Hausarbeit im Seminar:
Effiziente Programmierung

MPI Datentypen

Vorgelegt am 14. März 2021

Michel Böker

michel.boeker@studium.uni-hamburg.de
Studiengang Wirtschaftsinformatik
Matr.-Nr. 7029479

Betreuer: Dr. Hermann Lenhart

Abstract

Durch das Message Passing Interface wird eine Schnittstelle angeboten, die eine nachrichtenbasierte Kommunikation bei parallelen Berechnungen ermöglicht und zudem als Standard akzeptiert wird. Diese Kommunikation zwischen den Systemen verursacht immer auch Kosten – zusätzlich zu den entstehenden Kosten für die Berechnungen. Durch die Verwendung und Konstruktion von abgeleiteten Datentypen innerhalb der MPI-Umgebung lässt sich die Anzahl der zu versendenden Nachrichten deutlich verringern und die Kosten dadurch möglichst geringhalten. Somit kann eine effiziente Kommunikation gewährleistet werden.

Inhaltsverzeichnis

1 Einleitung 3
2 Grundlagen 3
3 Kommunikation 4
3.1 Point-to-Point Kommunikation 4
3.2 Kollektive Kommunikation 5
4 Datentypen 6
4.1 Elementare Datentypen 6
5 Abgeleitete Datentypen 7
5.1 Datentyp Contiguous 8
5.2 Datentyp Vektor 9
5.3 Datentyp Indexed 10
5.4 Datentyp Struct 10
6 Zusammenfassung 11
Literaturverzeichnis 12

1 Einleitung

Das Message Passing Interface ist eine Bibliothek für die nachrichtenbasierte Kommunikation bei parallelen Berechnungen auf verteilten Systemen, die weitgehend als Standard angesehen wird.[8] Wie der Name bereits aussagt, handelt es sich bei MPI jedoch lediglich um eine Schnittstelle. Die konkrete Implementation wird beispielsweise durch die beiden größten MPI-Anbieter Open MP und MPICH frei zur Verfügung gestellt. Als Programmiersprache für MPI-Programme werden überwiegend Fortran oder C benutzt. Es werden jedoch auch bereits Implementierungen für Python angeboten.

Die Grundlage für das Message Passing Interface ist das gleichnamige Programmiermodell. In diesem Modell geschieht der Datenaustausch zwischen nebenläufigen Prozessen durch explizites Versenden und Empfangen von Nachrichten.[9] Ein großer Vorteil hierbei ist, dass der Nachrichtenaustausch auch über Rechnergrenzen hinweg funktioniert. Dadurch können beispielsweise verschiedene Rechner über ein gemeinsames Netzwerk an einem großen Problem arbeiten, das hohe Rechenleistung bzw. Speicherkapazität bedarf.

Eine weitere wichtige Eigenschaft von MPI und der damit verbundenen Nachrichtenübertragung ist die Tatsache, dass nicht - wie sonst üblich in der Programmierung – lediglich der entstehende Aufwand für die Berechnung betrachtet werden muss, sondern zudem der Aufwand für die Übertragung der Daten. Um die Kosten für die Kommunikation möglichst gering zu halten, ist es daher wichtig, die zu übermittelnden Daten möglichst effizient in einer Nachricht unterzubringen. Dabei ist es sinnvoller, eine große Nachricht als viele kleine Nachrichten zu versenden. Um dies zu erreichen werden abgeleitete Datentypen benötigt, auf die im Kapitel 4.2 näher eingegangen wird.

2 Grundlagen

Bevor es um den konkreten Aufbau eines MPI-Programms und die wichtigsten MPI-Routinen geht, wird der Fokus zuerst auf die Vorteile des Message Passing Interfaces gelegt. Wie bereits im Kapitel 1 erwähnt, bietet MPI ein standardisiertes System zur Nachrichtenübertragung. Einhergehend mit der Standardisierung gibt es zugleich auch eine gewisse Portabilität, die es erlaubt, geschriebenen Programmcode auf anderen Plattformen wiederzuverwenden, sofern diese ebenfalls den MPI-Standard unterstützen. Zudem ist MPI innerhalb der parallelen Programmierung und des Hochleistungsrechnens weit verbreitet. Das liegt unter anderem auch an der Skalierbarkeit. MPI erlaubt es explizit, dem zur Verfügung stehenden Speicher eine gewisse Anzahl an Prozessen zuzuweisen.[7] Zudem gibt es eine umfassende Palette an unterschiedlichen

