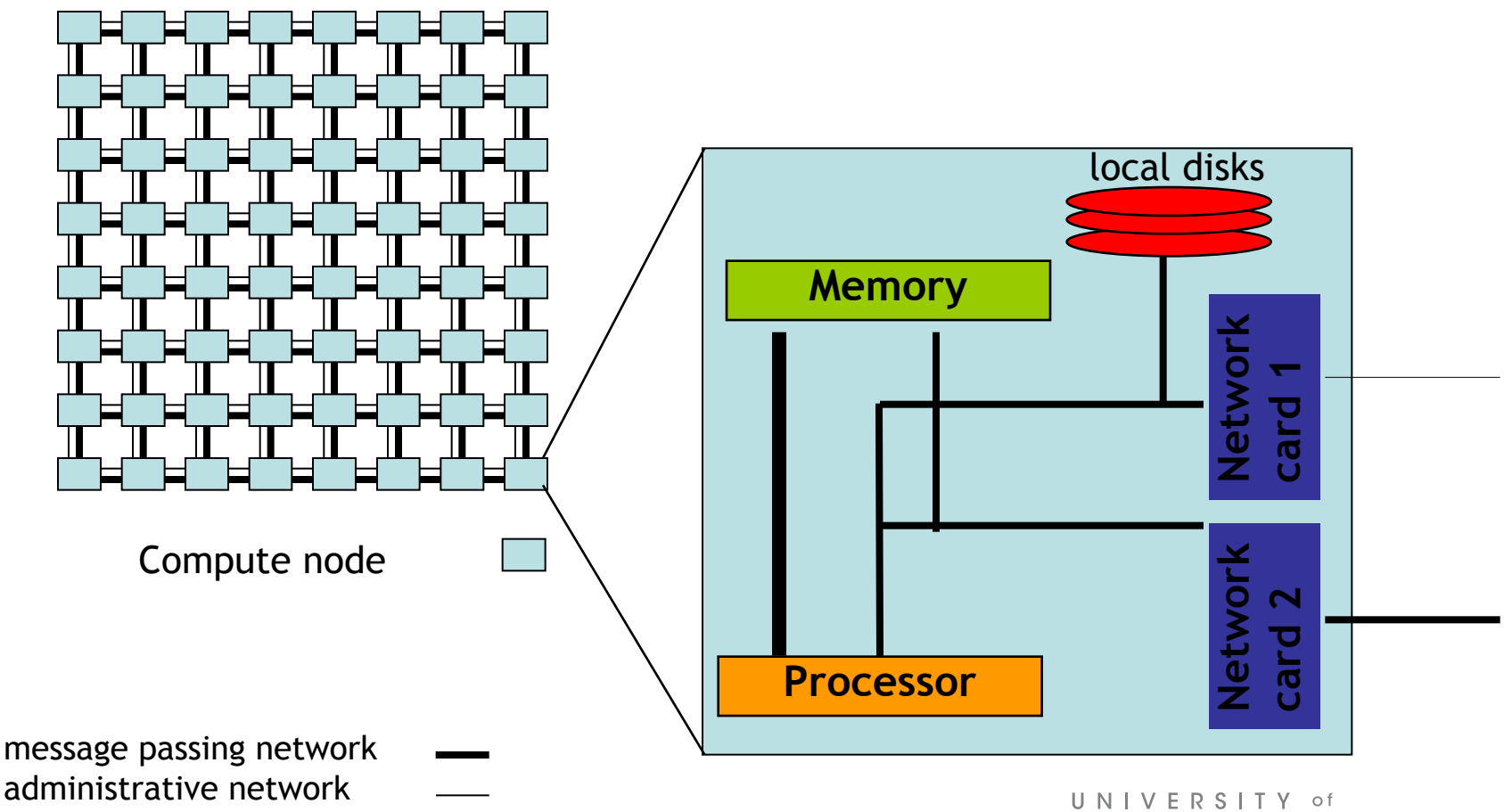
Introduction to Parallel Computing
MPI – Message Passing Interface

Vistas in Advanced Computing / Summer 2017

UNIVERSITY of HOUSTON
CENTER FOR ADVANCED COMPUTING & DATA SYSTEMS

Distributed memory machines

- Each compute node represents an independent entity with its own main memory, operating system etc.



