Message Passing Interface (MPI) in Python

Python im Hochleistungsrechnen

Alexander Michael Gerlach

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

2019-06-24

Gliederung (Agenda)

1 Einleitung

2 Grundlagen

3 Verwendung von MPI

4 Benchmarks

5 Zusammenfassung

6 Literatur

Was ist MPI?

- **MPI** = **M**essage **P**assing **I**nterface
- Weit verbreiteter und verwendeter Standard
- MPI ist ein Interface, **keine** Implementation
- MPI Standard spezifiziert, wie Prozesse untereinander Nachrichten austauschen können
- Basiert auf dem Distributed Memory Model

Quelle: [Bla19b]

Distributed Memory Model

- Jede CPU arbeitet unabhängig und hat seinen eigenen lokalen Speicher
- Vorteile: Skalierbar, schneller lokaler Speicherzugriff
- Nachteile: Programmierer verantwortlich für Datenaustausch, uneinheitliche Zugriffszeiten

Quelle: [Bla19a]

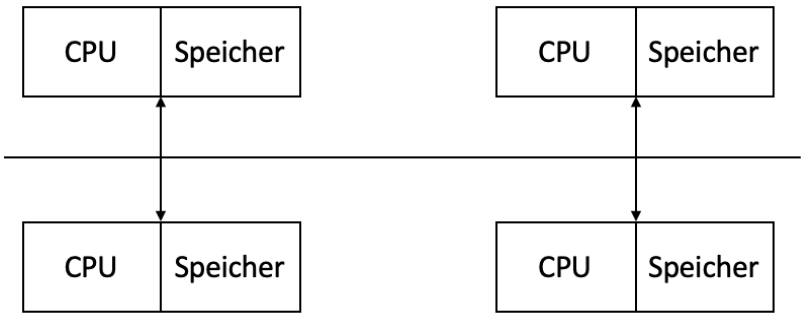

Abb.: Distributed Memory Model

Warum MPI?

Gründe für die Verwendung von MPI:

1. Steigerung der Performance
2. Standartisierung
3. Funktionalität
4. Viele Implementierungen verfügbar
5. Portabilität

Quelle: [Bla19b]

Wichtige Begriffe

- Prozess - Unabhängige Ausführungsinstanz einer Anwendung mit eigenem Adressraum
- Thread - Ausführungspfad innerhalb eines Prozesses mit jeweils gleichem Adressraum
- (Parallel) Overhead - Zeit, die für die Koordination paralleler Prozesse benötigt wird, z.B. durch Kommunikation von Daten

Quelle: [Bla19a]

Wie arbeitet MPI?

- Communicator definiert eine Gruppe von Prozessen, welche miteinander kommunizieren können
- Jeder Prozess hat einen individuellen Rang
- Der Rang wird für die Übermittlung von Nachrichten zwischen Prozessen verwendet

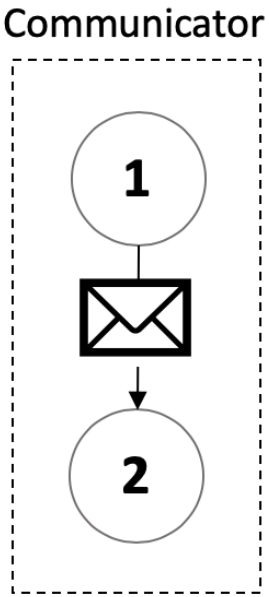

Abb.: Communicator mit 2 Prozessen

Quelle: [Ken19]

mpi4py

- mpi4py Modul wird verbreitet eingesetzt
- Basierend auf MPI 1/2/3 Standards
- Ähnelt der Semantic von MPI-2 in C++

Quelle: [Dal19]

Nachrichtenübertragung

- Übertragung von Nachrichten Erfolgt als Stream von Bytes
- mpi4py bietet Funktionen für die Übertragung von Python objects (lower case) und buffer like objects (upper case), wie z.B. numpy Arrays
- Python objekte müssen jedoch vor der Übertragung serialisiert werden $\Rightarrow$ Signifikanter overhead, insbesondere bei größeren Arrays
- Overhead bei Kommunikation von buffer like objects verschwindent gering

Quelle: [Dal19]

Installation von mpi4py

Voraussetzungen:

- MPI Implementierung (z.B. MPICH)

```
1 # Mac OS
2 sudo brew install mpich
3 # Ubuntu
4 sudo apt install mpich
```

- mpi4py

```
1 sudo pip install mpi4py
```

- Cython

```
1 sudo pip install Cython
```

Quelle: [Dal19]

