Parallel Computing

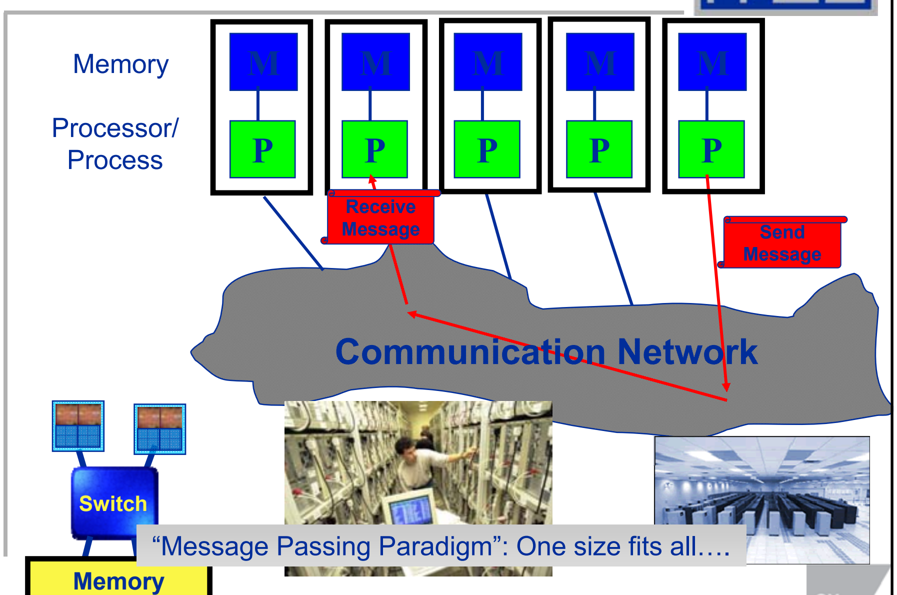
Programming Distributed Memory Architectures

Dr. Gerhard Wellein, Dr. Georg Hager
Regionales Rechenzentrum Erlangen (RRZE)

Vorlesung „Parallelrechner“
Georg-Simon-Ohm-Fachhochschule Nürnberg
03.03.-06.03.2008

Programming Distributed-Memory Architectures

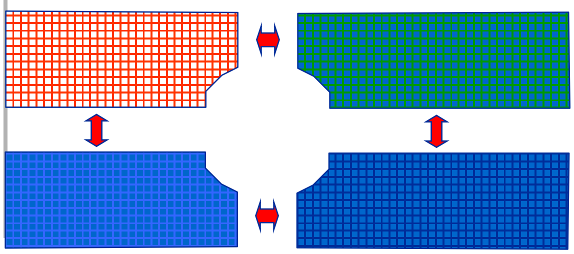
Schematic View



Programming Distributed-Memory Architectures
The Concept of Message Passing

- User explicitly distributes data
- User explicitly defines communication
- Compiler has to do no additional work

- Typically domain or work decomposition is used
- Typically communication across borders of domains is necessary



User defined communication

Programming Distributed-Memory Architectures
The Message Passing Paradigm

- The same program on each processor/machine (SPMD)
  - Restriction of the general MP model?
    → No, because processes can be distinguished by their rank (see later)
- The program is written in a sequential language (FORTRAN/C[++])

- All variables are local! → No concept of shared memory

- Data exchange between processes: Send / Receive messages via appropriate library
  - This is usually the most tedious but also the most flexible way of parallelization

- Widely accepted message passing standards:
  - **Message Passing Interface (MPI)**
  - Parallel Virtual Machine (actually MPMD) (waning importance..)

Programming Distributed-Memory Architectures
The Message Passing Paradigm

- **Messages: MP system moves data between processes**
- **MP System requires information about**
  - **Which processor is sending the message.**
  - **Where is the data on the sending processor.**
  - **What kind of data is being sent.**
  - **How much data is there.**

  - **Which processor(s) are receiving the message.**
  - **Where should the data be left on the receiving processor.**
  - **How much data is the receiving processor prepared to accept.**

Programming Distributed-Memory Architectures
MPI Basics

- MPI library (MPI-1): 127 subroutine calls
  - For basic functionality: <10 needed!
- MPI Errors:
  - C MPI routines : Return an `int` — may be ignored
  - FORTRAN MPI routines : `ierror` argument — must not be omitted!
  - Return value `MPI_SUCCESS` indicates that all went ok
  - Default: Abort parallel computation in case of other return values
- Problem: Need include files/libraries at compile/link time!
  - Most implementations provide `mpif77`, `mpif90`, `mpicc` or `mpiCC` scripts for compile and link step
  - These facilities are not standardized, so variations are to be expected, e.g. with Intel-MPI (`mpiifort`, `mpiicc` etc.).

Programming Distributed-Memory Architectures
MPI Basics - C and FORTRAN Interfaces

- Required header files:
  - C: `#include <mpi.h>`
  - FORTRAN: `include 'mpif.h'`
  - FORTRAN90: `USE MPI`
- Bindings:
  - C: `error = MPI_Xxxx(parameter,.....);`
  - FORTRAN: `call MPI_XXXX(parameter,...,ierror)`
  - MPI constants (global/common): Upper case in C
- Arrays:
  - C: indexed from 0 ⟺ FORTRAN: indexed from 1
- Here: concentrate on FORTRAN interface!
- Most frequent source of errors in C: call by reference with return values!

Programming Distributed-Memory Architectures
MPI Basics - Initialization and Finalization

- Each processor must start/terminate an MPI process
  - Usually handled automatically
  - More than one process per processor is often, but not always possible
- First call in MPI program: initialization of parallel machine!

  `call MPI_INIT(ierror)`
- Last call: shut down parallel machine!

  `call MPI_FINALIZE(ierror)`

  (Only process with rank 0 (see later) is guaranteed to return)
- **`ierror`** = integer argument for error report
- Usually: stdout/stderr of each MPI process is redirected to console where program was started (but depending on implementation)

Programming Distributed-Memory Architectures
MPI Basics - Initialization and Finalization

- Frequent source of errors: `MPI_Init()` in C
  C binding:

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

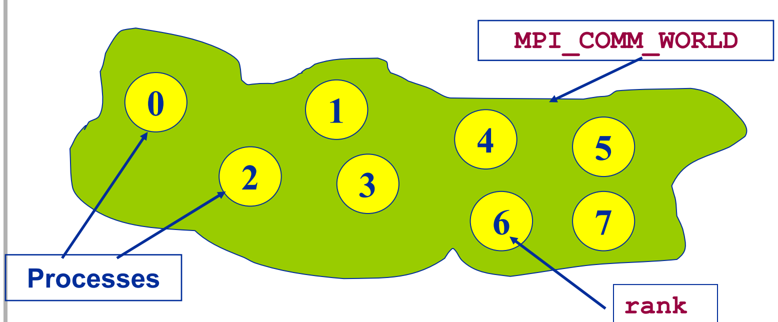
- If `MPI_Init()` is called in a function (bad idea anyway), this function must have pointers to the original data:

```
void init_all(int *argc, char***argv) {
   MPI_Init(argc, argv);
   …
}
…
   init_mpi(&argc, &argv);
```

- Depending on implementation, mistakes at this point might even go unnoticed until code is ported

Programming Distributed-Memory Architectures
MPI Basics - Communicator and Rank

- `MPI_INIT` defines "communicator" `MPI_COMM_WORLD`:



- `MPI_COMM_WORLD` defines the processes that belong to the parallel machine
- `rank` labels processes inside the parallel machine

Programming Distributed-Memory Architectures

MPI Basics - Communicator and Rank

- The rank identifies each process within the communicator (e.g. `MPI_COMM_WORLD`):
  - Get rank with `MPI_COMM_RANK`:

```
integer rank, ierror
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
```

