Parallel Programming

Message Passing

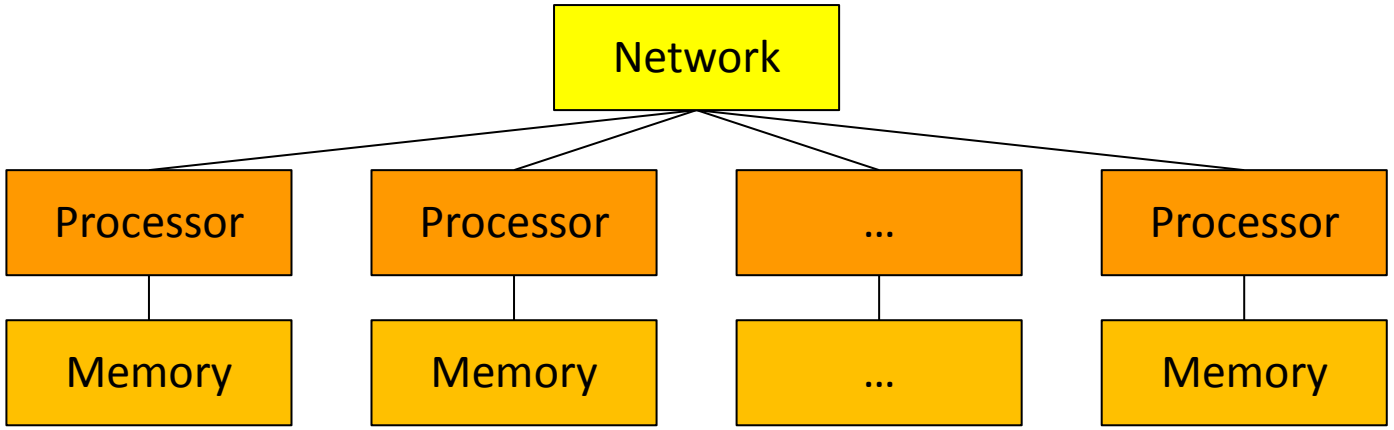
Jan Schönherr
Technische Universität Berlin
School IV – Electrical Engineering and Computer Sciences
Communication and Operating Systems (KBS)
Einsteinufer 17, Sekr. EN6, 10587 Berlin

Message Passing Paradigm

- Processes/threads exchange messages
  - No shared resources
- Said to less error prone than shared memory programming
- Used also in
  - Object-oriented programming
  - Interprocess communication

Application Area (i)

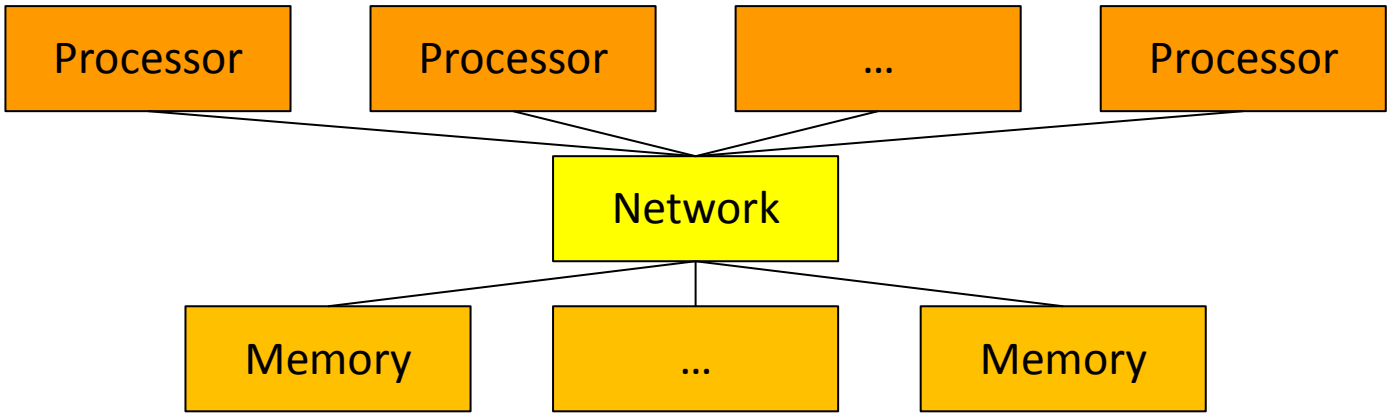
- Distributed memory systems
  - Multiple processing elements, each with its own memory
  - An interconnection network must be programmed to exchange data



