Shared Memory & Message Passing Programming on SCI-Connected Clusters

Joachim Worringen, RWTH Aachen

SCI Summer School 2000
Trinitiy College Dublin

Agenda

- How to utilize SCI-Connected Clusters
- SMI Library
  - We have SISCI & Smile – why SMI?
  - SMI Programming Paradigma
  - SMI Functionality
- SCI-MPICH – an Example for using SMI
  - Design of SCI-MPICH
  - Special features of SCI-MPICH

Usage of Clusters

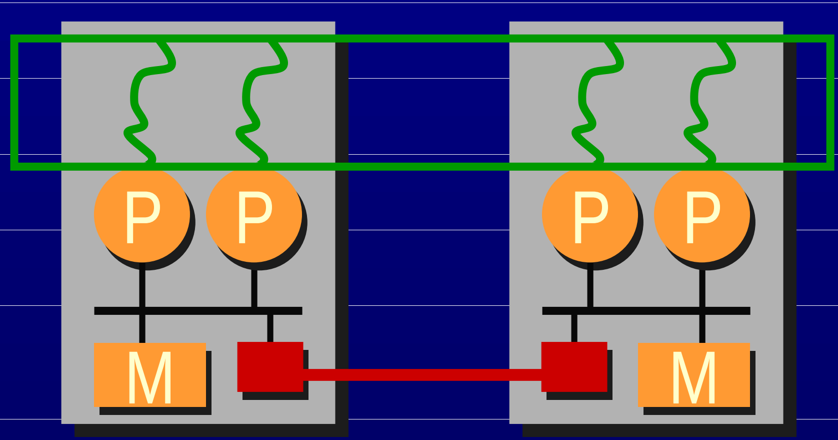
- All Clusters:
  - More throughput
  - Increased redundancy
  - Higher application performance
  - ⇨ Communication via OS services (TCP/IP)
- SCI-Connected Clusters:
  - SCI not offered as a standard OS service
  - I/O backend server
  - Make best use of high-speed interconnect: parallel multi-process applications

Why SMI?

- Higher abstraction level than SISCI:
  - Providing application environment
  - Single function calls for complex operattions
  - Hiding of node & segment IDs
  - Extension of SISCI functionality
  - Resource management
- Lower abstraction level than SMILE:
  - Utilization of multiple PCI-SCI adapters
  - Utilization of DMA & Interrupts
  - Full control of memory layout

SMI Programming Paradigma

- Basic Model: **SPMD**
- Independent processes form an application
- Processes share explicitly created *Shared Memory Regions*
- Multiple *processes* on each *node*



SMI Functionality

- Set of ~70 API functions, **but:**

  only 3 function calls to create an application with shared memory
- Collective vs. individual functions:
  - Collective: all processes must call to complete
  - Individual: process-local completion
- Some (intended) similarities to MPI
- C/C++ and Fortran 77 bindings
- Shared library for Solaris, Linux, Windows

SMI Availability

- SMP systems
- NUMA systems (SCI-Clusters)

Hardware platforms:

- Sparc, Intel, Alpha (soon)

Software platforms:

- Solaris, Linux, Windows NT/2000
- Uses threads, is partly threadsafe
- static or shared library

Initialization/Shutdown

Initialization: collective call to

`SMI_Init(int *argc, char ***argv)`

- ⇨ Passing references to argc and argv to SMI
- ⇨ Do not touch argc/argv before `SMI_Init()`!

Finalization: collective call to

`SMI_Finalize()`

Abort: individual call to

`SMI_Abort(int error)`

- ⇨ Implicitely frees all resources allocated by SMI

Information Gathering

- Topology information:
  - Number of processes: `SMI_Proc_size (int *sz)`
  - Local process rank: `SMI_Proc_rank (int *rank)`
  - Number of nodes: `SMI_Node_size (int *sz)`
  - Several more topology functions
- System/State information:
  `SMI_Query(smi_query_t q, int arg, void *result)`
  - SCI, SMI and system related information
- Timing functions:
  `SMI_Wtime(), ...`

Watchdog

- Observation of „heartbeat“ signals of all processes of an application
- Missing signal for a certain period indicates defunct process
  - ⇨ Termination of the whole application
  - ⇨ Freeing of all resources allocated via SMI
- Watchdog hinders debugging
  - ⇨ Turn off watchdog via SMI_Watchdog() or command line option on startup

Shared Memory Regions

- Inter-process communication is done solely via shared memory
  - ⇨ shared memory regions are always required
- Significant difference in access latency between local and remote memory
  - ⇨ Consider data locality
  - ⇨ Different type of shared memory regions
- Passing pointers between processes makes things easier

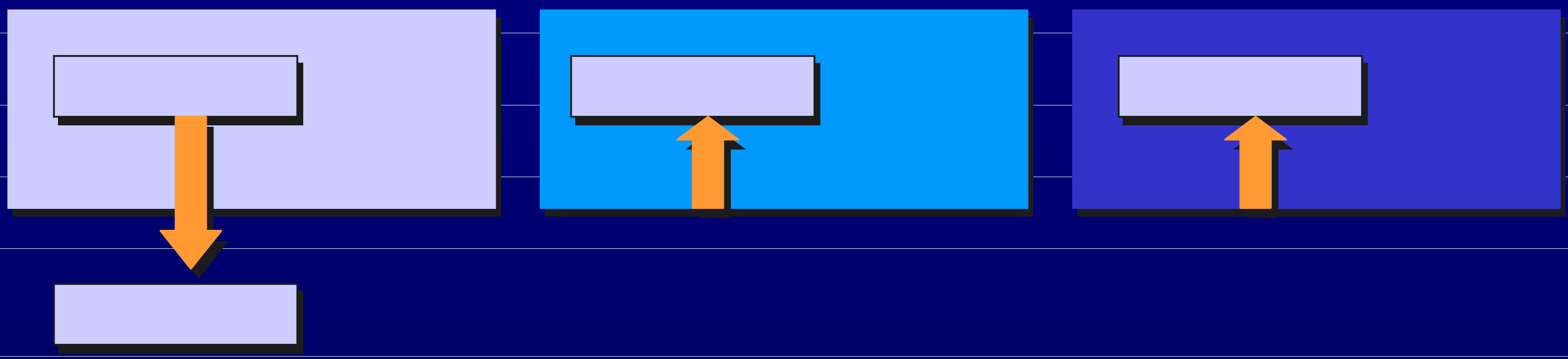
Setting up SHM Regions

- Creating a shared memory region:
```
SMI_Create_shreg(int type, smi_shreg_info_t
                 *reginfo, int *id, char **addr)
```
- Shared region information:
  – **Size** of the shared memory region
  – **PCI-SCI adapter** to use