Communication on the Internet

1st Process (Client)

mypc.my-university.edu
183.69.14.54

Host Name and Host Address

http request

Internet

2nd Process (Server)

webserver.provider.com
129.74.11.55

Protocol

Communication on the Internet

- Addressing:
  - hostname and/or IP Address
  - Globally unique address required
- Communication:
  - based on protocols, e.g. http or TCP/IP
  - Allows for communication between independent vendors / implementors
- Process start-up:
  - every process (= application) has to be started separately

The Message Passing universe

- Addressing:
  - Communication only required within a parallel job
  - Simple addressing scheme sufficient, e.g. an integer value between *0* and *n-1.*
- Communication:
  - Efficiency the primary goal
  - MPI defines interfaces how to send data to a process and how to receive data from a process.
  - It does not specify a protocol nor enforce a particular implementation
- Process start-up:
  - Want to start *n*-processes to work on the same problem <u>efficiently</u>

History of MPI

- Until the early 90's:
  - all vendors of parallel hardware had their own message passing library
  - Some public domain message passing libraries available
  - all of them being incompatible to each other
  - High efforts for end-users to move code from one architecture to another
- June 1994: Version 1.0 of MPI presented by the MPI Forum
- June 1995: Version 1.1 (errata of MPI 1.0)
- 1997: MPI 2.0 – adding new functionality to MPI
- 2008: MPI 2.1
- 2009: MPI 2.2
- 2012: MPI 3.0 released in Nov.
- 2015: MPI 3.1 released in June

Simple Example (I)

MPI command to start process

name of the application to start

number of processes to be started

shark.cs.uh.edu - shark - SSH Secure Shell

File Edit View Window Help

Quick Connect Profiles

```
gabriel@shark:~> mpirun -np 2 ./hello_world
MPI Hello World from process 0  part size = 2
MPI Hello World from process 1  part size = 2
gabriel@shark:~>
```

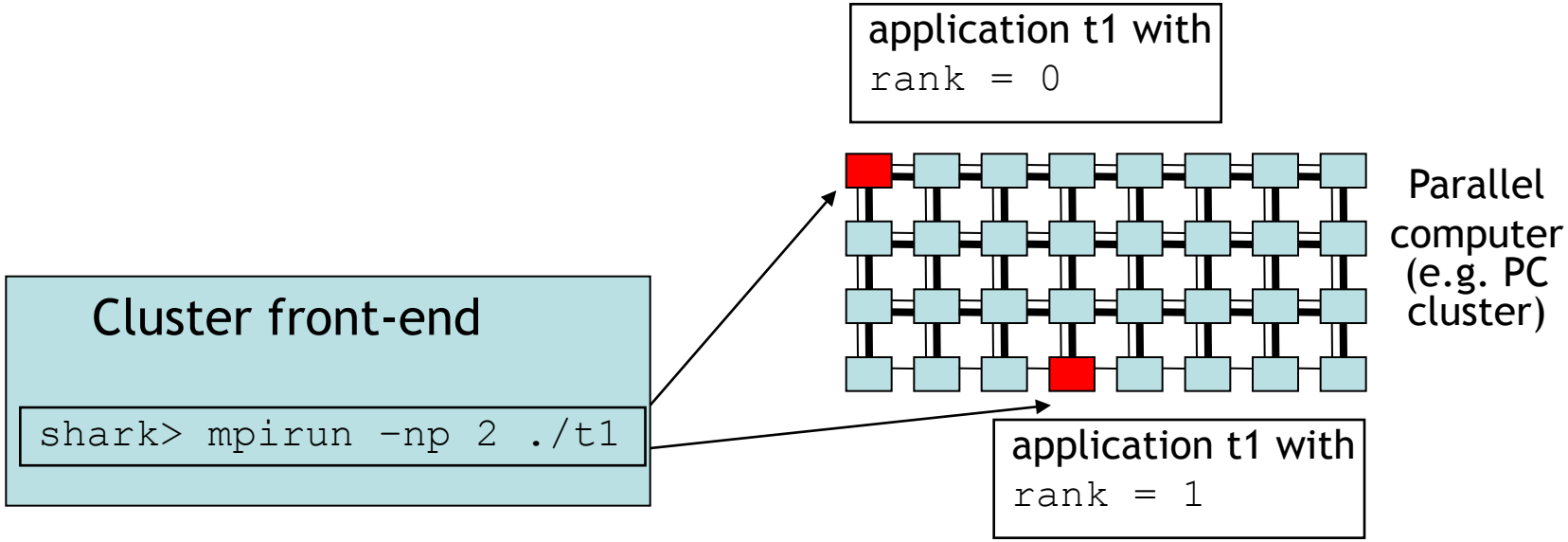
Number of processes which have been started

