übergeben werden die selben Parameter wie bei <code>MPI_Send()</code> . Die Funktion erzwingt das Zwischenpuffern der Daten.
<code>MPI_Ssend (void* buf, int cnt, MPI_Datatype dt, int dst, int tag, MPI_Comm com)</code>	Sendet eine Nachricht an einen entfernten Prozess. Übergeben werden die selben Parameter wie bei <code>MPI_Send()</code> . Die Funktion synchronisiert Sender und Empfänger.
<code>MPI_Comm_size (MPI_Comm com, int *size)</code>	Ermittelt die Anzahl der Prozesse im Kommunikationsraum.
<code>MPI_Comm_rank (MPI_Comm com, int *rank)</code>	Ermittelt die Identifikationsnummer des Prozesses.

Hinweise, Berichtigungen und Kritik zu den Übungsunterlagen bitte an:

- René Oertel <rene.oertel@cs.tu-chemnitz.de>

Literatur und wichtige Links

- [1] *MPI 1.0 Standard*
<http://www.mpi-forum.org/docs/mpi-10.ps>

- [2] *MPI 2.0 Standard*
<http://www.mpi-forum.org/docs/mpi-20.ps>
- [3] *TOP500-Liste der schnellsten Supercomputer*
<http://www.top500.org/>
- [4] *Homepage von Open MPI*
<http://www.open-mpi.org/>

8. Mai 2015