  Information specific to some region types:
  – **Owner** of the region: memory is local to the owner
  – **Custom distribution** information
  – **Remote Segment** information

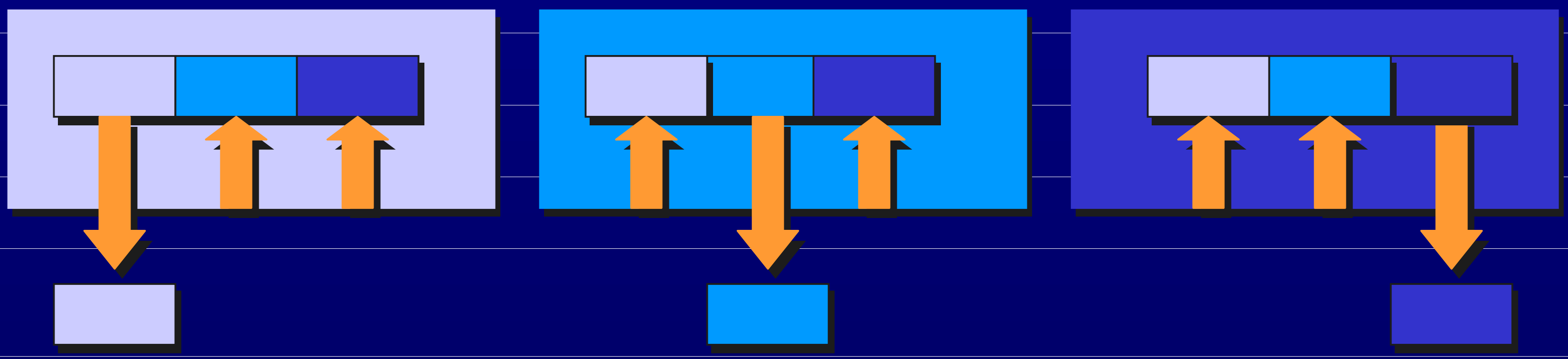
SHM-Type UNDIVIDED

- Basic Region Type:
  one process (**owner**) exports a segment, all others import it.
- FIXED or NON_FIXED, DELAYED
- Collective invocation



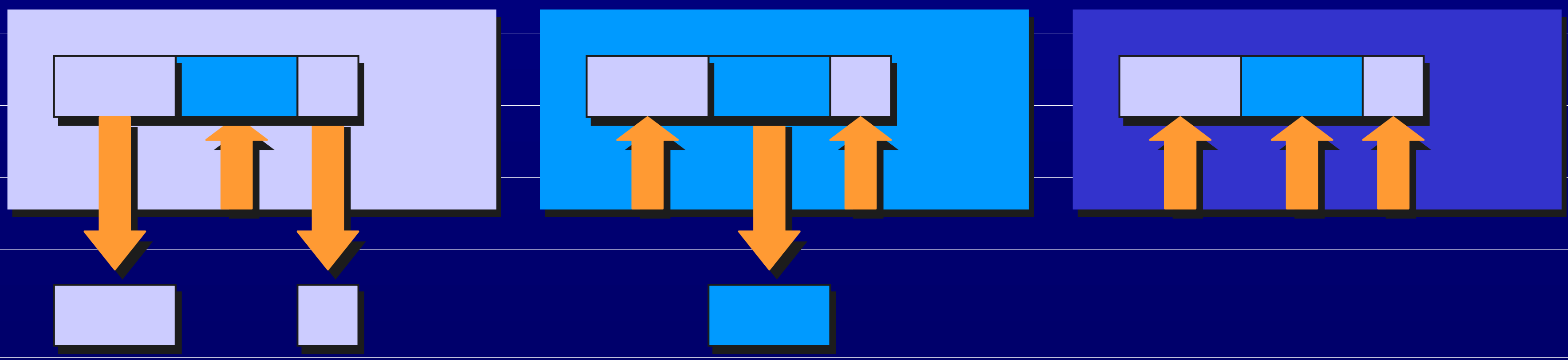
SHM-Type BLOCKED

- Each process exports one segment
- All segment get concanated
  - ⇨ Continous region is created
- Only FIXED, not DELAYED
- Collective invocation



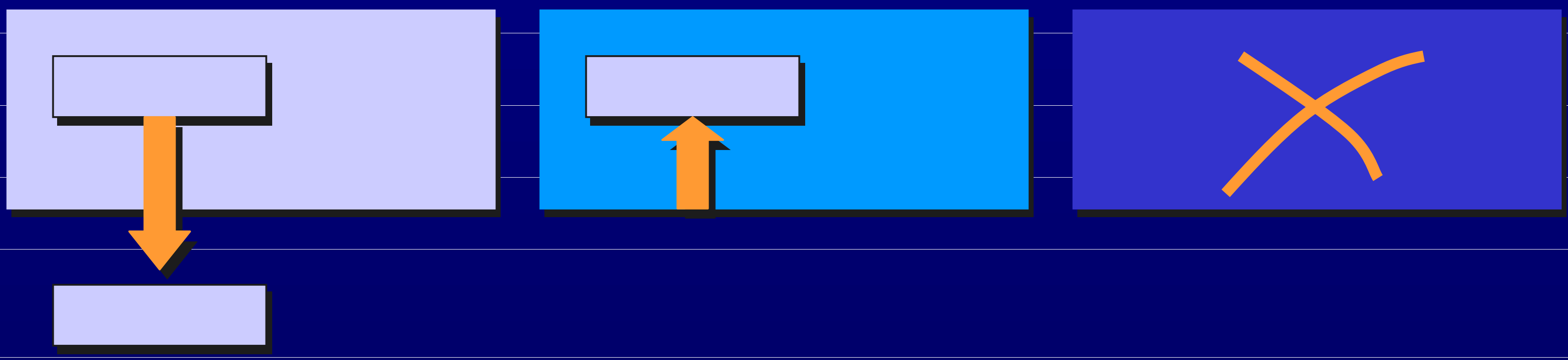
SHM-Type CUSTOMIZED

- User-defined distribution of segments
- All segments get concanated
  - ⇨ Continous region is created
- Only FIXED, not DELAYED
- Collective invocation



SHM-Type PT2PT

- Two processes share a memory segment
- FIXED or NON_FIXED, DELAYED
- Non-collective, but bi-lateral invocation



SHM-Type LOCAL

- A single process exports a segment
- No other process is involved
  - ⇨ Local completion semantics
- Segment is available for connections
- Only NONFIXED

