Parallel Programming Concepts

Message Passing

Peter Tröger

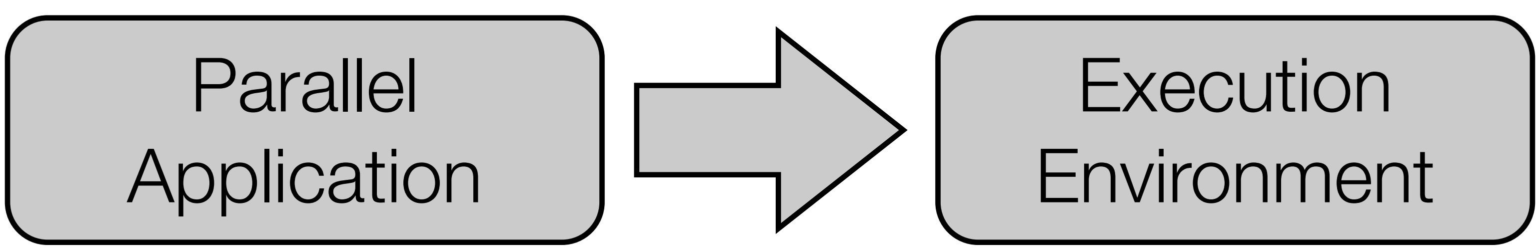
Sources:

Clay Breshears: The Art of Concurrency
Blaise Barney: Introduction to Parallel Computing
OpenMP 3.0 Specification
MPI2 Specification
Blaise Barney: OpenMP Tutorial, https://computing.llnl.gov/tutorials/openMP/

Parallel Programming

Multi-Tasking	PThreads, OpenMP, OpenCL, Linda, Cilk, ...
Message Passing	MPI, PVM, CSP Channels, Actors, ...
Implicit Parallelism	Map/Reduce, PLINQ, HPF, Lisp, Fortress, ...
Mixed Approaches	Ada, Scala, Clojure, Erlang, X10, ...

	Data Parallel / SIMD	Task Parallel / MIMD
Shared Memory (SM)	GPU, Cell, SSE, Vector processor ...	ManyCore/ SMP system ...
Shared Nothing / Distributed Memory (DM)	processor-array systems systolic arrays Hadoop ...	cluster systems MPP systems ...



Message Passing

Multi-Tasking	PThreads, OpenMP, OpenCL, Linda, Cilk, ...
Message Passing	MPI, PVM, CSP Channels, Actors, ...
Implicit Parallelism	Map/Reduce, PLINQ, HPF, Lisp, Fortress, ...
Mixed Approaches	Ada, Scala, Clojure, Erlang, X10, ...

The Parallel Virtual Machine (PVM)

- Intended for heterogeneous environments, integrated set of software tools and libraries
- User-configured host pool
- Translucent access to hardware, collection of virtual processing elements
- Unit of parallelism in PVM is a task, no process-to-processor mapping is implied
- Support for heterogeneous environments
- Explicit message-passing mode, multiprocessor support
- C, C++ and Fortran language

PVM (contd.)

- PVM tasks are identified by an integer task identifier (TID)
- User named groups
- Programming paradigm
    - User writes one or more sequential programs
    - Contains embedded calls to the PVM library
    - User typically starts one copy of one task by hand
    - Process subsequently starts other PVM tasks
    - Tasks interact through explicit message passing

