Distributed-Memory Programmiermodelle IV

Stefan Lang

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
INF 368, Raum 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 13/14

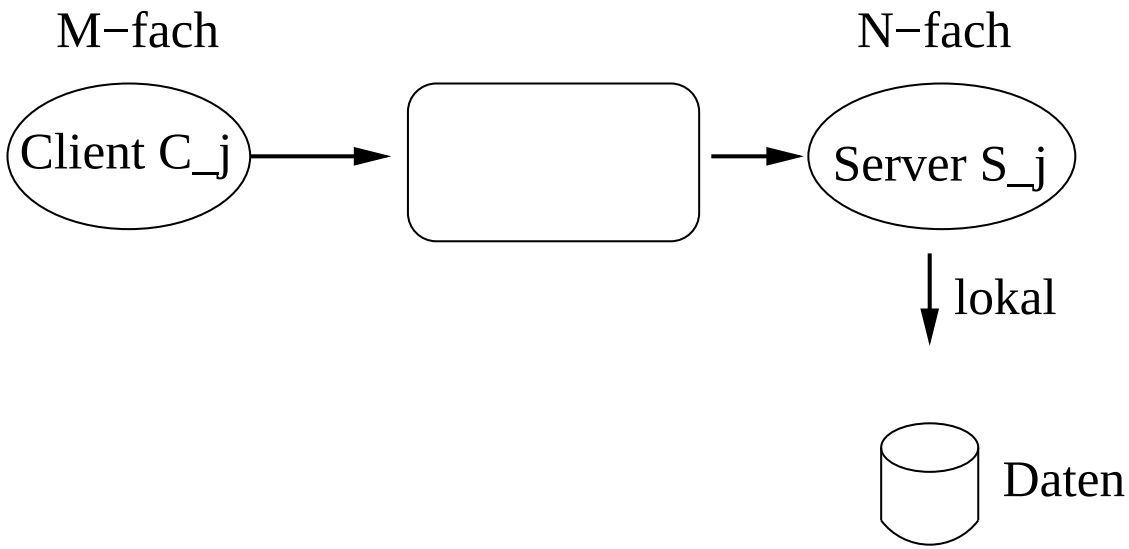
Client-Server Paradigma I

- **Server:** Prozess, der in einer Endlosschleife Anfragen (Aufträge) von Clients bearbeitet.
- **Client:** Stellt in unregelmäßigen Abständen Anfragen an einen Server.

Zum Beispiel waren im Fall der verteilten Philospohen die Philosophen die Clients und die Diener die Server (die auch untereinander kommunizieren).
Praktische Beispiele:

- File Server (NFS: Network File Server)
- Datenbank-Server
- HTML-Server

Weiteres Beispiel: File Server, Conversational Continutity
Über das Netzwerk soll der Zugriff auf Dateien realisiert werden.



Client-Server Paradigma II

- Client: öffnet Datei; macht beliebig viele Lese-/ Schreibzugriffe; schließt Datei.
- Server: bedient genau einen Client, bis dieser die Datei wieder schließt. Wird erst nach Beendigung der Kommunikation wieder frei.
- Allocator: ordnet dem Client einen Server zu.

process C [**int** $i \in \{0, \ldots, M-1\}$]
{
    send(A, OPEN , „foo.txt “);
    recv(A , ok , j);
    send(S_j , READ , *where*);
    recv(S_j , *buf*);
    send(S_j , WRITE , *buf* , *where*);
    recv(S_j , ok);
    send(S_j , CLOSE);
    recv(S_j , ok);
}

Client-Server Paradigma III

```
process A                                   // Allocator
{
    int free [N] = {1[N]} ;                 // alle Server frei
    int cut = 0;                            // wieviel Server belegt?
    while (1) {
        if ( rprobe( who ) ) {                          // von wem kann ich empfangen?
            if ( who ∈ {C_0, ..., C_{M-1}} && cut == N )
                continue;                               // keine Server frei
            recv( who , tag , msg );
            if ( tag == OPEN ){
                Finde freien Server j ;
                free [j] = 0 ;
                cut++;
                send( S_j , tag , msg , who );
                recv( S_j , ok );
                send( who , ok , j );
            }
            if ( tag == CLOSE )
                for ( j ∈ {0, ..., N - 1} )
                    if ( S_j == who ) {
                        free [j] = 1;
                        cut = cut - 1 ;
                    }
        }
    }
}
```