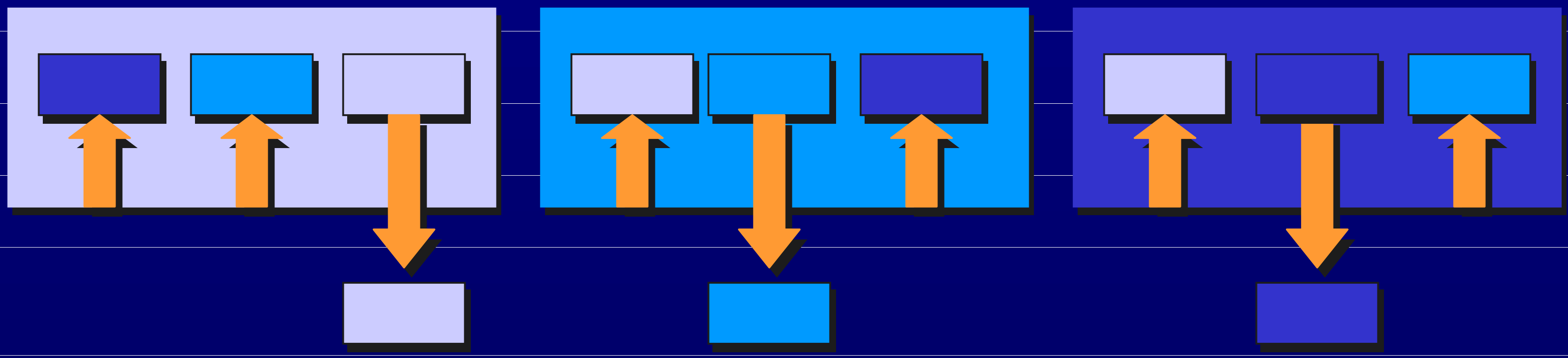
SHM-Type REMOTE

- A single process imports an existing remote segment
- No other process is involved
  - ⇨ Local completion semantics
- Only NONFIXED

Remote process needs information!

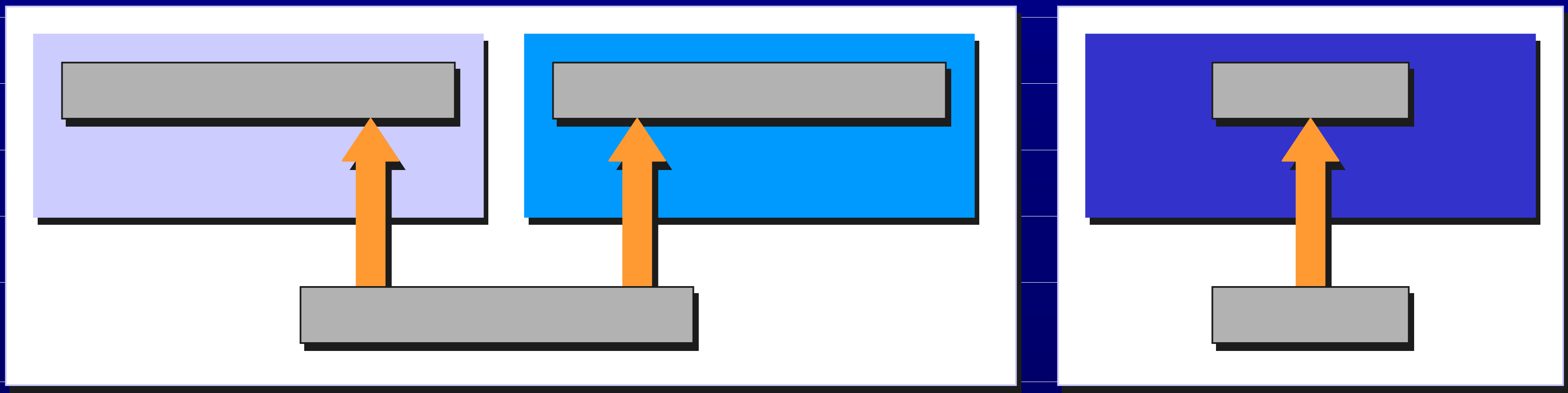
SHM-Type FRAGMENTED

- All processes export a segment and import all other segments
- Segments **do not** get concanated
  - ⇨ Non-contignous region is created
- Faster than creating *n* UNDIVIDED regions



SHM-Type SMP

- Create node-local shared regions
  - different memory backing on each node
  - different sizes possible on different nodes
- No remote memory access
- Collective operation



SHM-Region Flags

- Do not enforce identical addresses:
  `SMI_SHM_NONFIXED`
- Do not connect immeadeletly:
  `SMI_SHM_DELAYED`
- Register user memory as SCI segment:
  `SMI_SHM_REGISTER`
- Keep the segment private (no export):
  `SMI_SHM_PRIVATE`

Connecting to SHM Regions

- Why create regions with DELAYED flag?
  - Faster creation
  - Saving of resources if segment is not needed
- Determine connection state:
  `SMI_Query(SMI_Q_SMI_REGION_CONNECTED)`
- Connect to a region:
  `SMI_Connect_shreg(int id, char **addr)`
- The owner of a region is always connected
- Connecting does not do any harm

Deleting SHM Regions

- Delete a shared memory region:
  `SMI_Free_shreg (int id)`

- All processes who have created/connected to the region need to participate
- Access to a region after it has been free‘d
  ⇨ SIGSEGV

Memory Management

- Dynamic allocation of memory of shared regions (for any contignous region type)
- Region can be used directly ***or*** via SMI memory manager – not both!
- Initialize *Memory Management Unit*:
  `SMI_Init_shregMMU(int region_id)`
- Memory manager works with „buddy“ technique
  ⇨ Fast, but coarse granularity

Memory Allocation

- Individual allocation:
  `SMI_Imalloc(int size, int id, char **addr)`
- Collective allocation:
  `SMI_Cmalloc(int size, int id, char **addr)`
- Freeing allocated memory:
  `SMI_Ifree(char *addr)`
  `SMI_Cfree(char *addr)`
- Freeing mode must match allocation mode!