  - `rank` = 0,1,2,…, (number of processes – 1)
- Get number of processes within `MPI_COMM_WORLD` with:

```
integer size, ierror
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
```


Programming Distributed-Memory Architectures

MPI Basics - Communicator and Rank

- `MPI_COMM_WORLD` is a global variable and required as argument for nearly all MPI calls
- `rank`
  - is target label for MPI messages
  - can define what each process should do:

```
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
…
if (rank.EQ.0)
      *** do work for rank 0 ***
else
      *** do work for other ranks ***
end if
```

Programming Distributed-Memory Architectures

MPI Basics - A Very Simple MPI Program

```
program hello
implicit none
include 'mpif.h'

integer rank, size, ierror

call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

write(*,*) 'Hello World! I am ',rank,' of ',size

call MPI_FINALIZE(ierror)
end
```


Programming Distributed-Memory Architectures

MPI Basics - A Very Simple MPI Program

- Compile:

  `mpif90 -o hello hello.f90`

- Run on 4 processors:

  `mpirun -np 4 ./hello`

- Output:

Order undefined!

```
Hello World! I am 3 of 4
Hello World! I am 1 of 4
Hello World! I am 0 of 4
Hello World! I am 2 of 4
```

Programming Distributed-Memory Architectures

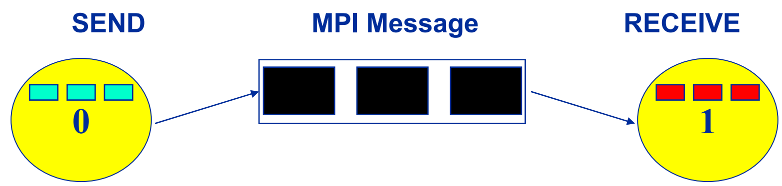
MPI Basics - Process Communication

- Communication between two processes:

  Sending / Receiving of MPI-Messages

- MPI-Message:

  Array of elements of a particular MPI datatype



- MPI datatypes:
  - Basic datatypes
  - Derived datatypes

Programming Distributed-Memory Architectures

MPI Basics - FORTRAN and C data types

MPI datatype	FORTRAN datatype
MPI_CHARACTER	CHARACTER(1)
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_BYTE	
MPI_PACKED	

MPI datatype	C datatype
MPI_CHAR / MPI_SHORT	signed char / short
MPI_INT / MPI_LONG	signed int / long
MPI_UNSIGNED_CHAR / …	unsigned char / …
MPI_FLOAT / MPI_DOUBLE	float / double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Programming Distributed-Memory Architectures

MPI Basics - Data Types

- **MPI_BYTE: Eight binary digits: do not use**
- **MPI_PACKED: can implement new data types → however, derived data types are available to built new data type at run time from basic data types**
- **Data-type matching: Same MPI data type in SEND and RECEIVE call**
  - **Data types must match on both ends in order for the communication to take place**
- **Supports heterogeneous systems/clusters**
  - **Automatic data type conversion between heterogeneous environments**

Programming Distributed-Memory Architectures

MPI Basics - Point-to-Point Communication

- **Communication between exactly two processes within the communicator**

- **Identification of source and destination by the rank within the communicator!**
- **Blocking: MPI call returns if the message to be sent or received can be modified or used …**

Programming Distributed-Memory Architectures
MPI Basics - Blocking Standard Send: MPI_SEND

- **Syntax (FORTRAN):**

  `MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)`

  - `buf`: Address of data to be sent
  - `count`: Number of elements to be sent
  - `datatype`: MPI data type of elements to be sent
  - `dest`: Rank of destination process
  - `tag`: Message marker
  - `comm`: Communicator shared by source & destination
  - `ierror`: Error code
- **Completion of `MPI_SEND`: Status of *destination* is not defined: Message may or may not have been received after return!**
- **Send buffer may be reused after `MPI_SEND` returns**

Programming Distributed-Memory Architectures
MPI Basics - MPI_SEND Example

- **Example: first 10 integers of array `field` to process #5**

```
      integer count, dest, tag, field(100)
      ...
      count=10
      dest=5
      tag=0
      call MPI_SEND(field, count, MPI_INTEGER, dest, tag,
     &                    MPI_COMM_WORLD, ierror)
```

Source and destination may coincide, but: danger of deadlocks!

Programming Distributed-Memory Architectures
MPI Basics - Blocking Receive: MPI_RECV

- `MPI_RECV`: **1) Receive data**

  2) Complete
- **Syntax (FORTRAN):**

```
MPI_RECV( buf, count, datatype, source, tag, comm,
                  status, ierror)
integer status(MPI_STATUS_SIZE)
```

  - `buf` **Size of buffer must be ≥ size of message !**
  - `count` Maximum **number of elements to receive**
  - `source, tag` **Wildcards may be used** (`MPI_ANY_SOURCE`, `MPI_ANY_TAG`)
  - `status` **Information from the message that was received (size, source, tag) (Wildcards!)**

Programming Distributed-Memory Architectures
MPI Basics - MPI_RECV Example

- **Example: receive array of REALs from any source**

```
  integer count, source, tag, status(MPI_STATUS_SIZE)
  real field(count)
  ...
  call MPI_RECV(field, count, MPI_REAL,
&               MPI_ANY_SOURCE, MPI_ANY_TAG,
&               MPI_COMM_WORLD, status, ierror)
  write(*,*) 'Received from #', status(MPI_SOURCE),
&             ' with tag ', status(MPI_TAG)
```

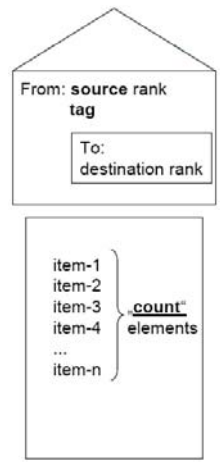
Get actual number of received items with `MPI_GET_COUNT`:

```
MPI_GET_COUNT(status, datatype, count, ierror)
```

Programming Distributed-Memory Architectures
MPI Basics: Requirements for Point-to-Point Comm.

For a communication to succeed:

- Sender must specify a valid destination.
- Receiver must specify a valid source rank (or `MPI_ANY_SOURCE`).
- Communicator must be the same (e.g. `MPI_COMM_WORLD`).
- Tags must match.
- Message data types must match.
- Receiver's buffer must be large enough.



Programming Distributed-Memory Architectures
MPI Basics: Summary

- **Beginner's MPI procedure toolbox:**
  - `MPI_INIT` **let's get going**
  - `MPI_COMM_SIZE` **how many are we?**
  - `MPI_COMM_RANK` **who am I?**
  - `MPI_SEND` **send data to someone else**
  - `MPI_RECV` **receive data from some-/anyone**
  - `MPI_GET_COUNT` **how much have I received?**
  - `MPI_FINALIZE` **finish off**
- **Standard send/receive calls provide most simple way of point-to-point communication**
- **Send/receive buffer may safely be reused after the call has completed**
- **`MPI_SEND` has to have a specific target/tag, `MPI_RECV` does not**

Programming Distributed-Memory Architectures
MPI Basics: First Complete Example

- **Task: Write parallel program in which a master process („root“) collects some data (e.g. numbers to sum up) from the others**

```
program collect
  implicit none
  include 'mpif.h'
  int i,size,rank,ierror,status(MPI_STATUS_SIZE)
  int number,sum

  call MPI_INIT(ierror)
  call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierror)

  if(rank.eq.0) then
      sum=0
      call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierror)
```


