Getting started with MPI

Parallel Programming with the Message Passing Interface

Carsten Kutzner, 12.10.2006

compiling & running • most useful commands • parallelization concepts • performance monitoring • MPI resources

What is MPI?

- a library, not a language
  - specifies names, parameters and results of functions / subroutines to be called from C, C++, and Fortran programs
  - programs that use MPI are compiled with ordinary compilers (gcc, icc) and linked with the MPI library
- MPI standard defined in 1994
  - MPI-1 core functionality
- MPI-2 add-ons 1995-97
  - remote memory operations
  - parallel I/O
  - dynamic process management
- MPI-3 will address future needs
  - interoperability among MPI implementations?
  - non-blocking collectives?
  - ...

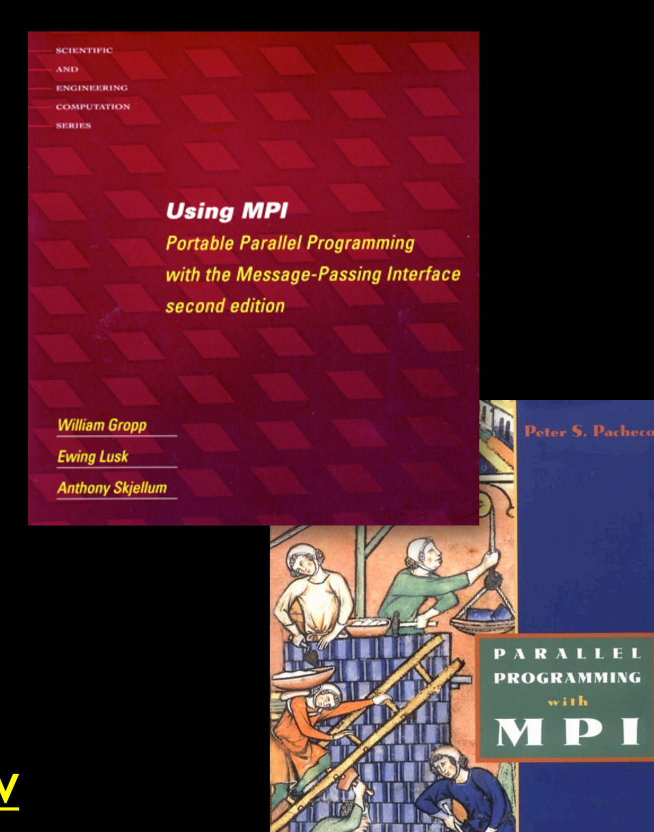
Why use MPI?

- high performance
  - today used from 1 – 131072 processors
- flexible
- portable

	MPI	**Cluster OpenMP**	**OpenMP**	**threads**
shared memory	yes	yes	yes	yes
distributed memory	yes	yes	not possible	not possible
performance	++	+/o	+	+
ease of use	o	+/o	+	o

MPI Resources

- Books
  - Pacheco 1997: *Parallel Programming with MPI*
  - Gropp, Lusk, Skjellum 1999: *Using MPI*
  - Gropp, Lusk, Thakur 1999: *Using MPI-2*
- On the web
  - the MPI forum: www-unix.mcs.anl.gov/mpi/ supplies the MPI standards as PS, PDF
  - web pages for MPI and MPE: www.mcs.anl.gov/mpi/www (multiple mirrors)
  - MPICH (MPI CHameleon): www-unix.mcs.anl.gov/mpi/mpich2
  - LAM/MPI (Local Area Multicomputer): www.lam-mpi.org
  - OpenMPI: www.open-mpi.org
  - en.wikipedia.org/wiki/Message_Passing_Interface
  - www.linux-mag.com/extreme/
  - newsgroup comp.parallel.mpi



Getting started with LAM

- edit your .bashrc
  - on your workstation:
    export PATH=/usr/local/lam-7.0.4/bin:$PATH
    export LAMHOME=/usr/local/lam-7.0.4
  - on the clusters (e.g. coral4) export
    /usr/local/Cluster-Apps/lam-7.1.1-64/
- compile ...
  - mpicc hellompi.c -o hellompi.x
  - mpicc – a compiler wrapper around gcc/icc/*cc that automatically links your code to the mpi library
  - mpif77, mpif90 – for fortran
  - mpic++ / mpiCC – for c++
- ... and run
  - lamboot
  - mpirun -np 2 ./hellompi.x

Getting started with LAM

- on multiple nodes a hostfile is needed:
  - lamboot -v bhost

```
node01 cpu=2
node27 cpu=2
node07 cpu=1
node09 cpu=1
```

- tests can be done with any number of processes even on your workstation!
  - lamboot
  - mpirun -np 17 ./hellompi.x