Initialisierung und Ausführung

- Erzeugen eines neuen Communicators

```
1 from mpi4py import MPI
2
3 comm = MPI.COMM_WORLD   # Communicator
4 comm.Get_size()   # Anzahl an Prozessen
5 comm.Get_rank()   # Rang des Prozesses
```

- Python Programm ausführen

```
1 # Execution with 4 processes
2 mpiexec -n 4 python file_name.py
```

Quelle: [fwDmG19]

Point-to-Point

- Zweistufiger Prozess, bei dem von einem Prozess eine Nachricht zu einem anderen Prozess gesendet wird
- Syntax:

```
1 # Python objects
2 comm.send(buffer, destination=0, tag=0)
3 comm.recv(object, source=0, tag=0, status)
4
5 # Buffer like objects
6 comm.Send(buffer, destination=0, tag=0)
7 comm.Recv([data, count, MPI.type], source=0,
      ↪ tag=0, status)
```

Quelle: [fwDmG19]

Point-to-Point

- 'status' Parameter ermöglicht Abrufen von Informationen über die Nachricht
- Instanz von Status Klasse
- Hilfreich um z.B. vor Empfang einer Nachricht die Größe des buffers zu bestimmen

```
1 info = MPI.Status()
2
3 info.Get_count() # Message size in bytes
4
5 info.Get_elements(dtype) # Number of dtype
      ↪ elements
6
7 info.Get_source() # Message source
```

Quelle: [fwDmG19]

Blocking / Non-blocking

Blocking Communications:

- Synchroner Kommunikationsmodus
- Folgeinstruktionen werden erst dann ausgeführt, wenn der Speicherort der zu kommunizierenden Daten sicher verwendet werden kann
- Kann zu Deadlocks führen

Non-blocking Communications:

- Asynchroner Kommunikationsmodus
- Folgeinstruktionen können direkt ausgeführt werden

Quelle: [fwDmG19]

Collective Send Operations

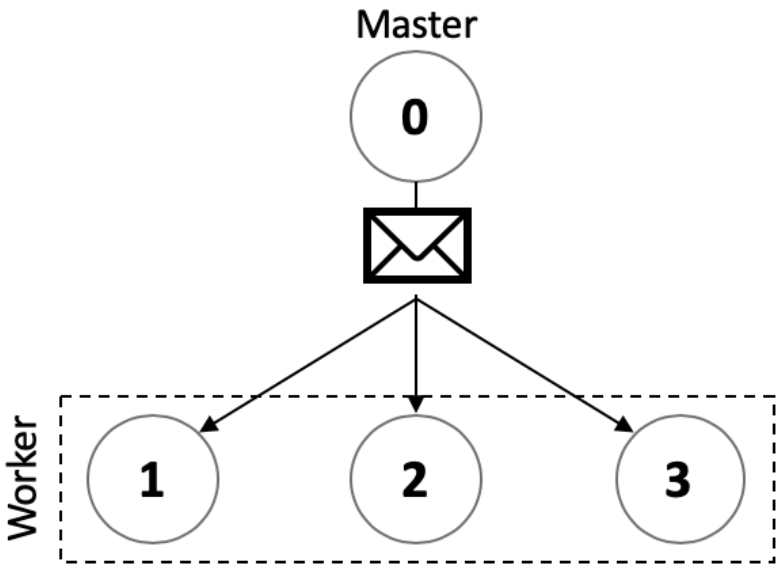

Abb.: Broadcast

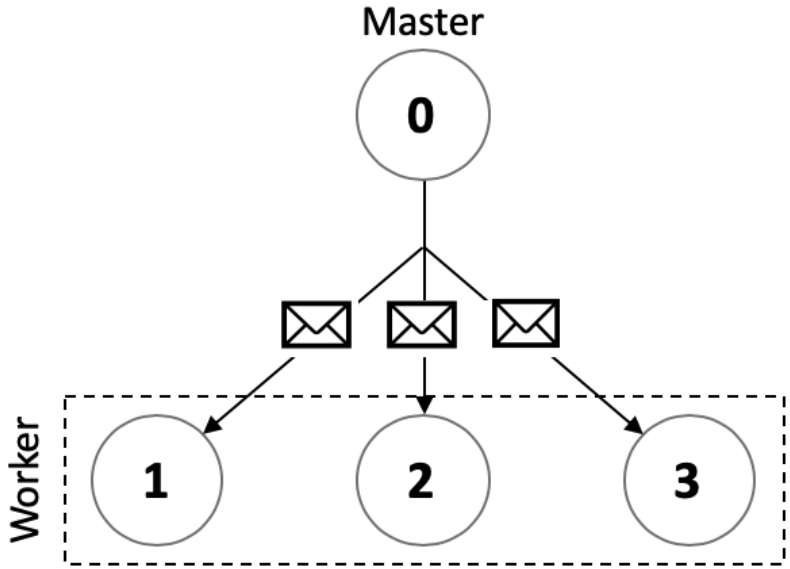

Abb.: Scatter

Collective Send Operations - Syntax

Collective Send Operations:

- Buffer als list [data, dsize, dtype]
- dsize & dtype entfallen wenn sie durch das buffer object impliziert sind
- Scatter sendbuf muss Vielfaches der Anzahl an Prozessen sein

```
1 # Broadcast
2 comm.Bcast(buf, root=0)
3
4 # Scatter
5 comm.Scatter(sendbuf, recvbuf, root=0)
```

Quelle: [fwDmG19]

Collective Receive Operations

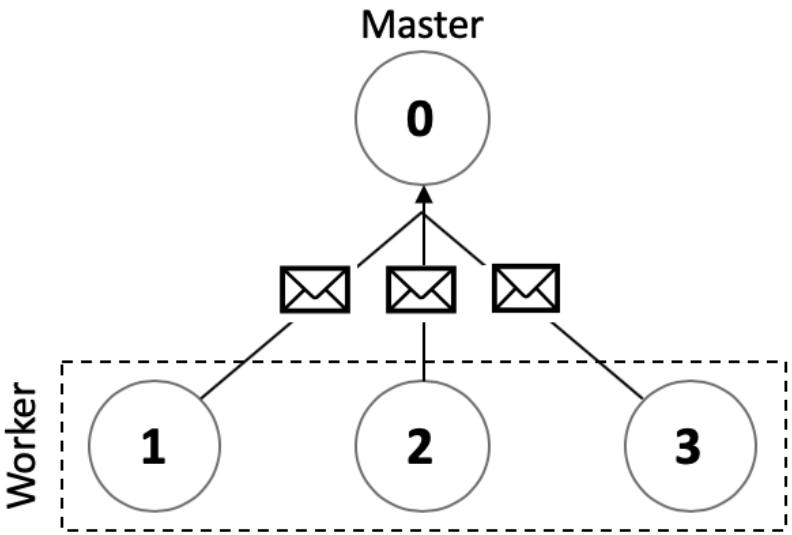

Abb.: Gather

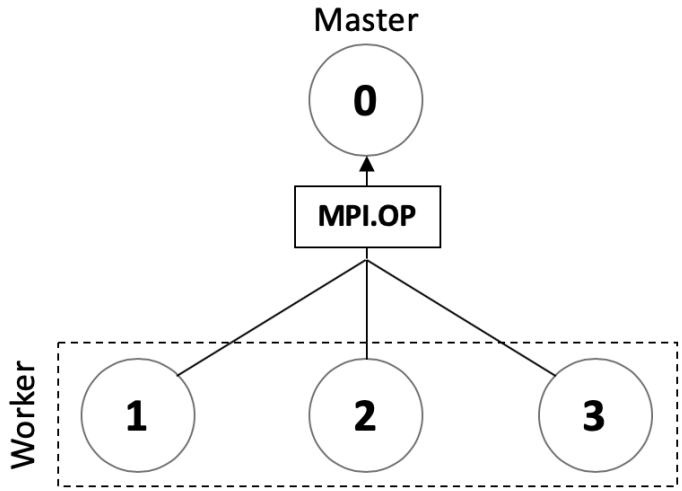

Abb.: Reduce

Collective Receive Operations - Syntax

Collective Receive Operations:

- sendbuf Größe identisch bei allen task
- recvbuf muss i-mal die Größe von sendbuf haben, wobei i die Anzahl an Prozessen ist

```
1 # Gather
2 comm.Gather(sendbuf, recvbuf, root=0)
3
4 # Reduce
5 comm.Reduce(sendbuf, recvbuf, operation, root=0)
```

Quelle: [fwDmG19]

