Hochleistungsrechnen 2026/2027

Aussagen aus dem MPI Standard 5.0

Jannek Squar

2026-11-xx

Scientific Computing
Universität Hamburg



Outline

- Minimales Setup
  - Programmstart
  - Programm-Ende
- Kommunikation
  - Punkt-zu-Punkt
  - Kollektive Kommunikation
  - Extras

Programmstart

Ziel: Standardkonformes MPI-Programm. Wir optimieren nicht für eine bestimmte Implementation

- MPI semi-manuell initialisiert[1]
  - Kein Spawn, nur Bekanntmachung
  - `MPI_Init(int *argc, char ***argv)`
    - `argc` und `argv` können auch `NULL` gesetzt werden
    - Standard bleibt vage, Implementation kann MPI-Parameter filtern
    - Davor nur ausgewählte MPI-Aufrufe erlaubt
  - `MPI_Init_Thread(int *argc, char ***argv, int required, int *provided)`
    - `MPI_THREAD_SINGLE`: Ein Thread
    - `MPI_THREAD_FUNNELED`: MT, MPI nur aus Main-Thread
    - `MPI_THREAD_SERIALIZED`: MT, keine parallelen MPI-Aufrufe
    - `MPI_THREAD_MULTIPLE`: Keine Vorschriften

[1] MPI 5.0, Ch. 11.2.1 Starting MPI Processes

Programmstart

Ziel: Standardkonformes MPI-Programm. Wir optimieren nicht für eine bestimmte Implementation

- MPI semi-manuell initialisiert[1]
  - Kein Spawn, nur Bekanntmachung
  - `MPI_Init(int *argc, char ***argv)`
    - `argc` und `argc` können auch `NULL` gesetzt werden
    - Standard bleibt vage, Implementation kann MPI-Parameter filtern
    - Davor nur ausgewählte MPI-Aufrufe erlaubt
  - `MPI_Init_Thread(int *argc, char ***argv, int required, int *provided)`
    - `MPI_THREAD_SINGLE`: Ein Thread
    - `MPI_THREAD_FUNNELED`: MT, MPI nur aus Main-Thread
    - `MPI_THREAD_SERIALIZED`: MT, keine parallelen MPI-Aufrufe
    - `MPI_THREAD_MULTIPLE`: Keine Vorschriften

[1] `MPI 5.0, Ch. 11.2.1 Starting MPI Processes`

Ziel: Standardkonformes MPI-Programm. Wir optimieren nicht für eine bestimmte Implementation

- MPI semi-manuell initialisiert[1]
  - Kein Spawn, nur Bekanntmachung
  - `MPI_Init(int *argc, char ***argv)`
    - `argc` und `argc` können auch `NULL` gesetzt werden
    - Standard bleibt vage, Implementation kann MPI-Parameter filtern
    - Davor nur ausgewählte MPI-Aufrufe erlaubt
  - `MPI_Init_Thread(int *argc, char ***argv, int required, int *provided)`
    - `MPI_THREAD_SINGLE`: Ein Thread
    - `MPI_THREAD_FUNNELED`: MT, MPI nur aus Main-Thread
    - `MPI_THREAD_SERIALIZED`: MT, keine parallelen MPI-Aufrufe
    - `MPI_THREAD_MULTIPLE`: Keine Vorschriften

[1] `MPI 5.0, Ch. 11.2.1 Starting MPI Processes`

Programm-Ende

- Jeder Prozess muss seinem Ende i.d.R. `MPI_Finalize`[2] aufrufen
- MPI-Kommunikation muss vorher lokal beendet worden sein
  - Auch nicht-blockierende Operationen!
- Vor letztem Finalize: Jede Kommunikation muss gematcht worden sein
- `MPI_Finalize` muss für mindestens Rang 0 zurückkehren
  - Mehr ist aber nicht garantiert!
  - Rang 1 usw. könnten weiterhin aktiv sein, müssen es aber nicht

    → Keine Annahmen treffen!
- Best Practise:
  - `MPI_Init` so früh wie möglich
  - Außer **`return`** kein Code nach `MPI_Finalize`

[2] MPI 5.0, Ch. 11.2.2 Finalizing MPI

Programm-Ende

- Jeder Prozess muss seinem Ende i.d.R. `MPI_Finalize`[2] aufrufen
- MPI-Kommunikation muss vorher lokal beendet worden sein
  - Auch nicht-blockierende Operationen!
- Vor letztem Finalize: Jede Kommunikation muss gematcht worden sein
- `MPI_Finalize` muss für mindestens Rang 0 zurückkehren
  - Mehr ist aber nicht garantiert!
  - Rang 1 usw. könnten weiterhin aktiv sein, müssen es aber nicht