Memory Transfers

- Memory transfers possible via load/store operations or memcpy()
  - ⇨ why SMI functionality to copy memory?
  - **secure**: including sequence check & store barrier
  - **optimized**: twice the performance
  - **asynchronous**: no CPU utilization

- Synchronous copying:

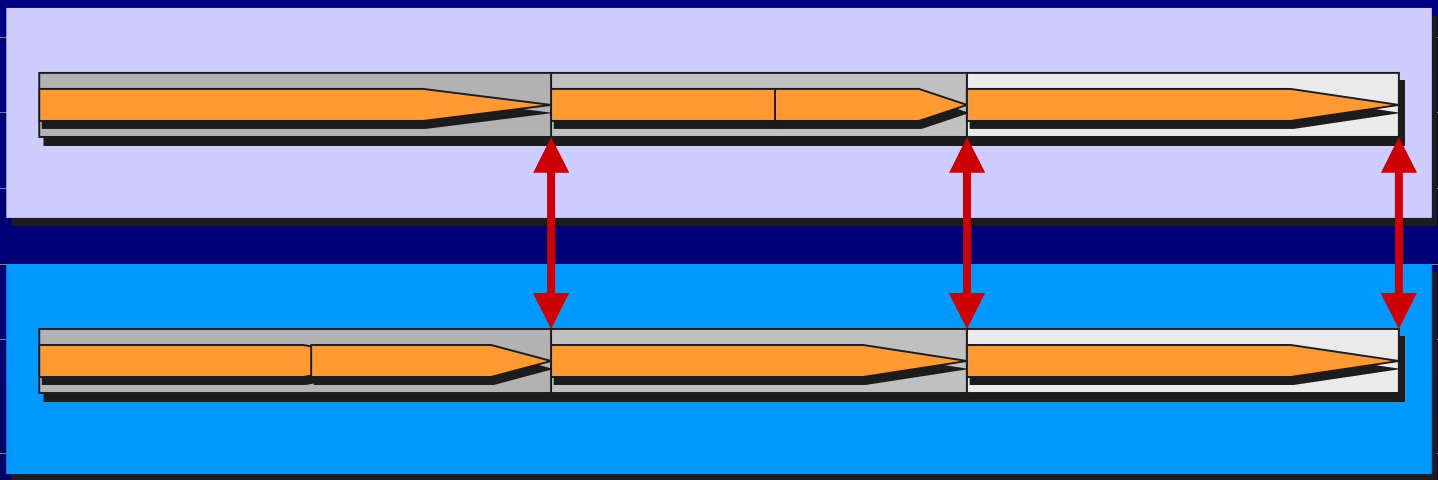
```
SMI_Memcpy(void *dst, void *src,
                                int len, int flags)
```

Synchronization

- Barrier Synchronization:
  `SMI_Barrier()`
- Mutual exclusion via **locks**:
  – Initialization:
  ```
  SMI_Mutex_init (int *id)
  SMI_Mutex_init_with_locality (int *id, int prank)
  ```
  – Acquisition:
  ```
  SMI_Mutex_lock (int id)
  SMI_Mutex_trylock (int id)
  SMI_Mutex_unlock (int id)
  ```
  - Destruction:
  ```
  SMI_Mutex_destroy (int id)
  ```

Progress Counters

- Each process has an atomic counter
- Use other processes‘ counter to synchronize
- Collective or non-collective
- Easier to use than locks and barriers



Progress Counters (cont.)

- Initialization / Reset:

```
SMI_Init_PC (int *pc_id)
SMI_Reset_PC (int pc_id)
```

- Incrementing Counter:

```
SMI_Increment_PC (int pc_id, int val)
```

- Reading / Waiting Counter:

```
SMI_Get_PC (int pc_id, int rank, int *val)

SMI_Wait_individual_PC (int pc_id, int rank,
                                    int val)
SMI_Wait_collective_PC (int pc_id, int val)
```

Signalization

- Wait for events (signal) from other processes:
  - wait for signal from specific process:
    `SMI_Signal_wait (int proc_rank)`
  - wait for signal from any process
    `SMI_Signal_wait (SMI_SIGNAL_ANY)`
  - waiting for a signal does not cost CPU cylces
    - ⇨threads can block for a signal
- Trigger an event:

```
SMI_Signal_send (int proc_rank)
SMI_Signal_send (SMI_SIGNAL_BCAST)
```

Callback Functions

- Set up a callback function

```
SMI_Signal_setCallback (int proc_rank,
         void (*cb_fcn)(void *), void *cb_arg,
         smi_signal_handle *sh)
```

  – `SMI_SIGNAL_ANY` can be used here, too

- Wait for completion of callback function:

```
SMI_Signal_joinCallback (smi_signal_handle *sh)
```

  – Joining does not cost CPU cycles

- Current implementation uses threads; SISCI callbacks will be used when available

„Message Passing“

- SMI is no message passing library
- BUT: minimized inter-process message exchange mechanisms
  - useful i.e. for LOCAL/REMOTE region setup
  - Message size limited to SMI_MP_MAXDATA
  - Blocking or non-blocking message transfer

```
SMI_Send (void *buf, int len, int dest)
SMI_Recv (void *buf, int len, int src)
SMI_Isend (void *buf, int len, int dest)
SMI_Send_wait (int dest)
```

Misc

Functionality of SMI not covered today:

- Load balancing:
  - Static loop splitting
  - Dynamic loop scheduling
- Different consistency modes:
  - Replication of a shared region
  - Different techniques to share a replicated region

Summary SMI

- Development started in 1996:
  - SBus-SCI adapters in Sun Sparcstation 20
  - no SISCI available
  - make SCI usage/NUMA programming less painful
- Marcus Dormanns until end of 1998:
  - API for creation of parallel applications on shared memory (SMP/NUMA/cc-NUMA) platforms
  - Ph.D. thesis: *Grid based parallelization techniques*
- Joachim Worringen since 1998:
  - extension of SMI as basis for other libraries or services on SCI-SMP-clusters

SCI-MPICH

- MPI-1 implementation for SCI-connected clusters
- Part of the MP-MPICH project:
  - NT, Solaris x86/Sparc, Linux x86 (soon Alpha)
  - Communication via Sockets, shared memory, SCI
  - ⇨ Heterogenous usage:
    mixed platforms, mixed interconnects
- Based on the MPICH implementation
- Open-source, freely available