Rank of the 2nd process

Rank of the 1st process

Connected to shark.cs.uh.edu SSH2 - aes128-cbc - hmac-md5 - n 80x16

Simple example (II)



`mpirun` starts the application `t1`

- two times (as specified with the -np argument)
- on two currently available processors of the parallel machine
- telling one process that his `rank` is 0
- and the other that his `rank` is 1

Simple Example (III)

```
#include "mpi.h"

int main ( int argc, char **argv )
{
  int rank, size;

  MPI_Init ( &argc, &argv );
  MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
  MPI_Comm_size ( MPI_COMM_WORLD, &size );

  printf ("MPI Hello World from process %d job size %d\n",
           rank, size);

  MPI_Finalize ();
  return (0);
}
```

MPI summary (I)

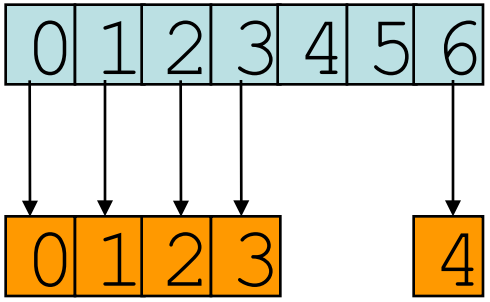
- `mpirun` starts the required number of processes
- every process has a unique identifier (`rank`) which is between `0` and `n-1`
  - no identifiers are duplicate, no identifiers are left out
- all processes which have been started by `mpirun` are organized in a process group (`communicator`) called `MPI_COMM_WORLD`
- `MPI_COMM_WORLD` is static
  - number of processes can not change
  - participating processes can not change

Ranks and process groups (II)

- The rank of a process is always related to a communicator
  - e.g. a process is only uniquely identified by a tuple (`rank, communicator`)
- A process can be part of several groups
  - i.e. a process has in each group a different rank

`MPI_COMM_WORLD, size=7`

`new communicator, size = 5`



Simple Example (IV)

Function returns the rank of a process within a communicator

Rank of a process within the communicator `MPI_COMM_WORLD`

```
---snip---

MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
MPI_Comm_size ( MPI_COMM_WORLD, &size );

---snip---
```

Default communicator containing all processes started by `mpirun`

Number of processes in the communicator `MPI_COMM_WORLD`

Function returns the size of a communicator

Simple Example (V)

Function sets up parallel environment:

- processes set up network connection to each other
- default communicator (`MPI_COMM_WORLD`) is set up
- should be the first function executed in the application