Weitere

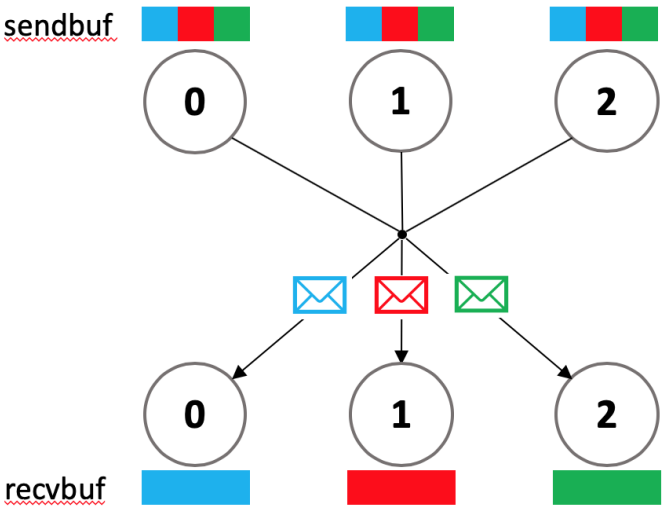

Abb.: Alltoall

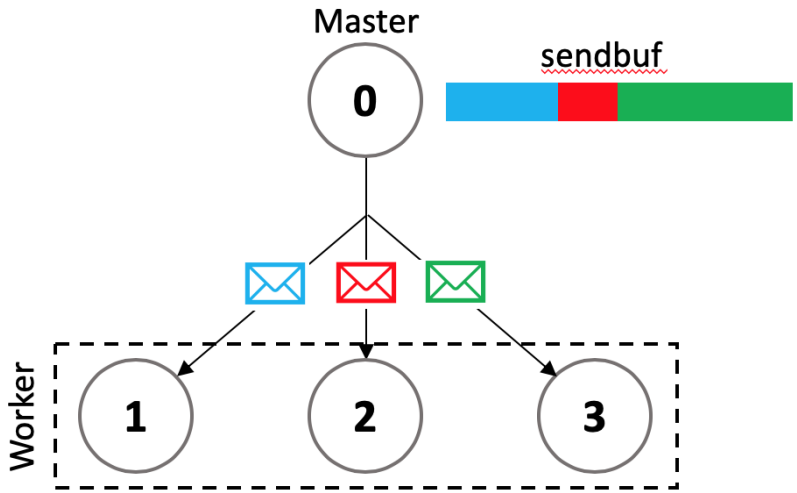

Abb.: Scatterv

One-Sided Communication

- Auch bekannt als **R**emote **M**emory **A**ccess (RMA)
- Two-Sided Communications: Empfänger muss vor versenden einer Nachricht bereit sein ⇒ Overhead
- One-Sided Communications: Bereitstellung von Public Memory Region (Window)
- Prozess A kann auf Speicher von Prozess B zugreifen durch put & get, ohne dass dieser intervenieren muss ⇒ Overhead wird reduziert

Abb.: One-Sided Communication

Quelle: [Ngu14]

Dynamic Process Management

- MPI-1: Parallele Anwendung statisch, d.h. es können keine Prozesse gelöscht / hinzugefügt werden
- Seit MPI-2 können neue Gruppen von Prozesse erstellt werden
- Hilfreich für serielle Anwendungen mit parallelen Modulen

```
1 MPI.Intracomm.Spawn()
```

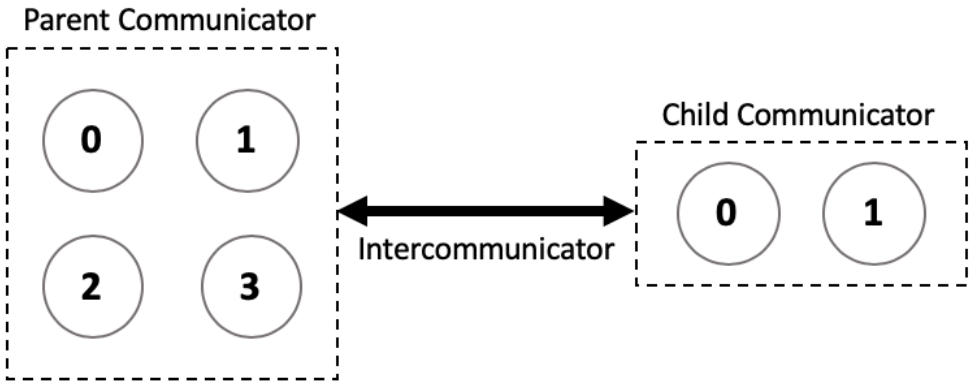

Abb.: Intercommunicator

Quelle: [Dal19]