  - Message passing is the natural way to program these systems

Application Area (ii)

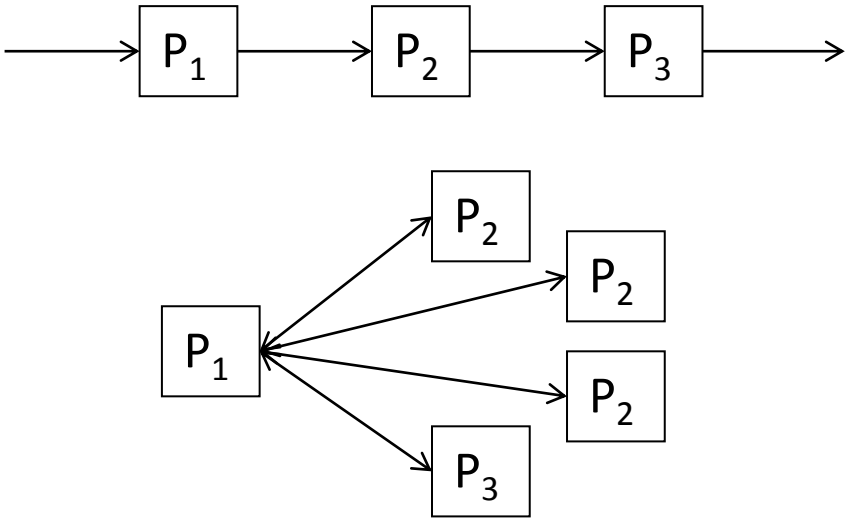
- Shared memory with non-uniform memory access
  - Multiple processing elements share an address space
  - An interconnection network maps addresses to memory
    - Some processing element/memory pairs perform better than others



  - Message passing might provide an advantage

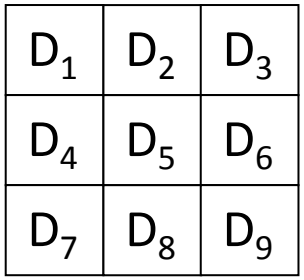
Using Message Passing (i)

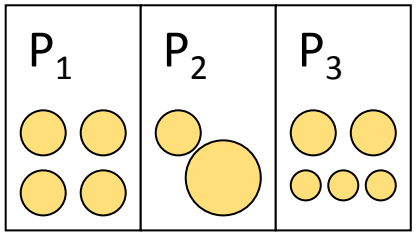
- Multiple Program Multiple Data (MPMD)
  - Each process executes a different program and has its own memory
  - Coarse grain task parallel view
    - Difficult to coordinate as a programmer
  - Possible use cases
    - Pipelines
    - Client/Server, Master/Worker structures
    - …



Using Message Passing (ii)

- Single Program Multiple Data (SPMD)
  - All processes execute the same program, but have their own memory
  - Data parallel view
    - Only one flow of control (at higher levels)
  - Possible use cases
    - Geometric decomposition
    - Fine-grained task parallelism
    - Recursive data
    - Divide and conquer
    - …





Message Passing Interface Standard

- Response to many different, incompatible message passing systems for parallel computers.
- Collaboration of 40 organizations

- Interface specification, no library!
  - Allows efficient implementations by vendors
  - Defines bindings for different languages
- Enables portable source code
  - Between architectures: behavior of functions is specified
  - To new versions of the MPI Standard

Short MPI History

- MPI-1
  - MPI-1.0 (June 1994) – initial version, 128 functions
  - MPI-1.1 (June 1995) – clarifications
  - MPI-1.2 (July 1997) – clarifications, 1 additional function
  - MPI-1.3 (May 2008) – clarifications
- MPI-2
  - MPI-2.0 (July 1997) – new functionality, 299 functions
  - MPI-2.1 (June 2008) – clarifications
  - MPI-2.2 (Sep. 2009) – clarifications, 7 additional functions
- MPI-3
  - MPI-3.0 (Sep. 2012) – revised functionality, 414 functions