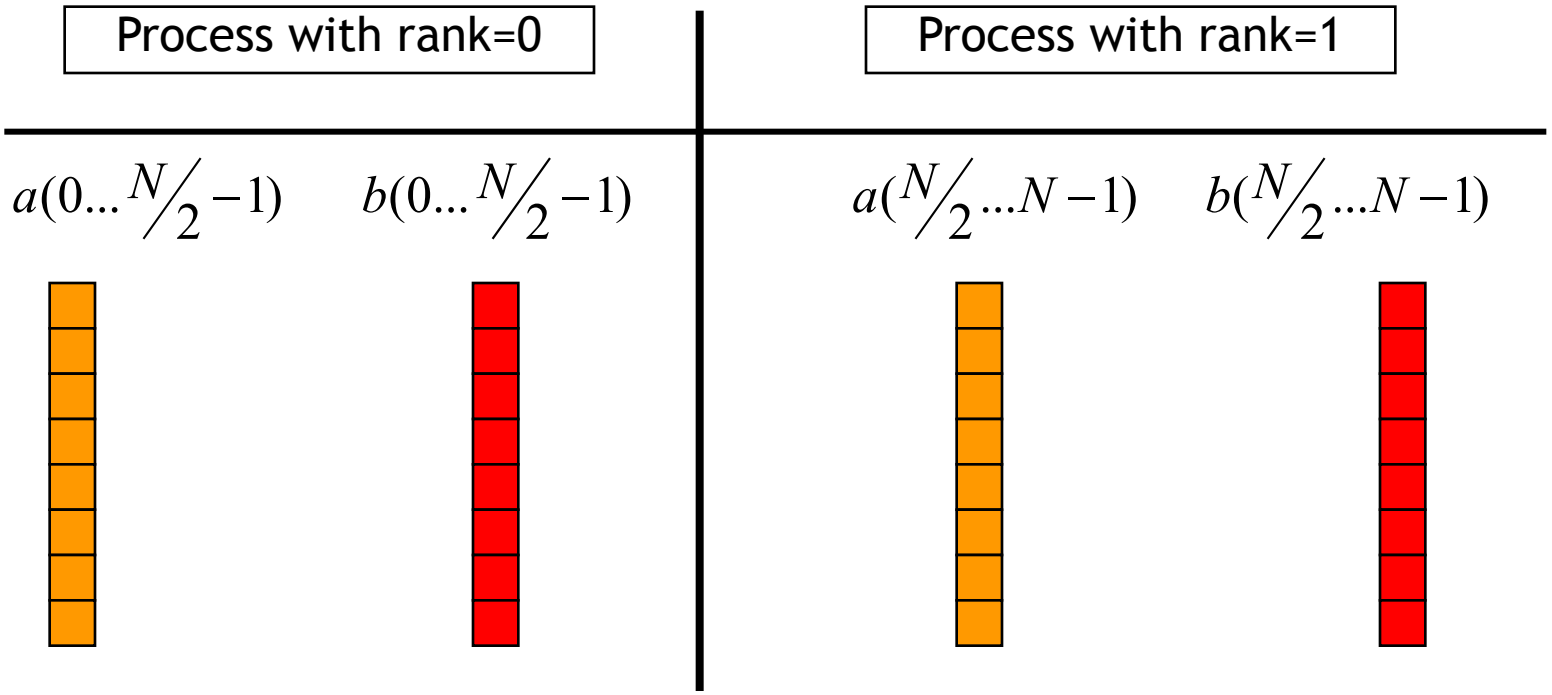
```
---snip---
MPI_Init (&argc, &argv );
---snip---
MPI_Finalize ();
---snip---
```

Function closes the parallel environment

- should be the last function called in the application
- might stop all processes

Scalar product of two vectors

- Two vectors are distributed on two processors
  - each process holds half of the overall vector



Scalar product (II)

- Logical/Global view of the data compared to local view of the data

Process with rank=0	Process with rank=1
$a(0 \ldots N/2 - 1)$	$a(N/2 \ldots N-1)$
$a_{local}(0) \Rightarrow a(0)$	$a_{local}(0) \Rightarrow a(N/2)$
$a_{local}(1) \Rightarrow a(1)$	$a_{local}(1) \Rightarrow a(N/2+1)$
$a_{local}(2) \Rightarrow a(2)$	$a_{local}(2) \Rightarrow a(N/2+2)$
$\vdots$	$\vdots$
$a_{local}(n) \Rightarrow a(N/2 - 1)$	$a_{local}(n) \Rightarrow a(N-1)$

Scalar product (III)

- Scalar product:

$$s = \sum_{i=0}^{N-1} a[i] * b[i]$$

- Parallel algorithm

$$s = \sum_{i=0}^{N/2-1} (a[i] * b[i]) + \sum_{i=N/2}^{N-1} (a[i] * b[i])$$

$$= \underbrace{\sum_{i=0}^{N/2-1} (a_{local}[i] * b_{local}[i])}_{rank=0} + \underbrace{\sum_{i=0}^{N/2-1} (a_{local}[i] * b_{local}[i])}_{rank=1}$$

  - requires communication between the processes

Scalar product (IV)

```
#include "mpi.h"

int main ( int argc, char **argv )
{
  int i, rank, size;
  double a_local[N/2], b_local[N/2];
  double s_local, s;

  MPI_Init ( &argc, &argv );
  MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
  MPI_Comm_size ( MPI_COMM_WORLD, &size );

 /* Set the values for the arrays a_local and b_local
    e.g. by reading them from a file */

  s_local = 0;
  for ( i=0; i<N/2; i++ ) {
    s_local = s_local + a_local[i] * b_local[i];
  }
```

