Distributed-Memory Programming Models IV

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 368, Room 532
D-69120 Heidelberg
phone: 06221/54-8264
email: Stefan.Lang@iwr.uni-heidelberg.de

WS 14/15

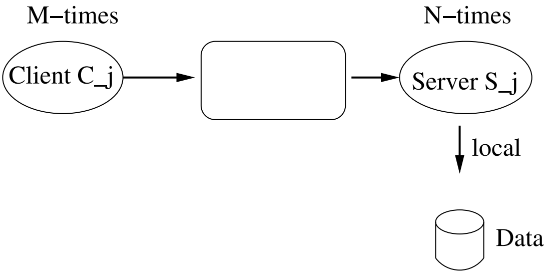
Client-Server Paradigm I

- **Server:** Process, that processes in an endless loop requests (tasks) of clients.
- **Client:** Sends in irregular distances requests to a server.

For example the distributed philosophers have been the clients and the servants the servers (that communicate beneath each other).
Practical Examples:

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

Further Example: File Server, Conversational Continuity
Access onto files shall be realized over the network.



Client-Server Paradigm II

- Client: opens file; performs an arbitrary number of read/write accesses; closes file.
- Server: serves exactly one client, until this closes the file again. Will be releases after finalising the communication.
- Allocator: maps a client to a server.

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 Paradigm III

```
process A                                   // Allocator
{
    int free [N] = {1[N]} ;                 // all servers free
    int cut = 0;                            // how many servers occupied?
    while (1) {
        if ( rprobe( who ) ) {                          // from whom may I receive?
            if ( who ∈ {C_0, ..., C_{M-1}} & & cut == N )
                continue;                                       // no servers free
            recv( who , tag , msg );
            if ( tag == OPEN ){
                Find free 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 Paradigm IV

```
process S [ int j ∈ {0, . . . , N − 1}]
{
    while (1) {
        // wait for message of A
        recv( A , tag , msg , C );              // my client
        if ( tag ≠ OPEN ) → error;
        open file 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;
            }
        }
    }
}
```

Remote Procedure Call I

- Is abbreviated with RPC (Remote Procedure Call). A process calls a procedure/function of another process.
- Π_1:

```
⋮
y = Square(x);
⋮
```

Π_2:

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

- It applies thereby:
  - The processes can run on distinct (remote) processors.
  - The caller blocks as long as the results have not arrived.
  - A two-way communication is established, this means arguments are sent forth and results are sent back. For the client-server paradigm this is the ideal configuration.
  - Many clients can call a remote procedure at a time.

Remote Procedure Call II

- We realise the RPC by assigning the key word `remote` to the procedure of interest. These can then be called by other processes.

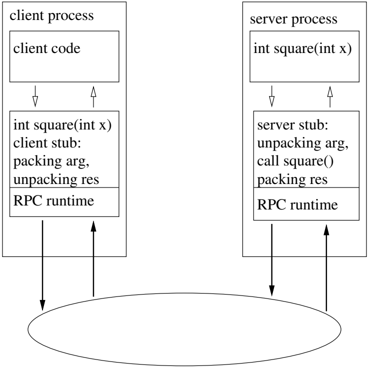
Program (RPC-Syntax)

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

Remote Procedure Call III

During a call of a function in another process via RPC the following happens:

- The arguments are packed on the caller side into a message, sent across the network and unpacked on the other side.
- Now the function can be called completely normal.
- The return value of the function is sent back to the caller in the same kind.



Remote Procedure Call IV

A quite frequently used implementation of RPC comes from the company SUN. The most important properties are:

- Portability (client/server applications on different architectures). This means, that the arguments and return values have to be transported in a architecture-independent representation over the network. This is performed by the `XDR` library (external data representation).
- Few knowledge about network programming is necessary.

We now realize step by step the example from above via SUN's RPC.

Client-Server Paradigm with RPC I

- (1) Construct a RPC specification in file `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) Compile the description with the command

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

Client-Server Paradigm with RPC II

generates the following 4 files in a **completely automatic** way:

`square.h`: data types for arguments, procedure heads (cutout)

```
#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 side of the function, packing of arguments

```
#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 Paradigm with RPC III

square_svc.c: Complete server, that reacts on the procedure call.
square_xdr.c: Function for data conversion in a heterogeneous environment:

```
#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 Paradigm with RPC IV

- (3) Now the client needs to be written, that calls the procedure. (`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 Paradigm with RPC V

- (4) Now the client can be build:

```
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) Finally the function on the server side has to be written (`server.c`):

```
square_out * squareproc_1_svc(square_in *inp, struct svc_req *rqstp)
{
    static square_out out; /* since we return pointers */