Getting started with LAM

- you can also use our network of workstations:
  - you will need a homedir on each workstation with a .bashrc reading
    ```
    export PATH=/usr/local/lam-7.0.4/bin:$PATH
    export LAMHOME=/usr/local/lam-7.0.4
    ```
  - you must be able to login via ssh without password:
    execute ssh-keygen (if you not have already) on your machine
    ```
    cd .ssh
    scp id_dsa.pub you@remotehost:.ssh/authorized_keys
    ```
  - ```
    export LAMRSH="ssh"
    lamboot -v bhost
    ```

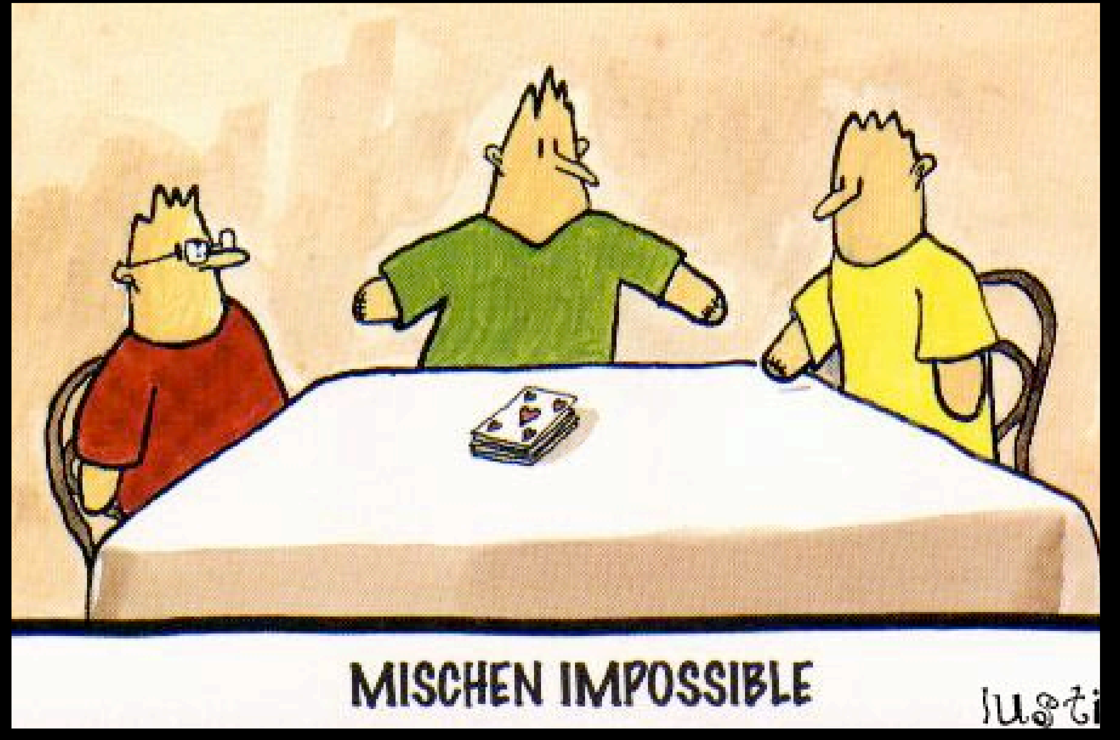
    ```
    wes         cpu=2
    platypus    cpu=2
    tivoli      cpu=2
    ```

    ```
    ckutzne@wes:~> lamnodes
    n0    wes.mpibpc.gwdg.de:2:origin,this_node
    n1    platypus.mpibpc.gwdg.de:2:
    n2    tivoli.mpibpc.gwdg.de:2:
    ```
  - the executable has to reside on all of the workstations in the same directory, e.g. /netmount/coral4/you/
    ```

Do’s and don’ts

- never mix LAM versions!!!
  - LAM–A-compiled program might run on LAM–B, but nothing is guaranteed!
- recompile when you move your program from A to B

Cluster	LAM v.	arch
tivoli	7.0.4	32 bit
orca1–2	7.0.6	32 bit
coral1–4	7.1.1	64 bit
beluga1–3	7.1.1	64 bit
kea1–3	7.1.1–2	64 bit
slomo	7.1.1	32 bit



MPI commands & keywords

MPIO_Request_c2f
MPIO_Request_f2c
MPIO_Test
MPIO_Wait
MPI_Abort
MPI_Address
MPI_Allgather
MPI_Allgatherv
MPI_Allreduce
MPI_Alltoall
MPI_Alltoallv
MPI_Attr_delete
MPI_Attr_get
MPI_Attr_put
MPI_Barrier
MPI_Bcast
MPI_Bsend
MPI_Bsend_init
MPI_Buffer_attach
MPI_Buffer_detach
MPI_CHAR
MPI_Cancel
MPI_Cart_coords
MPI_Cart_create
MPI_Cart_get
MPI_Cart_map
MPI_Cart_rank
MPI_Cart_shift
MPI_Cart_sub
MPI_Cartdim_get
MPI_Comm_compare
MPI_Comm_create
MPI_Comm_dup
MPI_Comm_free
MPI_Comm_get_name
MPI_Comm_group
MPI_Comm_rank
MPI_Comm_remote_group
MPI_Comm_remote_size
MPI_Comm_set_name
MPI_Comm_size
MPI_Comm_split
MPI_Comm_test_inter
MPI_DUP_FN

MPI_Dims_create
MPI_Errhandler_create
MPI_Errhandler_free
MPI_Errhandler_get
MPI_Errhandler_set
MPI_Error_class
MPI_Error_string
MPI_File_c2f
MPI_File_close
MPI_File_delete
MPI_File_f2c
MPI_File_get_amode
MPI_File_get_atomicity
MPI_File_get_byte_offset
MPI_File_get_errhandler
MPI_File_get_group
MPI_File_get_info
MPI_File_get_position
MPI_File_get_position_shared
MPI_File_get_size
MPI_File_get_type_extent
MPI_File_get_view
MPI_File_iread
MPI_File_iread_at
MPI_File_iread_shared
MPI_File_iwrite
MPI_File_iwrite_at
MPI_File_iwrite_shared
MPI_File_open
MPI_File_open
MPI_File_preallocate
MPI_File_read
MPI_File_read_all
MPI_File_read_all_begin
MPI_File_read_all_end
MPI_File_read_at
MPI_File_read_at_all
MPI_File_read_at_all_begin
MPI_File_read_at_all_end
MPI_File_read_ordered
MPI_File_read_ordered_begin
MPI_File_read_ordered_end
MPI_File_read_shared
MPI_File_seek

MPI_File_seek_shared
MPI_File_set_atomicity
MPI_File_set_errhandler
MPI_File_set_info
MPI_File_set_size
MPI_File_set_view
MPI_File_sync
MPI_File_write
MPI_File_write_all
MPI_File_write_all_begin
MPI_File_write_all_end
MPI_File_write_at
MPI_File_write_at_all
MPI_File_write_at_all_begin
MPI_File_write_at_all_end
MPI_File_write_ordered
MPI_File_write_ordered_begin
MPI_File_write_ordered_end
MPI_File_write_shared
MPI_Finalize
MPI_Finalized
MPI_Gather
MPI_Gatherv
MPI_Get_count
MPI_Get_elements
MPI_Get_processor_name
MPI_Get_version
MPI_Graph_create
MPI_Graph_get
MPI_Graph_map
MPI_Graph_neighbors
MPI_Graph_neighbors_count
MPI_Graphdims_get
MPI_Group_compare
MPI_Group_difference
MPI_Group_excl
MPI_Group_free
MPI_Group_incl
MPI_Group_intersection
MPI_Group_range_excl
MPI_Group_range_incl
MPI_Group_rank
MPI_Group_size
MPI_Group_translate_ranks

MPI_Group_union
MPI_Ibsend
MPI_Info_c2f
MPI_Info_create
MPI_Info_delete
MPI_Info_dup
MPI_Info_f2c
MPI_Info_free
MPI_Info_get
MPI_Info_get_nkeys
MPI_Info_get_nthkey
MPI_Info_get_valuelen
MPI_Info_set
MPI_Init
MPI_Init_thread
MPI_Init_thread
MPI_Initialized
MPI_Int2handle
MPI_Intercomm_create
MPI_Intercomm_merge
MPI_Iprobe
MPI_Irecv
MPI_Irsend
MPI_Isend
MPI_Issend
MPI_Keyval_create
MPI_Keyval_free
MPI_NULL_COPY_FN
MPI_NULL_DELETE_FN
MPI_Op_create
MPI_Op_free
MPI_Pack
MPI_Pack_size
MPI_Pcontrol
MPI_Probe
MPI_Recv
MPI_Recv_init
MPI_Reduce
MPI_Reduce_scatter
MPI_Request_c2f
MPI_Request_free
MPI_Rsend
MPI_Rsend_init
MPI_Scan

MPI_Scatter
MPI_Scatterv
MPI_Send
MPI_Send_init
MPI_Sendrecv
MPI_Sendrecv_replace
MPI_Ssend
MPI_Ssend_init
MPI_Start
MPI_Startall
MPI_Status_c2f
MPI_Status_set_cancelled
MPI_Status_set_elements
MPI_Test
MPI_Test_cancelled
MPI_Testall
MPI_Testany
MPI_Testsome
MPI_Topo_test
MPI_Type_commit
MPI_Type_contiguous
MPI_Type_create_darray
MPI_Type_create_indexed_block
MPI_Type_create_subarray
MPI_Type_extent
MPI_Type_free
MPI_Type_get_contents
MPI_Type_get_envelope
MPI_Type_hindexed
MPI_Type_hvector
MPI_Type_indexed
MPI_Type_lb
MPI_Type_size
MPI_Type_struct
MPI_Type_ub
MPI_Type_vector
MPI_Unpack
MPI_Wait
MPI_Waitall
MPI_Waitany
MPI_Waitsome
MPI_Wtick
MPI_Wtime

MPI_Something

What you need to start

- Minimum subset to start with
  - MPI_Init – Initialize MPI
  - MPI_Comm_size – How many processes are there?
  - MPI_Comm_rank – My process number
  - MPI_Send – Send a message
  - MPI_Recv – Receive a message
  - MPI_Finalize – Close MPI universe
- Collective communication (powerful)
  - MPI_Bcast – Broadcast a variable to all processes
  - MPI_Reduce – Add up a variable across all processes
  - MPI_Alltoall – Complete communication across all processes
  - MPI_Barrier – Synchronize all processes
  - …

Initializing & quitting

Fortran:

```
      PROGRAM hello

      INCLUDE 'mpif.h'
      INTEGER err

      CALL MPI_INIT(err)
      PRINT *, "Hello world!"
      CALL MPI_FINALIZE(err)

      END
```

C:

```
#include <stdio.h>
#include <mpi.h>

main (int argc, char *argv[])
{
  int err;
  err = MPI_Init(&argc, &argv);
  printf("Hello world!\n");
  err = MPI_Finalize();
}
```

- a header file (mpi.h / mpif.h) has to be included which contains the MPI definitions and function prototypes
- MPI routines return an error code indicating whether or not they run successfully.

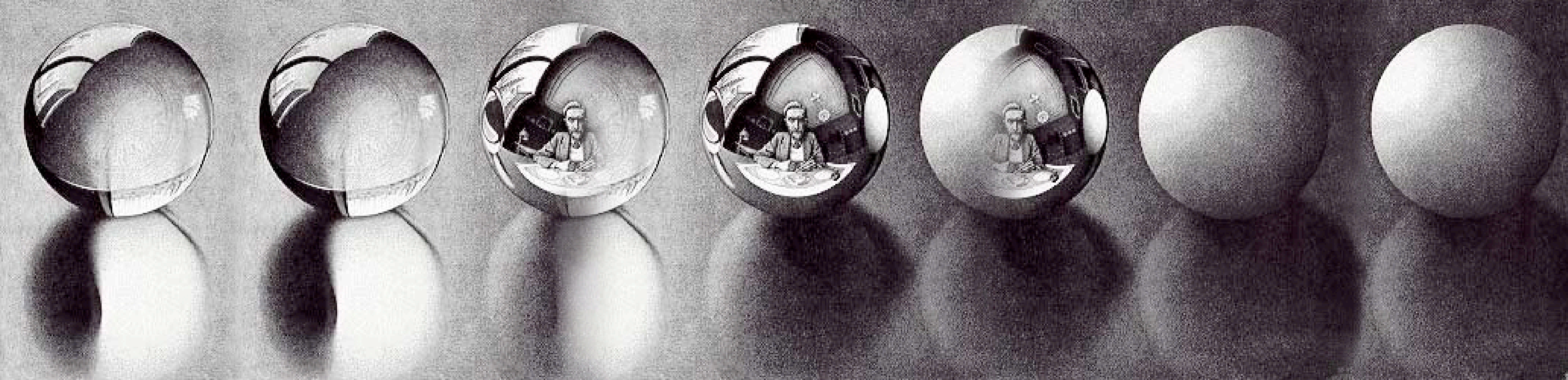
```
if (err == MPI_SUCCESS)
{
  … /* routine ran correctly */
}
else
{
}
```

- output

```
> mpirun -np 4 ./helloworld.x
Hello world!
Hello world!
Hello world!
Hello world!
```

Try to think parallel

- on mpirun -np N each of the N processes runs one copy of your code
- each variable is duplicated N times and may have different values on the different processes
- if you want to check the value of variables with printf, always output the rank with each print statement!



Finding out who I am

```
#include <stdio.h>
#include <mpi.h>

int gmx_setup(int *nnodes)
{
  int    resultlen;
  int    mpi_num_nodes;
  int    mpi_my_rank;
  char   mpi_hostname[MPI_MAX_PROCESSOR_NAME];

  MPI_Comm_size(MPI_COMM_WORLD, &mpi_num_nodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &mpi_my_rank);
  MPI_Get_processor_name(mpi_hostname, &resultlen);

  fprintf(stderr,"NNODES=%d, MYRANK=%d, HOSTNAME=%s\n",
          mpi_num_nodes,mpi_my_rank,mpi_hostname);

  *nnodes=mpi_num_nodes;

  return mpi_my_rank;
}
```

```
int main(int argc, char *argv[])
{
  int nnodes, nodeid;

  MPI_Init(&argc,&argv);
  nodeid = gmx_setup(&nnodes);
  MPI_Finalize();
  return 0;
}
```

```
> mpirun -np 6 ./gmxsetup.x
NNODES=6, MYRANK=2, HOSTNAME=node02
NNODES=6, MYRANK=0, HOSTNAME=node01
NNODES=6, MYRANK=1, HOSTNAME=node01
NNODES=6, MYRANK=4, HOSTNAME=node12
NNODES=6, MYRANK=3, HOSTNAME=node02
NNODES=6, MYRANK=5, HOSTNAME=node12
>
```

Sending and receiving data

- int MPI_Send(void *buf, int count, MPI_Datatype dtype,
  int dest, int tag, MPI_Comm comm);
  - body – read `count` elements of datatype `dtype` from memory address `buf`.
  - envelope: – send this message to the process with rank `dest` in comm and label it `tag`.
  - error code
- int MPI_Recv(void *buf, int count, MPI_Datatype dtype,
  int source, int tag, MPI_Comm comm,
  MPI_Status *status);
  - body – write count elements of datatype dtype to memory address buf.
  - envelope – only accept a message tagged `tag` from process with rank `source` in communicator `comm`.
  - error code

Sending and receiving data

```
/* simple send and receive */
#include <stdio.h>
#include <mpi.h>

main (int argc, char **argv)
{
  int        myrank;
  MPI_Status status;
  double     a[100];

  MPI_Init(&argc, &argv);                        /* Initialize MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);        /* Get rank */

  if( myrank == 0 )                              /* Send a message */
    MPI_Send(a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD);
  else if( myrank == 1 )                      /* Receive a message */
    MPI_Recv(a, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD,
             &status);