Paralleler Merge Sort Algorithmus

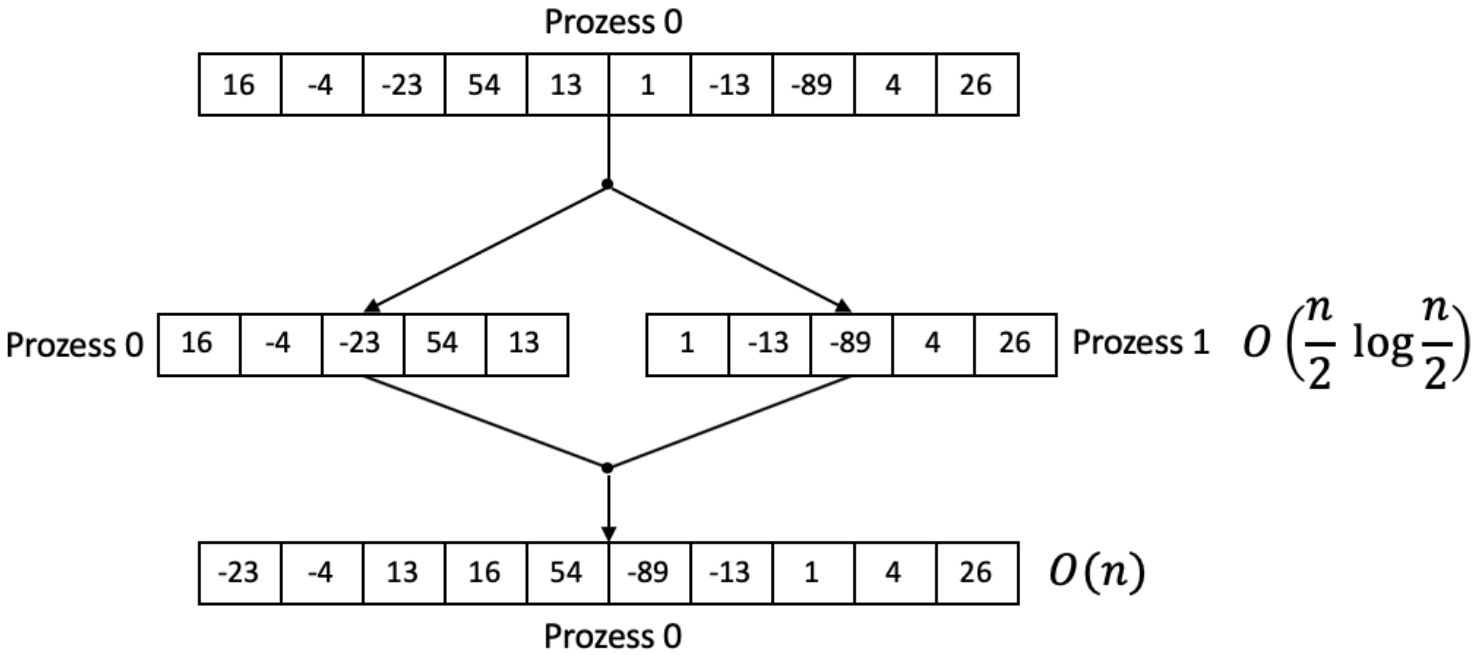

Abb.: Implementierungsansatz

Quelle: [ea19, MPI19]

Parallel vs. Seriell

Merge Sort Algorithmus:

a_size	comm_size	Parallel (t in sec)	Seriell (t in sec)
1.000	2	0.004096 s	0.007349 s
100.000	2	0.556190 s	1.068008 s
1.000.000	2	6.581953 s	13.821216 s

Tabelle: Merge Sort Geschwindigkeitsvergleich

C vs. Python

Berechnung des Durchschnitts einer Menge von Integers:

a_size	comm_size	C (t in sec)	Python (t in sec)
1.000	2	0.000586 s	0.003144 s
100.000	2	0.001593 s	0.005094 s
100.000.000	2	0.601536 s	0.928805 s

Tabelle: Anteilige MPI Berechnungszeit

a_size	comm_size	C (t in sec)	Python (t in sec)
1.000	2	0.000612 s	0.003666 s
100.000	2	0.001881 s	0.043040 s
100.000.000	2	0.891626 s	40.399889 s

Tabelle: Gesamte Berechnungszeit

Zusammenfassung

- MPI Standard
  - Verbreiteter Standard
  - Spezifiziert wie Prozesse untereinander kommunizieren können
- Kommunikationsarten
  - Two-sided Communication
  - One-sided Communication
- Verwendung von MPI in Python

Literatur

[Bla19a] Barney Blaise. Introduction to parallel computing, 2019.

[Bla19b] Barney Blaise. Message passing interface, 2019.

[Dal19] Lisandro Dalcin. Mpi for python, 2019.

[ea19] Moore et al. Merge sort, 2019.

[fwDmG19] Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen. Mpi4py, 2019.

[Ken19] Les Kendall. A comprehensive mpi tutorial resource, 2019.

[MPI19] MPICH. Mpich, 2019.

[Ngu14] Loc Q Nguyen. Mpi one-sided communication, 2014.