    → Keine Annahmen treffen!
- Best Practise:
  - `MPI_Init` so früh wie möglich
  - Außer **`return`** kein Code nach `MPI_Finalize`

[2] MPI 5.0, Ch. 11.2.2 Finalizing MPI

Programm-Ende

- Jeder Prozess muss seinem Ende i.d.R. `MPI_Finalize`[2] aufrufen
- MPI-Kommunikation muss vorher lokal beendet worden sein
  - Auch nicht-blockierende Operationen!
- Vor letztem Finalize: Jede Kommunikation muss gematcht worden sein
- `MPI_Finalize` muss für mindestens Rang 0 zurückkehren
  - Mehr ist aber nicht garantiert!
  - Rang 1 usw. könnten weiterhin aktiv sein, müssen es aber nicht

    → Keine Annahmen treffen!
- Best Practise:
  - `MPI_Init` so früh wie möglich
  - Außer `return` kein Code nach `MPI_Finalize`

[2] MPI 5.0, Ch. 11.2.2 Finalizing MPI

Programm-Ende

- Jeder Prozess muss seinem Ende i.d.R. `MPI_Finalize`[2] aufrufen
- MPI-Kommunikation muss vorher lokal beendet worden sein
  - Auch nicht-blockierende Operationen!
- Vor letztem Finalize: Jede Kommunikation muss gematcht worden sein
- `MPI_Finalize` muss für mindestens Rang 0 zurückkehren
  - Mehr ist aber nicht garantiert!
  - Rang 1 usw. könnten weiterhin aktiv sein, müssen es aber nicht

    → Keine Annahmen treffen!
- Best Practise:
  - `MPI_Init` so früh wie möglich
  - Außer **`return`** kein Code nach `MPI_Finalize`

[2] `MPI 5.0, Ch. 11.2.2 Finalizing MPI`

Finalize Beispiel I

Falsch:

```
1 // Prozess 0
2 MPI_Init();
3 MPI_Send(...,dest=1,...);
4 MPI_Finalize();
```

```
1 // Prozess 1
2 MPI_Init();
3
4 MPI_Finalize();
```

Finalize Beispiel I

Falsch:

```
1 // Prozess 0
2 MPI_Init();
3 MPI_Send(...,dest=1,...);
4 MPI_Finalize();
```

```
1 // Prozess 1
2 MPI_Init();
3
4 MPI_Finalize();
```

Richtig:

```
1 // Prozess 0
2 MPI_Init();
3 MPI_Send(...,dest=1,...);
4 MPI_Finalize();
```

```
1 // Prozess 1
2 MPI_Init();
3 MPI_Recv(...,source=0,...);
4 MPI_Finalize();
```

Finalize Beispiel II

Falsch:

```
1 // Prozess 0
2 MPI_Init();
3 MPI_Isend(...,dest=1,..., request);
4 MPI_Finalize();
```

```
1 // Prozess 1
2 MPI_Init();
3 MPI_Recv(...,source=0,...);
4 MPI_Finalize();
```

Finalize Beispiel II

Falsch:

```
1 // Prozess 0
2 MPI_Init();
3 MPI_Isend(...,dest=1,..., request);
4 MPI_Finalize();
```

```
1 // Prozess 1
2 MPI_Init();
3 MPI_Recv(...,source=0,...);
4 MPI_Finalize();
```

Richtig:

```
1 // Prozess 0
2 MPI_Init();
3 MPI_Isend(...,dest=1,...,request);
4 ...
5 MPI_Wait(request,...);
6 MPI_Finalize();
```

```
1 // Prozess 1
2 MPI_Init();
3 MPI_Recv(...,source=0,...)
4
5
6 MPI_Finalize();
```

Semantik[3]

Reihenfolge : Nachrichten überholen einander nicht

- Vorausgesetzt gleicher Prozess und Thread (gilt nicht für MT)
- Vorausgesetzt gleiche Ziel-Config (Tag, Dest, Comm)

Fortschritt : garantiert

Fairness : nicht garantiert

- Passende Nachrichten anderer Prozesse/Threads könnten immer vorgezogen werden

Ressourcen : Probleme möglich durch Limitierung

- insb. bei Buffern (z.B. `MPI_Bsend`)

[3] `MPI 5.0, Ch. 3.5 Semantics of Point-to-Point Communication`

Kommunikations-Modi[4]

Standard Buffered oder Synchronous

- Simpel

Buffered Lokal, braucht ausreichend Buffer

- Abschluss ohne passendes `MPI_Recv`
- Rate ich von ab

Synchronous Abschluss frühestens nach Start von Empfang

- „receiver has reached a certain point“
- Sicher aber langsam

Ready Passender Empfang muss(!) bereits gestartet sein

- Rate ich dringend von ab

[4] `MPI 5.0, Ch. 3.4 Communication Modes`

Blocking vs Non-Blocking

Blocking

- Lokaler Buffer: kann sofort verwendet werden
- Einfach
- Steht allein für sich
- Sicherer
- Ineffizienter durch Sequentialisierung

