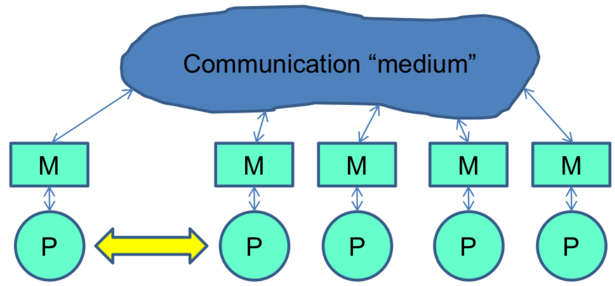
Message Paging model

→ Merkmal: habe Prozesse als aktive ge enten

- Gekapselt
- Arbeit lokal
- eigener Speicher / Adressraum → Eig. Prog.
- ASYNCHRON
- verwenden explizite Komm — Message Passing (wenn sie was voneinander wollen)
  send → ○ recv
- ~~SYNCHRONISATION~~ zw. Prozesse



P. = Endlich
→ Komm. über Medium ↗ Abhängig impl.
Keine impl. Synchronisation
↳ Nur Komm

Keine andere Komm. → SHARED Nothing

Fokus: Korrektheit

⊕

Locality: Isoliert, wo Austausch stattfindet (lokalisiert, wo Austausch stattfinden kann)
→ zwingt Entwickler lokal zu denken

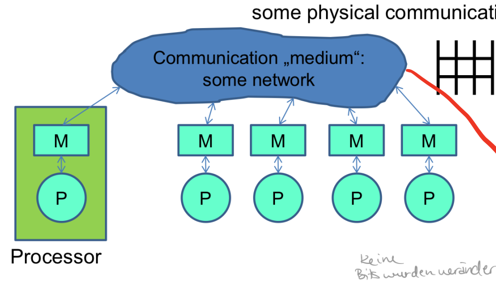
no race condition
↳ prim. DETERMINISTISCH

Lokalität oft ↑ Performance

Performance model:

- $ Comm. — 5c local comp.
- ↓ so wenig wie möglich
- ↳ Besser 1 große msg als viele kleine msg

Communication medium realized by some physical communication network



ist dahir verantw.

keine Bits wurden verändert

Routing protocol (algorithm) implements correct, reliable, deadlock-free communication between message passing processes. Not part of model but handled by "lower layers"

Message Passing Interface

- Am häufigsten benutzte Schnittstelle für Message Passing
- Kann sehr nah an d. HW programmieren
- Verwendet für:
  - Einzelanwendungen mit ~~trivialen~~ Komm. anforderungen für HPC-Sys. & Cluster mit ~~trivialen~~ Komm.flt
  - HPC (fast exclusively)
  - Paradigmatic Realisierung von message model
  - Viele zu lernen für andere Interfaces

Design Prinzipien, Ziele, etc.

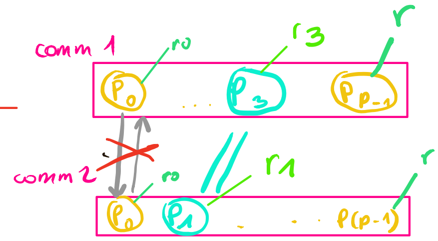
- High-Performance → wegen HW Nähe (low protocol stack overhead)
- Portability + Scalability (gab viele Portabilitätsprobleme) Kann SEHR GROSS skalieren
- memory efficient ← platzeffizient ← meistens im Userspace allozieren (Buffers)
- Kann koexistieren mit anderen Interfaces
- Unterstützt
  - Konstruktions-tools
  - heterogene Systeme
  - SPMD MIMD
  - libary Bauen → läuft in Bibliotheken
- Nur Spezifikation einer Bibliothek

⊕	⊖
Kann unabhängig von Compiler Support entwickelt werden	Umständlich zu nutzen → was Compiler weiß ↳ weiß Bibliothek nicht → muss bib. selber sagen

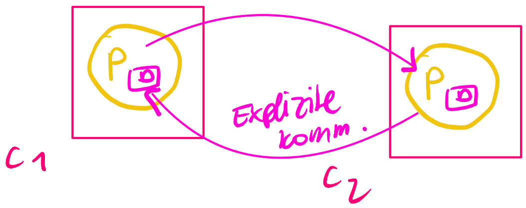
Programmier Model

- Set of processes — rang [0, …, p-1]
  - com. domain = communicator → können nicht einander interferieren
  - identifiziert durch Rang
  - können NUR innerhalb von comm. kommunizieren

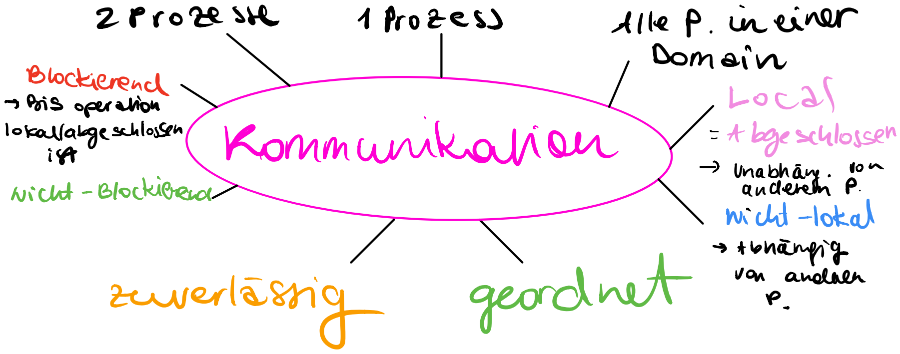
- kann mehrere comms geben
  - ↳ P. kann in 1+ comms sein
  - → mit anderem Rank

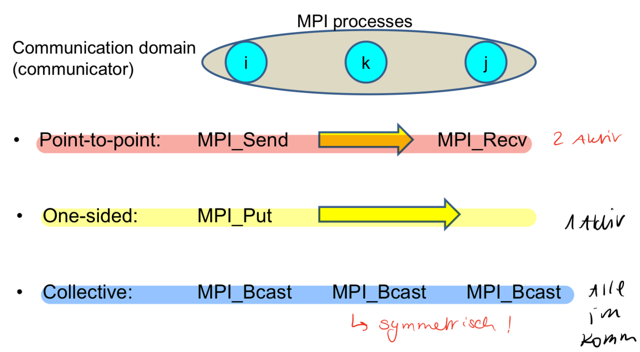


- P. hat lokale Daten



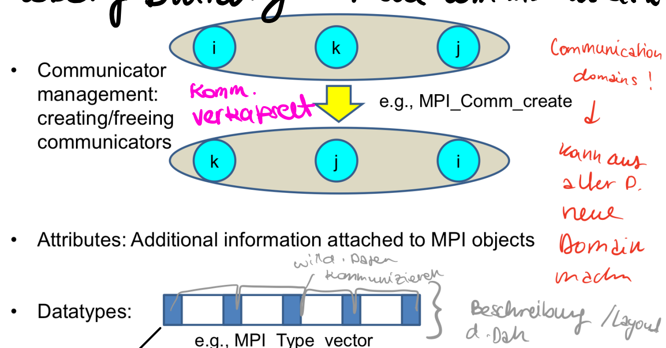
- Point to Point — 2 Prozesse
- 1-sided — 1 Prozess
- Collectiv — Alle P. in einer Domain





- Struktur von Daten ⊥ komm. modellen
- Comm. reflektiert physische topology
- kein Kosten model → Sagt nicht aus wie viel Ops kosten
  - MPI = rein semantisch
- Nicht sehr fehler tolerant!!

libary building → Neue comms aus alten!



Beschreiben layout / Struktur von Daten, die komm. werden

MPI Programm

First MPI program

```
#include <mpi.h>

int main(int argc, char *argv[])
{
  int rank, size;

  MPI_Init(&argc,&argv);   Init

  MPI_Comm_size(MPI_COMM_WORLD,&size);   } Build
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);   }

  fprintf(stdout,"Here is %d out of %d\n",rank,size);

  MPI_Finalize();   Ende
  return 0;
}
```

The 6 basic functions (plus two)…

Get time (in micro-seconds with suitably high resolution) since some time in the past:

```
double point_in_time = MPI_Wtime();
```

Synchronize the processes (really: only semantically); often used for benchmarking applications

```
MPI_Barrier(MPI_COMM_WORLD);
```

Good SPMD practice: Write programs to work correctly for any number of processes

```
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (size<10||size>1000000) MPI_Abort(comm,errcode);
if (rank==0) {
  // code for rank 0; may be special
} else if (rank%2==0) {
  // remainder even ranks
} else if (rank==7) {
  // another special one
} else {
  // all other (odd) processes…
}
```

Explicit assignment of work (code) to each process based on rank

Dangerous C practice (type error): Don't rely on C convention for Boolean expressions `if (rank) {…}`

The 6 basic functions

4

```
MPI_Init(&argc,&argv);
MPI_Finalize();
```

First and last call in MPI part of application; can only be called once

"Who/where am I?" in communication context/set of processes. Processes numbered (ranked) from 0 to size-1

```
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
```

Point-to-Point

Process rank i:

```
int a[N];
float area;
MPI_Send(a,N,MPI_INT,j,TAG1,MPI_COMM_WORLD);
MPI_Send(&area,1,MPI_FLOAT,j,TAG2,MPI_COMM_WORLD);
```

Process rank j:

```
int b[N];
float area;
MPI_Recv(b,N,MPI_INT,i,TAG1,MPI_COMM_WORLD,&status);
MPI_Recv(&area,1,MPI_FLOAT,i,TAG2,MPI_COMM_WORLD,
         &status);
```

Good practice for writing libraries

```
int my_library_init(comm,&libcomm)
{
  MPI_Comm_dup(comm,&libcomm); // store somewhere

  // library communication wrt. libcomm
  // initialize other library data structures
  // could be cached with libcomm (attributes)
}
```

Communication inside library uses libcomm, and will never interfere with communication using other communicators

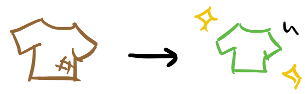
`MPI_Comm_dup`:
Collective operation, MUST be called by all processes in `comm`

Communicators

<program> executes

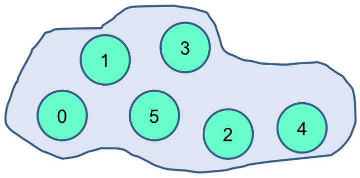
```
MPI_Init(&argc,&argv);
// sets up internal data structures, incl:
…
MPI_Comm_size(MPI_COMM_WORLD,&size);
```

können keine neuen coms. erstellen bzw. ändern

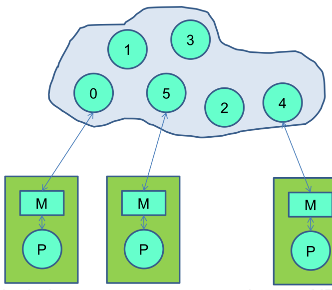
Nur  „aus all mach neu“

↳ Also aus alten Comm. neue machen

`MPI_COMM_WORLD`: Initial communicator containing all started processes. Static, never changes!



Communicator: distributed, global object with communication context, finite ordered set of processes that can communicate



Communicator: Distributed, global object, communication context, finite set of ordered processes that can communicate

Physical processors may run more than one MPI process (but most often only one)

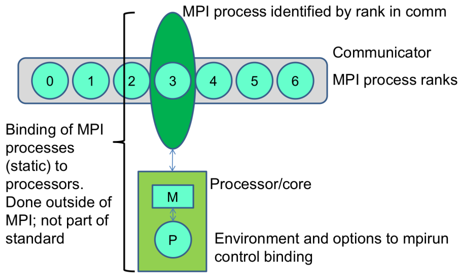
P. sind bestimmten cores zugewiesen

↳ wie Zuordnung von außen

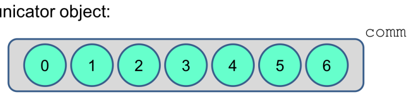
1 MPI P. ⟷ 1 physischen Prozessor

meist FIX

1+ MPI P. ausführen



Communicator object:



- All processes in a communicator can communicate
- All models (point-to-point, one-sided, collective; all other functionality)
- Has a size: number of processes
- Each process has a rank (0≤rank<size)

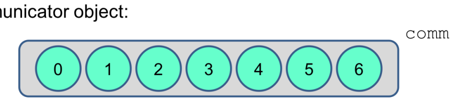
```
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);
```

MPI process: (normally) statically bound to some processor; can have different ranks in different communicators; canonically identified by rank in `MPI_COMM_WORLD`

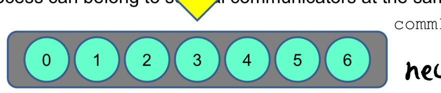
1. Duplikat machen

Für libs.