Contents of the MPI Standard

- Point-to-point communication
- Datatypes (revised in MPI-2)
- Collective communication (extended in MPI-3)
- Groups & Communicators
- Topologies
- Process Management (new in MPI-2)
- One-Sided Communication (new in MPI-2, revised in MPI-3)
- File I/O (new in MPI-2)
- (+ some other things)

Point-to-Point Communication (i)

- Two types of send/receive
  - *Blocking:* Caller is blocked, until communication operation has completed (see below).
  - *Non blocking (prefix I):* Caller can continue with other instructions concurrently to the execution of the communication operation.
- Four modes for send
  - *Synchronous (prefix S):* Operation completes as soon as the corresponding receive is started.
  - *Buffered (prefix B):* Operation completes as soon as the data is buffered.
  - *Standard:* Implementation decides whether *synchronous* or *buffered* is used.
  - *Ready (prefix R):* Similar to *standard*, but may only be used if the corresponding receive has already been started.

Point-to-Point Communication (ii)

- Semantics
  - *Order:* Multiple messages from one source to the same destination are received in the order they were sent.
  - *Progress:* If there are matching send/receive pairs, an operation will complete.
  - *Fairness:* Fairness is not guaranteed.
  - *Resource limitations:* Developer must take buffer behavior into account.
- Other things
  - Persistent Communication
  - Send and Receive in one call