Programming Distributed-Memory Architectures
MPI Basics: First Complete Example

```
      do i=1,size-1
         call MPI_RECV(number,1,MPI_INTEGER, &
         MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD, &
         status,ierror)
         sum=sum+number
      enddo
      write(*,*) 'The sum is ',sum
  else
      call MPI_SEND(rank,1,MPI_INTEGER,0,0, &
         MPI_COMM_WORLD,ierror)
  endif
  call MPI_FINALIZE(ierror)
end
```

Blocking Point-to-Point Communication in MPI

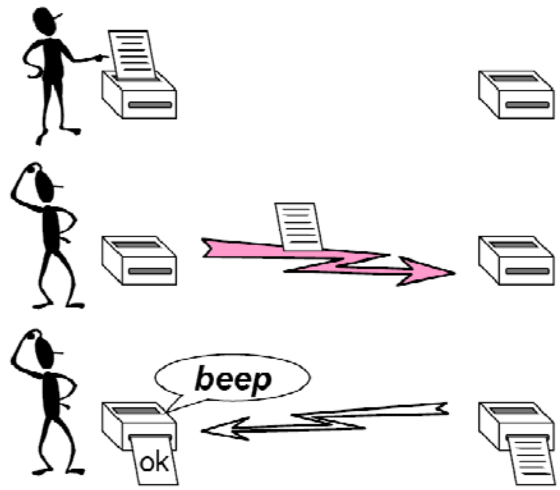
Programming Distributed-Memory Architectures
Blocking Point-to-Point Communication

- **"Point-to-Point communication"**
  - **One process sends a message to another, i.e. communication between exactly** two **processes**
  - **Two types of point-to-point communication: Synchronous send vs. buffered = asynchronous send**
- **"Blocking"**
  - **Operations are local activities on the sending and receiving processes - may block one processes until partner process acts:**
    - **Synchronous send operation blocks until receive is posted**
    - **Asynchronous send blocks until message can be changed on sender process**
    - **Receive operation blocks until message is sent**
  - **After a blocking subroutine returns, you may change the buffer without changing the message to be sent**

Programming Distributed-Memory Architectures
Blocking Point-To-Point Comm.: *Synchronous Send*

„Sending process“

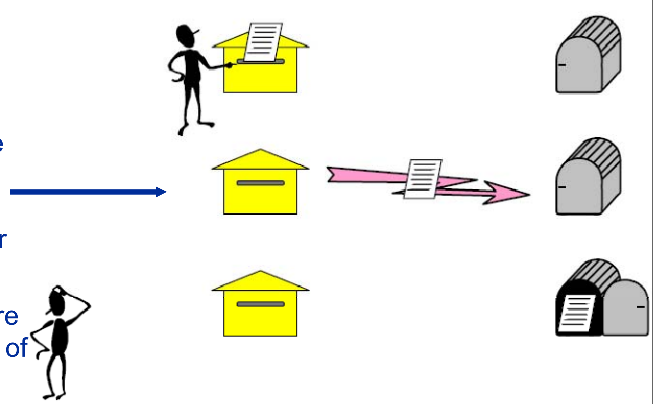
- The sender gets an information that the message is received.
- Analogue to the *beep* or *okay-sheet* of a fax.



Programming Distributed-Memory Architectures
Blocking Point-To-Point Comm.: *Asynchronous Send*

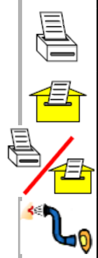
„Sending process“

- One only knows when the message has left
- Message to be sent is put in a separate (system) buffer
- No need to care about the time of delivery.



Programming Distributed-Memory Architectures
Point-to-Point Communication: Blocking Communication

- **Completion of send/receive ↔ buffer can safely be reused!**



Communication mode	Completion condition	MPI Routine (Blocking)
Synchronous Send	**Only completes when the receive has started.**	`MPI_SSEND`
Buffered Send	*Always completes, irrespective of the receive process.*	*`MPI_BSEND`*
Standard Send	**Either synchronous or buffered.**	`MPI_SEND`
Ready Send	*Always completes, irrespective whether the receive has completed.*	*`MPI_RSEND`* DO NOT USE
Receive	**Completes when a message has arrived.**	`MPI_RECV`

Programming Distributed-Memory Architectures
Point-to-Point Communication: MPI_SSEND

- **`MPI_SSEND` completes after message has been accepted by the destination ("handshaking").**
- **Synchronization of source and destination!**
- **Predictable and safe behavior!**
- **`MPI_SSEND` should be used for debugging purposes!**
- **Problems:**
  - **Performance (high latency, risk of serialization – best bandwidth)**
  - **Deadlock situations (see later)**
- **Syntax (FORTRAN): same as `MPI_SEND`**

```
MPI_SSEND( buf, count, datatype, dest, tag, comm,
   ierror)
```

Programming Distributed-Memory Architectures

Point-to-Point Communication: Example - Deadlocks

- **Example with 2 processes, each sending a message to the other:**

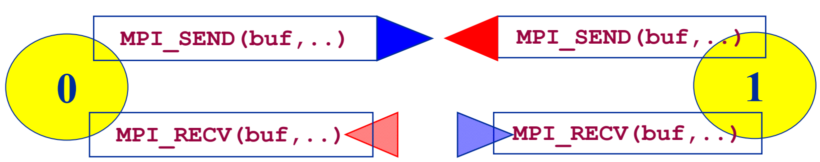
```
  integer buf(200000)
  if(rank.EQ.0) then
         dest   =1
         source =1
  else if(rank.EQ.1) then
         dest   =0
         source =0
  end if
  MPI_SEND(buf, 200000, MPI_INTEGER, dest, 0,
&               MPI_COMM_WORLD, ierror)
  MPI_RECV(buf, 200000, MPI_INTEGER, source, 0,
&               MPI_COMM_WORLD, status, ierror)
```

This program will not work correctly on all systems!

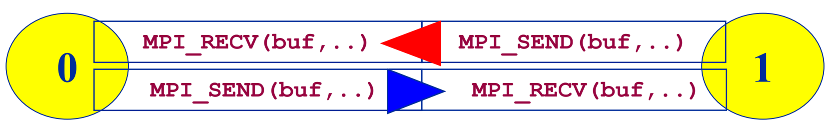
Programming Distributed-Memory Architectures

Point-to-Point Communication: Example - Deadlocks

- **Deadlock: Some of the outstanding blocking communication cannot be completed (program stalls)**
- **Example: `MPI_SEND` is implemented as synchronous send for large messages!**



One remedy: reorder send/receive pair on one process (e.g. rank 0):

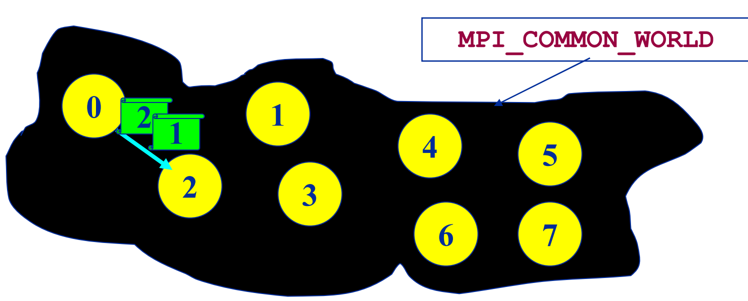


Programming Distributed-Memory Architectures
Point-to-Point Communication: Example - Deadlocks

```
integer buf(200000),buf_tmp(200000)
if(rank.EQ.0) then
       dest=1
       source=1
       MPI_SEND(buf, 200000, MPI_INTEGER, dest, 0,
     &                MPI_COMM_WORLD, ierror)
       MPI_RECV(buf, 200000, MPI_INTEGER, source, 0,
     &                MPI_COMM_WORLD, status, ierror)
else if (rank.EQ.1) then
       dest=0
       source=0
       MPI_RECV(buf_tmp, 200000, MPI_INTEGER, source, 0,
     &                MPI_COMM_WORLD, status, ierror)
       MPI_SEND(buf, 200000, MPI_INTEGER, dest, 0,
     &                MPI_COMM_WORLD, ierror)
       buf=buf_tmp
end if
```