  MPI_Finalize();                                /* Terminate MPI */
}
```

MPI datatypes

MPI datatype	**C type**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	(none)

Sending stuff to everyone

p0 → p1 → p2 → … → p(N-2) → p(N-1)

```
int main(int argc, char *argv[])
{
  int nnodes, nodeid, right, left;
  float test=0.0;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  nodeid = gmx_setup(&nnodes);
  gmx_left_right(nnodes, nodeid, &left, &right);

  if (nodeid == 0)
    test = 3.1415;

  if (nodeid != 0)
    MPI_Recv(&test, 1, MPI_FLOAT, left, 0, MPI_COMM_WORLD,
             &status);
  if (nodeid != (nnodes-1))
    MPI_Send(&test, 1, MPI_FLOAT, right, 0, MPI_COMM_WORLD);

  MPI_Finalize();
}
```

```
void
gmx_left_right(int nnodes,int nodeid,
int *left,int *right)
{
  *left  = (nnodes+nodeid-1) % nnodes;
  *right = (nodeid       +1) % nnodes;
}
```

Printf output from all procs

```
ckutzne@coral4:~/mpi-techtea> mpirun -np 5 ./gmxsetup.x
NNODES=5, MYRANK=0, HOSTNAME=coral4
NNODES=5, MYRANK=1, HOSTNAME=coral4
Nodeid: 0, value of test 3.141500
Nodeid: 0, value of test 3.141500
Nodeid: 1, value of test 0.000000
Nodeid: 1, value of test 3.141500
NNODES=5, MYRANK=2, HOSTNAME=coral4
NNODES=5, MYRANK=3, HOSTNAME=coral4
NNODES=5, MYRANK=4, HOSTNAME=coral4
Nodeid: 2, value of test 0.000000
Nodeid: 2, value of test 3.141500
Nodeid: 3, value of test 0.000000
Nodeid: 3, value of test 3.141500
Nodeid: 4, value of test 0.000000
Nodeid: 4, value of test 3.141500
```

```
int main(int argc, char *argv[])
{
  int nnodes, nodeid, right, left;
  float test=0.0;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  nodeid = gmx_setup(&nnodes);
  gmx_left_right(nnodes, nodeid, &left, &right);

  if (nodeid == 0)
    test = 3.1415;

  printf("Nodeid: %d, value of test: %d\n", nodeid, test);

  if (nodeid != 0)
    MPI_Recv(&test, 1, MPI_FLOAT, left, 0, MPI_COMM_WORLD,
             &status);
  if (nodeid != (nnodes-1))
    MPI_Send(&test, 1, MPI_FLOAT, right, 0, MPI_COMM_WORLD);

  printf("Nodeid: %d, value of test: %d\n", nodeid, test);
  MPI_Finalize();
}
```

Printf output from all procs

```
ckutzne@coral4:~/mpi-techtea> mpirun -np 5 ./gmxsetup.x
NNODES=5, MYRANK=0, HOSTNAME=coral4
NNODES=5, MYRANK=2, HOSTNAME=coral4
NNODES=5, MYRANK=1, HOSTNAME=coral4
NNODES=5, MYRANK=3, HOSTNAME=coral4
NNODES=5, MYRANK=4, HOSTNAME=coral4
Nodeid: 0, value of test 3.141500
Nodeid: 2, value of test 0.000000
Nodeid: 1, value of test 0.000000
Nodeid: 4, value of test 0.000000
Nodeid: 3, value of test 0.000000
Nodeid: 0, value of test 3.141500
Nodeid: 2, value of test 3.141500
Nodeid: 1, value of test 3.141500
Nodeid: 3, value of test 3.141500
Nodeid: 4, value of test 3.141500
```

```
int main(int argc, char *argv[])
{
  int nnodes, nodeid, right, left;
  float test=0.0;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  nodeid = gmx_setup(&nnodes);
  gmx_left_right(nnodes, nodeid, &left, &right);

  if (nodeid == 0)
    test = 3.1415;
  MPI_Barrier(MPI_COMM_WORLD);
  printf("Nodeid: %d, value of test: %d\n", nodeid, test);

  if (nodeid != 0)
    MPI_Recv(&test, 1, MPI_FLOAT, left, 0, MPI_COMM_WORLD,
             &status);
  if (nodeid != (nnodes-1))
    MPI_Send(&test, 1, MPI_FLOAT, right, 0, MPI_COMM_WORLD);
  MPI_Barrier(MPI_COMM_WORLD);
  printf("Nodeid: %d, value of test: %d\n", nodeid, test);
  MPI_Finalize();
}
```

Master-slave scheme

MPI_Send

Master

1
2
3

```
MPI_Recv(data, …, MPI_ANY_TAG, …, status)
if (status->MPI_TAG == 1)
{
  do_work;
  MPI_Send(results back to master);
}
```

Master waits for any message