Funktionen, die innerhalb der MPI-Umgebung benutzt werden können. Jedes MPI-Programm besteht dabei mindestens aus den folgenden vier Routinen:

- MPI_INIT: Aufruf immer als erste Funktion innerhalb des Programms
- MPI_FINALIZE: Aufruf immer als letzte Funktion innerhalb des Programms
- MPI_COMM_SIZE: Liefert die Anzahl der Prozesse innerhalb des Kommunikators
- MPI_COMM_RANK: Liefert den Rang des aktuellen Prozesses innerhalb des Kommunikators

Ein Kommunikator legt fest, in welchem Geltungsbereich die Operationen ausgeführt werden sollen. Prozesse eines Kommunikators können dabei nicht mit Prozessen eines anderen Kommunikators interagieren.

3 Kommunikation

Der grundlegende Mechanismus für die Kommunikation in MPI ist das Senden und Empfangen von Nachrichten. Die Nachrichten haben dabei je nach aufrufender Routine einen vorgegebenen Aufbau. Dadurch wird explizit festgelegt, welche Informationen von einem Prozess an den anderen gesendet werden.

3.1 Point-to-Point Kommunikation

Bei der Point-to-Point Kommunikation gibt es genau einen Sender und einen Empfänger. Diese werden durch ihren Rang im Kommunikator identifiziert und beim Nachrichtenaustausch angesprochen. MPI unterscheidet zwischen blockierender und nicht-blockierender Kommunikation. Blockierend bedeutet in diesem Kontext, dass der Puffer erst wieder verwendet bzw. gelesen werden darf sobald der Funktionsaufruf terminiert. Im Rahmen dieser Ausarbeitung wird bei der Point-to-Point Kommunikation immer von der blockierenden Variante ausgegangen. Die Kommunikation läuft dabei in drei Phasen ab:

1. Die Daten werden aus dem Sendepuffer kopiert und eine Nachricht wird erstellt
2. Die Nachricht wird vom Sender zum Empfänger übertragen
3. Die Daten von der eingegangenen Nachricht werden in den Empfangspuffer geschrieben

In der Programmiersprache Fortran sieht die Syntax für die Versende- und Empfangsroutine wie folgt aus:

- **MPI_SEND** (buf, count, datatype, dest, tag, comm, ierror)

- **MPI_RECV** (buf, count, datatype, source, tag, comm, status, ierror)

Die Funktionsparameter sind bei beiden Routinen sehr ähnlich. Zuerst wird mit *buf* ein Zeiger mitgegeben, der auf den Sende- bzw. Empfangspuffer weist. Der Integerwert *count* spezifiziert die Anzahl der Elemente, die von einem bestimmten MPI-Datentyp *datatype* versendet bzw. empfangen werden. Die Identifizierung von Sender und Empfänger innerhalb des Kommunikators geschieht, wie bereits erwähnt, durch den Rang des jeweiligen Prozesses. Der Prozess, der die Nachricht verschickt, muss den Rang des empfangenden Prozesses als *dest* mitgegeben. Äquivalent verhält es sich mit dem Verweis auf den Rang des Versenders durch *source* bei der Empfangsroutine. Das *tag* ist ein Integerwert und dient als eine Art Etikett. Es erlaubt die Unterscheidung von Nachrichtentypen und sollte bei beiden Prozessen, die miteinander kommunizieren, identisch sein. Alternativ kann dem Programm durch *MPI_ANY_TAG* mitgeteilt werden, dass das Etikett bei der Nachricht keine Verwendung hat. Der Kontext in welchem die Nachrichten verschickt und empfangen werden, wird durch den Kommunikator mitgegeben. Bei einfacher, schwach strukturierter Kommunikation kann als solcher immer *MPI_COMM_WORLD* verwendet werden. Durch diesen werden alle Prozesse innerhalb der Laufzeitumgebung angesprochen.