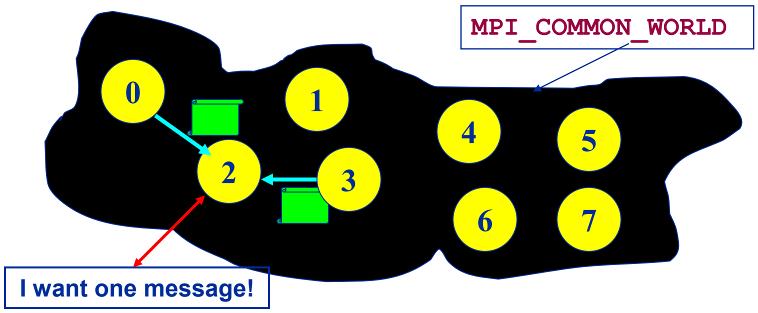

Programming Distributed-Memory Architectures
Point-to-Point Communication: Semantics

- **Deadlocks are always introduced by the programmer!**
- **MPI semantics guarantees progress for standard compliant programs**
- **Semantics: Rules, guaranteed by MPI implementations**
  - **Message Order Preservation (within same communicator)**



Programming Distributed-Memory Architectures
Point-to-Point Communication: Semantics

- **Progress: It is not possible for a matching send and receive pair to remain permanently outstanding.**
  - **Matching means:** data types, tags **and** receivers **match**

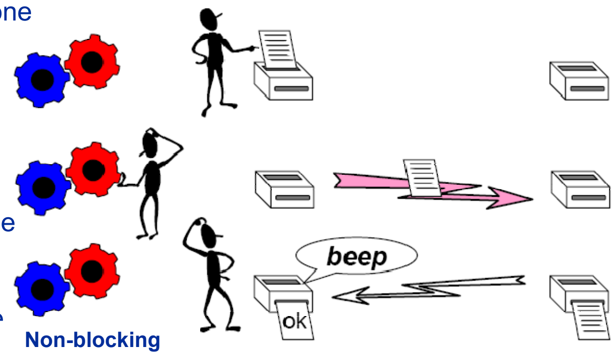


Non-Blocking Point-to-Point Communication in MPI

Programming Distributed-Memory Architectures
Non-Blocking Point-To-Point Communication: Basics

Idea of Non-Blocking Communication:
Overlap communication & work and enhance flexibility

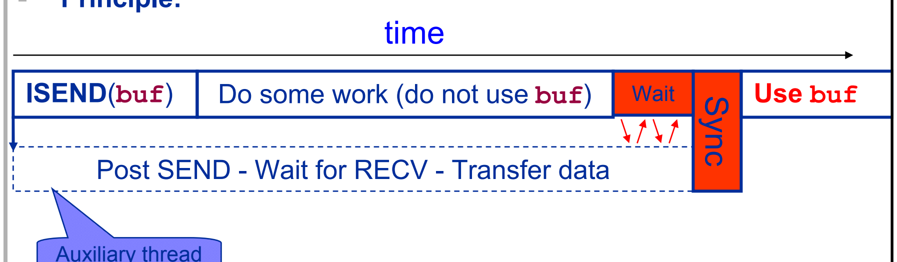
- After initiating the communication one can return to perform other work.
- At some later time one must ***test*** or ***wait*** for the completion of the non-blocking operation.



Non-blocking synchronous send

Programming Distributed-Memory Architectures
Non-Blocking Point-to-Point Communication: Basics

- **Motivation:**
  - **Avoid deadlocks**
  - **Avoid idle processors**
  - **Avoid useless synchronization**
  - **Overlap communication and useful work (hide the 'communication cost')**
- **Principle:**



Programming Distributed-Memory Architectures
Non-Blocking Point-to-Point Communication: Basics

- Detailed steps for non-blocking communication

1) Setup communication operation (MPI)
2) Build unique request handle (MPI)
3) Return request handle and control to user program (MPI)
4) User program continues while MPI system performs communication (asynchronously)
5) Status of communication can be probed by the request handle

All non-blocking operations must have matching wait (or test) operations as some system or application resources can be freed only when the non-blocking operation is completed.

Programming Distributed-Memory Architectures
Non-Blocking Point-to-Point Communication: Basics

- The return of non-blocking communication call does not imply completion of the communication
- Check for completion: Use request handle !
- Do not reuse buffer until completion of communication has been checked !
- Data transfer can be overlapped with user program execution (if supported by hardware)
- Blocking send matches a non-blocking receive and vice-versa!

Programming Distributed-Memory Architectures
Non-Blocking Point-to-Point Comm.: MPI_ISEND/IRECV

- **Standard non-blocking send**

```
MPI_ISEND(sendbuf, count, datatype, dest, tag,
          comm, request, ierror)
```

`request`: integer argument as request handle

  - Do not reuse `sendbuf` before `MPI_Isend` has been completed!

- **Standard non-blocking receive**

```
MPI_IRECV(recvbuf, count, datatype, source, tag,
          comm, request, ierror)
```

  - Do not reuse `recvbuf` before `MPI_Irecv` has been completed!
  - No **status** array necessary – will be used in `MPI_WAIT/MPI_TEST`

Programming Distributed-Memory Architectures
Non-Blocking Point-to-Point Comm.: Test for Completion

- **Test one communication for completion – basic calls:**

```
MPI_WAIT( request, status, ierror);

MPI_TEST( request, flag, status, ierror);
```

Parameter:

- `request`: request handle
- `status`: status object (cf. MPI_RECV)
- `flag`: `logical` to test for success

Programming Distributed-Memory Architectures
Non-Blocking Point-to-Point Communication: Example

- **Example: 2 processes, each sending a message to the other:**

```
integer buf(200000), buf_tmp(200000)
if(rank.EQ.0) then
       dest=1
       source=1
else if(rank.EQ.1) then
       dest=0
       source=0
end if
MPI_ISEND(buf, 200000, MPI_INTEGER, dest, 0,
&                 MPI_COMM_WORLD, REQUEST, ierror)
MPI_RECV(buf_tmp, 200000, MPI_INTEGER, source, 0,
&               MPI_COMM_WORLD, status, ierror)
MPI_WAIT(REQUEST , STATUS, ierror)
buf=buf_tmp
```


Programming Distributed-Memory Architectures
Non-Blocking Point-to-Point Comm.: Others

- **Communication models for non-blocking communication**

Non-Blocking Operation	MPI call
Standard send	MPI_ISEND()
Synchronous send	MPI_ISSEND()
Buffered send	MPI_IBSEND()
Ready send	MPI_IRSEND() DO NOT USE
Receive	MPI_IRECV()

Collective Communication in MPI

Programming Distributed-Memory Architectures

Collective Communication: Introduction

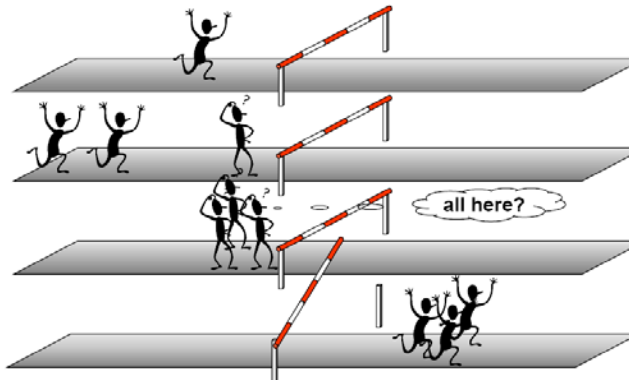
Collective communication always involves <u>every</u> process in the specified communicator