```
MPI_Recv(…,
  MPI_ANY_SOURCE,
  MPI_ANY_TAG, …,
  status)
```

Master

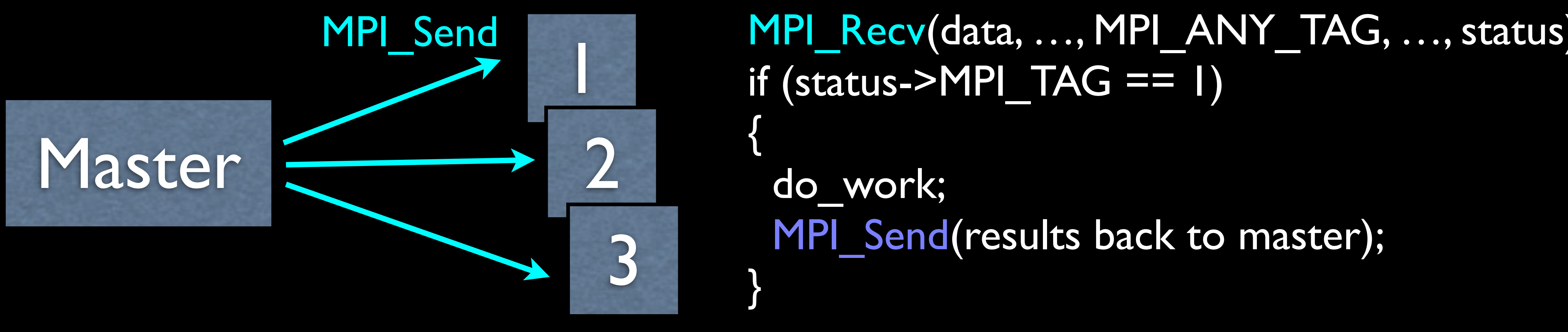
MPI_Recv

1
2
3

accumulates the data …

- status needed when using wildcards
  - status -> MPI_SOURCE
  - status -> MPI_TAG

Master-slave scheme



Master waits for any message
MPI_Recv(…,
MPI_ANY_SOURCE,
MPI_ANY_TAG, …,
status)

Master

MPI_Recv

1
2
3

accumulates the data …

Pitfall: deadlock

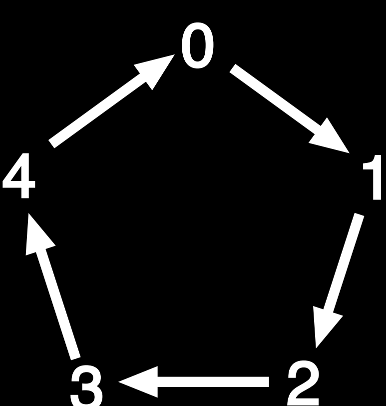
- MPI_Send and MPI_Recv block, so deadlock may occur

Time	Process 0	Process 1
1	MPI_Send to 1	MPI_Send to 0
2	MPI_Recv from 1	MPI_Recv from 0

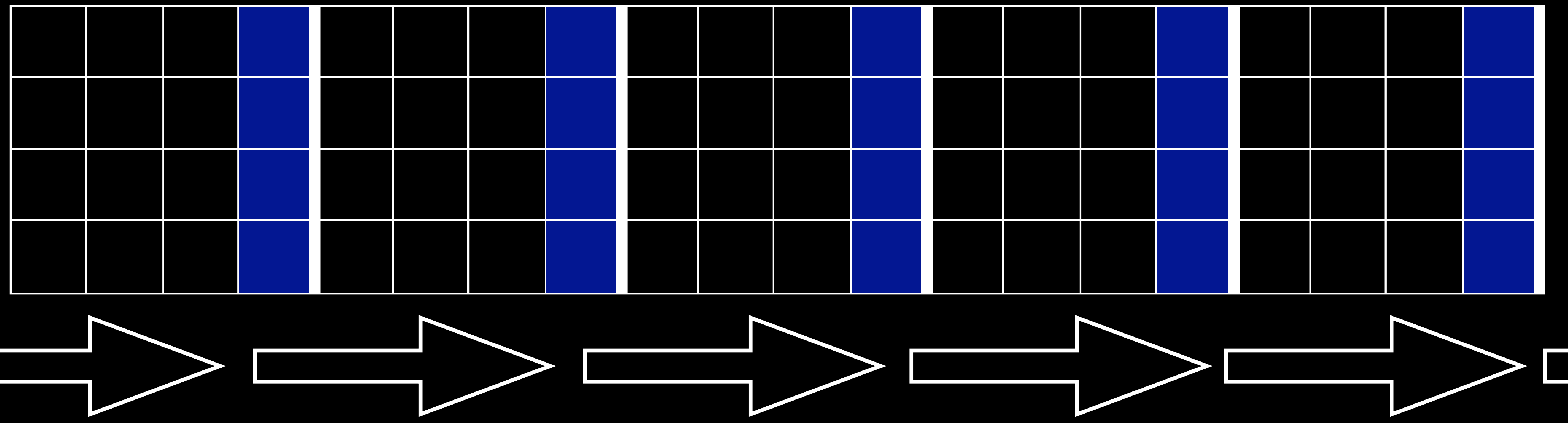
- if the messages cannot be buffered, the order becomes relevant!

Time	Process 0	Process 1
1	MPI_Send to 1	MPI_Recv from 0
2	MPI_Recv from 1	MPI_Send from 0

Boundary exchange



- Send and receive with a single call
  - int MPI_Sendrecv(
    void *sbuf, int scount, MPI_Datatype stype, int dest, int stag,
    void *rbuf, int rcount, MPI_Datatype rtype, int source, int rtag,
    MPI_Comm comm, MPI_Status *status);
  - MPI cares that no deadlock occurs!
  - Send to right neighbor, receive from left neighbor



Broadcasting data

- int MPI_Bcast(void *buf, int count, MPI_Datatype dtype, int rank, MPI_Comm comm);
  - Broadcasts data from process rank to all other processes in comm
  - error code
- Pitfall: Matching broadcast with receive

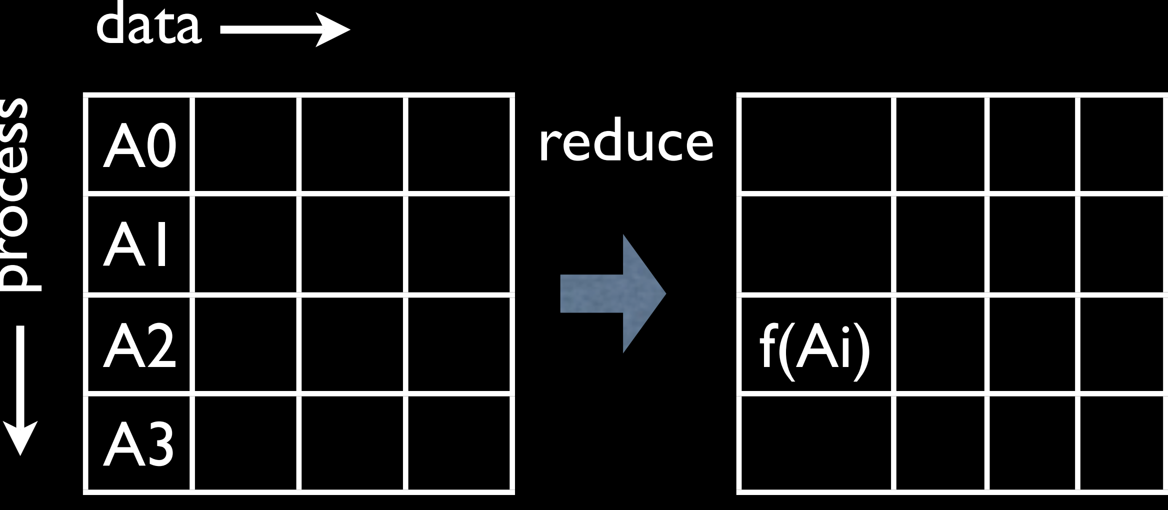
```
#include <mpi.h>
main(int argc, char *argv[])
{
  int rank;
  double param;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  if(rank==5) param=23.0;
  MPI_Bcast(&param,1,MPI_DOUBLE,5,MPI_COMM_WORLD);
  printf("P:%d after broadcast param is %f \n",rank,param);
  MPI_Finalize();
}
```