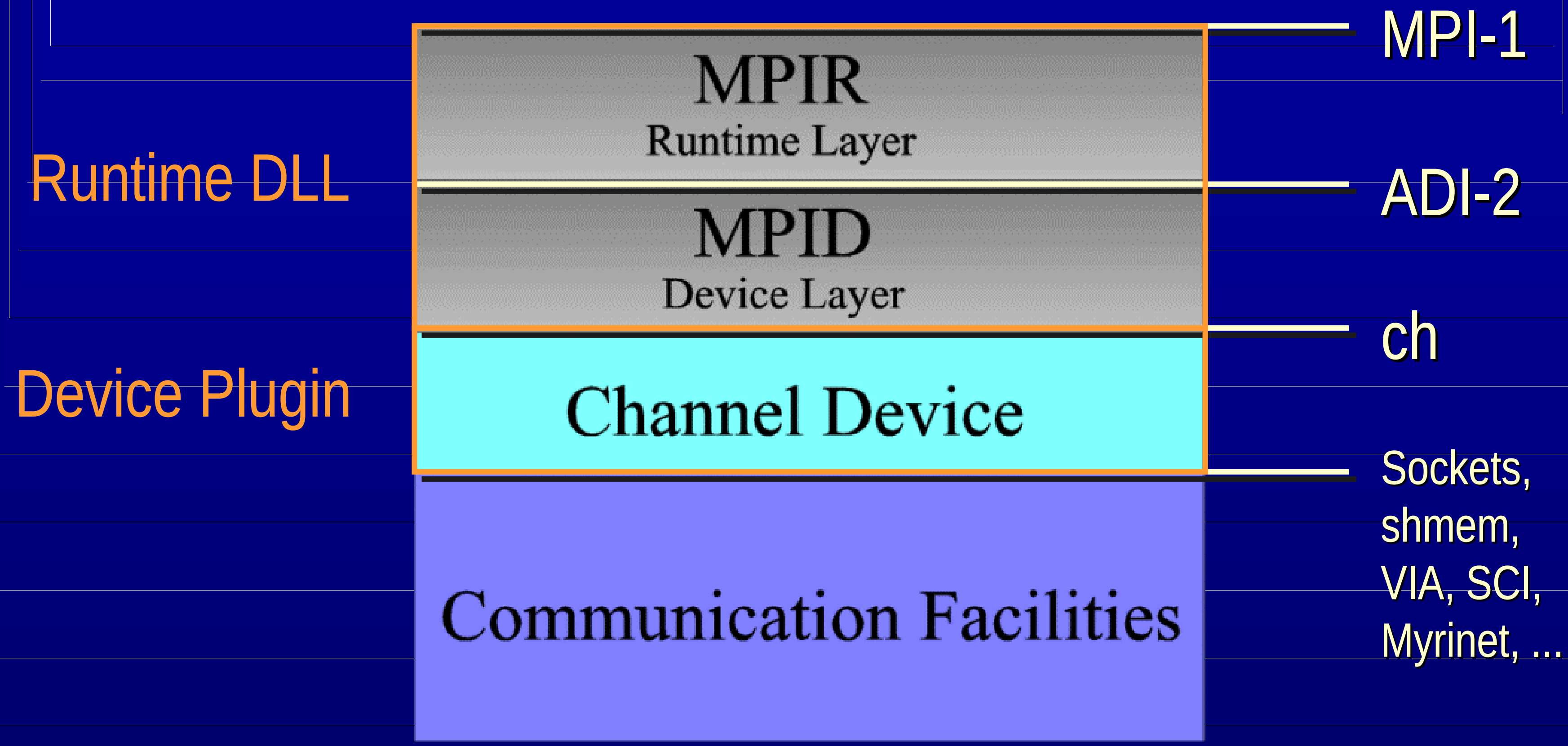
Development History

Starting point: MPICH shared memory device

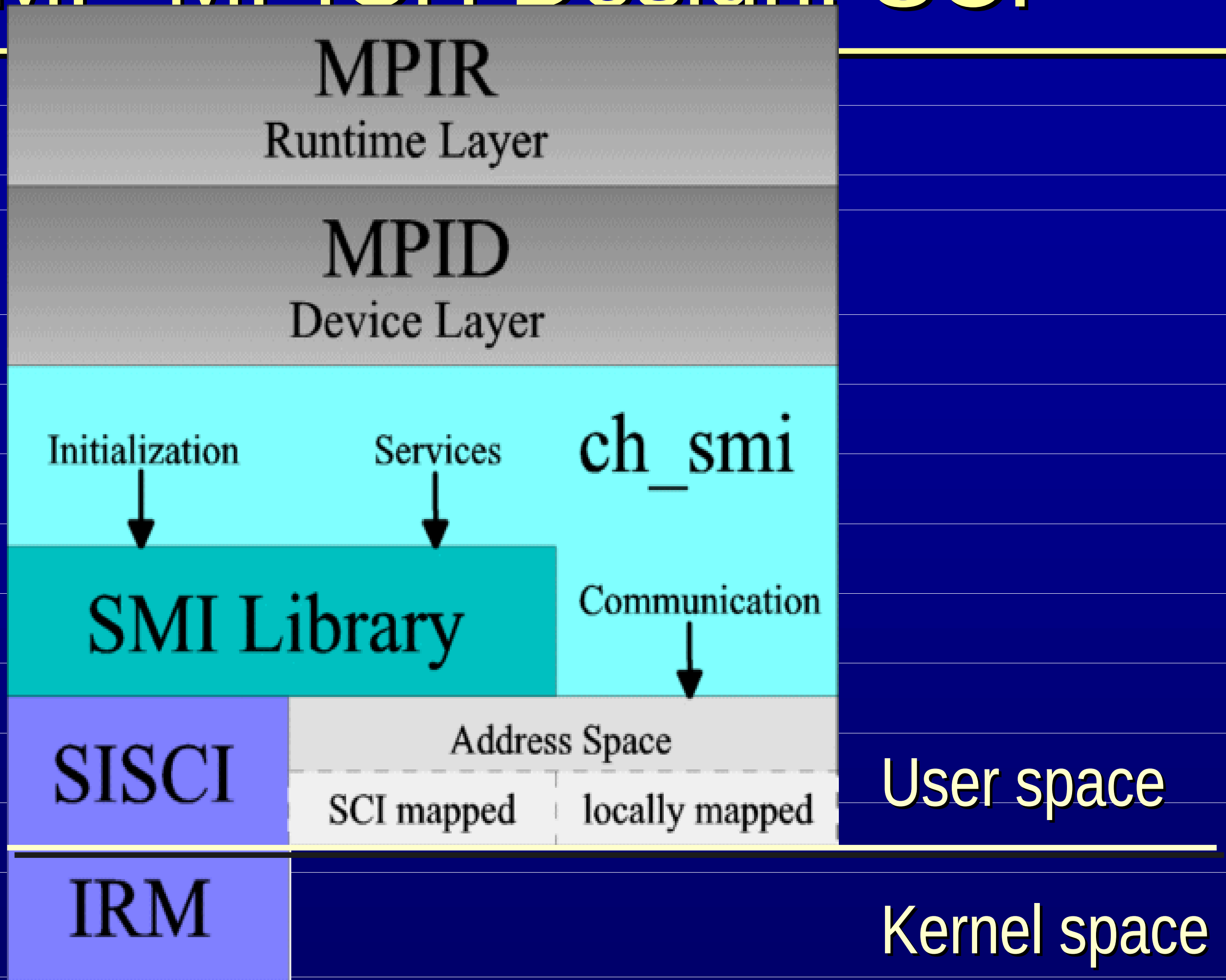
- replacement of shared memory allocation functions with SMI functions
  - ⇨ working MPI, but bad performance (10% peak)
- ⇨ Optimized layout of data structures
  - ⇨ performance doubled (20% peak)
- ⇨ New communication protocols, completely new data structures
  - ⇨ Good performance! (> 95% peak)