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

- (6) Now the server can be build:

```
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 Paradigm with RPC VI

- (7) Starting of the processes works as follows:
  Test, whether the portmapper runs: `rpcinfo -p`
  Start server via `server &`
  Start client:

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

  By default the server answers the request sequentially after each other. A multi-threaded server is created as follows:

- generate RPC code via `rpcgen -C -M ...`
- make the procedures reentrant. Trick with `static` variables does not work anymore. Solution: Pass the result back in a call-by-value parameter.

Client-Server Paradigm: CORBA I

Example works with MICO (`http://www.mico.org`), an free CORBA implementation (C++), that has been developed at the university of Frankfurt.

- (1) IDL definition of the class `account.idl`:

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

- (2) Automatic generation of client/server classes

```
idl account.idl
```

  generates the files `account.h` (class definitions) and `account.cc` (implementation of the client side).

Client-Server Paradigm: CORBA II

- (3) Call of the client side: `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 Paradigm: CORBA III

- (4) Server contains the implementation of the class, generates the objects and the server itself: `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 Paradigm: 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 Paradigm: CORBA V

To start the server is run again: `server &`

And the client is called:

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

Object naming: Here over a „stringified object reference“. Exchange over shared readable file, email, etc. Is global unique and contains IP numbers, server process, object.

Alternatively: Separate naming services.

Advanced MPI

Some innovative aspects of MPI-2

- Dynamic process creation and management
- Communicators: Inter- and Intracommunicators
- MPI and Threads
- One-sided communication

MPI-2 Process Control

- MPI-1 specifies neither how the processes are spawned nor how they create a communication infrastructure
- MPI-2 enables dynamic creation of processes
  - `MPI_Comm_spawn()` starts MPI processes and creates a communication infrastructure
  - `MPI_Comm_spawn_multiple()` starts binary-distinct programs or the same program with different arguments below the same communicator `MPI_COMM_WORLD`
- MPI uses the existing group abstractions to represent processes. A (group,rank) pair identifies a process in a unique way. A process determines a unqiue (group,rank) pair, since it may be part of several groups.
- MPI does not provide any operating system services, e.g. starting and stopping of processes , and therefore implies implicitly the existence of a runtime environment, within which a MPI-application can run.
- The newly created child processes possess their own communicator `MPI_COMM_WORLD`. With `int MPI_Comm_get_parent(MPI_Comm *parent)` you receive the same intercommunicator, that the parent processes have received during their creation.

MPI-2 Process Control

Interface to create new processes during runtime

- Syntax:
  `int MPI_Comm_spawn( command, argv, maxprocs, info, root, comm, intercomm, errorcodes)`
- `int MPI_Comm_spawn()` is a collective function. First if all child processes have called `MPI_Init()` it is finished.
- Arguments are specified in the following:

argument type	name	description
`char *` (IN)	`command`	name of the program to be created (only root)
`char *` (IN)	`argv`	arguments for `command` (only root)
`int` (IN)	`maxprocs`	maximal count of processes to be created
`MPI_Info` (IN)	`info`	a set of key-value pairs, that provides the runtime system info, where and how the processes are to be created (only root)
`int` (IN)	`root`	the rank of the process in which `argv` is evaluated
`MPI_Comm` (IN)	`comm`	Intracommunicator for generated processes
`MPI_Comm *` (OUT)	`intercomm`	Intercommunicator between original group and newly generated group
`int` (OUT)	`errorcodes[]`	A code per process

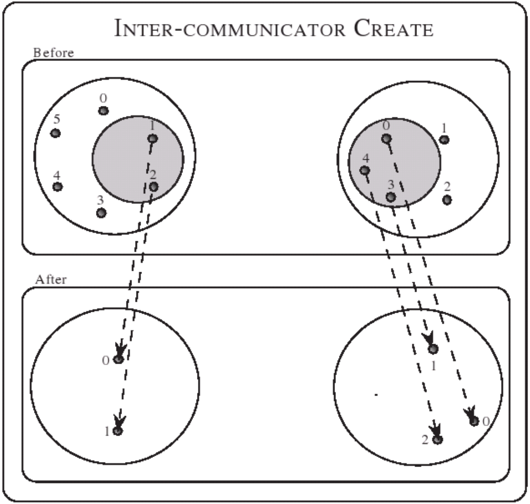
MPI-2 Enhanced Shared Communication

- MPI-1: shared communication operations for **intra**communicators, only `MPI_Intercomm_create()` and `MPI_Comm_dup()` to create **intercommunicators**
- MPI-2: extension of many MPI-1 communication operations to intercommunicators, further possibilites to create intercommunicators, 2 new routines for shared communication.

constructors for intercommunicators:

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

MPI-2: Intercommunicator Construction



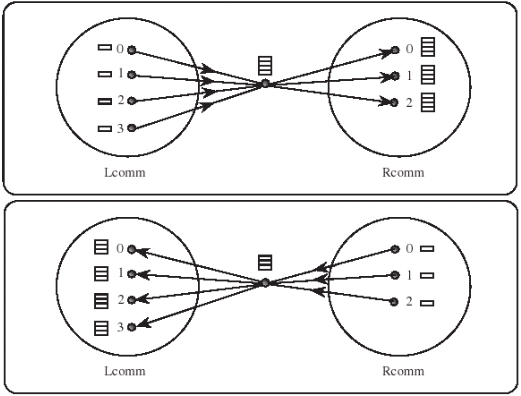
from MPI-2 standard document

MPI-2: Collective Communication inside Intercommunicator

- **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: Collective Communication in Intercommunicator

- Description of operations with source and target group.
  - within intracommunicators these groups are identical
  - within intercommunicators these groups are distinct
- Messages and data flow within `MPI_Allgather()`



from MPI-2 standard document

MPI-2: Collective Communication in the Intracommunicator

Generalised Alltoall function (w) (we already known this one!)

- Declaration:

```
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;
```

- The j-th block that sends process i is stored by process j in the i-th block of recvbuf.
- The blocks can have different size
- Type signatures and data extend have to be consistent: `sendcounts[j],sendtypes[j]` of process i fits to `sendcounts[i],sendtypes[i]` of prozess j
- No in-place option

MPI-2: Collective Communication in the Intracommunicator

Exclusive scan operation, inclusive scan already in MPI-1

- Declaration:

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

- Performs a prefix reduction on data, that are distributed across the group
- Value in `recvbuf` of process 0 is undefined
- Value in `recvbuf` of process 1 is defined by the value of `sendbuf` of process 0
- Value in `recvbuf` of process *i* with $i < 1$ is the value of reduction operation `op` applied to the `sendbuf`s of processes $0, \dots, i-1$
- no in-place option

Hybrid Programming: MPI and Threads I

Basic Assumptions

- Thread library according to POSIX standard
- MPI process can be run multithreaded without limitations
- Each thread can call MPI functions
- Threads of an MPI process can not be distinguished
  rank specifies a MPI process not thread
- The user has to avoid conditions, that can be generated by contradictionary communication calls
  This can e.g. occur by thread specific communicators

Minimal requirements for thread-aware MPI

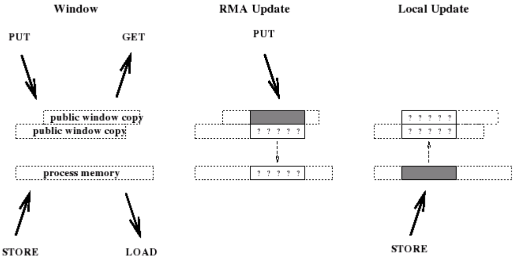
- All MPI calls are thread save, this means two concurrent threads may execute MPI calls, the result is invariant concerning the call sequence, also by interleaving of the calls in time
- Blocking MPI calls block only the calling thread, while further threads can be active, especially these may execute MPI calls.
- MPI calls can be made thread save when one only executes one call at a time. This can be performed with one MPI process with individual lock.

Hybrid Programming: MPI and Threads II

- `MPI_Init()` and `MPI_Finalize()` should be called by the same thread, so called main thread
- Initialisation of MPI and thread environment with
  `int MPI::Init_thread (int& argc, char **& argv, int required)`
  The argument `required` specifies a necessary thread level
  - `MPI_THREAD_SINGLE`: only a thread will be executed
  - `MPI_THREAD_FUNNELED`: the process can be multi-threaded, MPI calls are performed only by the main thread
  - `MPI_THREAD_SERIALIZED`: the process can be multi-threaded and several threads may execute MPI calls, but at each point in time only one (thus no concurrency of MPI calls)
  - `MPI_THREAD_MULTIPLE`: Several threads may call MPI without constraints
- The user has to ensure the correspondence of MPI collective operations on a communicator via interthread synchronisation
- It is not guaranteed, that the exception handling is done by the same thread, that has executed the MPI call causing the exception.
- Request of the current thread level with `int MPI::Query_thread()` determination whether main thread `bool MPI::Is_thread_main()`

MPI-2 One-sided Communication

- One-sided communication is an extension of communication mechanism by Remote Memory Access (RMA)
- Three communication calls:
  `MPI_Put()`, `MPI_Get()` and `MPI_Accumulate()`
- Different synchronization calls: Fence, Wait, Lock/Unlock
- Advantage: Usage of architecture characteristics (shared memory, hardware supported put/get operations, DMA engines)
- Initialisation of memory window
- Management via opaque object for storage of process group, that has access, and of window attributes
  `MPI::Win MPI::Win::Create()` and `void MPI::Win::Free()`



Literature

- 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