Non-Blocking

- Lokaler Buffer: Unbekannter Zustand
  - Ohne expliziten Abschluss, Finger weg vom Buffer!
- Komplex
- Muss(!) abgeschlossen werden
- Fehleranfälliger
- Effizienter durch Überlagerung

Auch nicht-blockierende kollektive Operationen verfügbar

Blocking vs Non-Blocking

Blocking

- Lokaler Buffer: kann sofort verwendet werden
- Einfach
- Steht allein für sich
- Sicherer
- Ineffizienter durch Sequentialisierung

Non-Blocking

- Lokaler Buffer: Unbekannter Zustand
  - Ohne expliziten Abschluss, Finger weg vom Buffer!
- Komplex
- Muss(!) abgeschlossen werden
- Fehleranfälliger
- Effizienter durch Überlagerung

Auch nicht-blockierende kollektive Operationen verfügbar

Übersicht

<table>
<tr><th></th><th>Send</th><th>Receive</th><th>Testing</th></tr>
<tr><td rowspan="2">Blocking</td><td>MPI_Send
MPI_Ssend
MPI_Rsend
MPI_Bsend</td><td>MPI_Recv</td><td rowspan="2">MPI_Probe
MPI_Wait</td></tr>
<tr><td colspan="2">MPI_Sendrecv</td></tr>
<tr><td rowspan="2">Non-blocking</td><td>MPI_Isend
MPI_Issend
MPI_Irsend
MPI_Ibsend</td><td>MPI_Irecv</td><td rowspan="2">MPI_Iprobe
MPI_Test</td></tr>
<tr><td colspan="2">MPI_Isendrecv</td></tr>
</table>

Tabelle 1: Basierend auf Übersicht ©Michael Kuhn

Senden und Empfangen

- Vier Sende-Modi, ein Empfangs-Modus $\rightarrow$ Passt beliebig zusammen
- Achtung:
  - Send kann zurückkehren, bevor Receive begonnen hat
  - Receive kann zurückkehren, nachdem Sende-Operation begonnen hat
  - Receive kann zurückkehren, bevor Sende-Operation zurückgekehrt ist

  $\Rightarrow$ Send-Recv ungeeignet für Synchronisation!

Fokus: Standard Senden

`MPI_Send`

- Verhalten von Nachrichtengröße abhängig:
  - Für kleine Nachricht wie `MPI_Bsend`
  - Für große Nachricht wie `MPI_Ssend`
  - System entscheidet $\rightarrow$ potentiell bessere Performance
- Gefahr:
  - Verhalten nicht vorhersagbar
  - Potentieller Deadlock
  - Klassiker: Kleiner Test funktioniert, großer Produktionslauf verklemmt

Beispiel: Standard Senden I

Fehlerhaftes Programm:[5]

```
1 // Prozess 0
2 MPI_Send(...,dest=1,...);
3 MPI_Recv(...,source=1,...);
```

```
1 // Prozess 1
2 MPI_Send(...,dest=0,...);
3 MPI_Recv(...,source=0,...);
```

Korrekte Alternative: Alternierend

```
1 // Prozess 0
2 MPI_Send(...,dest=1,...);
3 MPI_Recv(...,source=1,...);
```

```
1 // Prozess 1
2 MPI_Recv(...,source=0,...);
3 MPI_Send(...,dest=0,...);
```

[5] Auch wenn es manchmal funktioniert, das ist ein Fehler!

Beispiel: Standard Senden I

Fehlerhaftes Programm:[5]

```
1 // Prozess 0
2 MPI_Send(...,dest=1,...);
3 MPI_Recv(...,source=1,...);
```

```
1 // Prozess 1
2 MPI_Send(...,dest=0,...);
3 MPI_Recv(...,source=0,...);
```

Korrekte Alternative: Alternierend

```
1 // Prozess 0
2 MPI_Send(...,dest=1,...);
3 MPI_Recv(...,source=1,...);
```

```
1 // Prozess 1
2 MPI_Recv(...,source=0,...);
3 MPI_Send(...,dest=0,...);
```

[5] Auch wenn es manchmal funktioniert, das ist ein Fehler!