```
P: 0  after broadcast param is  23.
P: 5  after broadcast param is  23.
P: 2  after broadcast param is  23.
P: 3  after broadcast param is  23.
P: 4  after broadcast param is  23.
P: 1  after broadcast param is  23.
P: 6  after broadcast param is  23.
```

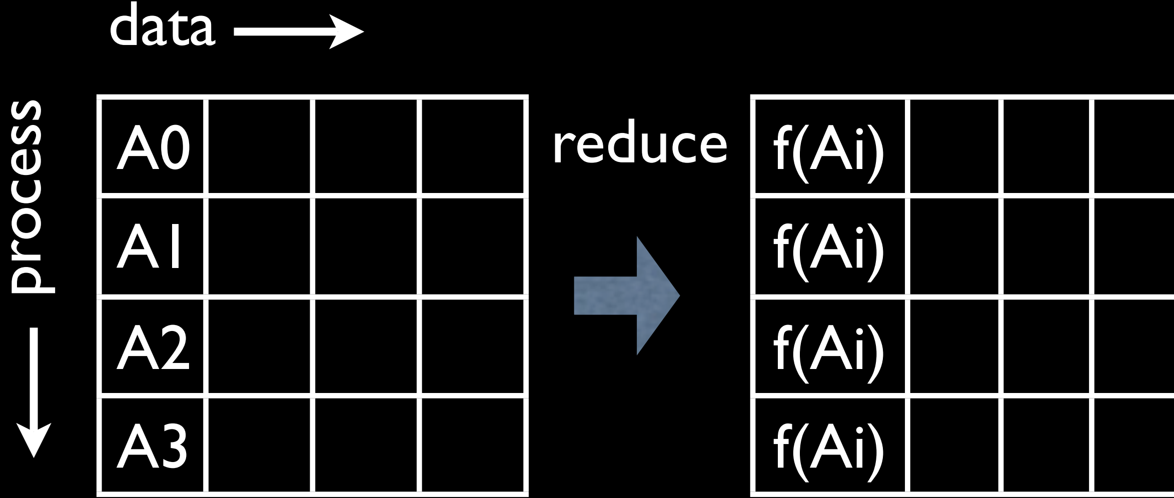
Combine data from all procs

- int MPI_Reduce(void *operand, void *result, int count, MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm comm);
  - Combines the operands using the operator op and stores the result in result on process root
  - error code



Combine data from all procs

- int MPI_Allreduce(void *operand, void *result, int count, MPI_Datatype dtype, MPI_Op op, MPI_Comm comm);
  - Combines the operands using the operator op and stores the result in result on all processes
  - error code

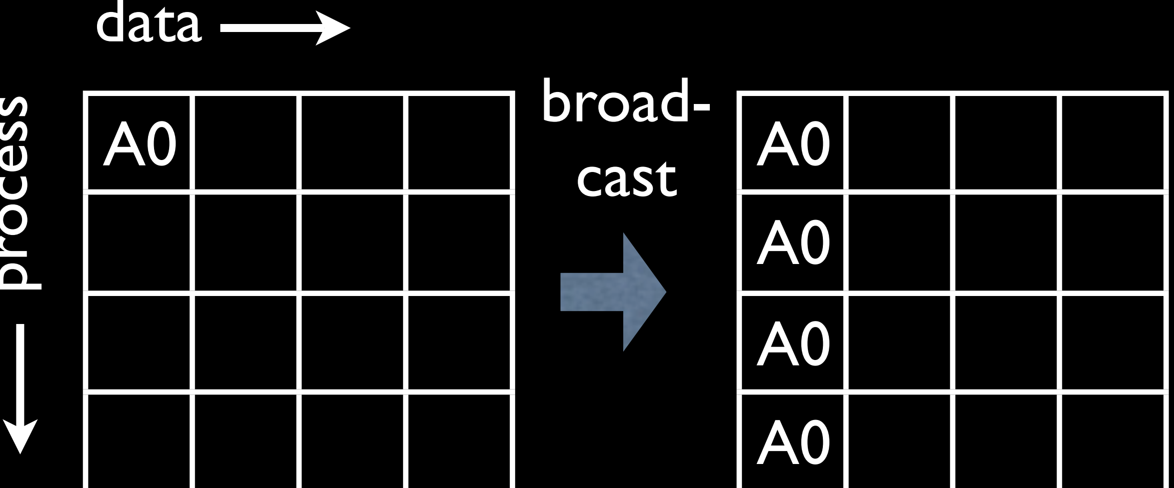


- Pitfall: Aliasing
  - attempt to store the result in the same location as the operand
  - MPI_Reduce(&operand, &operand, 1, MPI_FLOAT, MPI_SUM, 0, comm)

Predefined reduction ops

Operator name	**Meaning**
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical EXCLUSIVE OR
MPI_BXOR	Bitwise EXCLUSIVE OR
MPI_MAXLOC	Maximum & location of maximum
MPI_MINLOC	Minimum & location of minimum

More collectives



all-
gather

A0			
B0			
C0			
D0			

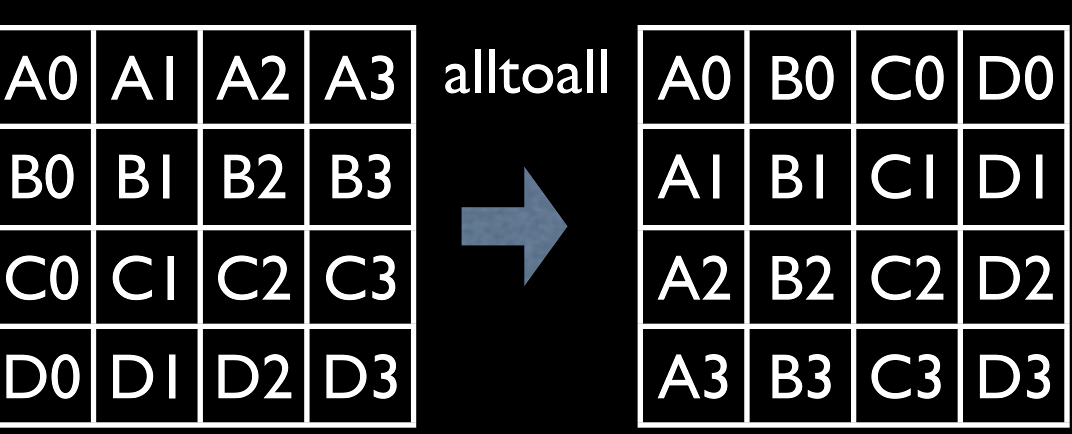
A0	B0	C0	D0
A0	B0	C0	D0
A0	B0	C0	D0
A0	B0	C0	D0

scatter

gather

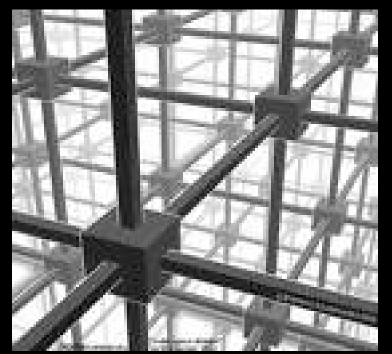
A0	A1	A2	A3

A0			
A1			
A2			
A3			



- Pitfall: Calling collectives from only some of the processes