Client-Server Paradigma IV

```
process S [ int j ∈ {0, . . . , N − 1}]
{
    while (1) {
        // warte auf Nachricht von A
        recv( A , tag , msg , C );                    // mein Client
        if ( tag ≠ OPEN ) → error;
        öffne Datei msg
        send( A , ok );
        while (1) {
            recv( C , tag , msg );
            if ( tag == READ ) {
                . . .
                send( C , buf );
            }
            if ( tag == WRITE ) {
                . . .
                send( C , ok ); }
            }
            if ( tag == CLOSE ){
                close file;
                send( C , ok );
                send( A , CLOSE , dummy );
                break;
            }
        }
    }
}
```

Entfernter Prozeduraufruf I

- Wird abgekürzt mit RPC (Remote Procedure Call). Ein Prozess ruft eine Prozedur in einem anderen Prozess auf.
- Π_1:

  ⋮

  y = Square(x);

  ⋮

  Π_2:

  int Square(**int** x)
  {
  return x · x;
  }

- Es gilt dabei:
  - Die Prozesse können auf verschiedenen Prozessoren sein.
  - Der Aufrufer blockiert solange, bis die Ergebnisse da sind.
  - Es findet eine Zweiweg-Kommunikation statt, d.h. Argumente werden hin-/ und Ergebnisse zurückgeschickt. Für das Client-Server Paradigma ist das die ideale Konfiguration.
  - Viele Clients können gleichzeitig eine entfernte Prozedur aufrufen.

Entfernter Prozeduraufruf II

- Wir realisieren den RPC dadurch, dass Prozeduren durch das Schlüsselwort `remote` gekennzeichnet werden. Diese können dann von anderen Prozessen aufgerufen werden.

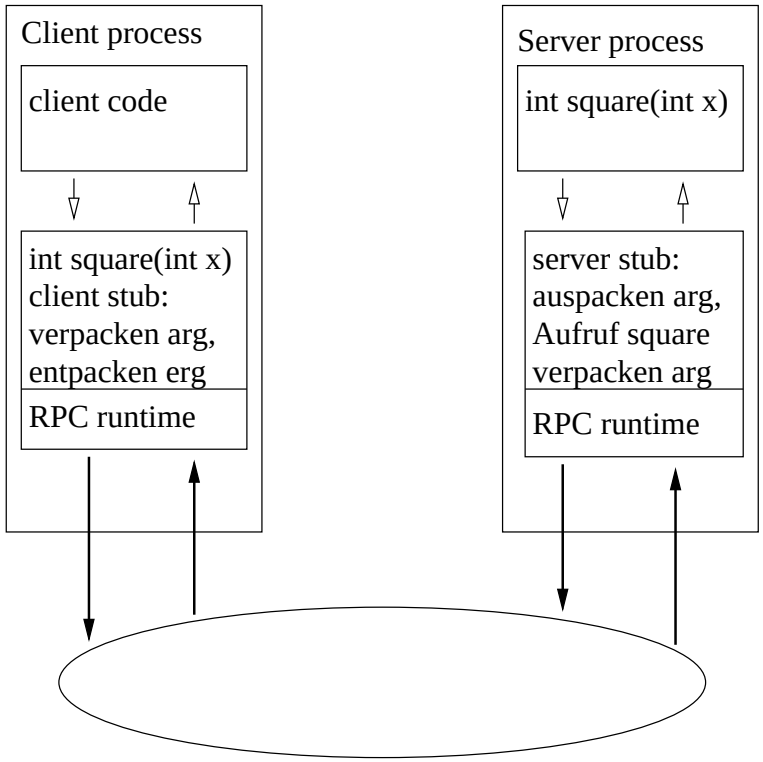
Programm (RPC-Syntax)

```
parallel rpc-example
{
    process Server
    {
        remote int Square(int x)
        {
            return x · x;
        }
        remote long Time ( void )
        {
            return time_of_day;
        }
        . . . Initialisierungscode
    }
    process Client
    {
        y = Server.Square(5);
    }
}
```

Entfernter Prozeduraufruf III

Beim Aufruf einer Funktion in einem anderen Prozess mittels RPC geschieht folgendes:

- Die Argumente werden auf Aufruferseite in eine Nachricht verpackt, über das Netzwerk geschickt und auf der anderen Seite wieder ausgepackt.
- Nun kann die Funktion ganz normal aufgerufen werden.
- Der Rückgabewert der Funktion wird auf dieselbe Weise zum Aufrufer zurückgeschickt.



Entfernter Prozeduraufruf IV

Eine sehr häufig verwendete Implementierung des RPC stammt von der Firma SUN. Die wichtigsten Eigenschaften sind:

- Portabilität (Client/Server Anwendungen auf verschiedenen Architekturen). Dies bedeutet, dass die Argumente und Rückgabewerte in einer rechnerunabhängigen Form über das Netzwerk transportiert werden müssen. Dazu dient die XDR-Bibliothek (external data representation).
- Es sind wenig Kenntnisse über Netzwerkprogrammierung erforderlich.