- **Features:**
  - **All processes must call the subroutine**
    Remarks:
    - **All processes must call the subroutine!**
    - **All processes must call the subroutine!!**
  - **Always blocking: buffer can be reused after return**
  - **May or may not synchronize the processes**
  - Cannot interfere with point-to-point communication
  - **Datatype matching**
  - **No tags**
  - **Sent message** must **fill receive buffer (count is exact)**
- Can be “built” out of point-to-point communications by hand, however, collective communication may allow optimized internal implementations, e.g., tree based algorithms

Programming Distributed-Memory Architectures

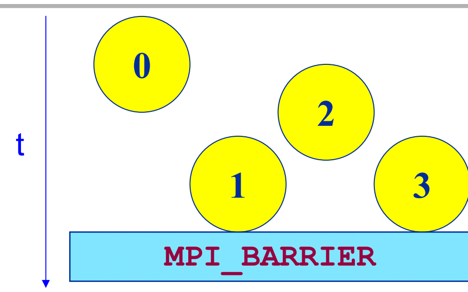
Collective Communication: Barriers

Synchronize processes (MPI_BARRIER):
At this point of the runtime all processes have to wait until the last one reaches a barrier



Programming Distributed-Memory Architectures

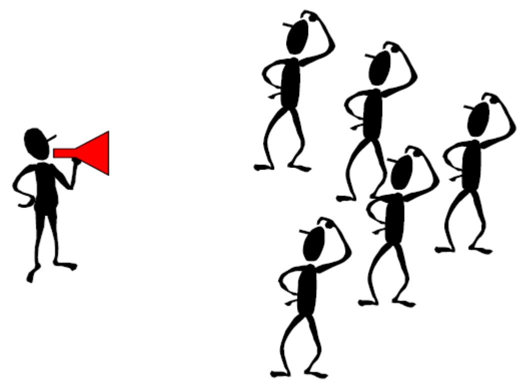
Collective Communication: Synchronization



- **Syntax:**
  `MPI_BARRIER(comm, ierror)`
- `MPI_BARRIER` **blocks the calling process until all other group members (=processes) have called it.**
- `MPI_BARRIER` **is normally never needed – all synchronization is done automatically by the data communication – however: debugging, profiling, …**

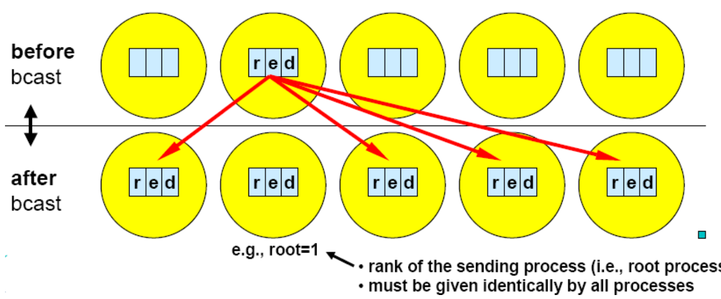
Programming Distributed-Memory Architectures
Collective Communication: Broadcast

BROADCAST (MPI_BCAST): A one-to-many communication.



Programming Distributed-Memory Architectures
Collective Communication: Broadcast

- **Every process receives one copy of the message from a root process**



Syntax

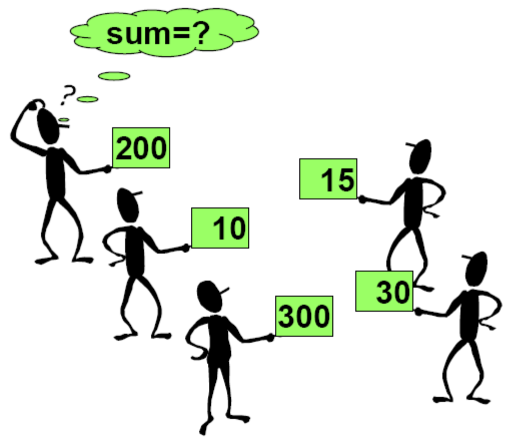
```
MPI_BCAST(buffer, count, datatype, root, comm,
ierror)
```

(e.g.: root = 0, but there is no "default" root process)

Programming Distributed-Memory Architectures
Collective Communication: Reduction Operations

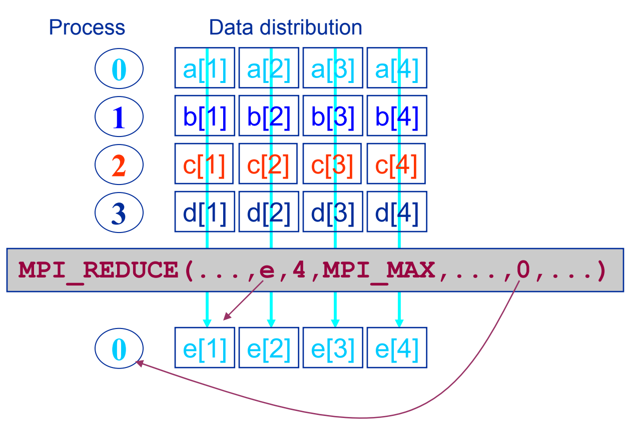
REDUCTION (MPI_REDUCE):

Combine data from several processes to produce a single result.



Programming Distributed-Memory Architectures
Collective Communication: Reduction Operations

Compute `e(i) = max{ a(i),b(i),c(i),d(i)}`
`i=1,2,3,4`



Programming Distributed-Memory Architectures
Collective Communication: Reduction Operations

- **Results stored on root process**

  `MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)`

- **Result in recvbuf on root process.**
- **Status of recvbuf on other processes is undefined.**
- **count > 1: Perform operations on all 'count' elements of an array**

If results should be stored on all processes:

- **`MPI_ALLREDUCE`: No root argument**
  - **Combination of `MPI_REDUCE` and `MPI_BCAST`**

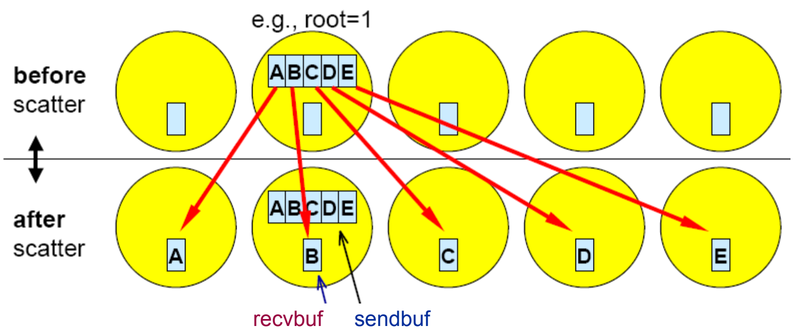
Programming Distributed-Memory Architectures
Collective Communication: Reduction Operations

Predefined operations in MPI

Name	Operation	Name	Operation
MPI_SUM	Sum	MPI_PROD	Product
MPI_MAX	Maximum	MPI_MIN	Minimum
MPI_LAND	Logical AND	MPI_BAND	Bit-AND
MPI_LOR	Logical OR	MPI_BOR	Bit-OR
MPI_LXOR	Logical XOR	MPI_BXOR	Bit-XOR
MPI_MAXLOC	Maximum+ Position	MPI_MINLOC	Minimum+ Position