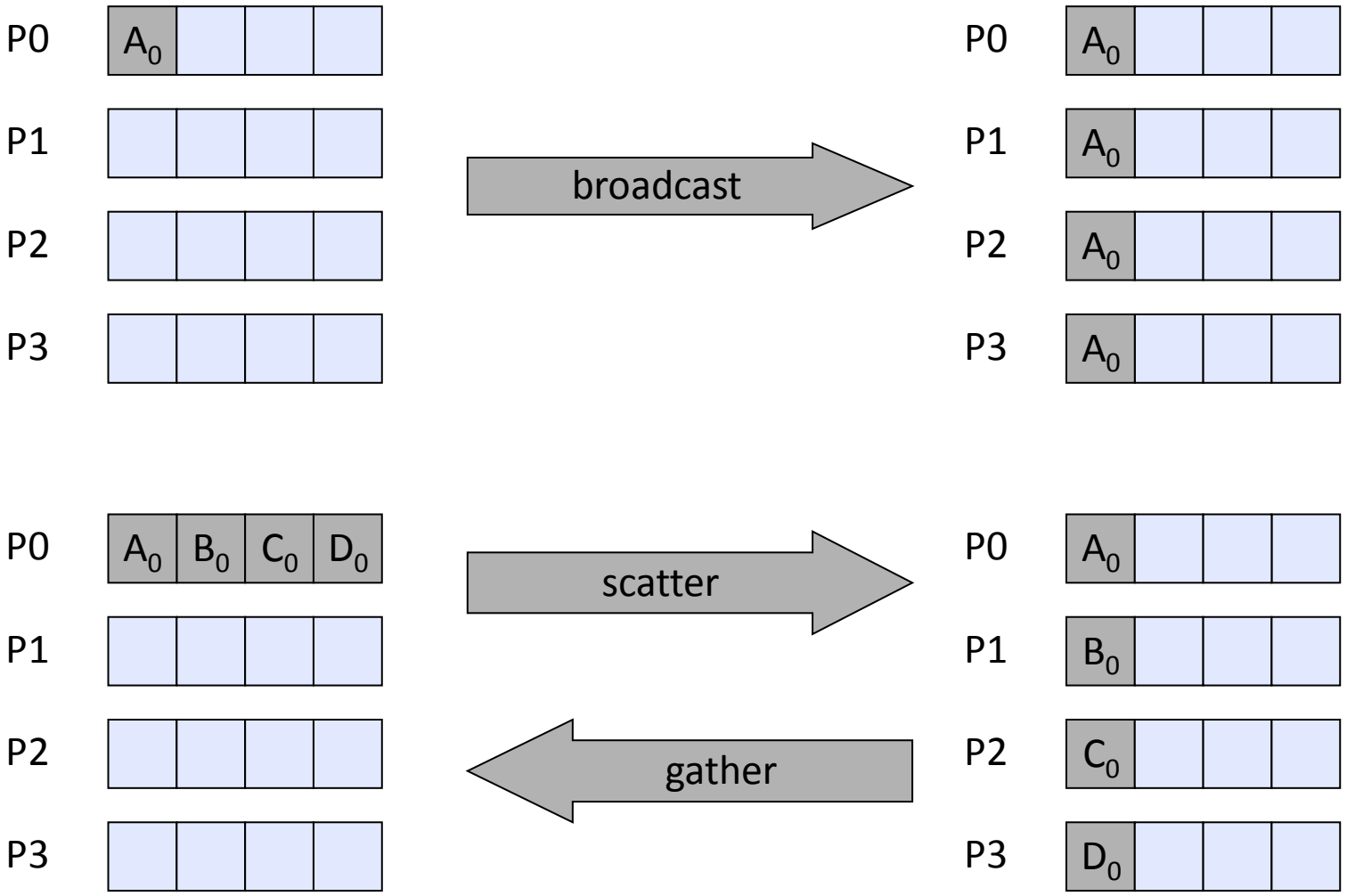
Datatypes

- Necessary for heterogeneous environments
- Allows complex user defined data types
  - Not necessarily contiguous, e. g.
    - A block or column of a matrix
    - Cyclic or completely irregular mappings
  - Mixing of different datatypes possible
- Revised in MPI-2
  - (MPI-1 source will still work in MPI-2, but not vice versa)

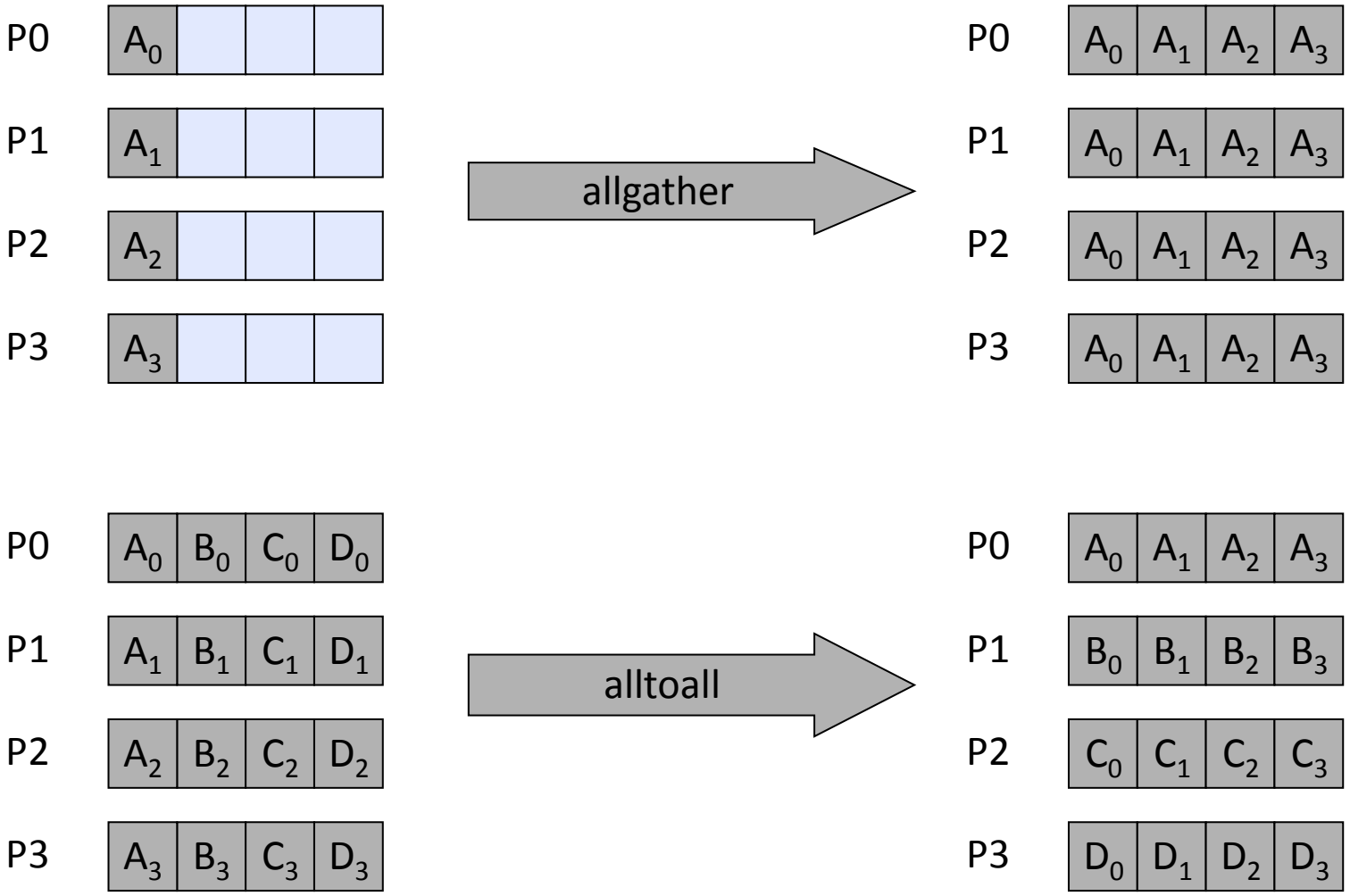
Collective Communication (i)