PVM Example

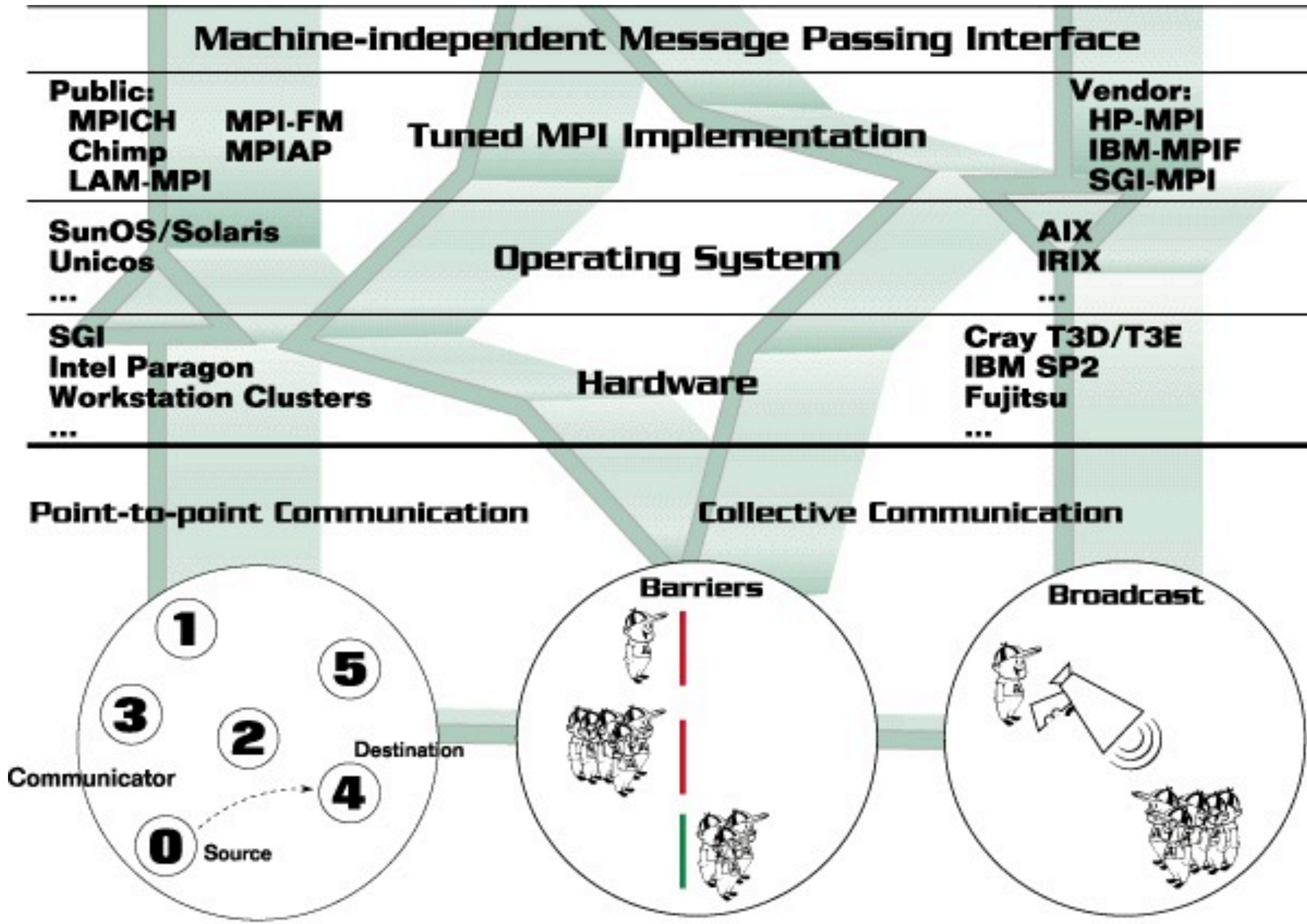
```
main() {
  int cc, tid, msgtag;
  char buf[100];
  printf("i'm t%x\n", pvm_mytid());  //print id
  cc = pvm_spawn("hello_other",
           (char**)0, 0, "", 1, &tid);
  if (cc == 1) {
     msgtag = 1;
     pvm_recv(tid, msgtag); // blocking
     pvm_upkstr(buf);   // read msg content
     printf("from t%x: %s\n", tid, buf);
  } else
     printf("can't start it\n");
  pvm_exit();
}
```

PVM Example (contd.)

```
main() {
  int ptid, msgtag;
  char buf[100];
  ptid = pvm_parent(); // get master id
  strcpy(buf, "hello from ");
  gethostname(buf+strlen(buf), 64); msgtag = 1;
  // initialize send buffer
  pvm_initsend(PvmDataDefault);
  // place a string
  pvm_pkstr(buf);
  // send with msgtag to ptid
  pvm_send(ptid, msgtag); pvm_exit();
}
```

Message Passing Interface (MPI)