Scalar product (V)

```
  if ( rank == 0 ) {
    /* Send the local result to rank 1 */
    MPI_Send ( &s_local, 1, MPI_DOUBLE, 1, 0,
               MPI_COMM_WORLD);
    /* Receive data from rank 0 */
    MPI_Recv ( &s, 1, MPI_DOUBLE, 1, 0,
               MPI_COMM_WORLD, &status );

  }
  if ( rank == 1 ) {
    MPI_Recv ( &s, 1, MPI_DOUBLE, 0, 0,
               MPI_COMM_WORLD, &status );
    MPI_Send ( &s_local, 1, MPI_DOUBLE, 0, 0,
               MPI_COMM_WORLD);
  }
  /* Calculate global result */
  s = s + s_local;

  MPI_Finalize ();
  return (0);
}
```

Sending Data

Data element which shall be send

Number of elements which shall be send

Data Type of the element which shall be send

```
---snip---

MPI_Send (&s_local, 1, MPI_DOUBLE, 1, 0,
          MPI_COMM_WORLD );

---snip---
```

Process group containing all processes started by `mpirun`

a user defined integer (tag) for uniquely identifying a message

Rank of processes in the process group `MPI_COMM_WORLD` to which the message shall be sent

Receiving Data

Buffer to use for receiving the data

Number of elements to be received

Data type of the element to be received

```
---snip---

MPI_Recv (&s_local, 1, MPI_DOUBLE, 0, 0,
          MPI_COMM_WORLD, &status );

---snip---
```

Process group

Status information about the message

Rank of processes in the process group which sent the message

a user defined integer (tag) for uniquely identifying a message

MPI Summary (II)

- MPI started np processes/copies of the same executable
- The same code / executable is used for all ranks
- Every process executes the entire code
  - Code sections not to be executed by every process can be excluded by using the rank of a process in if- statements

- Each process has its own address space
  - E.g. `s_local, a_local` etc. on rank 0 and on rank 1 have nothing in common

- A data item is identified in MPI through the tuple

  `(buffer pointer, count, datatype)`

Typical mistakes (I)

- Sender mismatch:
  - MPI library can recognize if source rank does not exist (e.g. `rank > size` of `MPI_COMM_WORLD`), and return an error
  - if source rank is valid (`0<rank<size` of `MPI_COMM_WORLD`) but does not send a message => `MPI_Recv` waits forever

    => deadlock

```
    if ( rank == 0 ) {
     /* Send the local result to rank 1 */
     MPI_Send ( &s_local, 1, MPI_DOUBLE, 1, 0,
                MPI_COMM_WORLD);
}
  if ( rank == 1 ) {
     MPI_Recv ( &s, 1, MPI_DOUBLE, 5, 0,
                MPI_COMM_WORLD, &status );

}
```

Typical mistakes (II)

- Tag mismatch:
  - MPI library can recognize if tag is outside of the valid range (e.g. `0<tag<MPI_TAG_UB`)
  - If tag used in `MPI_Recv` different then tag used in `MPI_Send` => `MPI_Recv` waits forever => deadlock

```
    if ( rank == 0 ) {
     /* Send the local result to rank 1 */
     MPI_Send ( &s_local, 1, MPI_DOUBLE, 1, 0,
                MPI_COMM_WORLD);
}
  if ( rank == 1 ) {
     MPI_Recv ( &s, 1, MPI_DOUBLE, 0, 18,
                MPI_COMM_WORLD, &status );

}
```

What you've learned so far

- Six functions are sufficient to write a parallel program using MPI

```
MPI_Init(int *argc, char ***argv);
MPI_Finalize ();

MPI_Comm_rank (MPI_Comm comm, int *rank);
MPI_Comm_size (MPI_Comm comm, int *size);

MPI_Send (void *buf, int count, MPI_Datatype dat,
          int dest, int tag, MPI_Comm comm);
MPI_Recv (void *buf, int count, MPI_Datatype dat,
          int source, int tag,  MPI_Comm comm,
          MPI_Status *status);
```

So, why not stop here?

- Performance
  - need functions which can fully exploit the capabilities of the hardware
  - need functions to abstract typical communication patterns

- Usability
  - need functions to simplify often recurring tasks
  - need functions to simplify the management of parallel applications

So, why not stop here?

- Performance
  - asynchronous point-to-point operations
  - collective operations
  - derived data-types
  - parallel I/O
  - hints

- Usability
  - process grouping functions
  - environmental and process management
  - error handling
  - object attributes
  - language bindings

Some Links

- MPI Forum:
  - http://www.mpi-forum.org
- Open MPI:
  - http://www.open-mpi.org
- MPICH:
  - http://www-unix.mcs.anl.gov/mpi/mpich/