- To be called on all processes of a communicator
- Blocking (or non-blocking since MPI-3, prefix *I*)
- May complete as soon as own contribution is performed
- Operations
  - Barrier
  - Broadcast, Scatter, (All-)Gather, All-to-all
  - (All-)Reduce, Reduce-Scatter
  - (Ex-)Scan
- Some operations support different amounts of data per process (suffix *v*)

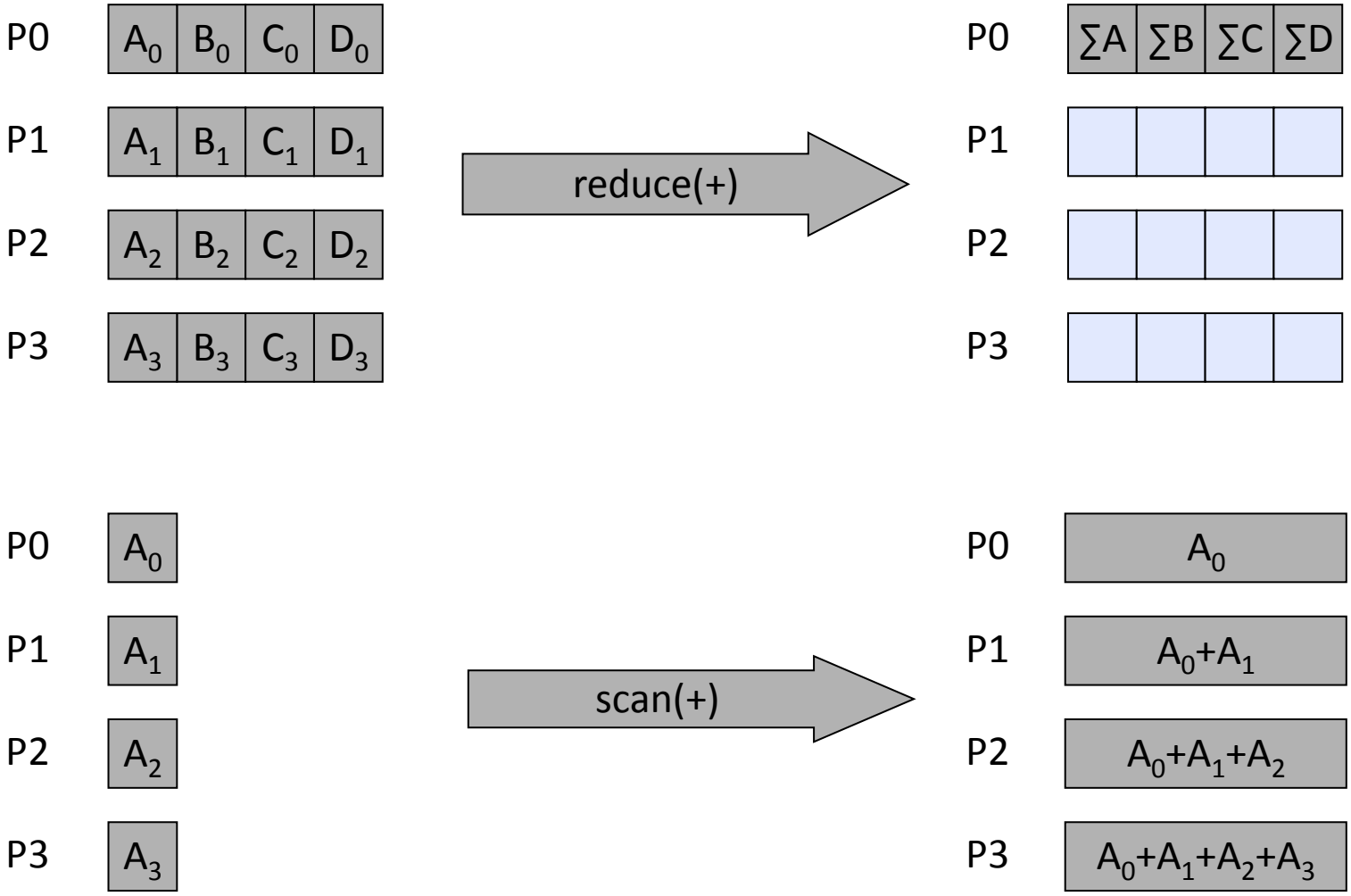
Collective Communication (ii)