- Communication library for sequential programs
  - Definition of syntax and semantics for source code portability
  - Maintain implementation freedom on high-performance messaging hardware - shared memory, IP, Myrinet, propietary ...
  - MPI 1.0 (1994) and 2.0 (1997) standard, developed by MPI Forum
- Fixed number of processes, determined on startup
  - Point-to-point communication
  - Collective communication, for example group broadcast
- Focus on efficiency of communication and memory usage, not interoperability
- Fortran / C - Binding



Machine-independent Message Passing Interface
Public:
MPICH
MPI-FM
Chimp
MPIAP
LAM-MPI
Tuned MPI Implementation
Vendor:
HP-MPI
IBM-MPIF
SGI-MPI
SunOS/Solaris
Unicos
...
Operating System
AIX
IRIX
...
SGI
Intel Paragon
Workstation Clusters
...
Hardware
Cray T3D/T3E
IBM SP2
Fujitsu
...
Point-to-point Communication
Collective Communication
1
5
3
2
Destination
Communicator
4
0
Source
Barriers
Broadcast

Basic MPI

- Communicators (process group handle)
  - `MPI_COMM_SIZE (IN comm, OUT size),`
    `MPI_COMM_RANK (IN comm, OUT pid)`
  - Sequential process ID‘s, starting with zero
- `MPI_SEND (IN buf, IN count, IN datatype, IN destPid, IN msgTag, IN comm)`
  `MPI_RECV (IN buf, IN count, IN datatype, IN srcPid, IN msgTag, IN comm, OUT status)`
  - Source / destination identified by 3-tupel tag, source and comm
  - MPI_RECV can block, waiting for specific source
- Constants: `MPI_COMM_WORLD, MPI_ANY_SOURCE, MPI_ANY_DEST`
- Data types: `MPI_CHAR, MPI_INT, ..., MPI_BYTE, MPI_PACKED`

MPI Data Conversion

- *„MPI does not require support for inter-language communication.“*
- *„The type matching rules imply that MPI communication never entails type conversion.“*
- *„On the other hand, MPI requires that a representation conversion is performed when a typed value is transferred across environments that use different representations for the datatype of this value.“*
- Type matching through name similarity (without MPI_BYTE and MPI_PACKED)

Communication Modes

- Blocking communication
  - Do not return until the message data and envelope have been stored away
  - *Standard*: MPI decides whether outgoing messages are buffered
  - *Buffered*: MPI_BSEND returns always immediately
    - Might be a problem when the internal send buffer is already filled
  - *Synchronous*: MPI_SSEND completes if the receiver started to receive
  - *Ready*: MPI_RSEND should be started only if the matching MPI_RECV is already available
    - Can omit a handshake-operation on some systems
- Blocking communication ensures that the data buffer can be re-used