Wir gehen nun schrittweise durch wie man obiges Beispiel mittels SUN's RPC realisiert.

Client-Server Paradigma mit RPC I

- (1) Erstelle RPC Spezifikation in Datei `square.x`

```
struct square_in {          /* first argument   */
    int arg1;
} ;

struct square_out {         /* return value     */
    int res1;
} ;

program SQUARE_PROG {
    version SQUARE_VERS {  /* procedure number */
        square_out SQUAREPROC(square_in) = 1;
    } = 1;                  /* version number   */
} = 0x31230000 ;            /* program number   */
```

- (2) Übersetzen der Beschreibung mittels

```
rpcgen -C square.x
```

Client-Server Paradigma mit RPC II

generiert folgende 4 Dateien **vollautomatisch**:

`square.h`: Datentypen für Argumente, Prozedurköpfe (Ausschnitt)

```
#define SQUAREPROC 1
extern  square_out * squareproc_1(square_in *, CLIENT *);     /* die ruft Client */
extern  square_out * squareproc_1_svc(square_in *, struct svc_req *); /* Server  */
```

`square_clnt.c`: Client-Seite der Funktion, Verpacken der Argumente

```
#include <memory.h> /* for memset */
#include "square.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

square_out * squareproc_1(square_in *argp, CLIENT *clnt)
{
    static square_out clnt_res;

    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, SQUAREPROC,
        (xdrproc_t) xdr_square_in, (caddr_t) argp,
        (xdrproc_t) xdr_square_out, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}
```

Client-Server Paradigma mit RPC III

`square_svc.c`: Kompletter Server, der auf den Prozeduraufruf reagiert.
`square_xdr.c`: Funktion für Datenkonversion in heterogener Umgebung:

```
#include "square.h"

bool_t xdr_square_in (XDR *xdrs, square_in *objp)
{
    register int32_t *buf;

     if (!xdr_int (xdrs, &objp->arg1))
         return FALSE;
    return TRUE;
}

bool_t xdr_square_out (XDR *xdrs, square_out *objp)
{
    register int32_t *buf;

     if (!xdr_int (xdrs, &objp->res1))
         return FALSE;
    return TRUE;
}
```

Client-Server Paradigma mit RPC IV

- (3) Nun ist der Client zu schreiben, der die Prozedur aufruft. (`client.c`):

```
#include "square.h"   /* includes also rpc/rpc.h */
int main (int argc, char **argv)
{
    CLIENT *cl;
    square_in in;
    square_out *outp; /* can only return a pointer */

    if (argc!=3) {
        printf("usage: client <hostname> <integer-value>\n");
        exit(1);
    }

    cl = clnt_create(argv[1],SQUARE_PROG,SQUARE_VERS,"tcp");
    if (cl==NULL) {
        printf("clnt_create failed\n");
        exit(1);
    }
    in.arg1 = atoi(argv[2]);
    outp = squareproc_1(&in,cl); /* remote procedure call */
    if (outp==NULL) {
        printf("%s",clnt_sperror(cl,argv[1]));
        exit(1);
    }

    printf("%d\n",outp->res1);
    exit(0);
}
```

Client-Server Paradigma mit RPC V

- (4) Nun kann der Client gebaut werden:

```
gcc -g -c client.c
gcc -g -c square_xdr.c
gcc -g -c square_clnt.c
gcc -o client client.o square_xdr.o square_clnt.o
```

- (5) Schlussendlich ist die Funktion auf der Serverseite zu schreiben (`server.c`):

```
square_out * squareproc_1_svc(square_in *inp, struct svc_req *rqstp)
{
    static square_out out; /* weil wir pointer zurueckgeben werden */

    out.res1 = inp->arg1 * inp->arg1;
    return (&out);
}
```

- (6) Nun kann der Server gebaut werden:

```
gcc -g -c server.c
gcc -g -c square_xdr.c
gcc -g -c square_svc.c
gcc -o server server.o square_xdr.o square_svc.o
```

Client-Server Paradigma mit RPC VI

- (7) Laufenlassen geht mittels
  Stelle sicher, dass portmapper läuft: `rpcinfo -p`
  Starte server mittels `server &`
  Starte Client:

```
josh> client troll 123
15129
```

  Per default beantwortet der Server die Anfragen sequentiell nacheinander. Einen multithreaded Server kriegt man so:

- generiere RPC code mittels `rpcgen -C -M ...`
- Mache die Prozeduren reentrant. Trick mit `static` Variable geht dann nicht mehr. Lösung: Gebe Ergebnis in einem call-by-value Parameter zurück.