Communicator object:



- All processes in a commun tor can communicate
- All models (point-to-point, o -sided, collective; all other functionality)
- Has a size: number of proc es
- Each process has a rank (0 ank<size)
- A process can belong to se al communicators at the same time



neuer Comm.

```
MPI_Comm_dup(comm,&comm1);
```

`MPI_COMM_WORLD`, `comm1`, `comm2`, …:
An MPI (predefined) handle used to access and perform operations on MPI objects (communicators, windows, datatypes, attributes)

- Handles are (almost always) opaque: Internal MPI data structures cannot be accessed; but only manipulated through the operations defined on them
  (undurchsichtig → können nicht rein)
- MPI does not define how handles are represented (index into table, or pointer, or …)
- Handles in C and Fortran may be different

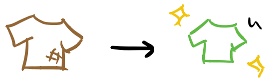
Example: `MPI_Comm_f2c(comm)`:
Returns C handle of Fortran communicator (no error code here)

- `MPI_Comm`: Communicators (distributed object)
- `MPI_Group`: Process groups (local object)
- `MPI_Win`: Windows for one-sided communication (distributed)

- `MPI_Datatype`: Description of data layouts (basic/primitive – or user-defined/derived; local)
- `MPI_Op`: Binary operators (built-in or user-defined)

- `MPI_Request`: Request handle for point-to-point
- `MPI_Status`: Communication status

- `MPI_Errhandler`:
- …

Nur  „aus alt mach neu"

Splitting communicators

Partition set of MPI processes (communicator) into disjoint sets (communicators) that can communicate independently

```
MPI_Comm_split(comm1,color,key,&comm2);
MPI_Comm_create(comm1,group,&comm2);
...
```

gibt sich beim Aufruf einen key

Calling MPI process may have different ranks in comm1 and comm2

- color, key: integers determine subsets and relative order, processes in comm2 will be in key order
- group: MPI_Group of ordered processes

```
MPI_Comm_rank(comm1,&rank);

MPI_Comm_split(comm1,color,key,&comm2);

// process’ role in comm2
MPI_Comm_size(comm2,&newsize
MPI_Comm_rank(comm2,&newrank
```

MPI_Comm_split (collective operation):
All processes with same color are grouped, order determined by key

- Calling processes are sorted into groups of same color
- In each group, processes are sorted according to key, ties broken by rank in calling communicator

Example: Even-odd split

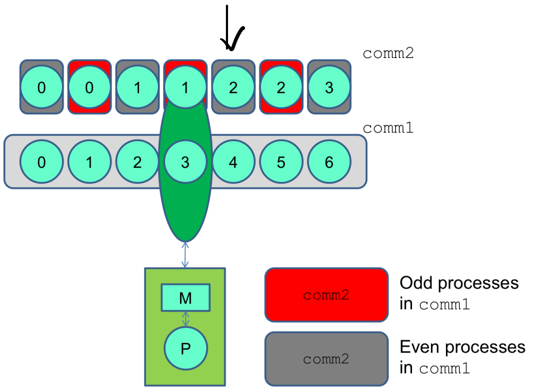
Even-numbered processes (in communicator comm1) shall work together, independently of the odd-numbered processes

General use: Divide-and-conquer type algorithms (Quicksort-like, see later)

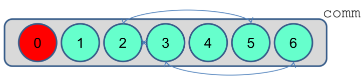
```
MPI_Comm_rank(comm1,&rank);

MPI_Comm_split(comm1,rank%2,0,&comm2);

// find process’ new ro   in comm2
MPI_Comm_size(comm2,&new  ze);
MPI_Comm_rank(comm2,&new  ank);
```

Ineren & odd aufteilen

Even numbered processes (in comm1) have color==0, odd processes color==1



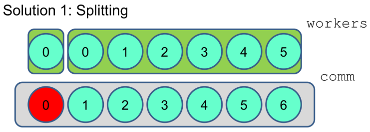
Example: Master-worker (non-scalable) pattern



- Master distributes work to individual workers, workers send results/new work to master
- Workers want to synchronize etc. independently of master

For workers: MPI_Barrier(comm), MPI_Allgather(comm), … (collective operations, see later) would deadlock, if master is doing something else
Solution: Split communicator, workers only communicator

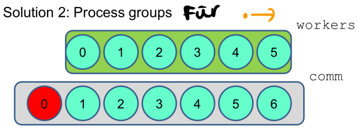
Solution 1: Splitting




```
MPI_Comm_split(comm,(rank>0 ? 1 : 0),0,&workers);
// workers for rank>0 in comm: all workers
// workers for rank==0 in comm: only master
```

MPI_COMM_SELF: Communicator with only process itself, size==1
MPI_COMM_NULL: No or invalid communicator

Solution 2: Process groups Für .→



One sided Komm

```
MPI_Comm_group(comm,&group); // get processes in comm
ranklist[0] = 0; // rank 0 to be excluded
MPI_Group_excl(group,1,ranklist,&workgroup);
MPI_Comm_create(comm,workgroup,&workers);
// rank 0 (in comm) not in workers
// workers==MPI_COMM_NULL for rank 0 in co
```

Group operation

→ Gruppen Operationen lokal
→ 1 P. kann drauf arbeiten ohne, dass was passiert

Communicator (MPI_Comm): A distributed, global object that can be manipulated through collective operations (MPI_Comm_split, MPI_Comm_dup, …)

Process group (MPI_Group): A process local object, ordered set of processes that can be manipulated locally by an MPI process

- MPI_Group_union, MPI_Group_intersection
- MPI_Group_incl, MPI_Group_excl
- MPI_Group_translate_ranks
- MPI_Group_compare
- …

See later, one-sided communication

Use: Building special communicators, one-sided communication

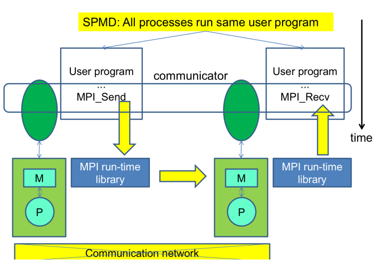
wenn am Ende d. Prg

```
MPI_Comm_free(&comm);
```

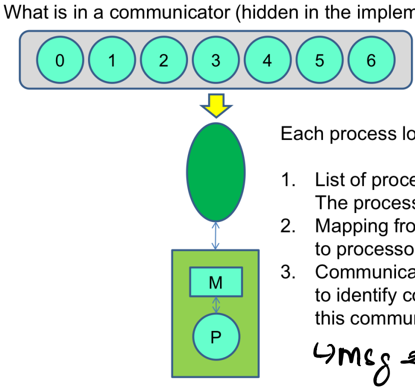
frees created communicator comm

Note:
MPI_COMM_WORLD and MPI_COMM_SELF cannot be freed

Good MPI practice:
Free any allocated MPI object after use (communicator, window, datatype, …)



What is in a communicator (hidden in the implementation)?



Each process locally maintains:

1. List of processes (ordered set): The process group
2. Mapping from rank to process to processor (implicit or explicit)
3. Communication context: A tag to identify communication on this communicator

welcher Prozess

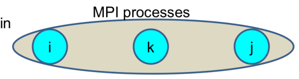
kann damit Komm. differenzieren

→ msg zuordenbar

Point to Point (2 Prozesse)

Point-to-point communication

Communication domain (communicator)



- Point-to-point: MPI_Send → MPI_Recv

→ Nicht lokale Semantik (kann nicht davon ausgehen, dass Send abgeschlossen werden kann, ohne, dass Recv, was gemacht hat)

Communication between two processes. Both explicitly involved in communication, one as sender, one as receiver

Point-to-point communication succeeds if

1. Sender specifies a valid rank within communicator (0≤rank<size) – and a valid (allocated) send buffer!
2. A receive with a matching source rank and tag is eventually posted on the same communicator
3. The amount of data sent is smaller or equal to the amount to be received (note: collective operations have different rule)
4. The type signature of the data sent matches the type signature of the data to be received

Comments:

1. Mistakes normally caught by MPI_Send: error (abort)
2. If not, deadlock
3. Otherwise, `MPI_ERR_TRUNCATE` or memory corruption (big trouble) at receiver!
4. `MPI_INT` matches `MPI_INT`, and so forth… rarely checked/enforced, user's responsibility

→ Does not work:

Process rank i:

```
double a;
MPI_Send(&a,1,MPI_DOUBLE,j,1111,MPI_COMM_WORLD);
```

Communication would likely take place, but violates 4.

Process rank j:

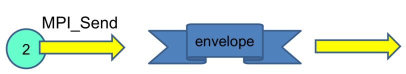
```
double a;
MPI_Recv(a,sizeof(double),MPI_BYTE,i,1111,
         MPI_COMM_WORLD,MPI_STATUS_IGNORE);
```

Very bad practice: Type information (double) lost (process i and j could be on different types of processors, little/big endian)

Programmierfehler

MPI macht KEINE Typkonversion / Datenkonversion

Message in transit identified by "envelope" (meta-information):



1. Communication context (identifier): Distinguishes communication on different communicators
2. Source (implicit)
3. Destination
4. Tag
5. Message type information (header, data block, error, …)

Implementation: The "envelope" is not accessible to application programmer (and not specified by MPI standard)

Performance: MPI is designed to allow small message envelope

Implementation: The "envelope" is not accessible to application programmer (and not specified by MPI standard)

Performance: MPI is designed to allow small message envelope

Not part of envelope: Typ-Information

- Type and structure of message: What is being sent is a sequence of basic datatypes (`int`, `double`, `char`, …); and it is expected that receiving process has space for exactly the same sequence of basic datatypes

Properties Point-to-Point

DEADLOCK FREE → **NON-OVERTAKING** — **RELIABLE**

DEADLOCK FREE:
→ Blockierend Send & Recv
⇒ wenn zurück kommt Op aus lokaler Sicht abgeschlossen
⇒ Puffer leer (Resourcen frei)

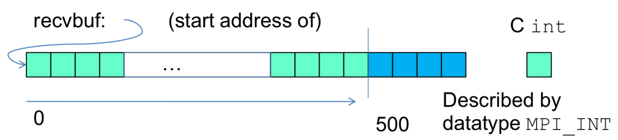
NON-OVERTAKING:
Prozesse können sich nicht überholen
→ ORDERED

RELIABLE:
verlässlich komm.

Kann Deadlocks prog. !!

```
int recvbuf[600]; // large enough
count = 600; // equal or larger to what is being sent
MPI_Status status;
success =
MPI_Recv(recvbuf,count,MPI_INT,2,THISMSG,comm,&status);
```

Returns when a message from rank 2 has been received; information about data in status object. Takes forever (deadlock), if nothing is sent from 2



Point-to-point semantics

`MPI_Send(sendbuf,…,rank,tag,comm):`

Initiates and completes send to designated process rank:

- Completion is non-local: Call return may depend on receiving rank having initiated receive operation(*)
- Operation is blocking: When call returns all of buffer can be reused
- Sent messages to same rank with same tag are non-overtaking

(*)but may also buffer message

`MPI_Recv(recvbuf,…,rank,tag,comm,&status):`

Completes receive operation from specific rank, or ANY

- Operation is blocking: Call returns when full message has been received

Status object (half opaque): Information on communication

```
MPI_Status status;  // status handle
MPI_Recv(…,&status);
```

Status contains information on what was received:

Fixed fields in C:
status.`MPI_SOURCE`:
status.`MPI_TAG`
status.`MPI_ERROR`

Why? Don't we know this??

Fixed fields in FORTRAN:
status(MPI_SOURCE)
status(MPI_TAG)
status(MPI_ERROR)

If so: Consider `MPI_STATUS_IGNORE` as status argument in `MPI_Recv`

Reasoning about point-to-point communication

Deterministic messages are non-overtaking:

Messages sent to the same destination (rank) arrive in sent order at destination

MPI_Send → Msg 1 → MPI_Recv gets msg 1

MPI_Send → Msg 2 → MPI_Recv gets msg 2

Local time

No races

Example: Synchronization with 0-count message

```
MPI_Send(NULL,0,MPI_INT,dest,tag,comm);
```

0-count message must be sent! Why?

```
MPI_Recv(NULL,0,MPI_INT,source,tag,comm,&status);
```

Can be used for process synchronization: Receiving process (with rank dest) cannot proceed before sending process (rank source) has reached send operation. For pairwise synchronization, send 0-message back

Alternative: MPI_Ssend — Briefly, later

But MPI processes are not synchronized

Therefore, possibly… warte zeit kann entstehen } Versuchen zu vermeiden

MPI_Send → Msg 1 → At receiver: "unexpected message"

Possibly idle time

MPI_Recv

MPI_Recv

Local time

Definitely idle time

MPI_Send → Msg 2

unerwartete Nachricht → noch kein Receiver

MPI_Send `MPI_Send(sendbuf,…,rank,tag,comm):`

MPI_Send Determinate: Message always sent to specific rank (in comm) with specific tag

```
MPI_Recv(recvbuf,…,
         rank/MPI_ANY_SOURCE,
         tag/MPI_ANY_TAG,
         comm,&status):
```

} kann nicht-det. werden

receives from specific rank or some non-determined (ANY) rank, with specific or non-determined (ANY) tag

um Puffer richtig anzulegen

```
MPI_Probe(source,tag,comm,&status);
```

Return when a message with given source (or MPI_ANY_SOURCE) and tag (or MPI_ANY_TAG) in comm has arrived and is ready for reception; the count for message (and error code etc.) in status

After probe:
Receive message with `MPI_Recv(buffer,count,…,comm);`

Advanced note:
Can cause problems in multi-threaded MPI applications, not "thread safe" (fix in MPI 3.0)

Sources of non-determinism (1)

weiß nicht wer zuerst kam ; i || j ?

Process i — Process k — Process j

MPI_Send(tag) →

← MPI_Send(tag)

Local time

brauche genug Puffer für 2 msg

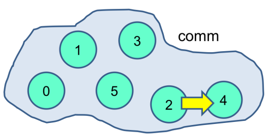
MPI_Recv(buf1, …, MPI_ANY_SOURCE,tag)

MPI_Recv(buf2, …, MPI_ANY_SOURCE,tag)

→ warte zeit zu minimieren

Either message may be received first. Problems if messages have different count and/or type

Deterministisch



"Process 2 to send 500 integers to process 4 (in comm)"

```
int THISMSG = 777; // message TAG
int count = 500;
if (rank==2) {
  int sendbuf[500] = {<the data>};
  MPI_Send(sendbuf,count,MPI_INT,4,THISMSG,comm);
} else if (rank==4) {
  int recvbuf[600]; // as large as send count
  MPI_Status status;
  MPI_Recv(recvbuf,count,MPI_INT,2,THISMSG,comm,
           &status);
}
```

P2 sendet

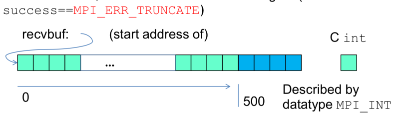
P4 empfängt

B Send

```
MPI_Recv(recvbuf,count,datatype,source,tag,comm,*status);
```

```
int recvbuf[600]; // large enough
count = 600; // equal or larger to what is being sent
MPI_Status status;
success =
MPI_Recv(recvbuf,count,MPI_INT,2,THISMSG,comm,&status);
```

"Start reception of message THISMSG from rank 2 in comm, store result in recvbuf, at most 600 consecutive integers (otherwise success==MPI_ERR_TRUNCATE)



lokal abgeschlossen
↳ können neues reinschreiben
→ ! muss nicht empfangen worden sein!