Non-Overtaking Message Order

- „If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending.“

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_BSEND (buf1, count, MPI_REAL, 1,
              tag, comm, ierr)
   CALL MPI_BSEND (buf2, count, MPI_REAL, 1,
              tag, comm, ierr)
ELSE   ! rank.EQ.1
   CALL MPI_RECV (buf1, count, MPI_REAL, 0,
              MPI_ANY_TAG, comm, status, ierr)
   CALL MPI_RECV (buf2, count, MPI_REAL, 0,
              tag, comm, status, ierr)
END IF
```

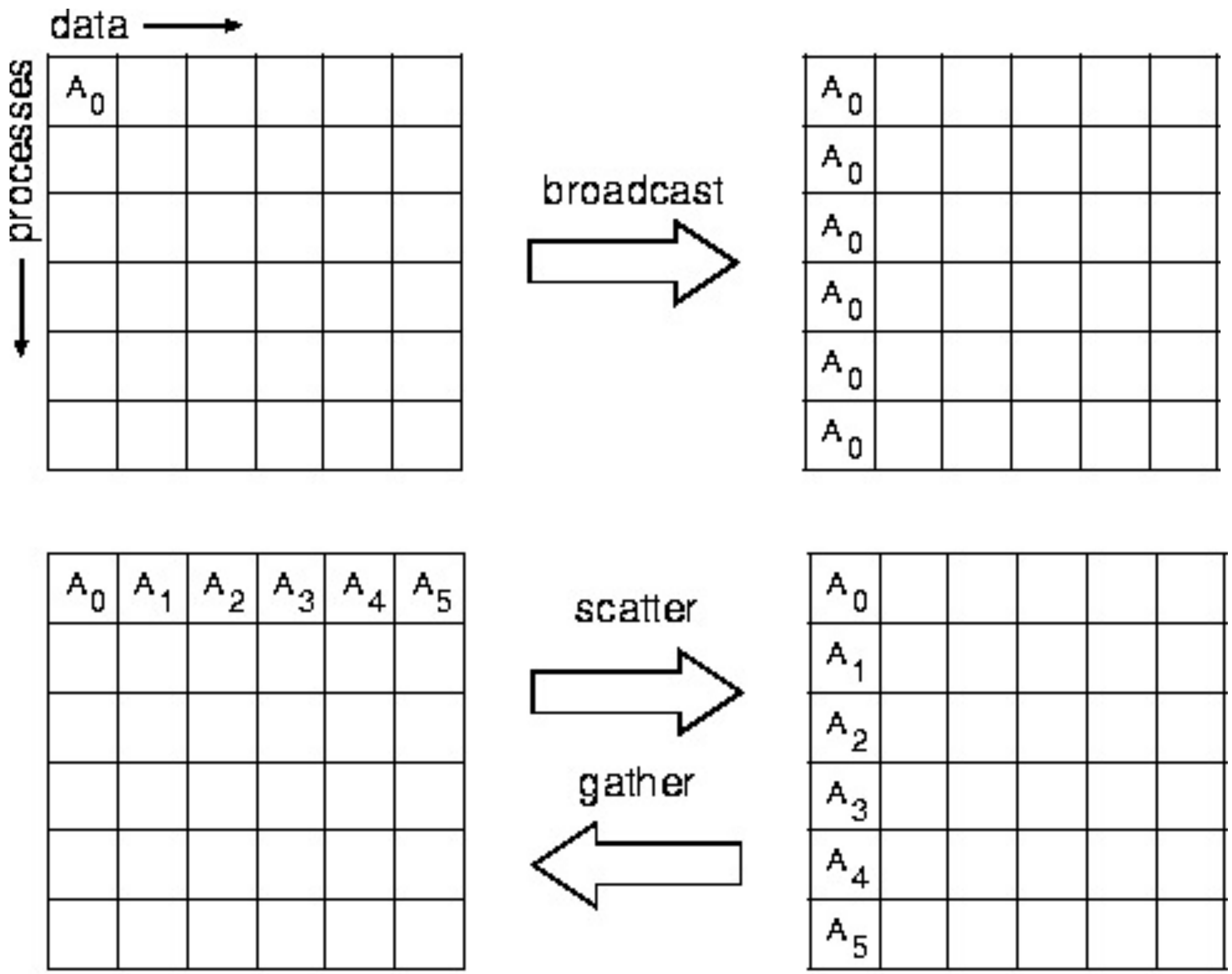
Non-Blocking Communication

- *Send/receive* start and *send/receive* completion call, with request handle
- Communication mode influences the behavior of the completion call
- Buffered non-blocking send operation leads to an immediate return of the completion call
- ‚Immediate send' calls
  - `MPI_ISEND, MPI_IBSEND, MPI_ISSEND, MPI_IRSEND`
- Completion calls
  - `MPI_WAIT, MPI_TEST, MPI_WAITANY, MPI_TESTANY, MPI_WAITSOME, ...`
- sending side: MPI_REQUEST_FREE

Collective Communication