⇨ New device **ch_smi**

MP-MPICH Design: Generic



MP-MPICH Design: SCI



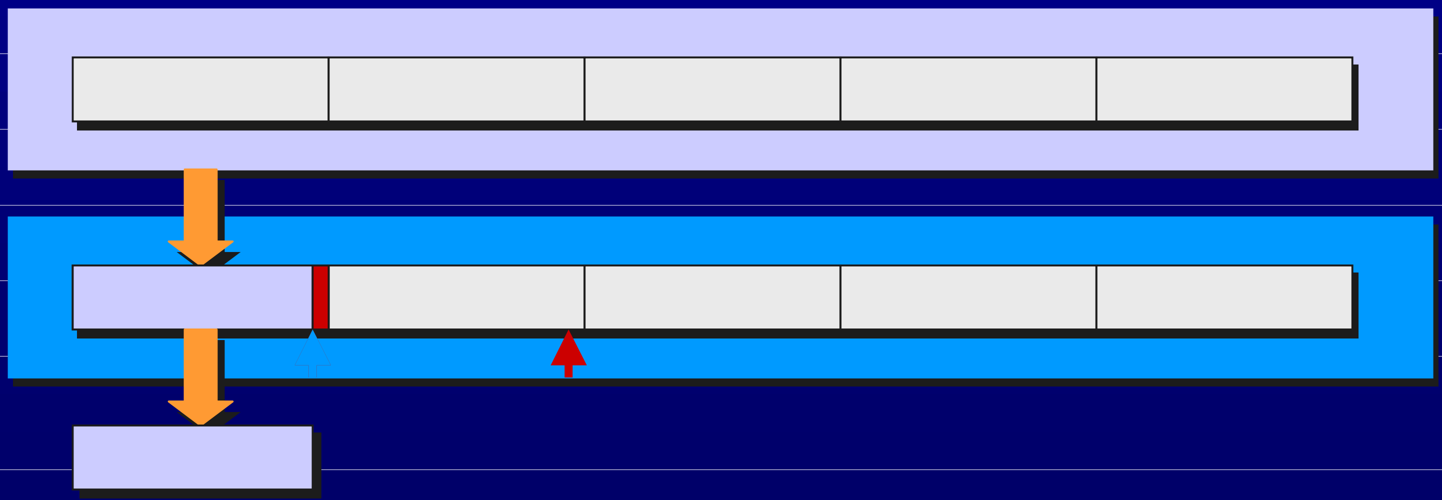
Protocols

Different protocols in SCI-MPICH:

- SHORT protocol:
  - Message length from 0 up to some 100‘s of byte
  - Also used for control messages
- EAGER protocol:
  - Message length up to some 10‘s of Kbyte
  - Uses preallocated buffers
- RENDEZVOUS protocol:
  - Arbitrary message length
  - May use multiple passes to transmit data

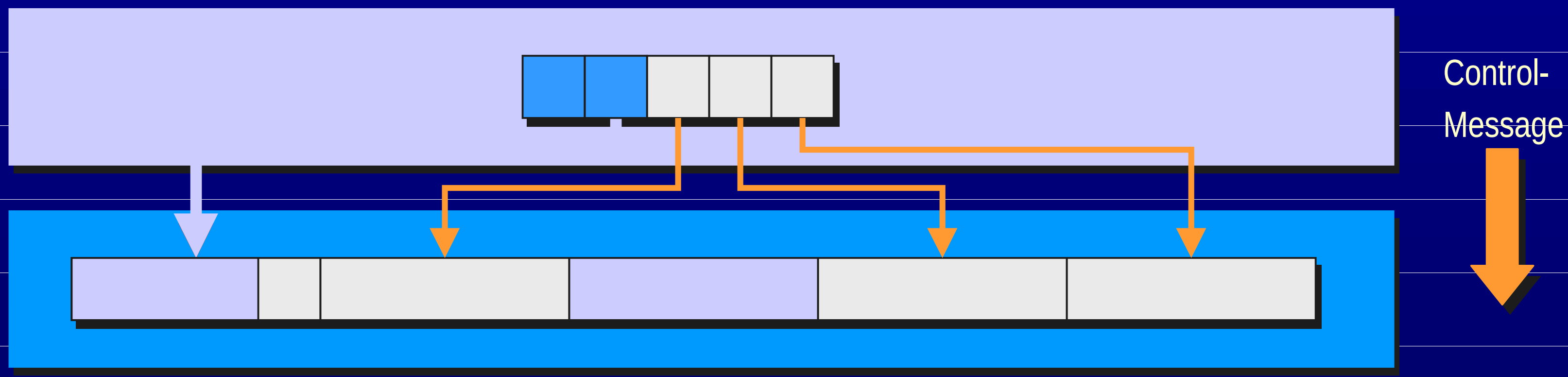
SHORT Protocol

- Separate message receive queues for each process
  - no queue-synchronization required
- Self-synchronizing messages
- Flexible size and number of message slots