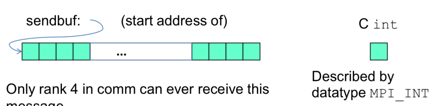
Recv B

```
MPI_Send(sendbuf,count,datatype,dest,tag,comm);
```

```
int sendbuf[500] = {<the data>};
count = 500;

MPI_Send(sendbuf,count,MPI_INT,4,THISMSG,comm);
```

→ obere Schranke

"Get message THISMSG stored in array sendbuf of 500 consecutive integers on the road to rank 4 in comm"



Only rank 4 in comm can ever receive this message

Blockierende operation
→ Nicht lokale semantik

Lokal / Abgeschlossen

Example: 2d-stencil, 5-point stencil — müssen alle Werte updaten über 10p

Für jeden Nachbarn / Elem wird d. gleiche Op verwendet
Anwendungsfall: Bildverarbeitung → glätten

Given mxn matrix u, for all 0≤i<m, 0≤j<n, update repeatedly
$u[i,j] \leftarrow \frac{1}{4}(u[i,j-1]+u[i,j+1]+u[i-1,j]+u[i+1,j]-h^2 f(i,j))$

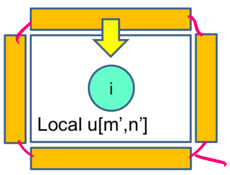
u[m,n]

u[i-1,j]
u[i,j-i] u[i,j] u[i+1,j]
u[i+1,j]

Special conditions on borders, i=0, i=m-1, j=0, j=n-1

Idee:
① Teile mxn matrix in ca gleich große m'xn' matrizen über p Prozesse
② Jeder Prozess hat eine Submatrix zum updaten
↯ Problem: Bei Grenzen müssen 2 P. miteinander kommunizieren → Komm

Per process (rank i) view of the parallelization



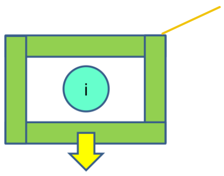
u[m',n']
u[i-1,j]
u[i,j-i] u[i,j] u[i+1,j]
u[i+1,j]

wird gebraucht für update

Before iteration, process needs to receive rows and columns of elements from the 4 neighboring processes

Per process (rank i) view of the parallelization

Halo



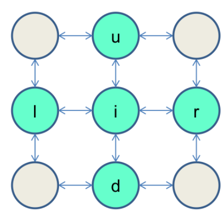
u[m',n']
u[i-1,j]
u[i,j-i] u[i,j] u[i+1,j]
u[i+1,j]

Send / Recv ↪ Symmetrisch

By symmetry, before iteration, process needs to send rows and columns of elements to the 4 neighboring processes

Arrange processes as a 2d mesh (process i needs to calculate ranks of its 4 neighboring processes l,r,u,d)

send ⟷ recv

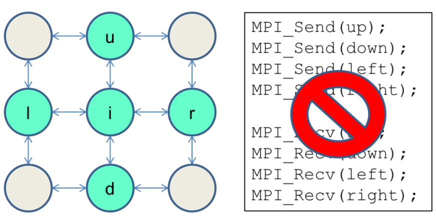


u[m',n']
u[i-1,j]
u[i,j-i] u[i,j] u[i+1,j]
u[i+1,j]

To update last row, values from first row of process d needed

May deadlock: MPI_Send has non-local completion semantics, each send may block until receive operation starts, and this may never happen

↯ Problem: nicht lokal Send Op.



Können NICHT annehmen, dass Send abgeschlossen sein kann, bevor Recv gestartet worden ist

By symmetry, all processes (except processes on the border of the process mesh) need to send/receive rows/columns with all four neighbor processes

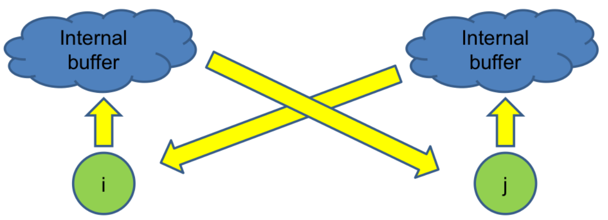
Send eigenartig ⇒ Non-lokal completion

MPI_Send

Short msg	Medium msg	Large msg
Buffered at sender / receiver → Processed later. MPI_Send can return IMMEDIATELY	May be buffered locally. Sent when rec. has been posted. MPI_Send can return fast	Participation of receiving process needed. MPI_Send cannot return before MPI_Recv becomes active

Probleme: Bei kleinem Datensatz geht es → bei großem Problem

Template MPI_Send implementation, short messages



MPI_Send(buffer, …,j, …);
MPI_Recv(buffer, …, j, …);

MPI_Send(buffer, …,i, …);
MPI_Recv(buffer, …, i, …);

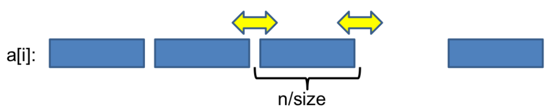
Drawback: Extra copy – costly for large buffers

MPI design principle: Library should not allocate unbounded buffers

→ deadlock

```
float *a = malloc((n/p+2)*sizeof(float));
a += 1; // offset, such that -1 and n/p can b addressed

if (rank>0) {
  MPI_Recv(&a[-1],1,MPI_FLOAT,rank-1,999,comm,&status);
  MPI_Send(&a[0],1,MPI_FLOAT,rank-1,999,comm);
}
if (rank<size-1) {
  MPI_Recv(&a[n/p],1,MPI_FLOAT,rank+1,999,comm,&status);
  MPI_Send(&a[n/p-1],1,MPI_FLOAT,rank+1,999,comm;
}
for (i=0; i<n/p; i++) {
  b[i] = a[i-1]+a[i]+a[i+1];
}
```

DEADLOCK! All processes attempt to receive, no progress, since receive operations cannot complete



Process j

MPI_Send(…,rank-1, …);
MPI_Recv(…,rank-1,…);

MPI_Send(…,rank-1,…);
MPI_Recv(…,rank-1,…);

MPI_Send(…,rank+1,…);
MPI_Recv(…,rank+1,…);

All processes trying to send, looks like deadlock

In MPI: Unsafe programming, if behavior (deadlock) depends on concrete implementation of send in MPI library.

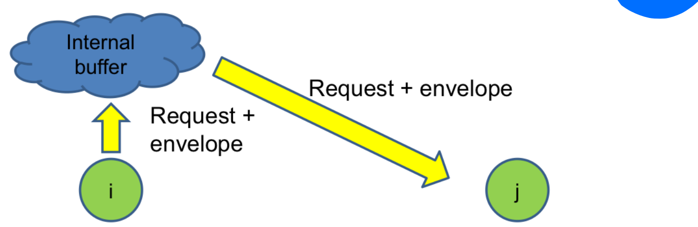
Safe(r) programming

Process 0		Process 1
`MPI_Send` `MPI_Recv`	⟷	`MPI_Send` `MPI_Recv`

Unsafe! Can be made safe by symmetry breaking (scheduling)

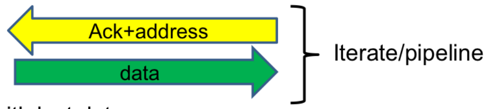
Process 0 (even)		Process 1 (odd)
`MPI_Send` `MPI_Recv`	⇄	`MPI_Recv` `MPI_Send`

Template MPI_Send implementation, long messages



MPI_Send(buffer, …,j, …);

MPI_Recv(buffer, …, i, …);



Send complete with last data

```
float *a = malloc((n/p+2)*sizeof(float));
a += 1; // offset, such that -1 and n/p can b addressed

if (rank>0) {
  MPI_Send(&a[0],1,MPI_FLOAT,rank-1,999,comm);
  MPI_Recv(&a[-1],1,MPI_FLOAT,rank-1,999,comm,&status);
}
if (rank<size-1) {
  MPI_Send(&a[n/p-1],1,MPI_FLOAT,rank+1,999,comm;
  MPI_Recv(&a[n/p],1,MPI_FLOAT,rank+1,999,comm,&status);
}
for (i=0; i<n/p; i++) {
  b[i] = a[i-1]+a[i]+a[i+1];
}
```

DEADLOCK? All processes attempt to send. Can a send complete without a matching receive operation? Depends on semantics and implementation of send operation

↳ Hängt von Send ab! (Library o. länged. msg)

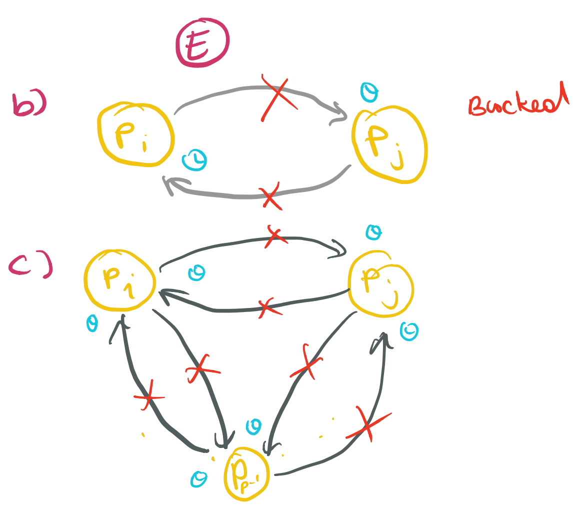
DEADLOCK:

a. All processes waiting for event that cannot happen
b. Process i waiting for action by process j, process j waiting for action by process i
c. Process i0 waiting for action by process i1, process i1 waiting for action by process i2, … process i(p-1) waiting for action by process i0

All deadlock forms are possible with MPI programs

Particularly problematic: Some deadlocks are context and MPI library implementation dependent. Such programs are called unsafe

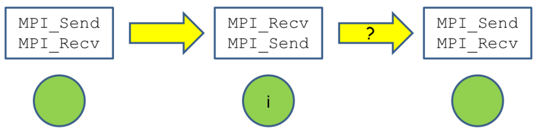
Definition: MPI program is unsafe if termination depends on whether messages are internally buffered by MPI_Send (or other MPI implementation properties).



Problem: What if the number of processes is odd?

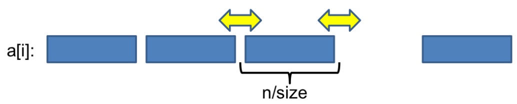
Example: Hillis-Steele prefix-sums algorithm

Recall from algorithms lecture



Round k:
Each process i receives partial sum from process $i-2^k$, and sends partial result to process $i+2^k$

Is process i odd or even in round k?



Process j

MPI_Send(…,rank-1, …); MPI_Recv(…,rank-1,…);
MPI_Send(…,rank-1,…); MPI_Recv(…,rank-1,…);

MPI_Recv(…,rank+1,…); MPI_Send(…,rank+1,…);
MPI_Recv(…,rank+1,…); MPI_Send(…,rank+1,…);

wollten das nicht!

Unfortunate even-odd scheduling leads to serialization. Last process size-1 receives after p=size steps!

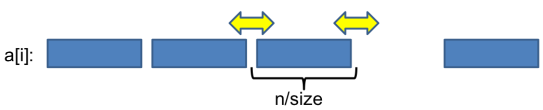
Safe(r) programming

Process 0	Process 1
MPI_Send MPI_Recv	MPI_Send MPI_Recv

Unsafe. Can be made safe safe by combined send-receive

Process 0		Process 1
MPI_Sendrecv	⟷	MPI_Sendrecv

keine Deadlocks!



MPI_Sendrecv(…,rank+1,…,rank-1,…);
MPI_Sendrecv(…,rank-1,…,rank+1,…);

Next attempt to parallelize data parallel loop

```
float *a = malloc((n/p+2)*sizeof(float));
a += 1;
if (rank>0) {
  MPI_Send(&a[0],1,MPI_FLOAT,rank-1,999,comm);
  MPI_Recv(&a[-1],1,MPI_FLOAT,rank-1,999,comm,&status);
}
if (rank<size-1) {
  MPI_Recv(&a[n/p],1,MPI_FLOAT,rank+1,999,comm);
  MPI_Send(&a[n/p-1],1,MPI_FLOAT,rank+1,999,comm,
           &status);
}
for (i=0; i<n/p; i++) {
   b[i] = a[i-1]+a[i]+a[i+1];
}
```

Symmetry breaking: Processes 0 and size-1 are special

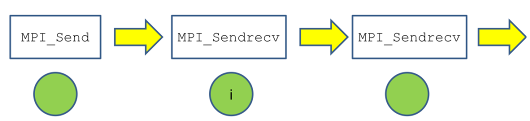
```
MPI_Sendrecv(void *sendbuf,
             int sendcount, MPI_Datatype sendtype,
             int dest, int sendtag,
             void *recvbuf,
             int recvcount, MPI_Datatype recvtype,
             int source, int recvtag,
              MPI_Comm comm, MPI_Status *status);
```

Combined send-receive operation, in one operation send to some process and receive from some (possibly different) process

Requirement: sendbuf and recvbuf must be disjoint

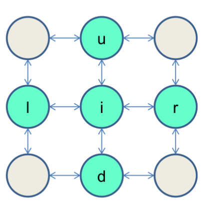
Performance advantage:
Can possibly better utilize (fully) bidirectional communication network

Example: Hillis-Steele prefix-sums algorithm (solution)



Round k:
Each process i receives partial sum from process $i-2^k$, and sends partial result to process $i+2^k$

Safe programming: Non-blocking communication




```
MPI_Request req[8];
MPI_Status stats[8];
```

```
MPI_Isend(up,&req[0]);
MPI_Isend(down,&req[1]);
MPI_Isend(left,&req[2]);
MPI_Isend(right,&req[3]);

MPI_Irecv(up,req[4]);
MPI_Irecv(down,&req[5]);
MPI_Irecv(left,&req[6]);
MPI_Irecv(right,&req[7]);

MPI_Waitall(8,req,stats);
```

weiß nicht, ob Puffer nach Isend wieder frei

ist op abgeschlossen?