- Global operations for a distributed application, could also be implemented manually
- `MPI_BARRIER (IN comm)`
  - returns only if the call is entered by all group members
- `MPI_BCAST (INOUT buffer, IN count, IN datatype, IN rootPid, IN comm)`
  - root process broadcasts to all group members, itself included
  - all group members use the same comm & root parameter
  - on return, all group processes have a copy of root‘s send buffer

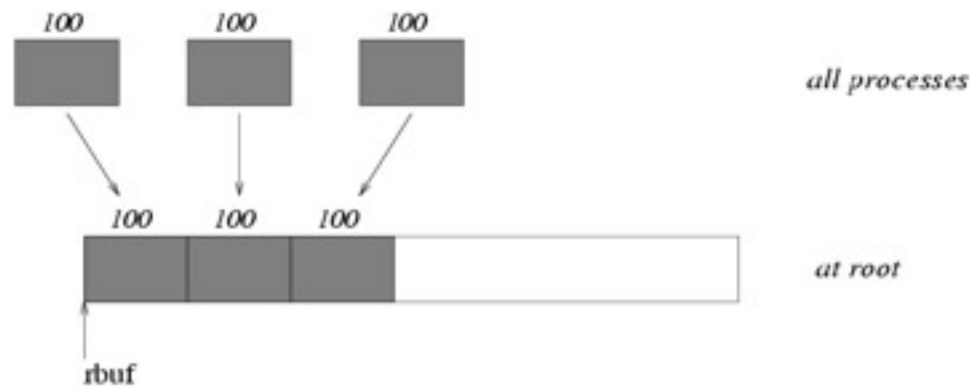
Collective Move Functions



Gather

- `MPI_GATHER ( IN sendbuf, IN sendcount, IN sendtype, OUT recvbuf, IN recvcount, IN recvtype, IN root, IN comm )`
  - Each process sends its buffer to the root process (including the root process itself)
  - Incoming messages are stored in rank order
  - Receive buffer is ignored for all non-root processes
  - `MPI_GATHERV` allows varying count of data to be received from each process
  - No promise for synchronous behavior

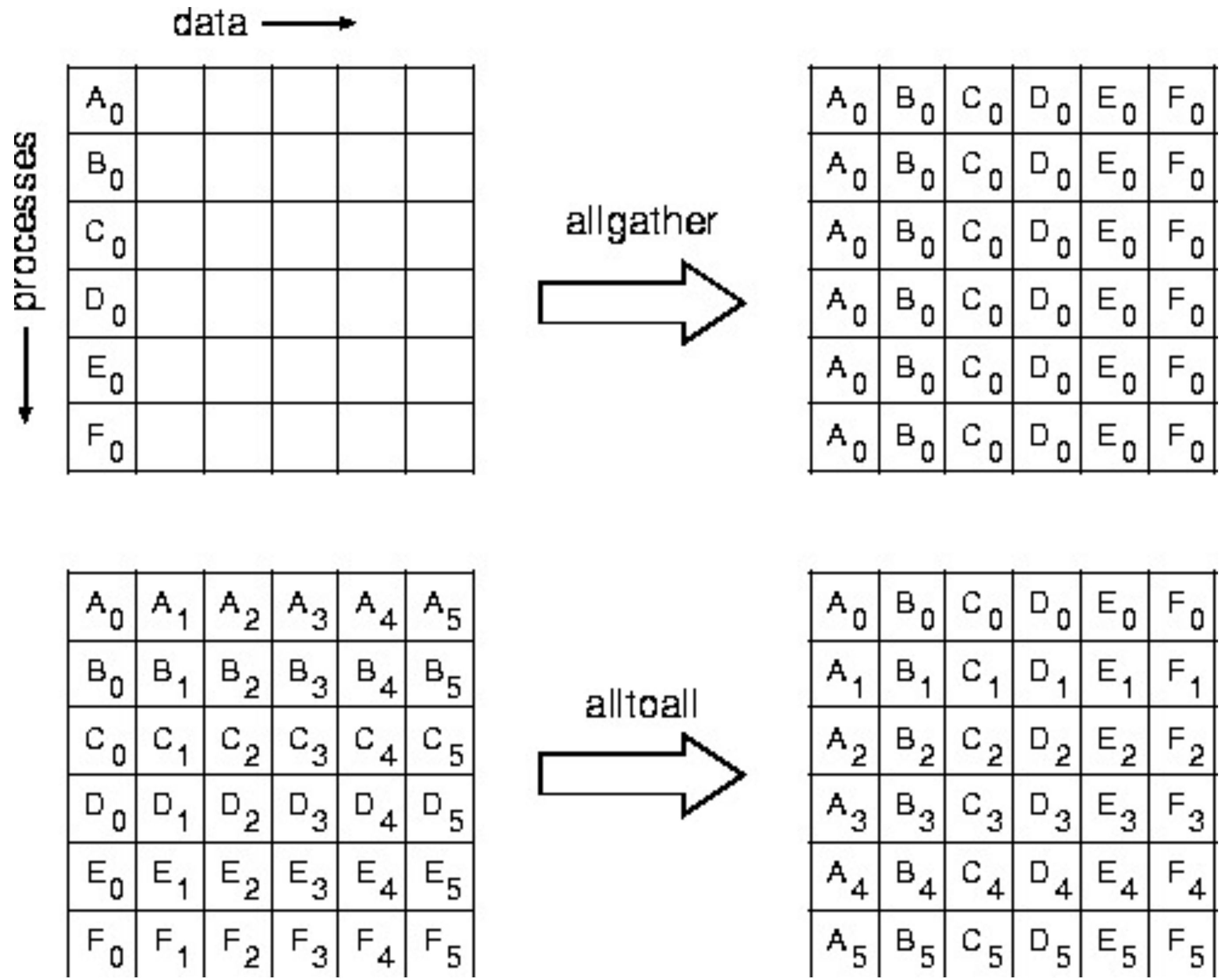
MPI Gather Example