```
if (…) {
  do_something();
} else {
  do_something_else();
  MPI_Barrier(comm);
}
```

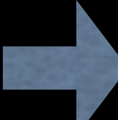


Design considerations

- “Quick & dirty”
  - compile your program in parallel, execute with mpicc, decide on the rank, which process operates on which part of the data
  - advantage & disadvantage: input data is read by every process
  - results can be written one after another into single output file or to separate files
- “Master” process reads data and broadcasts to everyone
  - slightly more work, but faster on more processors
  - results can be send back to the “master”, who then writes to file
- Master-slave parallelism <=> symmetric program?
- Make performance analysis a part of your development cycle!

Debugging parallel programs

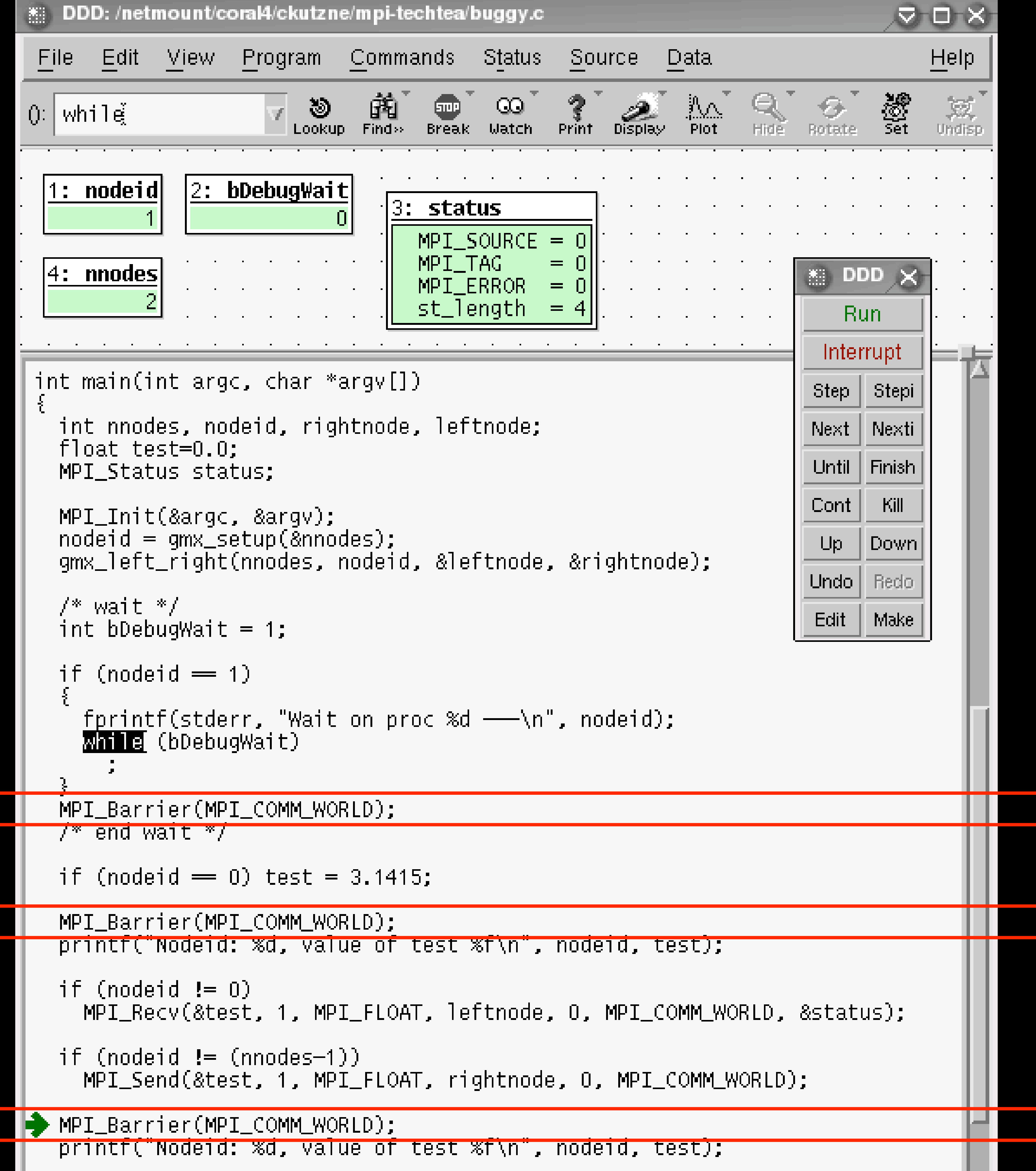
- serial style will not work!
  - mpicc -g ./buggy.c -o buggy.x
  - ddd mpirun -np 2 buggy.x will try to debug mpirun, not buggy.x
- what does the trick:
  - keep the program from running away with while statement & barriers
  - run the program:
    mpirun -np 2 buggy.x
  - find out process numbers:
    > ps -C buggy.x
    3209 buggy.x – MPI rank 0
    3210 buggy.x
  - attach debugger to process of interest:
    ddd buggy.x 3209 &



```
MPI_Init(&argc, &argv);
MPI_Comm_rank(&nodeid, MPI_COMM_WORLD);

/* wait */
if (nodeid == 0)
{
  int bDebugWait = 1;

  fprintf(stderr, "Wait on proc %d\n",
          nodeid);
  while (bDebugWait)
    ;
}
MPI_Barrier(MPI_COMM_WORLD);
/* end wait */
```



DDD: /netmount/coral4/ckutzne/mpi-techtea/buggy.c
File
Edit
View
Program
Commands
Status
Source
Data
Help
(): while
Lookup
Find»
Break
Watch
Print
Display
Plot
Hide
Rotate
Set
Undisp
1: nodeid
1
2: bDebugWait
0
3: status
MPI_SOURCE = 0
MPI_TAG = 0
MPI_ERROR = 0
st_length = 4
4: nnodes
2
DDD
Run
Interrupt
Step
Stepi
Next
Nexti
Until
Finish
Cont
Kill
Up
Down
Undo
Redo
Edit
Make
int main(int argc, char *argv[])
{
int nnodes, nodeid, rightnode, leftnode;
float test=0.0;
MPI_Status status;
MPI_Init(&argc, &argv);
nodeid = gmx_setup(&nnodes);
gmx_left_right(nnodes, nodeid, &leftnode, &rightnode);
/* wait */
int bDebugWait = 1;
if (nodeid == 1)
{
fprintf(stderr, "Wait on proc %d ---\n", nodeid);
while (bDebugWait)
;
}
MPI_Barrier(MPI_COMM_WORLD);
/* end wait */
if (nodeid == 0) test = 3.1415;
MPI_Barrier(MPI_COMM_WORLD);
printf("Nodeid: %d, value of test %f\n", nodeid, test);
if (nodeid != 0)
MPI_Recv(&test, 1, MPI_FLOAT, leftnode, 0, MPI_COMM_WORLD, &status);
if (nodeid != (nnodes-1))
MPI_Send(&test, 1, MPI_FLOAT, rightnode, 0, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
printf("Nodeid: %d, value of test %f\n", nodeid, test);

Performance evaluation

- Simple time measurements with MPI_Wtime:
  - MPI_Wtick() – timer accuracy
  - How long does something take on the individual processors? (N timings)
    ```
    t1 = MPI_Wtime();
    ... code to time ...
    elapsed = MPI_Wtime() -t1;
    ```
  - How long does it take on all parallel processes? (single timing)
    ```
    MPI_Barrier();
    t1 = MPI_Wtime();
    ... code to time ...'
    MPI_Barrier();
    elapsed = MPI_Wtime() - t1;
    ```
  - may be worth to look at the syncronization time!
    (How long does the barrier take?) (N timings, take maximum)
    ```
    t2 = MPI_Wtime();
    MPI_Barrier();
    synctime = MPI_Wtime() - t2;
    ```

Multi Processing Environment

Visualize & measure what is going on in your program

- Instrument your program
  - include <mpe.h>
  - MPE_Init_log();
  - MPE_Stop_log(); MPE_Start_log();
  - MPE_Describe_state(int start, int end, char *name, char *color);
  - MPE_Describe_event(int event, char *name);
  - MPE_Log_event(int event, int intdata, char *chardata);
- Compile with mpecc
  - mpecc -mpilog mpedemo.c -o mpedemo.x
- run
  - mpirun -np 6 mpedemo.x generates mpedemo.x.clog2
- view & analyze with jumpshot

MPE logging