Zudem müssen bei der Kommunikation gewisse Datentyp-Konventionen eingehalten werden. Die Überprüfung findet nicht durch MPI statt, sondern muss vom Programmierer selbst gewährleistet werden. Dazu gehört einerseits, dass die Daten im Sendepuffer zu MPI_Datatype in der Send-Routine passen müssen. Äquivalent verhält es sich bei dem Empfangspuffer und dem MPI_Datatype der Receive-Routine. Zudem müssen die Datentypen der miteinander kommunizierenden Prozesse übereinstimmen.

3.2 Kollektive Kommunikation

Zusätzlich zu der Kommunikation zwischen lediglich zwei Prozessen, gibt es in MPI noch die kollektive Kommunikation. Bei dieser läuft die Kommunikation zwischen einer oder mehrerer Gruppen von Prozessen ab. Es können sogar alle Prozesse innerhalb eines Kommunikators beteiligt sein. Die folgenden drei Arten der kollektiven Kommunikation dienen lediglich dem Einblick in diese Form der Kommunikation:

- **Broadcast**: Dieselben Daten werden von einem Prozess einer Gruppe zu allen anderen Prozessen geschickt
- **Scatter**: Verschiedene Daten derselben Größe werden von einem Prozess einer Gruppe zu allen anderen Prozessen geschickt
- **Gather**: Verschiedene Daten derselben Größe werden von allen Prozessen einer Gruppe gesammelt und an einen bestimmten Prozess gesendet

Scatter und Gather werden häufig nacheinander benutzt. Dabei können verschiedene Daten zuerst über den Aufruf *MPI_SCATTER* an alle Prozesse innerhalb der Gruppe verschickt werden. Dort werden Berechnungen auf den Daten ausgeführt und die neuen Daten über den Aufruf *MPI_GATHER* zurück an den ursprünglichen Prozess gesendet. Dieser führt die Daten wieder zusammen und wertet die Ergebnisse aus.
Ebenso wie bei der Point-to-Point Kommunikation muss auch bei der kollektiven Kommunikation darauf geachtet werden, dass die Daten sowohl bei dem Versender(n), als auch bei dem Empfänger(n) in Anzahl und Datentyp übereinstimmen.

4 Datentypen

Im einfachsten Fall wird bei der Kommunikation ein zusammenhängender Bereich von Daten eines identischen elementaren Typs übertragen. Es ist jedoch zu restriktiv, lediglich diesen einen Fall zu betrachten. Dies liegt einerseits daran, dass auch Nachrichten versendet werden sollen, die Werte unterschiedlicher Datentypen beinhalten (z.B. zuerst ein Zähler vom Typ Integer, gefolgt von mehreren Zahlen vom Typ Real). Andererseits sollen auch Daten versendet werden können, die nicht zusammenhängend sind (z.B. eine bestimmte Spalte einer Matrix). Aus diesem Grund besteht in MPI die Möglichkeit, eigene Datentypen zu definieren und in Kommunikationsoperationen zu benutzen.

4.1 Elementare Datentypen

Die nachfolgende Tabelle zeigt einige der elementaren Datentypen, sowie ihre korrespondierenden Datentypen in der Programmiersprache Fortran.

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Table 1. Vordefinierte Datentypen mit den zugehörigen Fortran-Datentypen [7]

Der elementare Datentyp *MPI_BYTE* ist in dieser Form nicht in Fortran zu finden. Dieser Typ besteht aus einem Byte, unabhängig davon welchen Datentyp die Variable hat, die diesen Byte nutzt. Dadurch kann die zuvor erläuterte Datentyp-Konvention umgangen werden, bei der die Datentypen identisch sein müssen. [6]
Ein anderer Datentyp ohne korrespondierenden Typ in Fortran, ist MPI_PACKED. Durch diesen können verschiedene Speicherregionen mittels der Routine *MPI_PACK* zusammengeführt und als Datentyp *MPI_PACKED* verschickt werden. Innerhalb des Prozesses, der die Daten empfängt, können diese nun mittels der Routine *MPI_UNPACK* extrahiert und weiterverarbeitet werden. [6]

5 Abgeleitete Datentypen

Wie bereits in der Einleitung erwähnt, verursacht nicht nur die Berechnung innerhalb eines MPI-Programms Kosten, sondern auch die Kommunikation. Genauer gesagt: Die Kosten, die durch die Nachrichtenübertragung entstehen, sind unter der Annahme, dass dieselbe Menge an Daten über das Netzwerk übertragen wird, sogar vergleichsweise deutlich höher. Daher sollten für eine hohe Effizienz möglichst wenige große anstatt viele kleine Nachrichten verschickt werden. Dies gelingt durch die Konstruktion von Datentypen.