Collective Communication (iii)



Collective Communication (iv)



Groups & Communicators

- Groups
  - Ordered set of processes (with ranks)
- (Intra-)Communicator
  - Scope of message passing operations
  - Contains a group
- Inter-Communicators
  - Allow communication between two disjoint groups

Topologies

- Helper functions to define virtual topologies
  - For general graphs and Cartesian structures
- Describe communication patterns
  - Simplify rank calculations
- May aid in a better utilization of the physical topology

Process Management

- New in MPI-2
- Spawn parallel programs within a parallel program
  - Communication between both programs with inter-communicators
  - Some details are implementation dependent (e. g. on which nodes the new program will execute)
- Establish inter-communicators between already running parallel programs
  - Less implementation specific code inside your program

One-Sided Communication

- New in MPI-2, revised in MPI-3
- More natural on platforms with shared memory
- Decouples communication and synchronization
  - Communication functions
    - *Put*, *Get*, *Accumulate*
    - Operate on previously created *Windows*
    - Non-blocking; execution may be deferred until a call to a synchronization function
  - Synchronization functions
    - *Fence*, *Post/Wait* + *Start/Complete*, *Lock/Unlock, Sync* + *Flush*
    - Accesses are structured in *epochs* by these functions
    - Simplified: During an epoch you may not issue conflicting accesses

File I/O

- New in MPI-2
- Parallel operations on files
  - Either collective, or non-collective
  - Blocking or non-blocking
- File structure is based on derived datatypes
  - A per process file view describes the data belonging to the process
    - Map complex internal data structures to a sequential representation
  - Support for different data representations (or file encodings)
    - Speed vs. portability
- Shifts I/O from application developers to MPI library developers

Writing Maintainable Code

- Rank specific computation
  - Is it really necessary?
  - Keep it as short as possible
  - Consider hiding it within a function
- Point-to-Point communication
  - Send and Receive always form a pair
  - Keep the two corresponding calls as close as possible
  - Consider collective communication
- Collective communication
  - Barrier, broadcast, reduce, scatter/gather, …
  - Have exactly one call for one collective operation

Example 1

```
main() {
        if (rank == 0) {
                initialize message
                send message
        } else if (rank == size - 1) {
                recv message
                print message
        } else {
                recv message
                send message
        }
}
```

```
main() {
        init-msg()
        forward-msg()
        print-msg()
}
init-msg() {
        if (rank == 0)
                initialize message
}
forward-msg() {
        if (rank > 0)
                recv message
        if (rank < size - 1)
                send message
}
print-msg() {
        if (rank == size - 1)
                print message
}
```

Example 2

```
main() {
        if (rank == 0)
                master()
        else
                slave()
}
master() {
        initialize data
        for each slave
                send part of data
        for each slave
                recv part of solution
        store solution
}
slave() {
        recv part of data
        compute part of solution
        send part of solution
}
```

```
main() {
        init-data()
        compute()
        store-solution()
}
init-data() {
        if (rank == 0)
                initialize data
        scatter data
}
compute() {
        compute part of solution
}
store-solution() {
        gather solution
        if (rank == 0)
                store solution
}
```