Beispiel: Standard Senden II

Fehlerhaftes Programm[6]

```
1 // Prozess 0
2 MPI_Isend(...,dest=1,...,request);
3 MPI_Wait(request);
4 MPI_Recv(...,source=1,...);
```

```
1 // Prozess 1
2 MPI_Isend(...,dest=0,...,request);
3 MPI_Wait(request);
4 MPI_Recv(...,source=0,...);
```

Korrekte Alternative:

```
1 // Prozess 0
2 MPI_Isend(...,dest=1,...,request);
3 MPI_Recv(...,source=1,...);
4 MPI_Wait(request);
```

```
1 // Prozess 1
2 MPI_Isend(...,dest=0,...,request);
3 MPI_Recv(...,source=0,...);
4 MPI_Wait(request);
```

[6] Auch wenn es manchmal funktioniert, das ist ein Fehler!

Beispiel: Standard Senden II

Fehlerhaftes Programm[6]

```
1 // Prozess 0
2 MPI_Isend(...,dest=1,...,request);
3 MPI_Wait(request);
4 MPI_Recv(...,source=1,...);
```

```
1 // Prozess 1
2 MPI_Isend(...,dest=0,...,request);
3 MPI_Wait(request);
4 MPI_Recv(...,source=0,...);
```

Korrekte Alternative:

```
1 // Prozess 0
2 MPI_Isend(...,dest=1,...,request);
3 MPI_Recv(...,source=1,...);
4 MPI_Wait(request);
```

```
1 // Prozess 1
2 MPI_Isend(...,dest=0,...,request);
3 MPI_Recv(...,source=0,...);
4 MPI_Wait(request);
```

[6] Auch wenn es manchmal funktioniert, das ist ein Fehler!

Beispiel: Standard Senden III

Korrekte Alternative: `MPI_Sendrecv`

```
1 // Prozess 0
2 MPI_Sendrecv(..., dest=1, ..., source=1, ...);
```

```
1 // Prozess 1
2 MPI_Sendrecv(..., dest=0, ..., source=0, ...);
```

Übersicht

- Arten kollektiver Operationen:[7]
  - Barriere
  - Broadcast
  - Gather
  - Scatter
  - Reduce
- Alle Prozesse des betroffenen Kommunikators müssen(!) teilnehmen
- Auch dann, wenn ein Prozess keinen Beitrag leistet
- Einzelne Prozesse können vorzeitig zurückkehren[8]

$\Rightarrow$ Außer der Barriere ist kein kollektive Aufruf geeignet für Synchronisation

[7] `MPI 5.0, Ch. 6 Collective Communication`

[8] Außer bei `MPI_Barrier`

Synchronisieren kollektive Operationen?

- „It is dangerous to rely on synchronization side-effects of the collective operations for program correctness. For example, even though a particular implementation may provide a broadcast routine with a side-effect of synchronization, the standard does not require this, and a program that relies on this will not be portable.“[9]
- „On the other hand, a correct, portable program must allow for the fact that a collective call may be synchronizing. Though one cannot rely on any synchronization side-effect, one must program so as to allow it.“[10]

[9] MPI 5.0, Ch. 6 Collective Communication
[10] MPI 5.0, Ch. 6 Collective Communication

Synchronisieren kollektive Operationen?

- „It is dangerous to rely on synchronization side-effects of the collective operations for program correctness. For example, even though a particular implementation may provide a broadcast routine with a side-effect of synchronization, the standard does not require this, and a program that relies on this will not be portable.“[9]
- „On the other hand, a correct, portable program must allow for the fact that a collective call may be synchronizing. Though one cannot rely on any synchronization side-effect, one must program so as to allow it.“[10]

[9] MPI 5.0, Ch. 6 Collective Communication
[10] MPI 5.0, Ch. 6 Collective Communication

Vorzeitiger Abbruch

- `MPI_Abort(MPI_Comm comm, int errorcode)`[11]
- `MPI_Cancel(MPI_Request *request)`[12]
  - Abbruch nicht-blockierender Operation
  - nicht-blockierend, lokal
  - Abschluss mit `MPI_Wait` etc. trotzdem notwendig!
  - Achtung: deprecated für Sende-Operationen

[11] MPI 5.0, Ch. 11.4.2 Aborting MPI Processes
[12] MPI 5.0, Ch. 3.8.4 Cancel

- `MPI_ANY_SOURCE`[13]
- `MPI_ANY_TAG`[14]
- `MPI_STATUS_IGNORE` bzw. `MPI_STATUSES_IGNORE`[15]
- `MPI_PROC_NULL`[16]
- `MPI_COMM_WORLD`
- `MPI_COMM_SELF`
- `MPI_IN_PLACE`

[13] MPI 5.0, Ch. 3.2.4 Blocking Receive
[14] MPI 5.0, Ch. 3.2.4 Blocking Receive
[15] MPI 5.0, Ch. 3.2.6 Passing MPI_STATUS_IGNORE for Status
[16] MPI 5.0, Ch. 3.10 Null MPI Processes