Safe: I(mmediate), non-blocking operations with local completion semantics

Point-to-point semantics: Immediate operations

MPI_Isend(sendbuf,…,rank,tag,comm,request):

Locally initiates send to designated process rank
- Completion is local: Returns immediately, send has been initiated
- Operation is non-blocking: Buffers cannot be reused before completion has been checked/enforced
- Sent messages to same rank with same tag are non-overtaking

MPI_Irecv(recvbuf,…,rank,tag,comm,request):

Initiates receive operation from specific rank, or ANY
- Operation is non-blocking: Call returns immediately, message received after completion

```
MPI_Status status;
MPI_Request request;
MPI_Isend(sendbuf,…,comm,&request);
```

starts ("posts") send operation, returns immediately – local completion semantics, independent of receiving side – sendbuf must NOT be modified before operation is complete

"progress" information in request object:

```
MPI_Test(&request,flag,&status);
```

If flag==1 operation has completed, information in status

```
MPI_Wait(&request,&status);
```

Wait: Returns when operation has completed, information in status

Test and completion calls

Completion checked/enforced for all non-blocking operations by

- `MPI_Wait`
- `MPI_Test`

- `MPI_Waitall(number,array_of_requests,array_of_statuses)`
- `MPI_Testall`

- `MPI_Waitany`
- `MPI_Testany`

- `MPI_Waitsome`
- `MPI_Testsome`

Details, see MPI Standard

Sources of non-determinism (2)

MPI_Wait(req1)

DEADLOCK!

tag1 message has matched MPI_ANY_TAG

Point-to-point communication performance rules

Send operations: Creating envelope in local buffer, initiating communication (e.g., $\alpha+\beta m$ transfer time)

Latency!

Rule-of-thumb: Avoid many small messages, group into fewer, larger messages (α may be large)

Performance: MPI_Send may or may not have to wait for acknowledgement; can sometimes be faster than other send operations

Semantics: MPI_Send may (for large messages) depend on activity of receiving process

```
MPI_Sendrecv(sendbuf,sendcount,sendtype,dest,sendtag,
             recvbuf,recvcount,recvtype,source,recvtag,
             comm,&status);
```

is equivalent to

```
MPI_Request request;
MPI_Irecv(recvbuf,recvcount,recvtype,source,recvtag,
          comm,&request);
MPI_Send(sendbuf,sendcount,sendtype,dest,sendtag,
          comm);
MPI_Wait(&request,status);
```

Summary: Send modes, send semantics

Mode		Remark	Semantics
MPI_Send	Standard Returns when sendbuf can be reused		Non-local (potentially)
MPI_Ssend	Synchronous Returns when sendbuf can be reused AND receiver has started reception		Strictly non-local
MPI_Bsend	Buffered, returns immediately, data may be copied into intermediate buffer	Intermediate buffer from user space must have been attached with MPI_Buffer_attach	local
MPI_Rsend	Ready, standard	Precondition: matching receive MUST have been posted	Non-local

Only one receive mode (blocking and nonblocking)

`MPI_Recv/MPI_Irecv`

Blocking/non-blocking and modes are orthogonal, and can be arbitrarily combined

tag
→ kann Reihung von msg kontrollieren

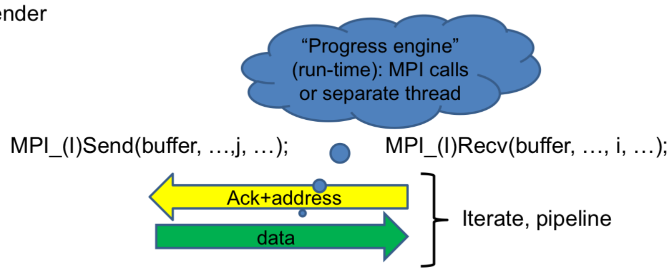
```
MPI_Iprobe(source,tag,comm,&flag,&status);
```

Non-blocking probe, `flag==1` means message with source and tag ready for reception in comm

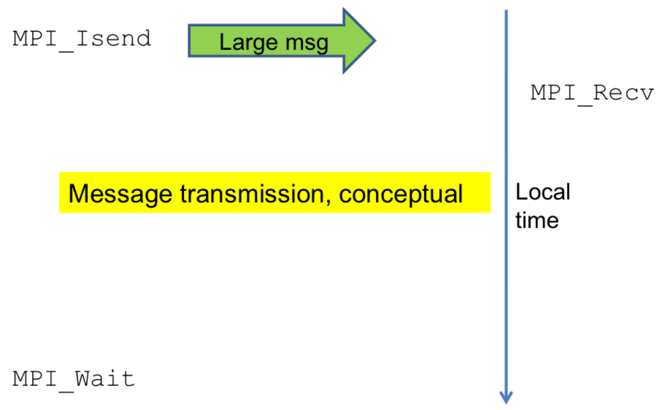
Information in `status` only when `flag==1`

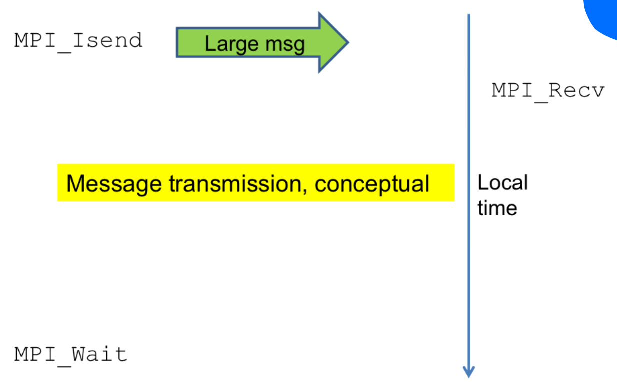
Point-to-point communication performance rules

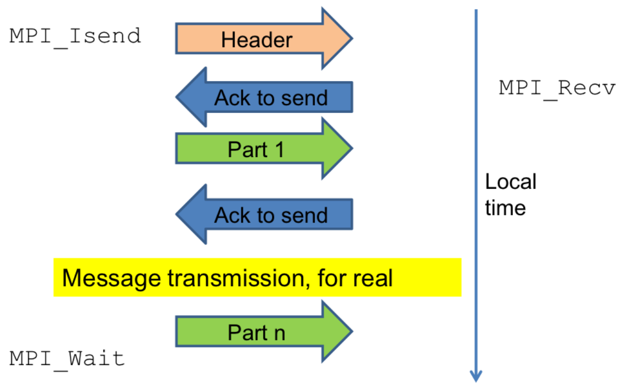
MPI_Isend: Can return immediately; progress and completion depends on activity of receiver AND often on activity/MPI calls by sender

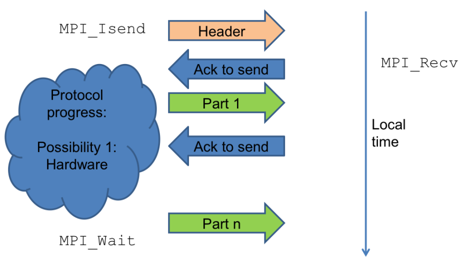


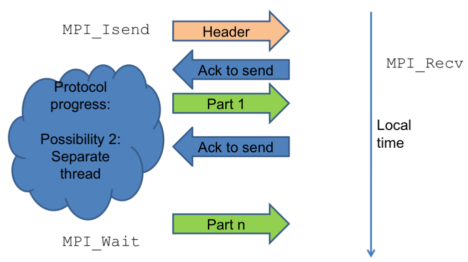
Again: Completion of MPI_Send and MPI_Isend does not imply anything about receiving process

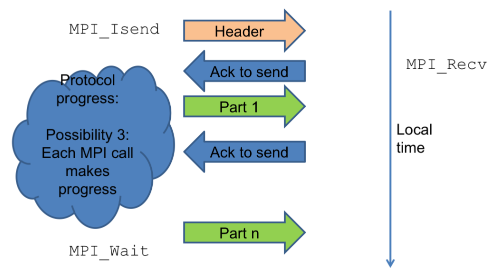






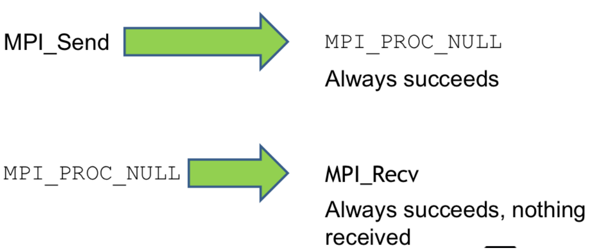






Other point-to-point communication features

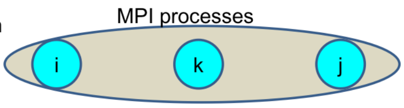
- MPI_PROC_NULL – "empty" process to send to and receive from
- (MPI_Ssend, MPI_Bsend)
- Persistent requests
- MPI_Cancel – dangerous!
- MPI_Sendrecv_replace



1 sided (1 Prozess) – Asynchron

One-sided communication

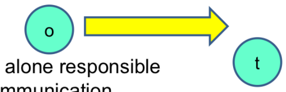
Communication domain (communicator)



- One-sided: MPI_Put → ; MPI_Get ← } ~~Blockierend~~

Communication between two processes, but only one process is explicitly involved with communication calls

MPI one-sided communication terminology



Origin process alone responsible for initiating communication, provides all arguments

Target process (semantically) not involved in communication

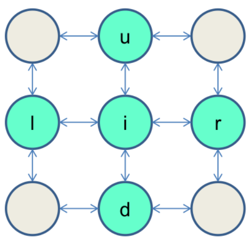
```
• MPI_Put(obuf,ocount,otype,…,win)
• MPI_Get(obuf,ocount,otype,…,win)
• MPI_Accumulate(obuf,ocount,otype,…,op,win);
```

Communication calls are non-blocking, local completion semantics: no guarantee that data have arrived before synchronization operation

Origin puts/get data from standard MPI buffer (buf,count,type)

One-sided communication by example

Safe neighbor exchange with one-sided (put) communication



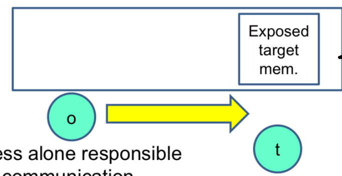

```
MPI_Put(up);
MPI_Put(down);
MPI_Put(left);
MPI_Put(right);
```

Process i actively puts data to neighbors

Issues:
- Where is the memory put to (and gotten from)?
- When are data ready/operations complete? brauchen Synch. um das sicher zu stellen

Bei o ⇄ o teil von Komm.

One-sided communication decouples communication and synchronization



Das dürfen andere Prozesse sehen

Origin process alone responsible for initiating communication, provides all arguments

Target process (semantically) not involved in communication

```
• MPI_Put(…,target,tdisp,tcount,ttype,win);
• MPI_Get(…,target,tdisp,tcount,ttype,win);
• MPI_Accumulate(…,target,tdisp,tcount,ttype,op,win);
```

rel. displacement

Data on target exposed in window structure, addressed with relative displacement

≈ window wie Komm.
↳ hat zusätzlich Speicherfläche

Communication window:
Distributed, global object containing memory for each process that can be accessed in one-sided communication operations

```
MPI_Win_create(base,size,dispunit,info,comm,&win);
```

Collective operation, all processes in comm provide a base address (size in Bytes may be 0), displacement unit

info (special MPI (key,value) object) can influence window properties (use MPI_INFO_NULL)

`MPI_Alloc_mem`: Special MPI memory allocator, sometimes beneficial (performance) for windows

Speicher ≠ Speicher

```
MPI_Put(obuf,…,target,targetdisp,…,win);
```

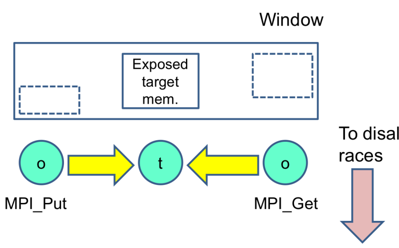
Data from obuf into target base+targetdispunit*targetdisp

NB: dispunit at target

Requirements: Origin data must fit into target buffer, type signatures must match, i.e., length of origin data at most length of target data, same sequence of basic types

Note:
Same rules as for point-to-point communication, `MPI_PROC_NULL` is a valid target process (nothing happens)

Vermeiden von Deadlock
2 x Put → disjunkt. adressen



To disallow data races

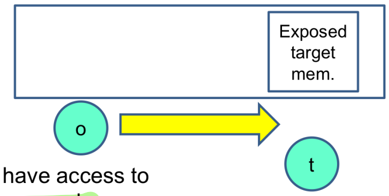
Rules:
Concurrent gets/puts must access disjoint target addresses. Very strict rules, violation is undefined (and usually not checked)

`MPI_Accumulate`: Atomic (at level of basic datatype) update at target, concurrent accumulates allowed (etwas wie sendrecv)

Synchronisierungsfrage

Communication epoch model

Teile P in Epochen
→ Darf bestimmte Dinge machen



Origin must have access to target: Access epoch

Target must expose memory: Exposure epoch
~ Zugriff möglich → welche P

End of epoch:
Access/exposure completed, data at origin processed (put or gotten), data on target arrived/accumulates complete

Globale Synchronisation / Kollektiv

Synchronization, epochs

Alle müssen d. ausführen
↳ gleich oft iterieren

Active (global) synchronization, both origin and target processes involved

```
MPI_Win_fence(assert,win);
```

Collective operation, all processes in comm of win must call.
Closes previous epoch, opens access epoch to all processes, opens exposure epoch for all processes

Assertion can control opening/closure behavior, use for tuning

Synchronization, epochs

Paarweise Synch.