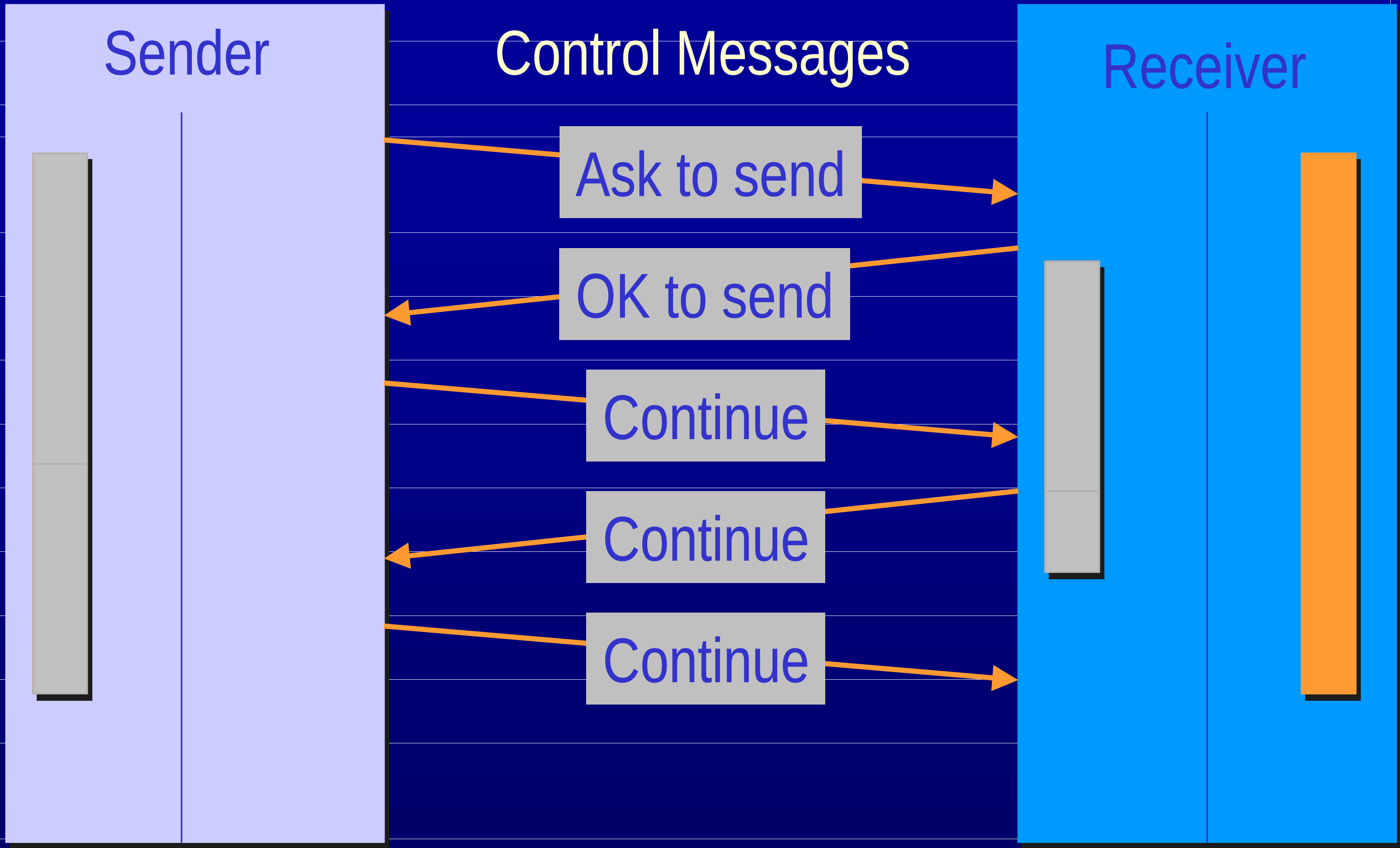


EAGER Protocol

- Use preallocated, fixed size receive buffers
- Send data „eagerly", without asking receiver
- Inform receiver of data via control message
- Configurable number and size of buffers



RNDV Protocol



Delayed Connections

- EAGER and RNDV messages are not necessarily exchanged between all process-pairs
  - Set up connections on demand: `SMI_SHM_DELAYED` and `SMI_Connect_shreg()`
- Startup-time is reduced
- Time to send first message is increased

⇨ Overall execution time (often) decreases

Global/Local Regions

- Intra-node and inter-node communication:
  - SMP region type for intra-node communication
  - other regions types for inter-node communication
- Identical protocols can be used
  - ⇨ SCI-MPICH is a good SMP-MPI, too
- Single-copy for intra-node messages:
  - Works great for Windows NT
  - Bad performance on Solaris
  - Additional kernel module for Linux (BIP)

MPI Transfer Types

- MPI offers different messager transfers types:
  - Synchronous: `MPI_Send() / MPI_Recv()`
    - When function returns, send buffer can be reused, and receive buffer contains new message
  - Asynchronous: `MPI_Isend() / MPI_Irecv()`
    - Posts send/receive job to the MPI library
    - Job is not complete until matching `MPI_Wait()` returns
- Asynchronous transfers allow overlapping of communication and computation
- **Problem**: many MPI implementations do not transfer really asynchronously!