Client-Server Paradigma: CORBA I

Beispiel arbeitet mit MICO (`http://www.mico.org`), einer an der Uni Frankfurt entwickelten, freien CORBA Implementierung (für C++).

- (1) IDL-Definition der Klasse `account.idl`:

```
interface Account {
    void deposit( in unsigned long amount );
    void withdraw( in unsigned long amount );
    long balance();
};
```

- (2) Automatisches Generieren der Client/Server Klassen

  `idl account.idl`

  generiert die Dateien `account.h` (Klassendefinitionen) und `account.cc` (Implementierung der Client-Seite).

Client-Server Paradigma: CORBA II

- (3) Aufruf auf der Client-Seite: `client.cc`

```
#include <CORBA-SMALL.h>
#include <iostream.h>
#include <fstream.h>
#include "account.h"

int main( int argc, char *argv[] )
{
    // ORB initialization
    CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
    CORBA::BOA_var boa = orb->BOA_init (argc, argv, "mico-local-boa");

    // read stringified object reference
    ifstream in ("account.objid");
    char ref[1000];
    in » ref;
    in.close();

    // client side
    CORBA::Object_var obj = orb->string_to_object(ref);
    assert (!CORBA::is_nil (obj));
    Account_var client = Account::_narrow( obj );

    client->deposit( 100 );
    client->deposit( 100 );
    client->deposit( 100 );
    client->deposit( 100 );
    client->deposit( 100 );
    client->withdraw( 240 );
    client->withdraw( 10 );
    cout « "Balance is " « client->balance() « endl;

    return 0;
}
```

Client-Server Paradigma: CORBA III

- (4) Server enthält Implementierung der Klasse, Erzeugen eines Objektes und den eigentlichen Server: `server.cc`:

```
#define MICO_CONF_IMR
#include <CORBA-SMALL.h>
#include <iostream.h>
#include <fstream.h>
#include <unistd.h>
#include "account.h"

class Account_impl : virtual public Account_skel {
    CORBA::Long _current_balance;
public:
    Account_impl ()
    {
        _current_balance = 0;
    }
    void deposit( CORBA::ULong amount )
    {
        _current_balance += amount;
    }
    void withdraw( CORBA::ULong amount )
    {
        _current_balance -= amount;
    }
    CORBA::Long balance()
    {
        return _current_balance;
    }
};
```

Client-Server Paradigma: CORBA IV

```
int main( int argc, char *argv[] )
{
    cout « "server init" « endl;

    // initialize CORBA
    CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
    CORBA::BOA_var boa = orb->BOA_init (argc, argv, "mico-local-boa");

    // create object, produce global reference
    Account_impl *server = new Account_impl;
    CORBA::String_var ref = orb->object_to_string( server );
    ofstream out ("account.objid");
    out « ref « endl;
    out.close();

    // start server
    boa->impl_is_ready( CORBA::ImplementationDef::_nil() );
    orb->run ();

    CORBA::release( server );
    return 0;
}
```

Client-Server Paradigma: CORBA V

Zum Start wird wieder der Server gestartet: `server &`

Und der Client aufgerufen:

```
josh > client
Balance is 250
josh > client
Balance is 500
josh > client
Balance is 750
```

Object-Naming: Hier über „stringified object reference“. Austausch über gemeinsam lesbare Datei, email, etc. Ist Global eindeutig und enthält IP-Nummer, Serverprozess, Objekt.

Alternativ: Separate naming services.

Advanced MPI

Einige innovative Aspekte von MPI-2

- Dynamische Prozess Erzeugung and Management
- Kommunikatoren: Inter- und Intrakommunikatoren
- MPI und Threads
- Einseitige Kommunikation