```
MPI_Comm comm;
int gsize,sendarray[100];
int root, myrank, *rbuf;
... [compute sendarray]
MPI_Comm_rank( comm, myrank);
if ( myrank == root) {
  MPI_Comm_size( comm, &gsize);
  rbuf = (int *)malloc(gsize*100*sizeof(int));
}
MPI_Gather ( sendarray, 100, MPI_INT, rbuf, 100,
            MPI_INT, root, comm );
```

Scatter

- `MPI_SCATTER ( IN sendbuf, IN sendcount, IN sendtype, OUT recvbuf, IN recvcount, IN recvtype, IN root, IN comm )`
  - Sliced buffer of root process is send to all other processes (including the root process itself)
  - Send buffer is ignored for all non-root processes
  - `MPI_SCATTERV` allows varying count of data to be send to each process


data
processes
A0
B0
C0
D0
E0
F0
allgather
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 A1 A2 A3 A4 A5
B0 B1 B2 B3 B4 B5
C0 C1 C2 C3 C4 C5
D0 D1 D2 D3 D4 D5
E0 E1 E2 E3 E4 E5
F0 F1 F2 F3 F4 F5
alltoall
A0 B0 C0 D0 E0 F0
A1 B1 C1 D1 E1 F1
A2 B2 C2 D2 E2 F2
A3 B3 C3 D3 E3 F3
A4 B4 C4 D4 E4 F4
A5 B5 C5 D5 E5 F5

What Else

- `MPI_SENDRCV` (useful for RPC semantic)
- Global reduction operators
- Complex data types
- Packing / Unpacking (`sprintf / sscanf`)
- Group / Communicator Management
- Virtual Topology Description
- Error Handling
- Profiling Interface

MPICH library

- Development of the MPICH group at Argonne National Laboratory (Globus)
- Portable, free reference implementation
- Drivers for shared memory systems (ch_shmem), Workstation networks (ch_p4) , NT networks (ch_nt) and Globus 2 (ch_globus2)
- Driver implements `MPIRUN` (fork, SSH, MPD, GRAM)
- Supports multiprotocol communication (with vendor MPI and TCP) for intra-/intermachine messaging
- MPICH2 (MPI 2.0) is available, GT4-enabled version in development
- MPICH-G2 is based on Globus NEXUS / XIO library
- Debugging and tracing support

Actor Model

- *Carl Hewitt, Peter Bishop and Richard Steiger. A Universal Modular Actor Formalism for Artificial Intelligence IJCAI 1973.*
  - Mathematical model for concurrent computation, inspired by lambda calculus, Simula, Smalltalk
  - No global system state concept (relationship to physics)
  - Actor as computation primitive, which can make local decisions, concurrently creates more actors, or concurrently sends / receives messages
  - Asynchronous one-way messaging with changing topology, no order guarantees
    - Comparison: CSP relies on hierarchy of combined parallel processes, while actors rely only on message passing paradigm only
  - Recipient is identified by *mailing address*, can be part of a message

Actor Model

- Principle of interaction: asynchronous, unordered, fully distributed messaging
- Fundamental aspects of the model
  - Emphasis on local state, time and name space - no central entity
  - Computation: Not global state sequence, but partially ordered set of events
    - Event: Receipt of a message by a target actor
    - Each event is a transition from one local state to another
    - Events may happen in parallel
  - Strict locality: Actor A gets to know actor B only by direct creation, or by name transmission from another actor C
  - Actors system are constructed inductively by adding events
- Messaging reliability declared as orthogonal aspect