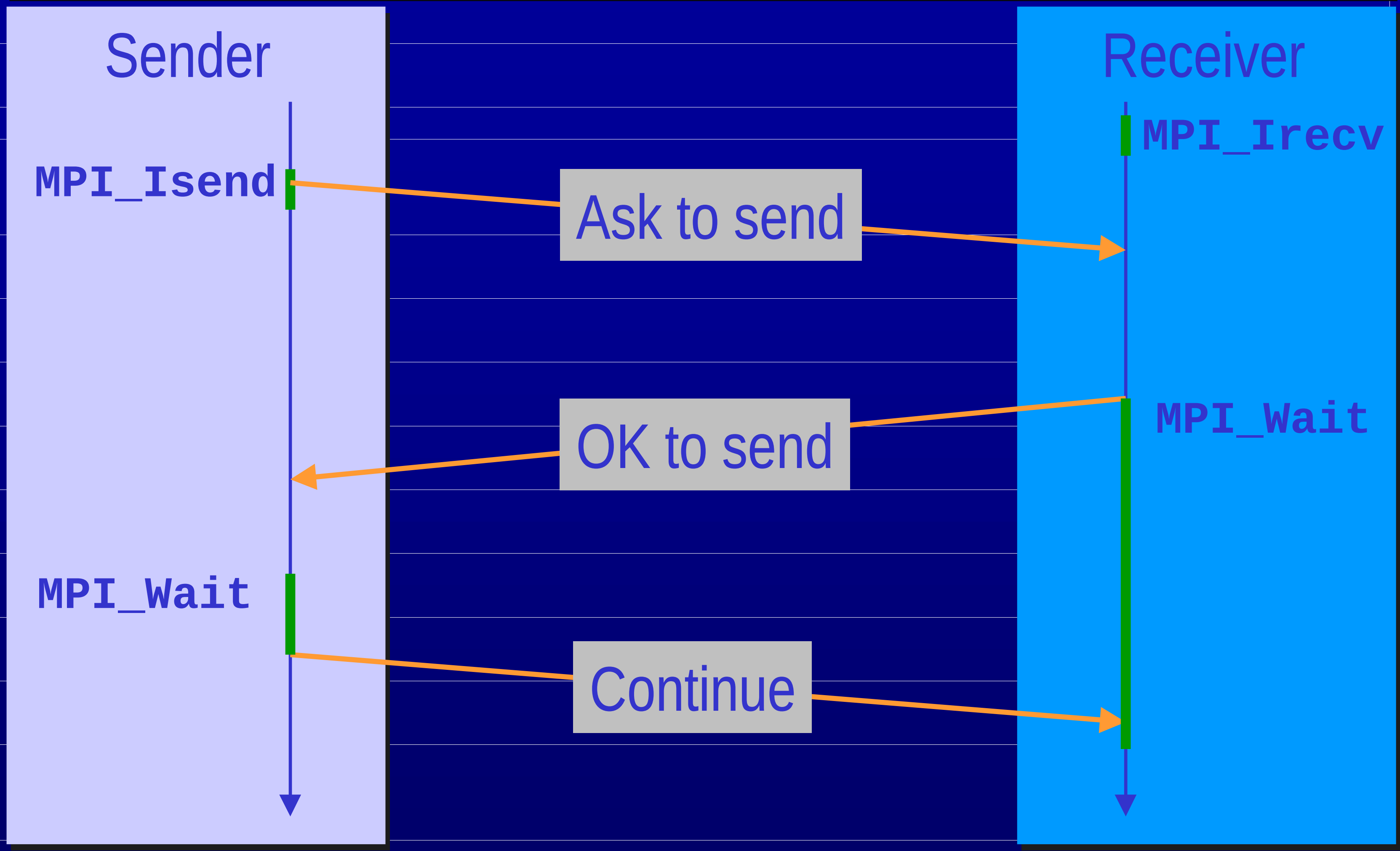
Overlapping

- MPI scenario for overlapping of computation and communication:

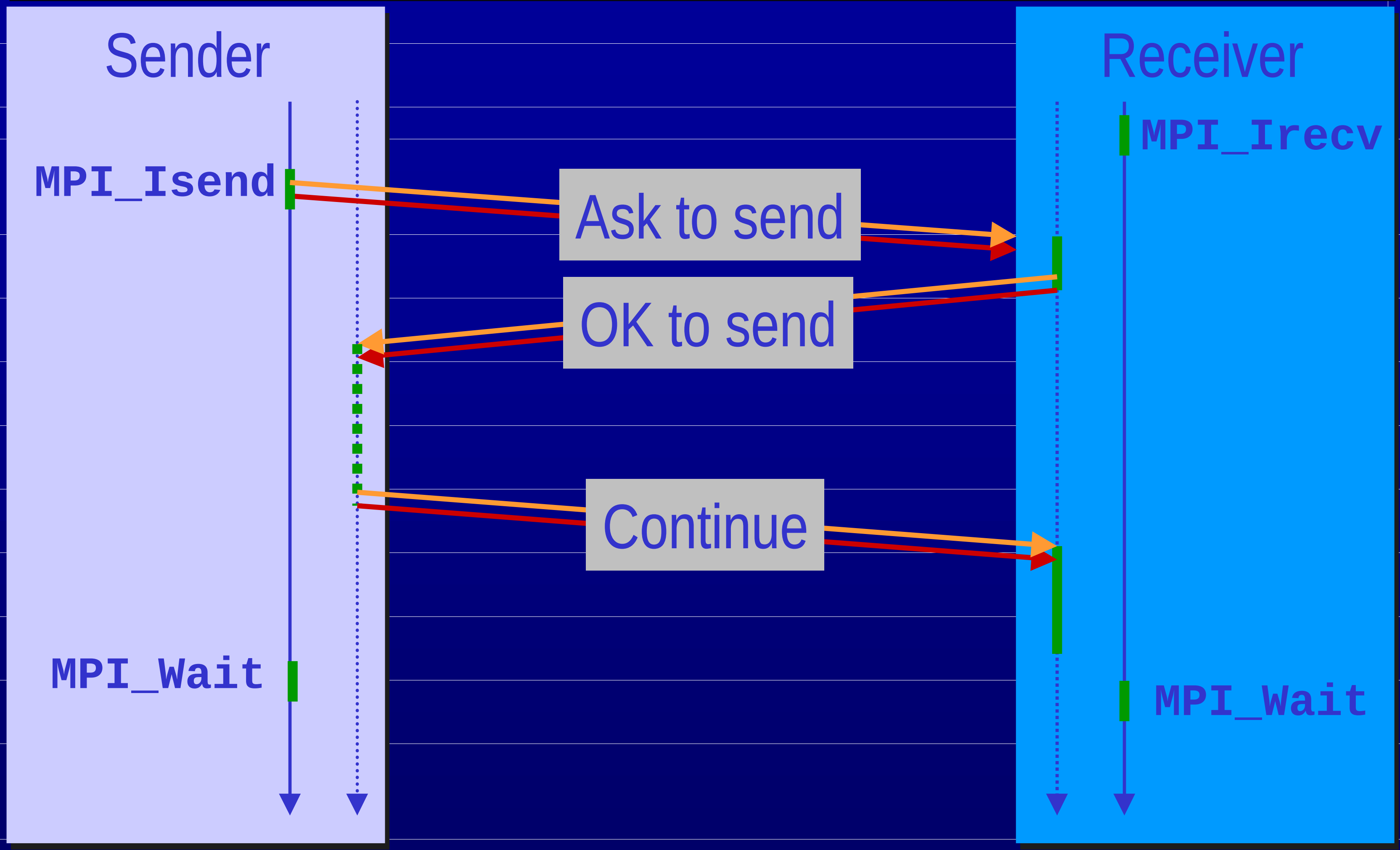
Sender	Receiver
prepare send buffer	setup receive buffer
`MPI_Isend()`	**`MPI_Irecv()`**
do computation	do computation
`MPI_Wait()`	**`MPI_Wait()`**
reuse send buffer	use receive buffer

⇨ Progress of communication !?

Synchronous Transfer



Asynchronous Transfer



Multi-Adapter Support

- SMI Library supports usage of multiple PCI-SCI-adapters
- Increase bisection bandwidth/throughput if multiple PCI-buses are available
- Possible adapter scheduling:
  - **DEFAULT**: use single default PCI-SCI adapter
  - **SMP**: each process uses another PCI-SCI adapter
  - **IMPEXP**: use different PCI-SCI adapter for importing and exporting segments

Configuration

- Many SCI-MPICH parameters are configurable on startup
- Different configuration settings may perform best for different applications
- Unreasonable settings are automatically corrected
- Device configuration file `ch_smi.conf`

⇨ More on this in the lab session!

RWTH Scalable Computing Aachen
Lehrstuhl für Betriebssysteme

Summary SCI-MPICH

- Open-source, free alternative to ScaMPI
- Based on MPCH: fully MPICH compatible
- Comparable performance on small to medium-sized clusters
- Runs on Dolphin and Scali SCI clusters
- Demonstrates usage of SMI for library development
- Part of MP-MPICH for heterogenous, cross-cluster MPI programming
- Stress-testing of SCI hardware & drivers