Active (dedicated) synchronization, both origin and target processes involved

```
MPI_Win_start(…,group);
MPI_Win_complete();
```

```
MPI_Win_post(…,group);
MPI_Win_wait();
```

Opens/closes access epoch, targets as process group (MPI_Group)

Opens/closes exposure epoch, origins as process group (MPI_Group)

Sagt explizit, welcher Prozesse für Access & welche zum exponieren

Kein Schema

Sagt, welche gruppe exposed is & zu welcher wir Access haben — können manipuliert werden

"generalized" pairwise synchronization…

Synchronization, epochs

Passiv ✓

Passive synchronization, only origin process (logically) involved

```
MPI_Win_lock(locktype,target,assertion,win);
MPI_Win_unlock(target,win);
```

→ hat Access & exponiert
→ schließt beide Epochen

Keine "lock" perse → warum? keine critical section, schwer MUTEX

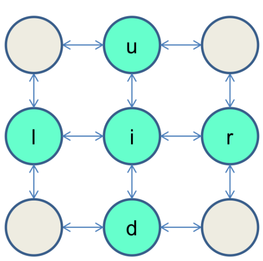
Opens/closes exposure epoch at origin, access epoch at target

Note 1: Not a lock (critical section), difficult to use for mutual exclusion (read-modify-write), weak mechanism

Note 2: Data at target may not be visible before target performs MPI_Win_lock on itself (and other weirdness)

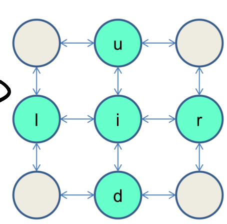
Target sieht nie was

Active, global synchronization with MPI_Win_fence




```
// prepare neighbor data
MPI_Win_fence(win);
MPI_Put(up);
MPI_Put(down);
MPI_Put(left);
MPI_Put(right);
MPI_Win_fence(win);
// data from neighbors ready
```

Active, dedicated synchronization with MPI_Win_start-post




```
// prepare neighbor data
MPI_Win_start([l,u,r,d],win);
MPI_Win_post ([l,u,r,d],win);
MPI_Put(up);
MPI_Put(down);
MPI_Put(left);
MPI_Put(right);
MPI_Win_wait(win);
MPI_Win_complete(win);
// data from neighbors ready
```

brauchen Access zu Nach. — Exponieren für nachs

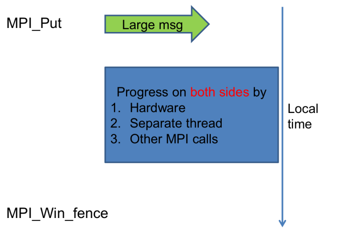
```
MPI_Win_free(win);
```

Free after use… (like other MPI objects); freeing the memory per process is NOT handled by MPI

```
base = (void*)malloc(size);
// or: MPI_Alloc_mem(size,MPI_INFO_NULL,&base);
MPI_Win_create(base,size,dispunit,info,comm,&win);

… // one sided communication epochs

MPI_Win_free(&win);
free(base);
// or: MPI_free_mem(base);
```

Window is leer → aber nicht d. Speicher!

A note on progress



Possible that all communicaton happens here…

Schauen, dass Epochen nicht zu lang sind

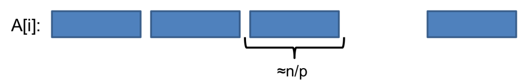
Example: Binary search with one-sided communication


```
l= -1; u = n;
do {
  m = (l+u)/2;
  if (x<A[m]) u = m; else l = m;
} while (l+1<u);
i = l;
```

Binary search for key x in array A[0,…,n-1] in at most ceil(log n) steps. Upon termination A[i]≤x<A[i+1]

Task: Implement sequential binary search in MPI (distributed memory, message passing), many processes search simultaneously

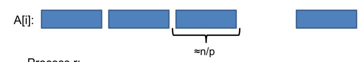
Side note: Binary search cannot be sped up (much) by parallel processing (Snir lower bound)




```
MPI_Comm_size(comm,&p);
MPI_Comm_rank(comm,&r);
```

Springe hin & her

Array A distributed (roughly) evenly over p MPI processes, process r, 0≤r<p, has block of (roughly) n/p elements

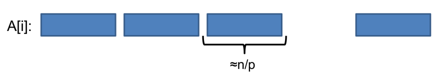


Process r:

```
l= -1; u = n;
do {
  m = (l+u)/2;
  mA = <get A[m]>
  if (x<mA) u = m; else l = m;
} while (l+1<u);
i = l;
```

Where is the middle element, and how to get it?

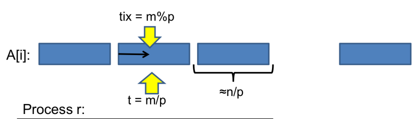
Process r can compute where A[m] is, but this process cannot know that r want to read an element: Fits one-sided model



Process r can compute where A[m] is, but this process cannot know that r want to read an element: Fits one-sided model

All processes make their block of A accessible to other processes in communication window

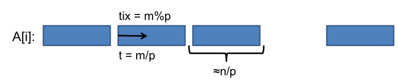
```
A = (int*)malloc(n/p*sizeof(int));
… // init block (see later)

MPI_Win_create(A,(n/p)*sizeof(int),sizeof(int),
               MPI_INFO_NULL,comm,&win);
```



Process r:

```
l= -1; u = n;
do {
  m = (l+u)/2;
  t = m/p; // target process
  tix = m%p; // index at target
  MPI_Get(&mA,1,MPI_INT,
          t,tix,1,MPI_INT,win);

  if (x<mA) u = m; else l = m;
} while (l+1<u);
i = l;
```

Problem: Need to make sure that data have been received



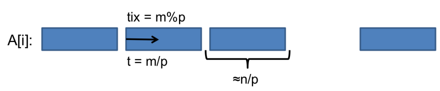
Process r:

```
l= -1; u = n;
do {
  m = (l+u)/2;
  MPI_Win_fence(0,win);
  t = m/p; // target process
  tix = m%p; // index at target
  MPI_Get(&mA,1,MPI_INT,
          t,tix,1,MPI_INT,win);
  MPI_Win_fence(0,win);
  if (x<mA) u = m; else l = m;
} while (l+1<u);
i = l;
```

Solution 1: Open/close access and exposure epochs with collective fence

Problem: Fence is collective, all processes in win must perform call

Must ensure that all processes do same number of iterations (calls to fence)



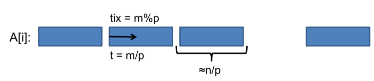
Process r:

```
l= -1; u = n;
do {
  m = (l+u)/2;
  MPI_Win_fence(MPI_NO_PRECEDE,win);
  t = m/p; // target process
  tix = m%p; // index at target
  MPI_Get(&mA,1,MPI_INT,
          t,tix,1,MPI_INT,win);
  MPI_Win_fence(MPI_NO_SUCCEED,win);
  if (x<mA) u = m; else l = m;
} while (l+1<u);
i = l;
```

Tuning: Use assertions.

`MPI_NO_PRECEDE` asserts that fence call does not complete any local Get/Put's

`MPI_NO_SUCCEED` asserts that fence does not start any local Put/Get's



Process r:

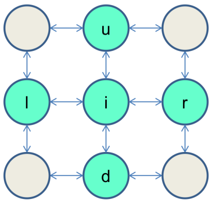
```
l= -1; u = n;
do {
  m = (l+u)/2;
  t = m/p; // target process
  MPI_Win_lock(MPI_LOCK_SHARED,t,0,
               win);
  tix = m%p; // index at target
  MPI_Get(&mA,1,MPI_INT,
          t,tix,1,MPI_INT,win);
  MPI_Win_unlock(t,win);
  if (x<mA) u = m; else l = m;
} while (l+1<u);
i = l;
```

Solution 2: Open/close access and exposure epochs with passive lock

Drawback: probably slow(er)?

But, recall that n»log n

Convenience functionality: d-dimensional MPI process naming

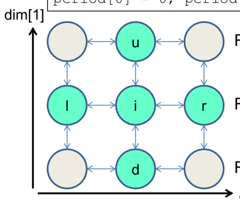


For d-dimensional stencil computation:

MPI processes are organized into a d-dimensional mesh: Each process rank needs to be able to compute efficiently in O(1) operations its 2d neighbor ranks

By-hand solution: Row-order numbering. Assume d=2, let p=rc (row-column), rank i, 0≤i<p, has coordinate (i/c,i%c), coordinate (a,b), 0≤a<r, 0≤b<c, has rank a*c+b

```
MPI_Comm_size(comm,&p);
r = sqrt(p); c = p/r; // or try MPI_Dims_create
dim[0] = c; dim[1] = r;
period[0] = 0; period[1] = 0;
```



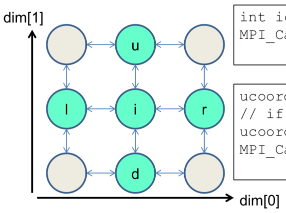
Rank 0: [0,0] Rank 1: [1,0] Rank 2: [2,0]

Rank 3: [0,1] Rank 4: [1,1] Rank 5: [2,1]

Rank 6: [0,2] Rank 7: [1,2] Rank 8: [2,2]

```
reorder = 0;
MPI_Cart_create(comm,d,dim,period,reorder,&newcomm);
```

Computing coordinates of u, d, l, r (Solution 1)




```
int icoord[2];
MPI_Cart_coords(newcomm,i,2,
                icoord);
```

```
ucoord[0] = icoord[0];
// if icoord[1]≠0
ucoord[1] = icoord[1]+1;
MPI_Cart_rank(newcomm,ucoord,
              &u);
```

Etc. MPI_Cart_rank requires coordinate in dimension i to be in range, when `period[i]==0`

MPI functionality: Cartesian communicators

```
MPI_Cart_create(comm,d,dim,period,reorder,&newcomm);
```

creates new communicator with helper functionality for d-dimensional Cartesian coordinate addressing. Collective operation, all processes in comm must call

`d`: number of dimensions (1, 2, 3, …)
`dim`: d-element array, `dim[i]` is size in i'th dimension, must hold that $\prod$`dim[i]`≤p
`period`: d-dimensional flag-array, if `period[i]==1` (true) the Cartesian grid is periodic in the i'th dimension

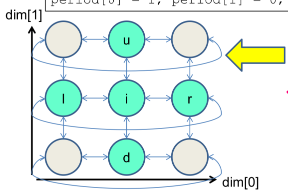
gibt # P. in Dimension

Periodisch

Here: Let `reorder=0` (false), otherwise MPI library may attempt to rerank processes to let virtual, Cartesian topology fit better with communication network topology

Umordnen

```
MPI_Comm_size(comm,&p);
r = sqrt(p); c = p/r; // or try MPI_Dims_create
dim[0] = c; dim[1] = r;
period[0] = 1; period[1] = 0;
```

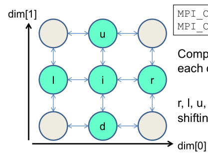


Periodic in dimension 0, processes on fringe are neighbors

→Ein Ring

```
reorder = 0;
MPI_Cart_create(comm,d,dim,period,reorder,&newcomm);
```

Computing coordinates of u, d, l, r (Solution 2)




```
MPI_Cart_shift(newcomm,0,1,&r,&l);
MPI_Cart_shift(newcomm,1,1,&u,&d);
```

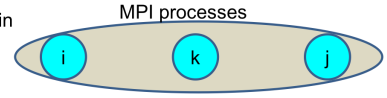
Compute neighbor ranks for one hop in each dimension

r, l, u, d will be `MPI_PROC_NULL` when shifting out of range

Collective komm (800) Alle Prozesse sind involviert

Collective communication

Communication domain (communicator)



} Jeder P. in comm. muss Operation ausführen

- Collective: MPI_Bcast MPI_Bcast MPI_Bcast

Communication among many (all) processes in communicator, all processes in communicator are explicitly involved, and must invoke same collective communication operation

→ Datenaustausch
→ symmetrisch
→ über Arg. Rolle bestimmen
→ z.B.: root

warum?

Collective operations: Motivation

Behauptung 1: Solche Operation kommen in Alg. öfters vor

→ Deswegen gut, Vorrat zu haben

Distributed data structure

```
while (!global convergence) {
  compute locally

  check for convergence
}
```

} Stencil update

Bsp.: Stencil

Distributed data structure

```
while (!global convergence) {
  compute locally
  check for local convergence

  All processes:
  globalflag = AND(localflags)
}
```

Collective operation: All processes take part in AND-computation (reduction with associative operation)

geht nur, wenn ALLE gehen!

Lokaler Konvergenzcheck
→ Checken, ob konvergiert
↳ Ist 1 P. nicht fertig
→ Nochmal eine Runde

Daten umverteilen
Rechnen lokal → kann sein, dass gewisse Ergebnisse, wo anders hin gehören

(1) Distributed data structure

```
while (!global convergence) {
  compute locally
  redistribute data
  check for local convergence

  All processes:
  globalflag = AND(localflags)
}
```

(2) Distributed data structure: Compute

```
while (!global convergence) {
  compute locally
  redistribute data
  check for local convergence

  All processes:
  globalflag = AND(localflags)
}
```

Alle Daten mit gleicher Farbe gehen zum gleichen Prozess →

Distributed data structure: Redistribute

```
while (!global convergence) {
  compute locally
  redistribute data
  check for local convergence

  All processes:
  globalflag = AND(localflags)
}
```

Collective operation: All-to-all communication; each process may have data for each other process; all processes take part in exchange

sehr intensiv

Distributed data structure: Initial distribution

```
distribute data from "master"
while (!global convergence) {
  compute locally
  redistribute data
  check for local convergence

  All processes:
  globalflag = AND(localflags)
}
collect result, write to file
```

Collective communication operations (non-symmetric), or collective I/O

Aufpassen mit Amdahl

Sammlung verteilen

COLLECTIVE OPERATIONS

MPI_Bcast: Data from root → all

not evenly +v { MPI_Scatter: Indiv. data root → all
{ MPI_Gather: Indiv. data all → root

Symm.