Abgeleitete Datentypen können erstellt werden, indem zuerst der gewünschte neue Datentyp mithilfe der dazugehörigen MPI-Routine konstruiert wird. Im Folgenden werden vier unterschiedliche Arten von MPI-Routinen betrachtet, die benutzt werden, um den jeweiligen Datentyp zu konstruieren:

1. MPI_TYPE_CONTIGUOUS
2. MPI_TYPE_VECTOR
3. MPI_TYPE_INDEXED
4. MPI_TYPE_CREATE_STRUCT

Sobald der neue Datentyp konstruiert wurde, muss dieser mittels *MPI_TYPE_COMMIT* übergeben werden. Erst danach ist es möglich, ihn innerhalb der Kommunikationsoperationen zu verwenden.

5.1 Datentyp Contiguous

```
MPI_TYPE_CONTIGUOUS (count, oldtype, newtype, ierror)
```

Die einfachste Art einen Datentyp zu konstruieren ist mithilfe von *MPI_TYPE_CONTIGUOUS.* Dieser erlaubt die Replikation eines Datentyps in zusammenhängende Speicherorte (daher auch der Name). Der Parameter *count* gibt an, wie viele Elemente des alten Datentyps *oldtype* der neue Datentyp *newtype* haben soll.

In der Praxis bietet dieser Datentyp keinerlei Effizienzvorteile. Wie bereits in Kapitel 3.1 besprochen, kann innerhalb der Versende-Routine ebenfalls ein *count*-Parameter mitgegeben werden. Falls also beispielsweise vier Integerwerte an einen anderen Prozess mit dem Rang 1 innerhalb des Kommunikators gesendet werden sollen, gibt es dafür zwei Möglichkeiten, die im Kern dasselbe tun:

1. Die vier Werte werden direkt aus dem Puffer gelesen und versendet:

```
1: MPI_SEND (data, 4, MPI_INT, 1, 99,
   MPI_COMM_WORLD
```

2. Es wird mithilfe von *MPI_TYPE_CONTIGUOUS* ein neuer Datentyp *mytype* konstruiert und anschließend versendet:

```
1: MPI_TYPE_CONTIGUOUS (4, MPI_INT, &mytype,
   ierror)
2: MPI_TYPE_COMMIT (&mytype)
3: MPI_SEND (data, 1, mytype, 1, 99,
   MPI_COMM_WORLD)
```

Es gibt jedoch auch bei diesem einfachen Beispiel einen Spezialfall, bei dem *MPI_TYPE_CONTIGUOUS* benutzt werden muss. Das liegt daran, dass der *count*-Parameter innerhalb der Kommunikationsoperationen ein Integerwert ist. Wenn sehr lange Nachrichten mit mehr als 2^31-1 Elementen (32bit Länge)

verschickt werden müssen, kann es daher zu Problemen kommen. Einige Systeme sind dazu nämlich nicht in der Lage, auch wenn die MPI Implementation intern 64bit Länge unterstützt. Die Lösung ist, einen neuen Datentyp der Länge m zu konstruieren und n Elemente des neuen Typs zu verschicken. Dadurch können nun insgesamt n*m Datenelemente verschickt werden, wobei n*m bis zu 2^62-2^32+1 groß sein kann.

Ein weiterer Grund für die Verwendung von Contiguous ist die Übersichtlichkeit. Angenommen es soll eine Zeile aus einer 4x4 Matrix versendet werden, wobei die Elemente zeilenweise in dem Speicher liegen. Dann könnte ein Datentyp abgeleitet werden, der genau eine bestimmte Zeile ausgibt.