MPI-2 Prozess Kontrolle

- MPI-1 spezifiziert weder wie Prozesse angestossen werden noch wie sie eine Kommunikationsinfrastruktur aufbauen
- MPI-2 ermöglicht dynamisches Erzeugen von Prozessen
  - `MPI_Comm_spawn()` startet MPI Prozesse und errichtet eine Kommunikationsinfrastruktur
  - `MPI_Comm_spawn_multiple()` startet binär-verschiedene Programme oder das selbe Programm mit unterschiedlichen Argumenten unter dem selben Kommunikator `MPI_COMM_WORLD`
- MPI benutzt die existierende Gruppenabstraktion um Prozesse zu repräsentieren. Ein (group,rank) Paar identifiziert eindeutig einen Prozess. Ein Prozess determiniert ein eindeutiges (group,rank) Paar, da er Bestandteil mehrerer Gruppen sein darf.
- MPI stellt keine Betriebssystemdienste zur Verfügung, z.B. Starten und Stoppen von Prozessen, und setzt somit implizit die Existenz einer Laufzeitumgebung, in welcher eine MPI-Anwendung abläuft, voraus.
- Die neu erzeugten Kind-Prozesse besitzen ihren eigenen Kommunikator `MPI_COMM_WORLD`. Mit `int MPI_Comm_get_parent(MPI_Comm *parent)` erhalten sie den selben Interkommunikator, welchen die Elternprozessen beim Erzeugen erhalten haben.

MPI-2 Prozeß Kontrolle

Interface um zur Laufzeit neue Prozesse zu erzeugen

- Syntax:
  `int MPI_Comm_spawn( command, argv, maxprocs, info, root, comm, intercomm, errorcodes)`
- `int MPI_Comm_spawn()` ist eine kollektive Funktion. Erst wenn alle Kind-Prozesse `MPI_Init()` gerufen haben wird sie beendet
- Argumente sind folgendermaßen spezifiziert

Argumenttype	Name	Beschreibung
`char *` (IN)	`command`	Name des zu erzeugenden Programms (only root)
`char *` (IN)	`argv`	Argumente für `command` (only root)
`int` (IN)	`maxprocs`	Maximalanzahl zu erzeugender Prozesse
`MPI_Info` (IN)	`info`	Ein Menge von key-value Paaren, welche dem Laufzeitsystem angeben, wo und wie die Prozesse zu erzeugen sind (only root)
`int` (IN)	`root`	Der Rang des Prozeßes in welchem `argv` ausgewertet wird
`MPI_Comm` (IN)	`comm`	Intrakommunikator für erzeugte Prozesse
`MPI_Comm *` (OUT)	`intercomm`	Interkommunikator zwischen ursprünglicher Gruppe und neu-erzeugter Gruppe
`int` (OUT)	`errorcodes[]`	Ein Code pro Prozess

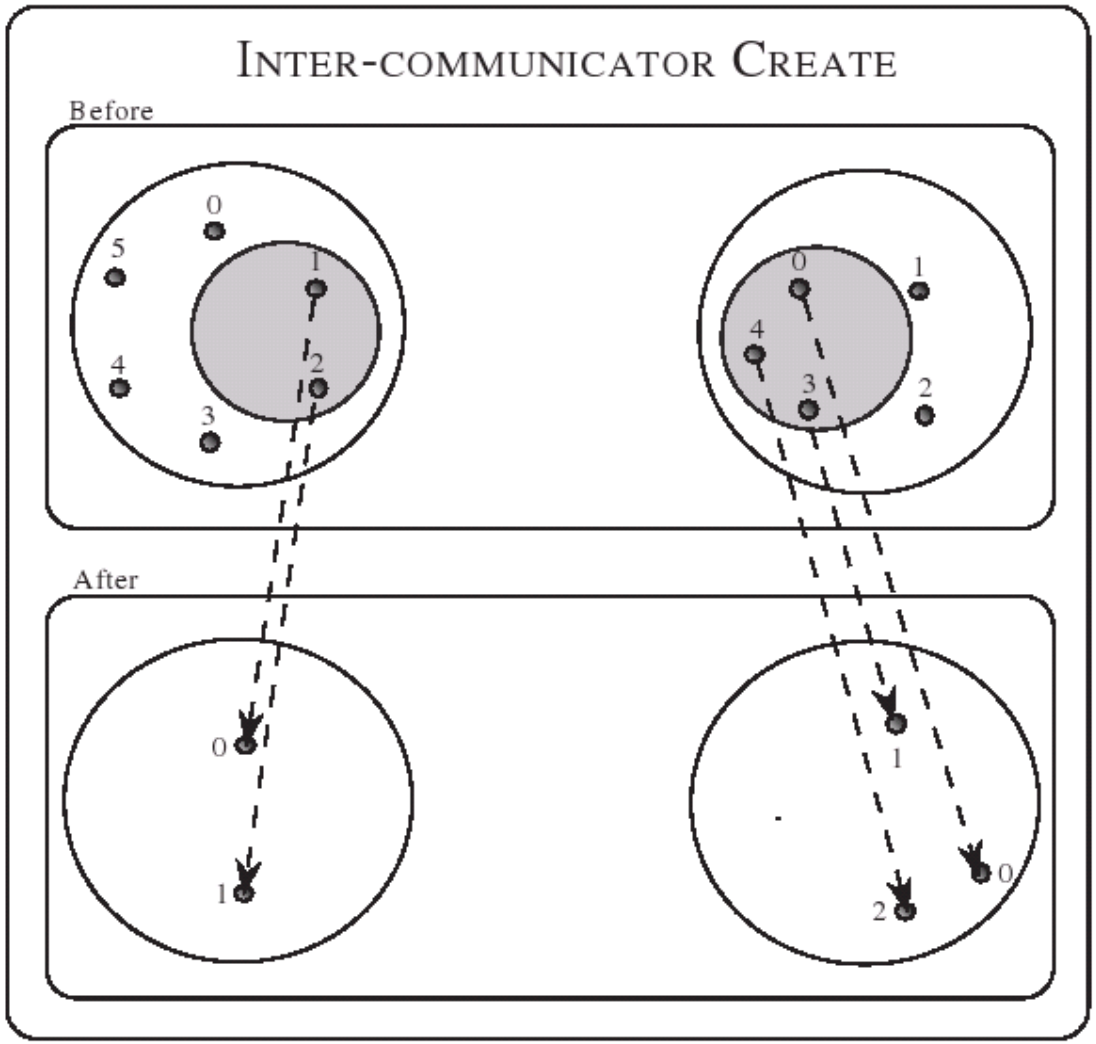
MPI-2 Erweiterte Gemeinsame Kommunikation