Programming Distributed-Memory Architectures
Collective Communication: Scatter

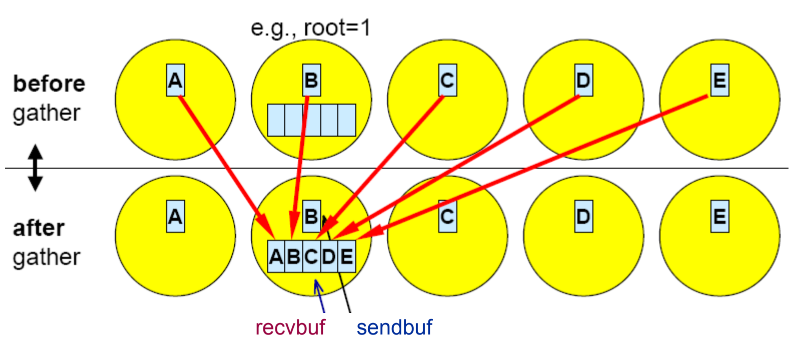
- Root process scatters data to all processes



- Specify root process (cf. example : root=1)
- send and receive details are different
- SCATTER: send-arguments significant only for root process

Programming Distributed-Memory Architectures
Collective Communication: Gather

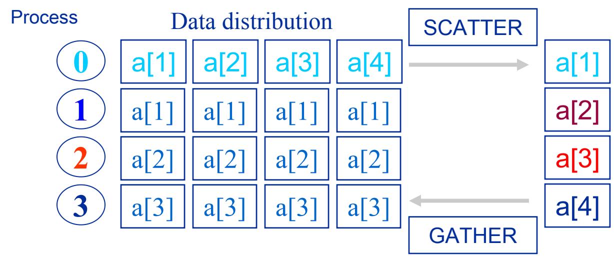
- Root process gathers data from all processes



- Specify root process (cf. example : root=1)
- send and receive details are different
- GATHER: receive-arguments significant only for root process

Programming Distributed-Memory Architectures
Collective Communication: Gather/Scatter

Gather / Scatter operations:
Root process scatters/gathers data to/from all processes



- **Specify root process (cf. example : root=0)**
- **send and receive details are different**
- **GATHER: recv-arguments significant only for root process**
- **SCATTER: send-arguments significant only for root process**

Programming Distributed-Memory Architectures
Collective Communication: Gather/Scatter

- **Gather:**
  ```
  MPI_GATHER( sendbuf, sendcount, sendtype,
  recvbuf, recvcount, recvtype, root, comm,
  ierror)
  ```
- **Each process sends sendbuf to root process**
- **root process receives messages and stores them in rank order**
- **In general: recvcount = sendcount**
- **recvbuf is ignored for all non-root processes**

Programming Distributed-Memory Architectures
Collective Communication: Gather/Scatter

- **Scatter:**
```
MPI_SCATTER( sendbuf, sendcount, sendtype,
recvbuf, recvcount, recvtype,
root, comm, ierror)
```
- **root process sends the *i-th*. segment of sendbuf to the *i-th.* process**
- **In general: recvcount = sendcount**
- **sendbuf is ignored for all non-root processes**

Programming Distributed-Memory Architectures
MPI Basics: Communication modes

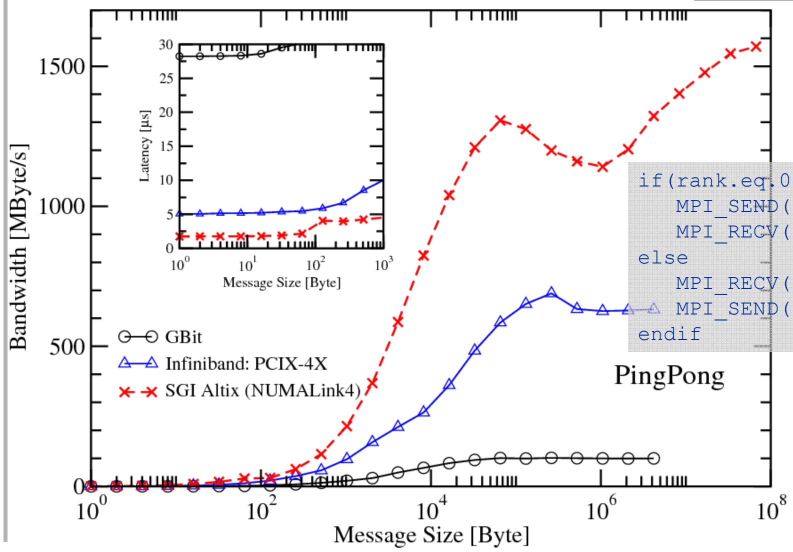
Can be mixed, e.g. MPI_Send can be matched by appropriate MPI_Irecv(…)		Point to Point	Collective

	Blocking	`MPI_SEND(mybuf….)`
		`MPI_SSEND(mybuf…)`
		`MPI_BSEND(mybuf…)`
		`MPI_RECV(mybuf…)`
		(`mybuf` can be modified after call returns)
			`MPI_BARRIER(…)`
			`MPI_BCAST(…)`
			`MPI_ALLREDUCE(…)`
			(All processes of the communicator must call the operation!)

	Non-blocking	`MPI_ISEND(mybuf…)`
		`MPI_IRECV(mybuf…)`
		(`mybuf` must not be modified after call returns – requires additional check for completion e.g.:MPI_Wait/Test
			--- 	

Programming Distributed-Memory Architectures

Basic Performance Characteristic of Networks




```
if(rank.eq.0) then
   MPI_SEND()
   MPI_RECV()
else
   MPI_RECV()
   MPI_SEND()
endif
```


Literature & Links

- MPICH Implementation available at:
  http://www-unix.mcs.anl.gov/mpi/mpich1/
- OpenMPI implementation available at:
  http://www.open-mpi.org/
- Full standard definition and more useful information:
  http://www.mpi-forum.org/
- W. Gropp, E. Lusk, A. Skjellum:
  Using MPI - Portable Parallel Programming with the Message-Passing Interface , MIT Press, 1994/1999.

MPI Exercise

MPI Exercise:
Matrix-Vector Multiply

- **Dense matrix vector multiply:**
  - **Common operation with eigenproblems**
- **Mathematically:**

$$c_i = c_i + \sum_j A_{ij} r_j \quad (i,j=1,...,n\_dim)$$

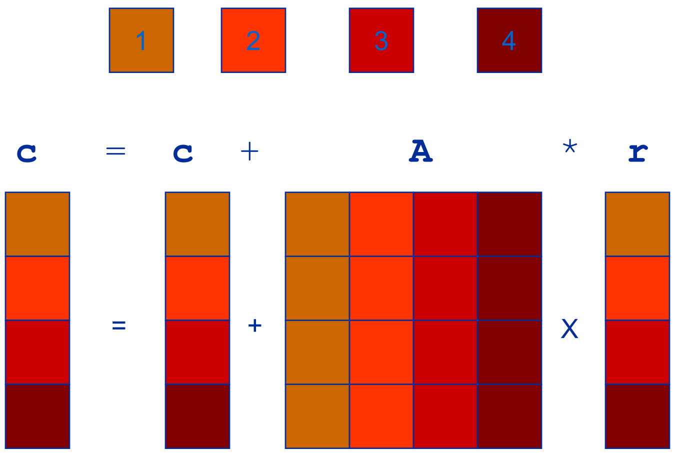
- **Serial code:**

```
do i = 1 , n_dim
   do j = 1 , n_dim
      c( i ) = c( i ) + A( i , j ) * r( j )
   enddo
enddo
```