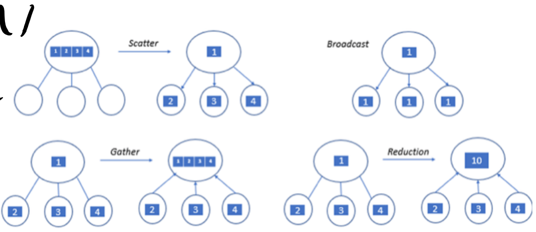
Blockierend / Nicht lokal

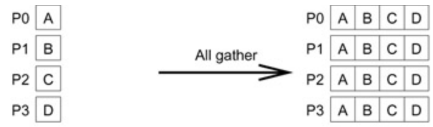
MPI_Alltoall: Indiv. all → all

MPI_Reduce: Apply ass. function (+, *, ...) to data from each process, res at root
MPI_Allreduce: res to all
MPI_Reduce_Scatter: res scattered / distr.

Pi (non-root) R(sbuf, rbuf, ... root, comm); Pj (root) R(sbuf, rbuf, root, comm); only sig. for root

Nicht Synch.



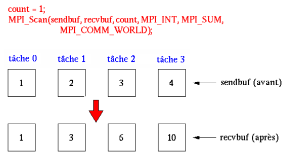


MPI_Scan: Prefix sum (reduction) → Nützlich für Lastverteilung

MPI_Exscan: value of process not used in reduction

MPI_Barrier: Sync. (waits until have reached routine)

MPI Scan



Collective operations: Motivation, why these?

Fit many applications (MPI_Allreduce for convergence tests, MPI_Scan for load balancing, MPI_Bcast for initialization, …)

Parallel, dense linear algebra (matrices, vectors) algorithms can be constructed from exactly these operations

Good symmetry and completeness properties (MPI_Gather, MPI_Scatter)

viele Alg. für Diskrete Algebra

Reduce

Example: MPI_Reduce

```
MPI_Reduce(void *sendbuf, void *recvbuf,
           int count, MPI_Datatype datatype,
           MPI_Op op, int root, MPI_Comm comm);
```

- `MPI_Op`: different operators (+, -, *, … even user-defined), must be associative (but not necessarily commutative)
- `MPI_Datatype`: Many different basic datatypes (`MPI_INT`, `MPI_DOUBLE`, …), even user-defined for user-defined op
- Any process can be root, communicator arbitrary
- `count`: Input per process is a vector, operation applied element-wise

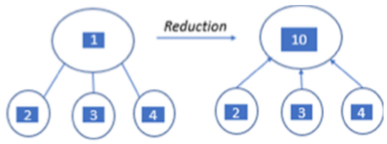
Collective operations: Motivation

Difficult to implement, need:

- Good performance on any given network, MPI library must work for any network and system
- Correctness, correct results for all possible (allowed) inputs, (communicators, datatypes, etc.)
- Safety, different collective operations must never interfere with each other and other communication

For many/most of the MPI collective operations, good algorithms exist, at least for fully connected networks

Implementing them well is not trivial



Set of numbers werden durch Fkt. reduziert

Better idea: Use properties of “+” to improve performance

Since “+” is associative

$$x_0+x_1+x_2+ \ldots + x_{p-1} = y$$

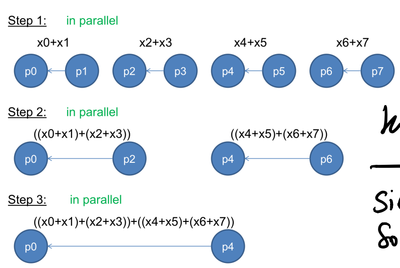
can be computed as

$$(x_0+x_1)+(x_2+x_3) + \ldots + x_{p-1} = y$$

and

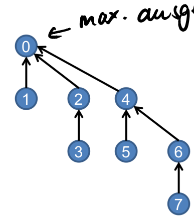
$$((x_0+x_1)+(x_2+x_3)) + \ldots((x_{p-2} + x_{p-1}) = y$$

etc.



kommt → sieht so aus

Binomial tree



← max. ausgelastet

Observation: Root node 0 active in each communication round

Same idea as round-optimal broadcast: “divide processes in two equal-sized sets, local reduction to local root in each, send result from one set to the set in which the global root is…”

Complexity of collective operations (fully connected network)

MPI Collective	Complexity
MPI_Barrier	O(log p)
MPI_Bcast	O(n+log p)
MPI_Gather/Scatter	O(n+log p)
MPI_Allgather	O(n+log p)
MPI_Alltoall	O(n+p) (*)
MPI_Reduce	O(n+log p)
MPI_Allreduce	O(n+log p)
MPI_Scan/Exscan	O(n+log p)

Important to find algorithms with small constants: See HPC lecture

(*) interesting tradeoffs possible

MPI_Bcast complexity is NOT O(n log p)

- p MPI processes (on p processors), n is total amount of data per process
- Strong (fully connected) network, linear communication cost

Complexity of collective operations (general)

MPI Collective	Complexity
MPI_Barrier	O(d)
MPI_Bcast	O(n+d)
MPI_Gather/Scatter	O(n+d)
MPI_Allgather	O(n+d)
MPI_Alltoall	O(n+p) (*)
MPI_Reduce	O(n+d)
MPI_Allreduce	O(n+d)
MPI_Scan/Exscan	O(n+d)

Important to find algorithms with small constants: See HPC lecture

(*) interesting tradeoffs possible

- p MPI processes (on p processors), n is total amount of data per process
- d = max(log p, diameter of network), determines latency

Collective operation semantics

Requirement:
If a process calls collective MPI_<A> on communicator C, then eventually all other processes in C must call MPI_<A> and no other collective inbetween (on that communicator)

gleiche root operation

Collective operations are safe: Collective communication on communicator C will not interfere with other communication on C

Requirement:
Collective operations must be called with consistent arguments: same root, same op, exactly matching amounts of data (see individual functions)

Collective operation semantics

If a process calls collective MPI_<A> on communicator C, then eventually all other processes in C must call MPI_<A> and no other collective inbetween (on that communicator)

Collective functions are blocking. A process returns when locally complete, buffers etc. can be reused. Completion semantics are non-local (most likely dependent on what other processes do) (*)

Collective functions are not synchronizing. A process may leave MPI_<A> as soon as it is locally complete (required local data sent and received)

Like MPI_Send

Exception: `MPI_Barrier(comm);`

(*) nonblocking collectives from MPI 3.0

} Blockierend

Incorrect:

Process i:
```
MPI_Bcast(buffer,…,root,comm);
MPI_Reduce(sbuf,rbuf,…,root,comm);
```

Process j:
```
MPI_Reduce(sbuf,rbuf,…,root,comm);
MPI_Bcast(buffer,…,root,comm);
```
} müsste umdrehen

Process local time

Note:
"incorrect" means that MPI may crash, deadlock, give wrong results! Or even work (for small counts): unsafe

Unsafe:

comm1: {i,j}
comm2: {i,j,k}

Process i:
```
MPI_Bcast(buffer,…,root,comm2);
MPI_Gather(sbuf,…,root,comm1);
```

Process k:
```
MPI_Bcast(buffer,…,root,comm2);
```

Process j:
```
MPI_Gather(sbuf,…,root,comm1);
MPI_Bcast(buffer,…,root,comm2);
```

Process local time

Unsafe:
May work for small counts, hang for large

können nicht sicher sein, dass Bcast i abgeschlossen ist bevor wir to Gather op kommen (von Pj?)

Kollektiv mit Anderen

Komm. Ops Arten

Safe:

Process i:
```
MPI_Bcast(buffer,…,root,comm);
MP_Recv(recvbuf,…,j,SOMETAG,comm,&status);
```

Process j:
```
MPI_Isend(sendbuf,…,i,SOMETAG,
    comm);
MPI_Bcast(buffer,…,root,comm);
```

Process local time

Point-to-point and one-sided and collective communication does not interfere

Correct:

Process i:
```
MPI_Bcast(buffer,…,root,comm);
```

Process j:
```
MPI_Bcast(buffer,…,root,comm);
```

Process local time

MPI_Bcast is blocking:
root: does not return before data have left buffer
Non-root: does not return before data from root have been received in buffer

Correct:

Process i:
```
MPI_Bcast(buffer,…,root,comm);
```

Process j:
```
MPI_Bcast(buffer,…,root,comm);
```

Process local time

MPI_Bcast is not synchronizing:
root: may return as soon as data have left buffer (independent of non-roots)
Non-root: may return as soon as data from root have been received in buffer (independent of other non-roots)

Correct:

comm1: {i,j}
comm2: {i,k}

Process i:
```
MPI_Bcast(buffer,…,root,comm2);
MPI_Gather(sendbuf,…,comm1);
```

Process k:
```
MPI_Bcast(buffer,…,root,comm2);
```

Process j:
```
MPI_Gather(sendbuf,…,comm1);
```

Process local time

Process involvement in/blocking behavior of collective call MPI_<A> is implementation dependent

Unsafe collective programming: Relying on synchronization properties

Observation:
Explicit MPI_Barrier calls are never (should never be) needed for correctness of MPI programs

If it seems so, there's probably something wrong

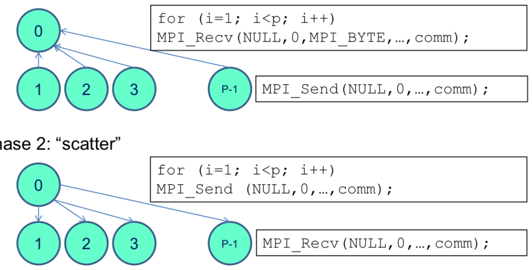
Bcast mit 0→0 umgesetzt

Example: A (legal) barrier implementation

Not suitable for timing. MPI libraries do something better…

Phase 1: "gather"

Phase 2: "scatter"

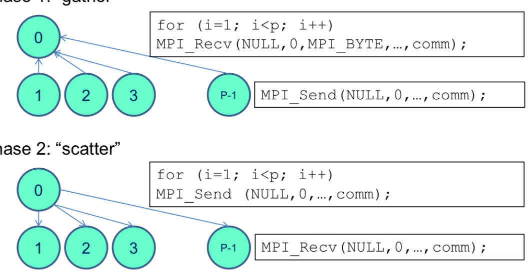


Example: A (legal) barrier implementation

Not suitable for timing. MPI libraries do something better…

Phase 1: "gather"

Phase 2: "scatter"



MPI "collectives" classification

	Class	regular	Irregular, vector
	Symmetric, no data	MPI_Barrier	
	Rooted	MPI_Bcast	
	Rooted	MPI_Scatter	MPI_Scatterv
	Rooted	MPI_Gather	MPI_Gatherv
	Symmetric, non-rooted	MPI_Allgather	MPI_Allgatherv
	Symmetric, non-rooted	MPI_Alltoall	MPI_Alltoallv, MPI_Alltoallw
	Rooted	MPI_Reduce	
(*)	Non-rooted	MPI_Reduce_scatter_block	MPI_Reduce_scatter
	Symmetric, non-rooted	MPI_Allreduce	
	Non-rooted	MPI_Scan	
	Non-rooted	MPI_Exscan	

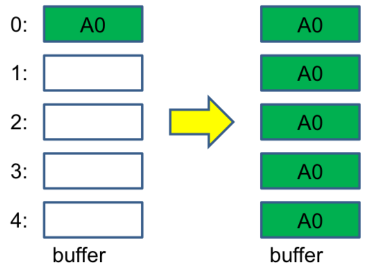
(*) MPI_Reduce_scatter_block: MPI 2.2 extension

Symmetric vs. non-symmetric: all processes have the same role in collective vs. one/some process (root) is special

Regular vs. irregular: each process contributes or receives the same amount of data from/to each other process vs. different pairs of processes may exchange different amounts of data

Bcast

```
MPI_Bcast(buffer,count,datatype,root,comm);
```

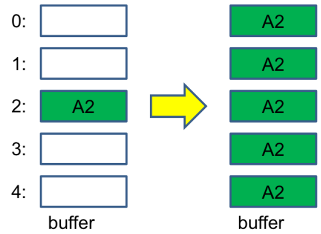


Example: root==0

Semantics: Data from root buffer is transferred to buffer of all non-root processes

Use: All processes broadcast with same root, buffer with same type signature (e.g., same count for basic datatypes like `MPI_FLOAT`)

```
MPI_Bcast(buffer,count,datatype,root,comm);
```



Example: root==2

Semantics: Data from root buffer is transferred to buffer of all non-root processes

Use: All processes broadcast with same root, buffer with same type signature (e.g., same count for basic datatypes like `MPI_FLOAT`)

MPI requirement

Collective functions MUST be called with consistent arguments:

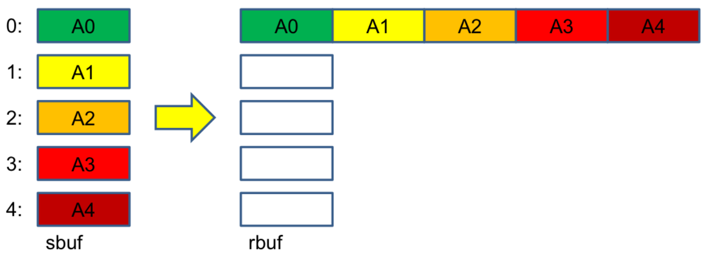
- Same root
- Matching type signatures (in particular: pairwise same size)
- Note: Number of elements sent and received must match exactly (unlike Send-Recv: sent≤recv and Get/Put)
- Same op (`MPI_Reduce` etc.)

```
int matrixdims[3]; // 3 dimensional matrix
if (rank==0) {
  MPI_Bcast(matrixdims,3,MPI_INT,0,comm);
} else {
  // do something on non-root first
  MPI_Bcast(matrixdims,2,MPI_INT,0,comm);
  // uhuh, Bcast probably works, but later…
}
```

Fehler

Gather

```
MPI_Gather(sbuf,scount,stype,rbuf,rcount,rtype,root,comm);
```

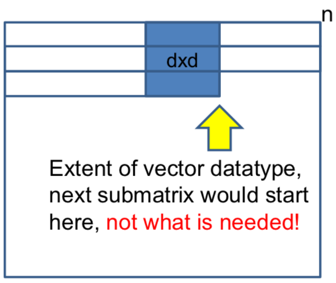


Semantics: Each process contributes a block of data to rbuf at root, blocks end up stored consecutively in rank order at root

Block from process i is stored at rbuf+i*rcount*extent(rtype)

Note: rcount is count of one block, not of whole rbuf

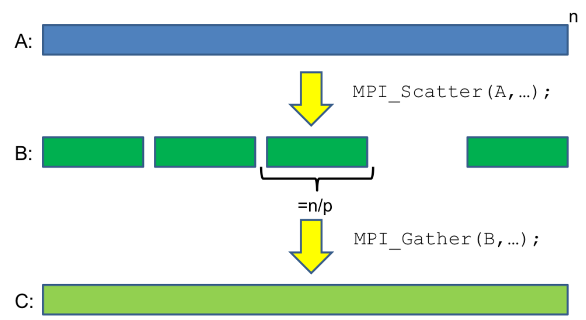
Challenge: Scattering submatrices of large input matrix



1. Describe submatrix as vector, block of d elements, stride n
2. `MPI_Scatter` will not work… (why?)
3. `MPI_Type_create_resize` with `MPI_Scatterv` can solve this problem

Advanced datatype use in HPC lecture

Example: Distributing initial array, collecting result



`MPI_Gather`/`MPI_Scatter`: All blocks same size

Further differences to point-to-point communication:

- Collective communication functions do not have tag argument
- Amount of data from process i to process j must equal amount of data expected by process j from process i
- Buffers of size 0 do not have to be sent

Process i:
```
MPI_Bcast(buffer,0,MPI_CHAR,…,root,comm);
```

Process j:
```
MPI_Bcast(buffer,0,MPI_CHAR,…,root,comm);
```

Correct! May be implemented as no-op, nothing broadcast

```
MPI_Allgather(sbuf,scount,stype,rbuf,rcount,rtype,comm);
```

Semantics: Each process contributes a block of data to rbuf at all processes, blocks end up stored consecutively in rank order

Block from process i is stored at rbuf+i*rcount*extent(rtype)

Fact:
Much better algorithms for MPI_Allgather than MPI_Gather+MPI_Bcast exist

A good MPI implementation will ensure that "best possible" algorithm is implemented, and that indeed MPI_Allgather always (all other things being equal) performs better than MPI_Gather+MPI_Bcast

Golden MPI rule

- Use collectives for conciseness and performance whereever possible
- Complain to MPI library implementer if performance anomalies are discovered

```
MPI_Allgather(sbuf,…rbuf,rcount,rtype,…comm);
```

is equivalent to

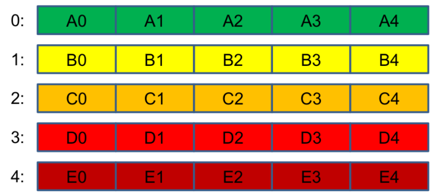
```
MPI_Gather(sbuf,…,rbuf,…,0,comm);
MPI_Bcast(rbuf,size*rcount,rtype,…,0,comm);
```

and

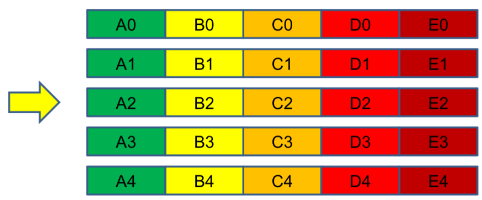
```
for (i) { // all-to-all broadcast
  if (i==rank) MPI_Bcast(sbuf,…,i,comm); else
  MPI_Bcast(rbuf+i*rcount*extent(rtype),…,i,comm);
}
memcpy(rbuf+rank*rcount*extent(rtype),sbuf,…);
```

But: Performance of library function should be better!

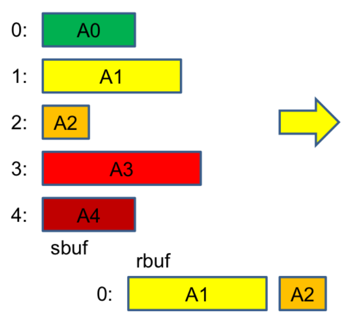
```
MPI_Alltoall(sbuf,scount,stype,rbuf,rcount,rtype,comm);
```



- Transpose
- All-to-all personalized communi-cation




```
MPI_Gatherv(sbuf,scount,stype,rbuf,rcount,rdisp,rtype,
            root,comm)
```



rbuf: address
rcount: count vector
rdisp: displacement vector
rtype: same receive type for all processes

Block from process i stored at rbuf+disp[i]*extent(rtype)

Displacement in extent(rtype) units; same for all processes (one rtype only)

Irregular (vector, v-) collectives:
Possibly different amounts of data destined to different processes

untersch. große Blöcke

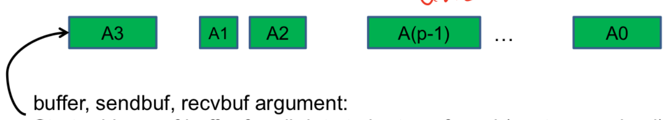
- MPI_Gatherv, MPI_Scatterv
- MPI_Allgatherv
- MPI_Alltoallv, MPI_Alltoallw

Data sizes and signatures must match pairwise, amount destined to a process must match what is required by that process

Processes can use different datatypes (data need not have the same structure, but signature must match)

Irregular collectives

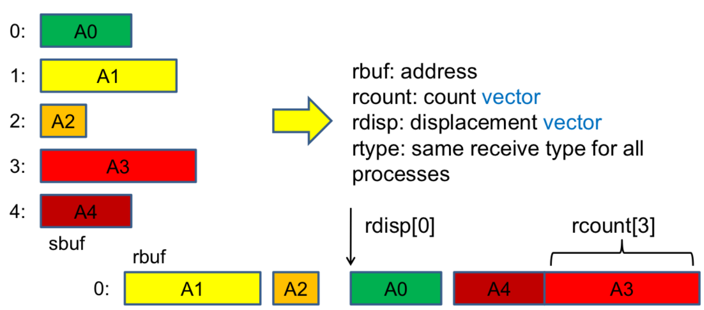
Jede Block einzeln beschrieben



buffer, sendbuf, recvbuf argument:
Start address of buffer for all data to be transferred (sent or received)

Segments to be transferred to/from different ranks may have different size (count[i]), and different displacement (displ[i]) relative to start address. Displacement is in datatype units

```
MPI_Gatherv(sbuf,scount,stype,rbuf,rcount,rdisp,rtype,
            root,comm)
```

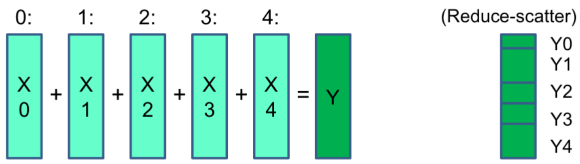


rbuf: address
rcount: count vector
rdisp: displacement vector
rtype: same receive type for all processes

Received data must not overlap. Displacement and count vectors significant only at root. Size/signature match pairwise

Zusammenspiel

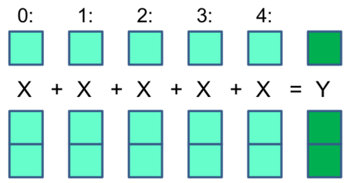
Reduction collectives



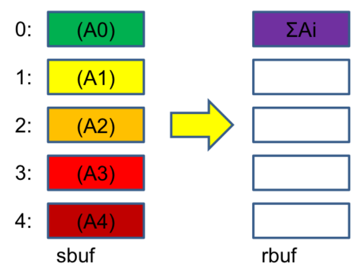
- Each process has vector of data X (same size, same type)
- Associative operation + (MPI builtin, MPI_SUM,…, user def)
- Element-wise reduction result Y=X0+X1+X2+ … + X(p-1) is stored at

1. Root: `MPI_Reduce`
2. All processes: `MPI_Allreduce`
3. Scattered in blocks (Y0 to 0, Y1 to 1, …): `MPI_Reduce_Scatter`

Reductions are performed elementwise on the input vectors



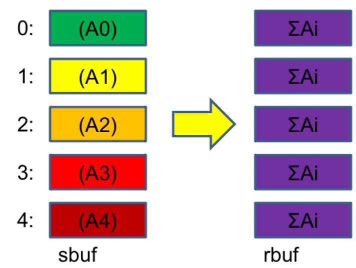

```
MPI_Reduce(sbuf,rbuf,count,type,op,root,comm);
```



Example: root==0

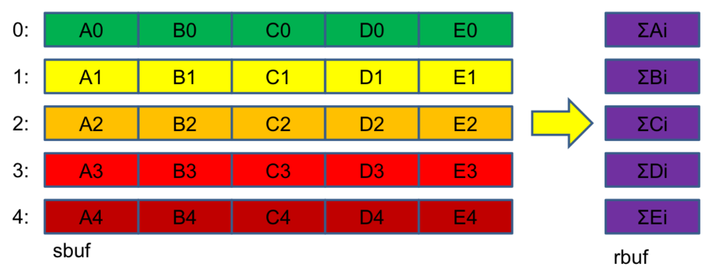
Semantics: All processes contribute sbuf (vector) of same size, elementwise result stored in rbuf at root. With `MPI_IN_PLACE` as sbuf, input is taken from rbuf

```
MPI_Allreduce(sbuf,rbuf,count,type,op,comm);
```



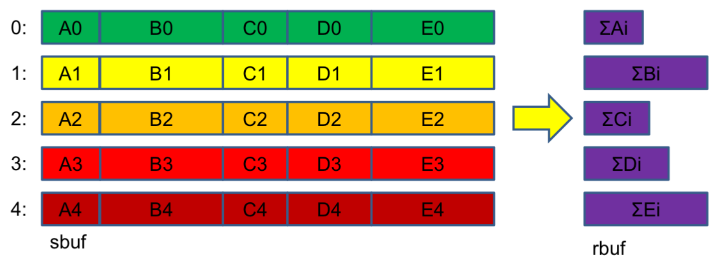
Semantics: All processes contribute sbuf (vector) of same size, elementwise result stored in rbuf at all processes. With `MPI_IN_PLACE` as sbuf, input is taken from rbuf

```
MPI_Reduce_scatter_block(sbuf,rbuf,count,type,op,comm);
```



Semantics: All processes contribute sbuf (vector) of same size, elementwise result is scattered in same sized blocks and stored in rbuf at each process. With `MPI_IN_PLACE` as sbuf, input is taken from rbuf

```
MPI_Reduce_scatter(sbuf,rbuf,counts,type,op,comm);
```



Semantics: All processes contribute sbuf (vector) of same size, elementwise result is scattered in blocks and stored in rbuf at each process. With `MPI_IN_PLACE` as sbuf, input is taken from rbuf. Since rbuf may have different size at different processes, counts[] is a vector. All processes provide same counts[] vector

MPI_Op	function	Operand type
MPI_MAX	max	Integer, Floating
MPI_MIN	min	Integer, Floating
MPI_SUM	sum	Integer, Floating
MPI_PROD	product	Integer, Floating
MPI_LAND	logical and	Integer, Logical
MPI_BAND	bitwise and	Integer, Byte
MPI_LOR	logical or	Integer, Logical
MPI_BOR	bitwise or	Integer, Byte
MPI_LXOR	logical exclusive or	Integer, Logical
MPI_BXOR	bitwise exclusive or	Integer, Byte
MPI_MAXLOC	max value and location of max	Special pair type
MPI_MINLOC	min value and location of min	Special pair type

```
MPI_Op_create(MPI_User_function *function,
              int commutative, MPI_Op *op);
```

makes it possible to define/register own, “user-defined”, binary, associative operators that can even work on derived datatypes

```
MPI_Op_free(MPI_Op *op);
```

Free it again after use…

müssen sagen wie verteilt

Solution 2: Matrix-vector multiplication

Assume p divides n, distribute M column-wise, each process has n/p columns of M, n/p elements of V

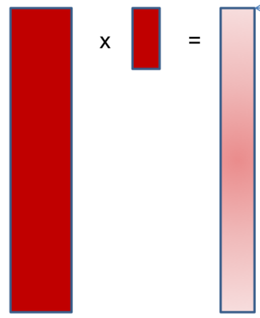
For process k, $0 \le k < p$, columns $c_i \le j < c_{i+1}$, where $c_i = k(n/p)$

$W[j] = \sum_{c0 \le i < c1} M'[j][i]*V'[i] + \sum_{c1 \le i < c2} M'[j][i]*V'[i] + \ldots + \sum_{c(p-1)0 \le i < cp} M'[j][i]*V'[i]$