- MPI-1: gemeinsame Kommunikationsoperationen für **Intra**kommunikatoren, nur `MPI_Intercomm_create()` und `MPI_Comm_dup()` zum Erzeugen von **Interkommunikatoren**
- MPI-2: Erweiterung vieler MPI-1 Kommunikationsoperationen auf Interkommunikatoren, weitere Möglickeiten um Interkommunikatoren zu erzeugen, 2 neue Routinen für gemeinsame Kommunikation.

Konstruktoren für Interkommunikatoren:

- ```
  MPI::Intercomm MPI::Intercomm::Create(const Group& group) const
  MPI::Intracomm MPI::Intracomm::Create(const Group& group) const
  ```
  ```

MPI-2: Interkommunikator Konstruktion



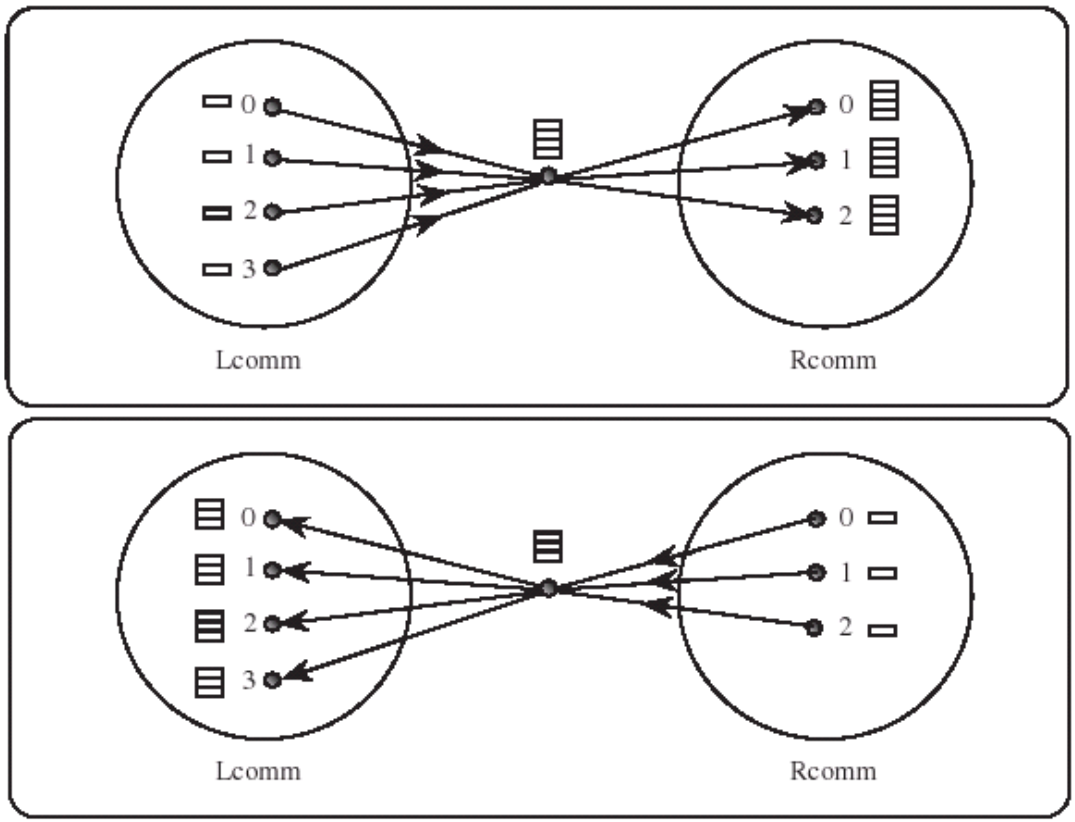
aus MPI-2 Standard Document

MPI-2: Kollektivkommunikation im Interkommunikator

- **All-To-All**
  - `MPI_Allgather, MPI_Allgatherv`
  - `MPI_Alltoall, MPI_Alltoallv`
  - `MPI_Allreduce, MPI_Reduce_scatter`
- **All-To-One**
  - `MPI_Gather, MPI_Gatherv`
  - `MPI_Reduce`
- **One-To-All**
  - `MPI_Bcast`
  - `MPI_Scatter, MPI_Scatterv`
- **Other**
  - `MPI_Scan`
  - `MPI_Barrier`

MPI-2: Kollektivkommunikation im Interkommunikator

- Beschreibung der Operationen mit Quell- und Zielgruppe.
  - bei Intrakommunikatoren sind diese Gruppen identisch
  - bei Interkommunikatoren sind diese Gruppen verschieden
- Nachrichten und Datenfluß bei `MPI_Allgather()`



aus MPI-2 Standard Document

MPI-2: Kollektivkommunikation im Intrakommunikator

Generalisierte Alltoall Funktion (w) (die kennen wir schon!)

- Deklaration:

```
void MPI::Comm::Alltoallw (const void* sendbuf, const int sendcounts[], const int sdispls[], const MPI::Datatype sendtype[], void *recvbuf, const int recvcounts[], const int rdispls[], const MPI::Datatype recvtypes[]) const = 0;
```

- Den j-ten Block den Prozess i schickt speichert Prozess j im i-ten Block von recvbuf.
- Die Blöcke können unterschiedliche Größe besitzen
- Typ Signaturen und Datenumfang müssen stimmen: `sendcounts[j],sendtypes[j]` von Prozess i passt zu `sendcounts[i],sendtypes[i]` von Prozess j
- Keine in-place Option

MPI-2: Kollektivkommunikation im Intrakommunikator

Exklusive Scan Operation, inklusive Scan bereits in MPI-1

- Deklaration:

```
MPI::Intracomm:Exscan (const void* sendbuf, void* recvbuf, int count, const
MPI::Datatype& datatype, const MPI::Op& op) const
```

- führt eine Prefix Reduktion auf Daten durch, welche über die Gruppe verteilt sind
- Wert in `recvbuf` von Prozess 0 ist undefiniert
- Wert in `recvbuf` von Prozess 1 ist als der Wert von `sendbuf` von Prozess 0 definiert
- Wert in `recvbuf` von Prozess i mit $i < 1$ ist der Wert der Reduktionsoperation op angewendet auf die `sendbufs` der Prozesse $0, \ldots, i-1$
- Keine in-place Option

Hybrid Programming: MPI und Threads

Grundsätzliche Annahmen

- Thread Bibliothek nach POSIX Standard
- MPI Prozess kann uneingeschränkt multithreaded ablaufen
- Jeder Thread kann MPI Funktionen aufrufen
- Threads eines MPI Prozesses sind nicht unterscheidbar
  rank spezifiziert einen MPI Prozess nicht Thread
- Der Benutzer muß race conditions verhindern, welche durch wiedersprüchliche Kommunikationsaufrufe entstehen können
  Dies kann z.B. durch thread-spezfifische Kommunikatoren geschehen

Minimale Anforderungen an thread verträgliches MPI

- Alle MPI Aufrufe sind thread sicher, d.h. zwei nebenläufige Threads dürfen MPI Aufrufe absetzen, das Ergebnis ist invariant bzgl. der Aufrufreihenfolge, auch bei zeitlicher Verschränkung der Aufrufe
- Blockierende MPI Aufrufe blockieren nur den rufenden Thread, während weitere Threads aktiv sein können, insbesondere können diese MPI Aufrufe ausführen.
- MPI Aufrufe kann man thread sicher machen indem man zu einem Zeitpunkt nur eine Aufruf ausführt. Dies kann man mit einem MPI Prozess eigenem Lock bewerkstelligen.

Hybrid Programming: MPI und Threads

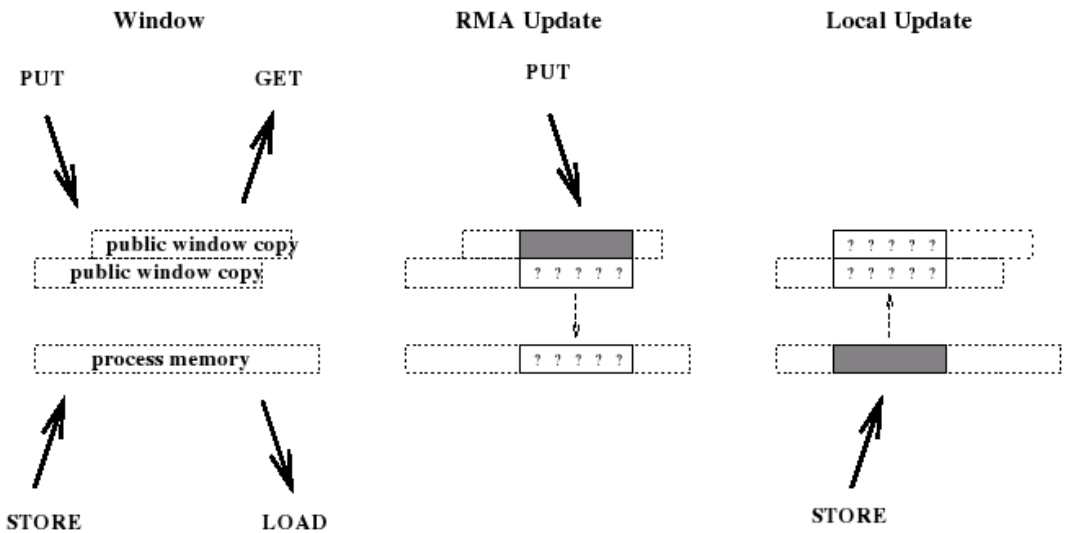
- `MPI_Init()` und `MPI_Finalize()` sollten vom selben Thread, sogenannter Hauptthread, gerufen werden
- Initialisierung von MPI und Thread Umgebung mit