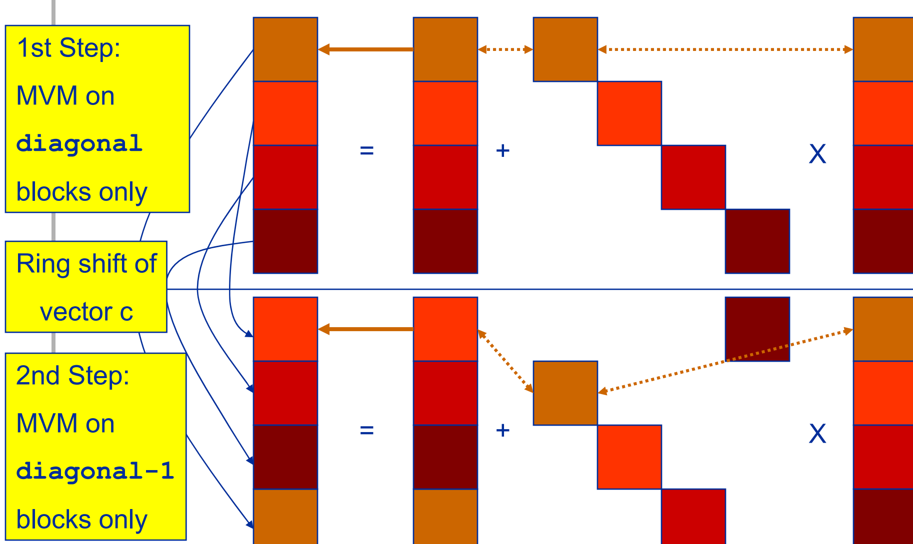
- **No reference to RISC optimizations here...**
- **Exercise: Implement parallel dense MVM**

MPI Exercise:
Matrix-Vector Multiply

- **Distribution of matrix and vector among the processors**



MPI Exercise:
Matrix-Vector Multiply: MPI Parallelization



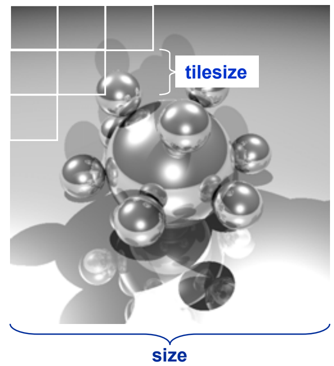
MPI Exercise:
Matrix-Vector Multiply: MPI Parallelization

- After 4 (np) steps:
  - the total MVM has been computed
  - the distribution of vector `c` to the processors has been restored
  - Vector `c` has been communicated np times
- Communication step (blocking):
  - Ring-shift with, e.g., `MPI_SEND/MPI_RECV`
- Communication step (non-blocking):
  - Idea: overlap communication and computation
  - Spend an additional temporary vector for asynchronous data transfer
  - Use non-blocking communication calls
  - Initialize next communication step before computation and check for completion afterwards
  - Start with `diagonal-1`; end with `diagonal` calculation

Using Performance Tools for MPI

Example: Parallel Ray Tracer

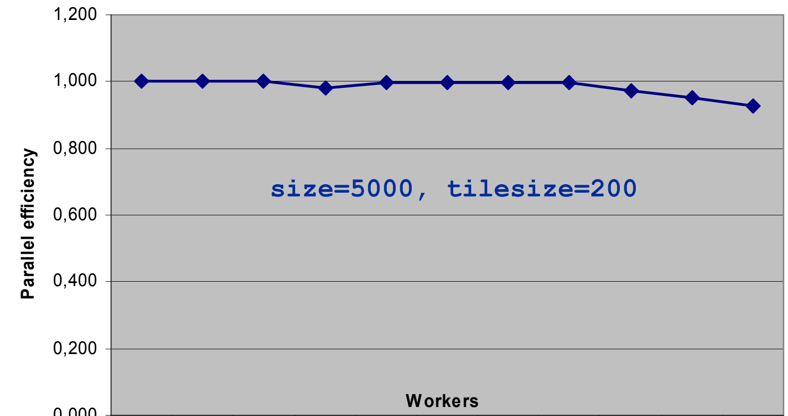
- Raytracing is an "embarrassingly parallel" task
- Each pixel is drawn by sending a "beam" through the scene and calculating its colour value
- All pixels are independent of each other
- Picture is divided into tiles which are distributed dynamically among the MPI processes
- "Master-Worker" scheme



Example: Parallel Ray Tracer
Pseudocode

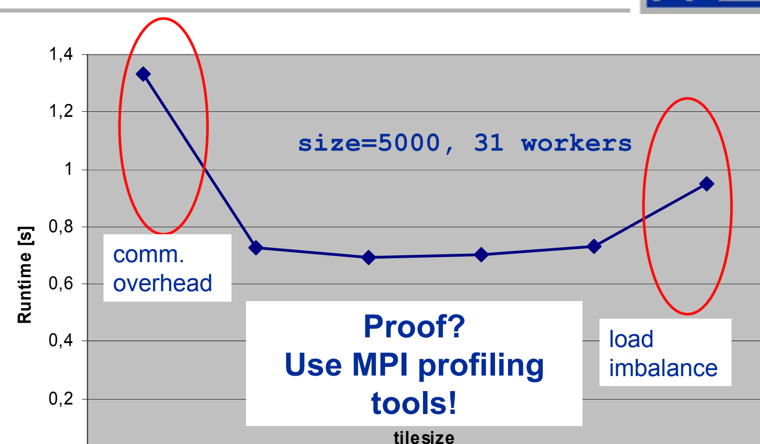
```
mpi_comm_rank(MPI_COMM_WORLD, &id);
if(id==0) {     // I am the master
  while(tiles_to_receive != 0) {
    … wait for anyone to send "ready" message …
    … store finished tile (if any) && tiles_to_receicve-- …
    if(tiles_to_send != 0)
      … send new tile coordinates to worker …
      tiles_to_send--
    else
      … send "finish" message to worker …

  }
} else {         // I am a worker
  … send tile request to master …
  while(1) {
    … receive tile coordinates …
    if(finish_received) break
    calculate_tile()
    … send tile data to master …
  }
}
```

Example:
Parallel ray tracing scalability



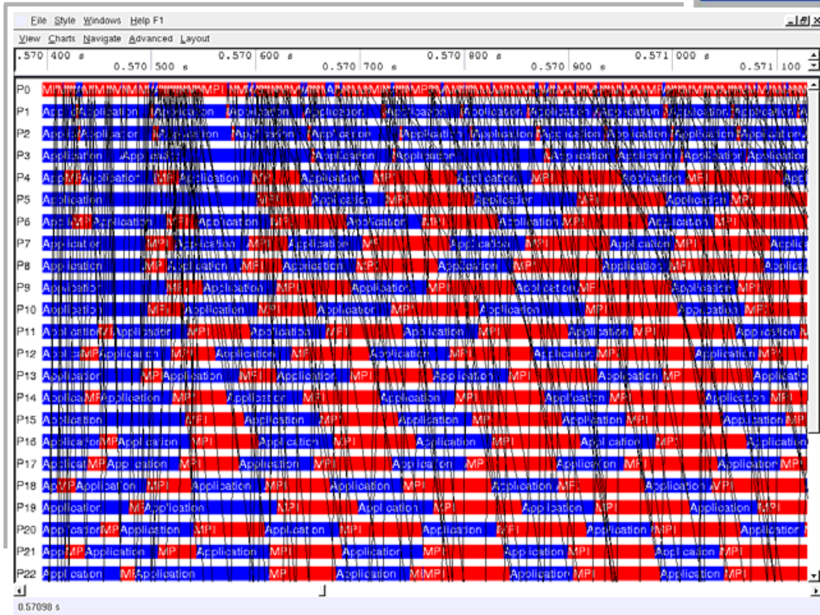
Workers	1	2	3	4	5	6	7	8	16	23	31
Reihe1	1,000	1,002	1,000	0,979	0,996	0,998	0,998	0,996	0,971	0,951	0,926