5.2 Datentyp Vektor

```
MPI_TYPE_VECTOR (count, blocklength, stride, oldtype, newtype,
ierror)
```

Eine weitere Art ein Datentyp abzuleiten geschieht durch die Routine *MPI_TYPE_VECTOR*. Es gibt auch hierbei die Parameter *count*, *oldtype* und *newtype*, Zusätzlich jedoch noch *blocklength* und *stride*, die den Datentyp Vektor von dem Datentyp Contiguous unterscheiden. Grundsätzlich handelt es sich bei Vektor um eine bestimmte Anzahl an Blöcken, die in gleichgroßen Abständen in den Speicher geschrieben werden. Durch den Parameter *count* wird die Anzahl der Blöcke angegeben. Jeder Block besteht aus Kopien des alten Datentyps, deren Anzahl durch *blocklength* festgelegt wird. Der Abstand zwischen den Blöcken wird durch *stride* bestimmt und ist ebenfalls ein Produkt der Größe des alten Datentyps. Dieser Wert sollte größer gleich der Blocklänge sein, da der Abstand immer vom Anfang eines zum Anfang des nächsten Blockes bestimmt wird. Ist der Wert kleiner, so würde es demzufolge zum Überschreiben von Datenwerten kommen.

Wie bereits im Kapitel 5.1 kann man sich auch den praktischen Nutzen des Datentyps Vektor anhand einer 4x4 Matrix gefüllt mit ganzen Zahlen veranschaulichen. Es wird wieder angenommen, dass die Datenwerte zeilenweise im Speicher liegen. Im Gegensatz zu Contiguous, durch den eine bestimmte Zeile definiert werden kann, kann Vektor nun zur Definition einer Spalte dienen. Die Konstruktion des Datentyps sähe wie folgt aus:

```
1:  MPI_TYPE_VECTOR (4, 1, 4, MPI_INT, &mycolumntype,
    ierror)
2:  MPI_TYPE_COMMIT (&mycolumntype)
```

Der erste Wert gibt in diesem Fall die vier Zeilen an. Der zweite die Tatsache, dass jeder Block lediglich aus einem Element besteht. Und zwar dem Wert, der in der jeweiligen Zeile in der gewünschten Spalte steht. Der Abstand zwischen den Blöcken ist viermal die Größe des alten Datentyps, da innerhalb einer 4x4 Matrix operiert wird. Hätte die Matrix stattdessen sechs Spalten und vier Zeilen (6x4), so müsste der Abstand mit sechs angegeben bei gleichbleibendem *count*-Wert.

5.3 Datentyp Indexed

```
MPI_TYPE_INDEXED (count, blocklengths, displs, oldtype, newtype,
ierror)
```

Ein Datentyp-Konstruktor, der noch allgemeiner ist als Vektor, kann durch *MPI_TYPE_INDEXED* aufgerufen werden. Es ist bereits anhand der Funktionssyntax ersichtlich, dass sich dieser Datentyp lediglich marginal vom vorherigen unterscheidet. Der einzige Unterschied besteht in der Definition der Blocklängen und der Abstände zwischen den Blöcken. Sowohl bei *blocklengths* als auch bei *displs* handelt es sich in diesem Fall nicht um einen einzelnen Integerwert, sondern um ein Array aus Integern. Der Indexed-Konstruktor erlaubt es dem Programmierer somit, ein nicht zusammenhängendes Datenlayout zu spezifizieren, wobei die Abstände zwischen den Blöcken unterschiedlich sein können.

Dadurch ist es beispielsweise möglich, beliebige Werte aus einem Array zu holen und diese innerhalb nur einer Nachricht zu verschicken. Einem empfangenden Prozess ist es gleichzeitig möglich, die Daten einer Nachricht an beliebige Stellen eines Arrays zu verteilen.

5.4 Datentyp Struct

```
MPI_TYPE_CREATE_STRUCT (count, blocklengths, displs, oldtypes,
newtype, ierror)
```

Die allgemeinste Form eines Datentyp-Konstruktors ist *MPI_TYPE_CREATE_STRUCT*. Wie bereits beim Typ Indexed können auch beim Typ Struct die Längen der Blöcke, sowie die Abstände zwischen ihnen individuell festgelegt werden. Zusätzlich ist es bei dieser Art von Konstruktor möglich, jedem Block einen unterschiedlichen Datentyp zuzuweisen.