  ```
  int MPI::Init_thread (int& argc, char **& argv, int required)
  ```

  Das Argument `required` spezifiziert den notwendigen Threadlevel
  - `MPI_THREAD_SINGLE`: nur ein Thread wird ausgeführt
  - `MPI_THREAD_FUNNELED`: der Prozess kann multithreaded sein, MPI Aufrufe werden nur vom Hauptthread gemacht
  - `MPI_THREAD_SERIALIZED`: der Prozess kann multithreaded sein und mehrere Threads dürfen MPI Aufrufe ausführen, aber zu einem Zeitpunkt nur einer (also keine Nebenläufigkeit von MPI Aufrufen)
  - `MPI_THREAD_MULTIPLE`: Mehrere Threads dürfen MPI ohne Einschränkungen aufrufen
- Der Benutzer hat die Korrespondenz von MPI Kollektivoperationen auf einen Kommunikator mittels Interthread Synchronisation sicherzustellen
- Es ist nicht garantiert, daß die Ausnahmenbehandlung vom gleichen Thread bewerkstelligt wird, welcher den MPI Aufruf, der die Ausnahmebehandlung verursacht hat, ausgeführt hat.
- Abfrage des aktuellen Threadlevels mit `int MPI::Query_thread()`
  Feststellung ob Hauptthread `bool MPI::Is_thread_main()`

MPI-2 Einseitige Kommunikation

- Einseitige Kommunikation ist eine Erweiterung des Kommunikationsmechanismus um Remote Memory Access (RMA)
- drei Kommunikationsaufrufe:
  `MPI_Put()`, `MPI_Get()` und `MPI_Accumulate()`
- verschiedene Synchronisationsaufrufe: Fence, Wait, Lock/Unlock
- Vorteil: Nutzung von Architekturmerkmalen (gemeinsamer Speicher, hardware unterstützte put/get Operationen, DMA Engines)
- Initialisierung eines Speicherfensters
- Verwaltung mittels opaquem Objekt zur Speicherung der zugriffsberechtigten Prozessgruppe und der Fensterattribute
  `MPI::Win MPI::Win::Create()` und `void MPI::Win::Free()`



Literatur

- Andrews, G. R.: Concurrent Programming - Principles and Practice, Benjamin/Cummings, 1991
- MPI: A Message-Passing Interface Standard, MPI-1.1, Message Passing Interface Forum, 1995
- MPI-2: Extensions to the Message-Passing Interface, Message Passing Interface Forum, 1997
- Grama A., Gupta A., Karypis G., Kumar V., Introduction to Parallel Computing, Benjamin/Cummings, 1994