Example:
Parallel ray tracing: Influence of granularity

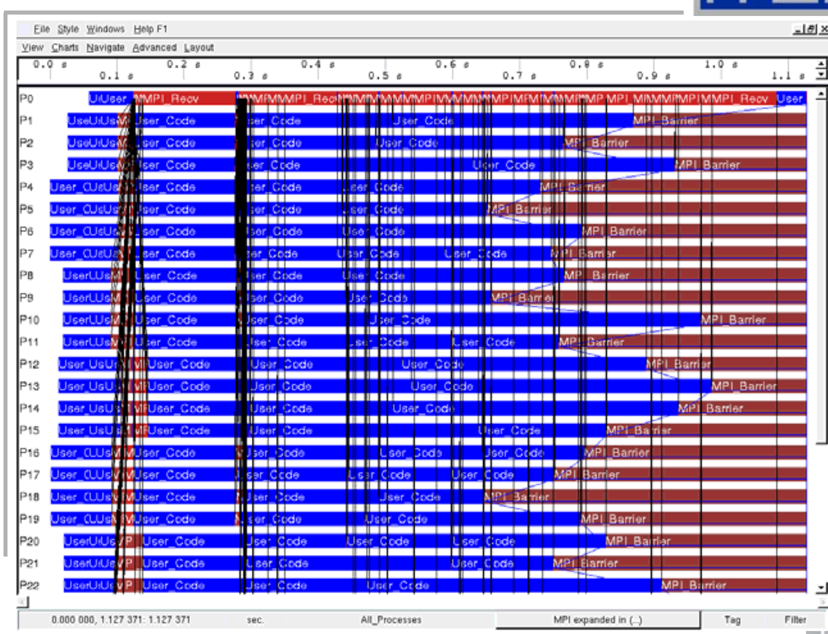


tilesize	10	20	40	100	200	500
Reihe1	1,33	0,725	0,692	0,702	0,73	0,95

Example: Parallel Ray Tracing
MPI event timeline (tilesize=10) (Intel Trace Analyzer)



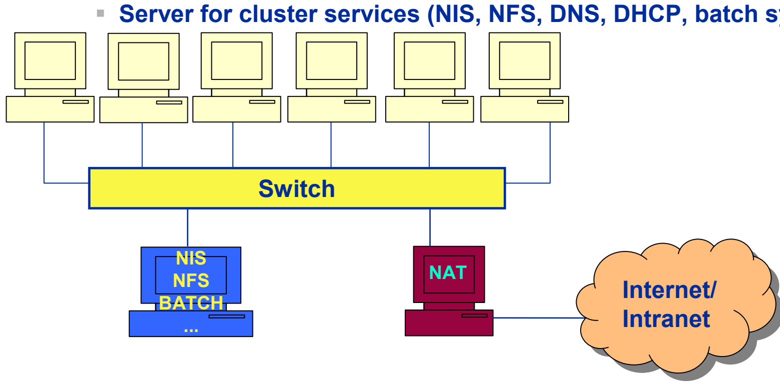
Example:Parallel Ray Tracing
MPI event timeline (tilesize=500)



Some Hints for Building a Compute Cluster

Clusterbuilder Hints
Hardware Components

- **Hardware Setup**
  - **Compute nodes: PCs with (at least) Ethernet**
  - **Switch (preferably non-blocking)**
  - **Network setup with NAT (easy SW updates)**
    - **"Head node" is gateway to internet / rest of intranet**
    - **Server for cluster services (NIS, NFS, DNS, DHCP, batch system)**



Clusterbuilder Hints
Software Components

- All systems: Linux/UNIX OS
- All systems are NFS clients
  - NIS-directed automounter
  - $HOME for all users on common NFS
- Compute nodes: Batch system daemon (Torque-MOM)
- Frontend/headnode
  - Batch system client commands (Torque clients)
  - Development SW (compilers, MPI, libs, tools)
  - NAT
- Server
  - Batch system server/scheduler (Torque)
  - NFS server
  - NIS server
  - DHCP server
  - DNS server/slave
  - Ganglia Monitoring Suite

Clusterbuilder Hints
Software Components

- Non-standard software:
- Compilers (GNU gcc/g++/g77/gfortran or Intel or…)
- MPI
  - Free implementation MPICH: `http://www-unix.mcs.anl.gov/mpi/`
  - `./configure` for use with compiler of your choice
  - Install static libs on frontend, dynamic libs (if built) on nodes
  - "make install" also installs MPI compiler scripts (mpicc…)
  - Might want to consider Pete Wyckoff's mpiexec for program startup `http://www.osc.edu/~pw/mpiexec/index.php`
  - MPI requires a node list (or file) to find the nodes to run processes on
    - batch system selects nodes automatically

Clusterbuilder Hints
Software Components

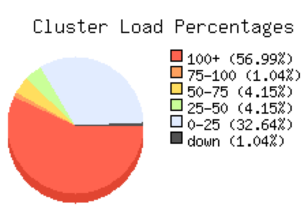
- **Batch system**
  - **Torque: Terascale Open-Source Resource and QUEue Manager**
    http://www.clusterresources.com/pages/products/torque-resource-manager.php
  - **Torque comes with a simple standard scheduler**
  - **Client commands (qsub, qstat, …), server (pbs_server), MOM (pbs_mom) and scheduler (pbs_sched) can be built separately**
  - **Server and scheduler go to server node, clients go to headnode, MOM goes to all compute nodes**
  - **Torque requires node file with list of nodes and properties:**

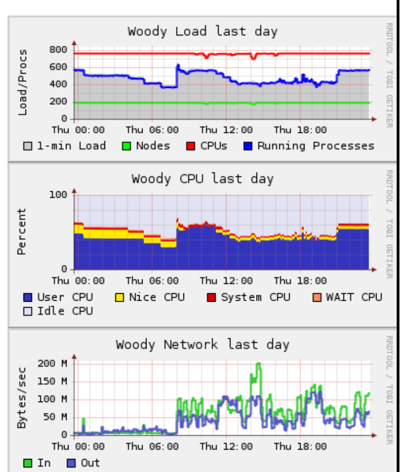
```
w0101 np=4 rack01 ib
w0102 np=4 rack01 ib
w0103 np=4 rack01 ib
w0104 np=4 rack01 ib
```

  - **Torque controls health state of all nodes**

Clusterbuilder Hints
Software Components

- **Ganglia Monitoring System**
  http://ganglia.sourceforge.net
- **Stores anv visualizes many metrics, global and node-based**
- **Highly configurable**
- **Integrates Torque**
  - **Job data**
  - **Job history**





Clusterbuilder Hints
Production Quality Clusters

- **For compute center quality of service, some elements have to be added**
  - **Cooling**
  - **Failure monitoring: Nodes and services going down must lead to admin notifications**
  - **Accounting: Who has drawn how much CPU time over some period?**
  - **Regular updates: Scheduled downtimes**
  - **Tools: Parallel debuggers, profilers**
  - **Documentation for users**

Clusterbuilder Hints
Links & References

- **Bauke & Mertens: *Cluster Computing*. ISBN 978-3540422990, Springer, Berlin, 2005**
- **ROCKS cluster package: `http://www.rocksclusters.org`**
- **Intel web pages on High Performance Computing: `http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/hpc/index.htm`**
- **Building clusters the easy way with OSCAR: `http://www.intel.com/cd/ids/developer/asmo-na/eng/66785.htm`**
- **Thomas Hofmann: *High Performance Computing Labor an der FH Nürnberg*. Systemdokumentation (on request)**