Struct ermöglicht dem Programmierer somit größtmögliche Freiheit bei der Konstruktion eines Datentyps. Beispielsweise kann ein Typ *persontype* mit den Parametern *age* (int), *height* (double) und *name* (char[20]) erstellt werden:

```
1:  count = 3
2:  lengths = [1, 1, 20]
3:  datatypes = [MPI_INT, MPI_DOUBLE, MPI_CHAR]
4:  MPI_TYPE_CREATE_STRUCT      (count,      lengths,
    displacements, datatypes, &persontype, ierror)
5:  MPI_TYPE_COMMIT (&persontype)
```

In diesem Fall fehlt jedoch noch das Array aus *displacements*. Dessen Berechnung erfordert bei der Konstruktion des Datentyps Struct einiges an Vorsicht. Der Abstand zwischen den Blöcken kann nicht wie bei Indexed im Verhältnis des alten Datentyps berechnet werden, da es mehrere Datentypen geben kann. Stattdessen muss der Abstand in Bytes angegeben werden. Um diesen zu ermitteln, müssen zuerst durch *MPI_GET_ADDRESS* die Adressen der jeweiligen Datenwerte im Speicher geholt werden. Anschließend kann mittels *MPI_AINT_DIFF* ein Displacement durch die Differenz der vorherigen Adresse und der jetzigen Adresse im Speicher berechnet werden. [5]

6 Zusammenfassung

In dieser Ausarbeitung wurde das Message Passing Interface (MPI) betrachtet, welches ein Standard für die parallele Berechnung auf verteilten Systemen ist. Mithilfe der in den Grundlagen besprochenen MPI-Routinen *MPI_INIT, MPI_FINALIZE, MPI_COMM_RANK, MPI_COMM_SIZE*, sowie der Sende- und Empfangsroutinen *MPI_SEND* und *MPI_RECV* lassen sich bereits eine Vielzahl unterschiedlicher MPI-Programme erstellen. Zudem sind diese sechs Routinen in (fast) jedem Programm zu finden und dienen einem grundlegenden Verständnis über die Kommunikation innerhalb von MPI. Der Fokus der Ausarbeitung wurde jedoch auf die verschiedenen Datentypen gelegt. Vordefinierte Datentypen, die bereits aus Fortran oder C bekannt sind, können mithilfe unterschiedlicher MPI-Routinen in andere Datentypen abgeleitet werden. Dies dient dazu, die Anzahl der zu verschickenden Nachrichten zu verringern und damit eine effizientere Kommunikation sowie eine besseren Überblick zu gewährleisten.

Literaturverzeichnis

[1] Message Passing Interface Standard – Version 3.1, https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[2] MPI-Datentypen, https://www.math.tu-cottbus.de/~kd/parallel/vorl/vorl/node25.html#:~:text=Die%20MPI%2DTypen%20sind%20keine,MPI%2DDatentypen%20definieren%20ein%20Speicherlayout.

[3] Einführung in MPI, https://www.fz-juelich.de/ias/jsc/EN/AboutUs/Staff/Hagemeier_A/docs-parallel-programming/MPI-Slides.pdf?__blob=publicationFile

[4] MPICH Documentation, https://www.mpich.org/static/docs/v3.2/

[5] MPI Datentyp Struct, https://www.rookiehpc.com/mpi/docs/mpi_type_create_struct.php

[6] Texas Advanced Computation Center (TACC): MPI Datatypes, https://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-data.html

[7] Marc Snir, Steve Otto, Steven Huss Lederman, David Walker, Jack Dongarra: MPI The Complete Reference: The MIT Press

[8] Wikipedia: Message Passing Interface, https://de.wikipedia.org/wiki/Message_Passing_Interface

[9] The Shared Memory and Message Passing Models of Interprocess Communication, http://www.umsl.edu/~siegelj/CS4740_5740/Overview/MPSM.html