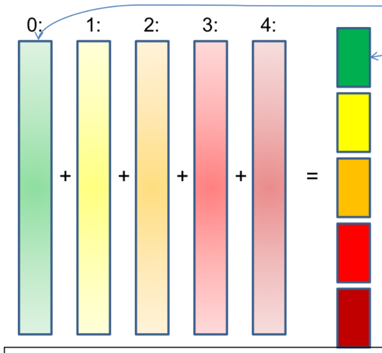
Correct since "+" is associative. Here i is index in global matrix, i-ck index in local column-matrix M' and local vector V'

Solution: distribute M column-wise, perform local M'V, sum partial results

1. Locally compute (nxn/p) matrix n/p vector product, M'V'



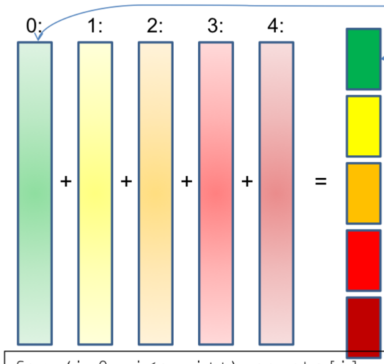
2. Sum partial result n vectors and scatter n/p blocks



Each rank stores n/p rows of result vector

```
MPI_Reduce_scatter_block(partial,result,n/p,MPI_FLOAT,
                         MPI SUM,comm);
```

2. Sum partial result n vectors and scatter n/p blocks



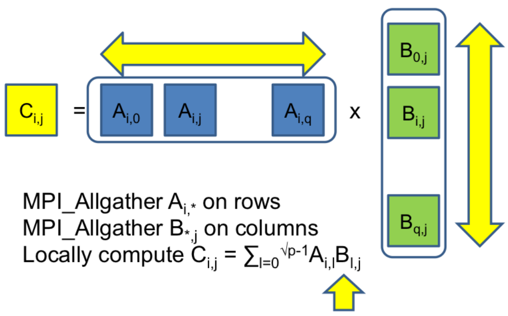
Beliebig Blocksize

Each rank stores n/p rows of result vector

```
for (i=0; i<p; i++) counts[i] = n/p;
MPI_Reduce_scatter(partial,result,counts,MPI_FLOAT,
                   MPI SUM,comm);
```

Folklore: Blockwise algorithm

Process (i,j) computes $C_{i,j}$ as $(A_{i,0},A_{i,1},\ldots A_{i,q})x(B_{0,j},B_{1,j},\ldots,B_{q,j})$, notation $q = \sqrt{p}-1$



1. MPI_Allgather $A_{i,*}$ on rows
2. MPI_Allgather $B_{*,j}$ on columns
3. Locally compute $C_{i,j} = \sum_{l=0}^{\sqrt{p}-1} A_{i,l}B_{l,j}$

Sequential subroutine

Main algorithm

C = 0
for $\sqrt{p}$ rounds, $l=0,\ldots,\sqrt{p}-1$
1. Broadcast $A_{i,l}$ to all processes on row i from process (i,l)
2. Broadcast $B_{l,j}$ to all processes on column j from process (l,j)
3. Update C = C+AB

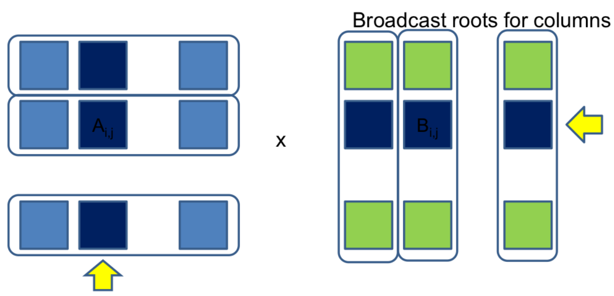
SUMMA: Scalable Universal Matrix-Multiplication Algorithm

Idea: Pipeline allgather (Recall: allgather = "p times bcast")

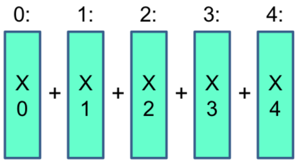
To compute $C_{i,j} = \sum_{l=0}^{\sqrt{p}-1} A_{i,l}B_{l,j}$, $\sqrt{p}$ communication rounds, in round l, $0 \le l < \sqrt{p}$, broadcast of $A_{i,l}$ on row communicator, broadcast of $B_{l,j}$ on column communicator

Note: This algorithm performs all sums in order, does not exploit commutativity of matrix addition

Main algorithm, round 1



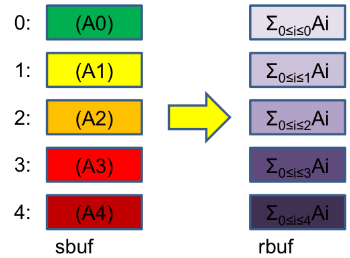
Scan collectives



wie viele sind vor mir?

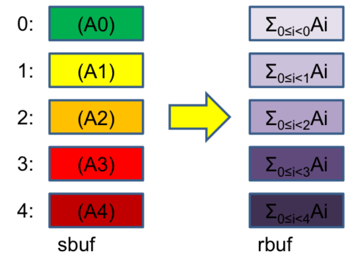
- Each process has vector of data X (same size, same type)
- Associative operation + (MPI builtin, `MPI_SUM`,…, user def)
- All prefix sums Yi=X0+…+Xi are computed and stored
- Yi at rank i: `MPI_Scan`
- Yi at rank i+1: `MPI_Exscan` (rank 0 undefined)

`MPI_Scan(sbuf,rbuf,count,type,op,comm);`



Semantics: Elementwise, inclusive prefix sums over sbuf vectors. Rank i stores sum from rank 0 to rank i (included)

`MPI_Exscan(sbuf,rbuf,count,type,op,comm);`



Semantics: Elementwise, exclusive prefix sums over sbuf vectors. Rank i stores sum from rank 0 to rank i (excluded)

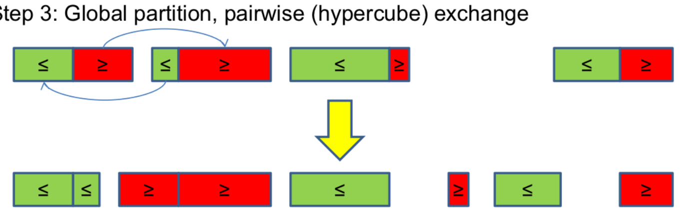
Quicksort

Step 1: Choose pivot, distribute to all processes

1. Process 0 chooses pivot, `MPI_Bcast`
2. Better, perhaps: Each process chooses pivot, `MPI_Allgather`, median of p as pivot

Step 2: Local partition

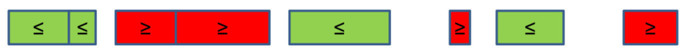
Step 3: Global partition, pairwise (hypercube) exchange



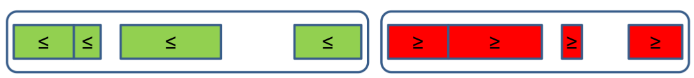
Even numbered process i sends larger than pivot elements to process i+1, and receives smaller than pivot elements from process i+1. Odd numbered process i receives larger than pivot elements from process i-1 and sends smaller than pivot elements to process i: `MPI_Sendrecv`

Implementation: May need to exchange number of elements first

Step 4: Split communicator and recurse



Use `MPI_Comm_split` with color `rank%2`: Even numbered processes will work on smaller than pivot elements, odd numbered processes on larger than pivot elements



After $\log_2 p$ steps, each process is in communicator by itself: Sort locally

Youran Lan, Magdi A. Mohamed: Parallel Quicksort in hypercubes. SAC 1992: 740-746

Drawbacks:
- Only for powers-of-two numbers of processors (hypercube algorithm)
- Load balance might be arbitrarily bad, one process could do all the work if pivot is bad

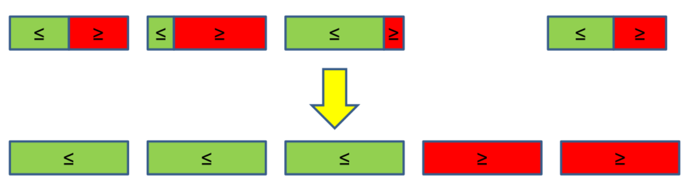
Analysis, assuming exact median pivot and Bcast:
T(n,p) = O(log p)+O(n/p)+T(n/2,p/2)
T(n,1) = O(n log n)

But what does `MPI_Comm_split` cost?

Since (n/2)/(p/2) = n/p, and $2^{\log p}$ = p, we get
T(n,p) = O($\log^2$p+(log p)(n/p))+O(n/p log(n/p)) =
O($\log^2$p)+O((log p)(n/p)+(n/p)(log n)-(n/p)(log p)) =
O($\log^2$p)+O(n/p log n)

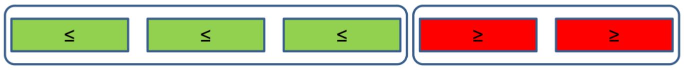
Speedup: O((n log n)/(n/p log n)) = O(p) for n»p

Improved step 3: Global partition by compaction



Smaller than pivot elements:
- For process i, compute how many smaller than pivot elements there are at process 0, 1, …, i-1: `MPI_Exscan` with result m_i
- Elements have to be sent to process m_i/(n/p) and (possibly) process m_i/(n/p)+1
- All processes need to inform their receiving processes (at most two) how many elements to receive: `MPI_Alltoall`
- Redistribution by `MPI_Alltoallv` or send and receive

Step 4: Split communicator and recurse



Use `MPI_Comm_split` with color=(`rank<middle ? 0 : 1`) to divide the processes into those with smaller than and those with larger than pivot elements

After $\log_2 p$ steps, each process is in communicator by itself: Sort locally

Same analysis applies, but good better balance irrespective of pivot choice

Example: Integer (bucket, counting) sorting

Given n integers in range [0,R[(=[0,1,2,…,R-1]) stored in array A

Bucket sort:
1. Count number of elements of each magnitude (“key”)

```
for (i=0; i<R; i++) bucket[i] = 0;
for (i=0; i<n; i++) bucket[A[i]]++;
```

2. Compute start index of each bucket: exclusive prefix-sums operation

```
for (i=1; i<R; i++) bucket[i] += bucket[i-1];
for (i=R-1; i>0; i--) bucket[i] = bucket[i-1];
bucket[0] = 0;
```

Given n integers in range [0,R[(=[0,1,2,…,R-1]) stored in array A

3. Distribute elements of A into buckets (in new array B)

```
for (i=0; i<n; i++) B[bucket[A[i]]++] = A[i];
```

4. Copy B back to A, if needed

Note:
As implemented, this bucket-sort is stable: Relative order or elements in A with same key is preserved

Example: Integer (bucket, counting) sorting in parallel

n integers in a given range [0,R[, distributed evenly across p MPI processes: m= n/p integers per process

A = [0 1 3 0 0 2 0 1 …] [ ]

B = (4, 2, 1, 3)

Input array distributed over p processes, A and B process local arrays of input elements and bucket sizes

n integers in a given range [0,R[, distributed evenly across p MPI processes: m= n/p integers per process

A = [0 1 3 0 0 2 0 1 …] [ ] B = (4, 2, 1, 3)

Step 1: Bucket sort locally, B[i] number of elements with key i

Step 2: `MPI_Allreduce(B,AllB,R,MPI_INT,MPI_SUM,comm);`

Step 3: `MPI_Exscan(B,RelB,R,MPI_INT,MPI_SUM,comm);`

Now:
Vector AllB contains the sizes of all buckets over all processes; RelB is for process rank the relative position of rank‘s elements in the buckets

n integers in a given range [0,R[, distributed evenly across p MPI processes: m= n/p integers per process

A = [0 1 3 0 0 2 0 1 …] [ ] B = (4, 2, 1, 3)

Step 1: Bucket sort locally, B[i] number of elements with key i

Step 2: `MPI_Allreduce(B,AllB,R,MPI_INT,MPI_SUM,comm);`

Step 3: `MPI_Exscan(B,RelB,R,MPI_INT,MPI_SUM,comm);`

Now:
Vector AllB contains the sizes of all buckets over all processes; RelB is for process rank the relative position of rank‘s elements in the buckets

n integers in a given range [0,R[, distributed evenly across p MPI processes: m= n/p integers per process

A = [0 1 3 0 0 2 0 1 …] [ ] B = (4, 2, 1, 3)

Step 5: compute number of elements to be sent to each other process, sendelts[i], i=0,…,p-1

Step 6:
`MPI_Alltoall(sendelts,1,MPI_INT,recvelts,1,MPI_INT,comm);`

Step 7: redistribute elements
```
MPI_Alltoallv(A,sendelts,sdispls,MPI_INT,C,recvelts,…,
              comm);
```

n integers in a given range [0,R[, distributed evenly across p MPI processes: m= n/p integers per process

A = [0 1 3 0 0 2 0 1 …] [ ] B = (4, 2, 1, 3)

Step 7: Redistribute elements
```
MPI_Alltoallv(A,sendelts,sdispls,MPI_INT,C,recvelts,…,
              comm);
```

Step 8: Reorder elements from C back to A

Possible optimization: Replace `MPI_Allreduce` by `MPI_Bcast`

Example: Integer (bucket, counting) sorting in parallel

The algorithm is stable → Radixsort

Choice of radix R depends on properties of network (fully connected, fat tree, mesh/torus, …) and quality of reduction/scan-algorithms

The algorithm is portable (by virtue of the MPI collectives), but tuning depends on systems: Concrete performance model needed, but analysis outside scope of MPI

Note: On strong network T(MPI_Allreduce(m)) = O(m+log p)

NOT: O(mlog p)

Quicksort and bucketsort data redistribution

In both cases, the redistribution has special structure, and it may be possible to do better with special algorithm than with MPI_Alltoallv

Worth trying:

- `MPI_Exscan` over block sizes; compute where data goes
- `MPI_Win_lock(MPI_LOCK_SHARED); MPI_Put; MPI_Win_unlock` to deliver data