```
int main(int argc, char *argv[])
{
  int nnodes, nodeid, rightnode, leftnode;
  float test;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size( MPI_COMM_WORLD, &nnodes );
  MPI_Comm_rank( MPI_COMM_WORLD, &nodeid );

  /* declare MPE variables & states and initialize MPE */
  int ev_print_start, ev_print_finish;
  ev_print_start  = MPE_Log_get_event_number();
  ev_print_finish = MPE_Log_get_event_number();
  MPE_Describe_state(ev_print_start, ev_print_finish, "output", "orange");
  MPE_Init_log();
  /* done MPE setup */

  gmx_left_right(nnodes, nodeid, &leftnode, &rightnode);
  if (nodeid == 0) test = 3.1415;

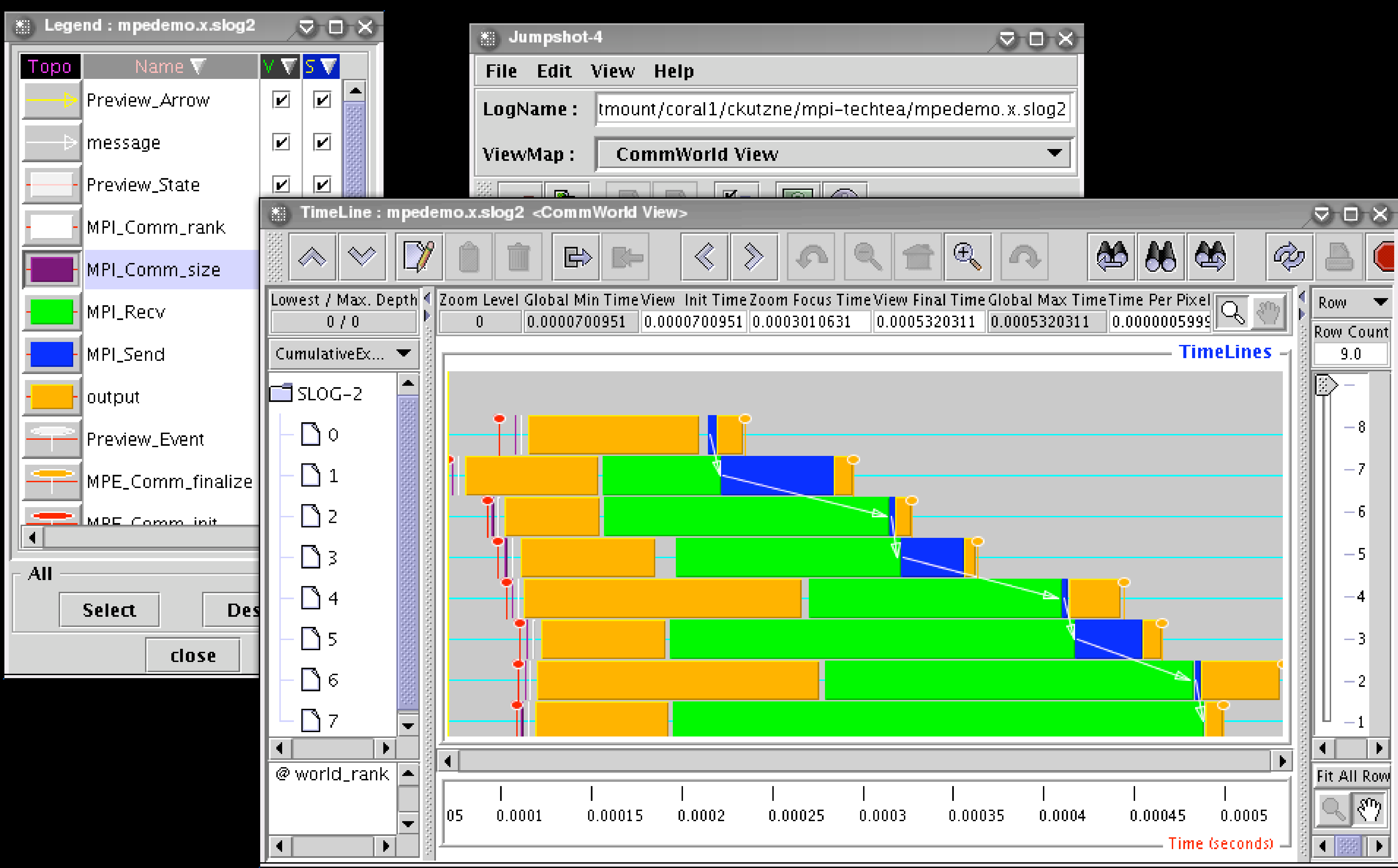
  MPE_Log_event(ev_print_start, 0, "start output");
  printf("Nodeid: %d, value of test %f\n", nodeid, test);
  MPE_Log_event(ev_print_finish, 0, "output finished");

  if (nodeid != 0)
    MPI_Recv(&test, 1, MPI_FLOAT, leftnode, 0, MPI_COMM_WORLD, &status);
  if (nodeid != (nnodes-1))
    MPI_Send(&test, 1, MPI_FLOAT, rightnode, 0, MPI_COMM_WORLD);

  MPE_Log_event(ev_print_start, 0, "start output");
  printf("Nodeid: %d, value of test %f\n", nodeid, test);
  MPE_Log_event(ev_print_finish, 0, "output finished");

  MPI_Finalize();
}
```

MPE logging



MPE graphics library

- simple graphics interface for parallel output
  - shared access by parallel processes to a single X display
  - each process can individually update the display
- usage:
  - include <mpe_graphics.h>
  - MPE_Open/Close_graphics(…)
  - MPE_Draw_point/line/circle/rectangle(…)
  - MPE_Update(…)
  - MPE_Num_colors(…)
  - MPE_Make_color_array(…)
- compile with mpecc
  - mpecc -graphics -lm mpedemo2.c -o mpedemo2.x

MPE graphics library

```
#include <stdio.h>
#include <mpi.h>
#include "mpe.h"
#include "mpe_graphics.h"

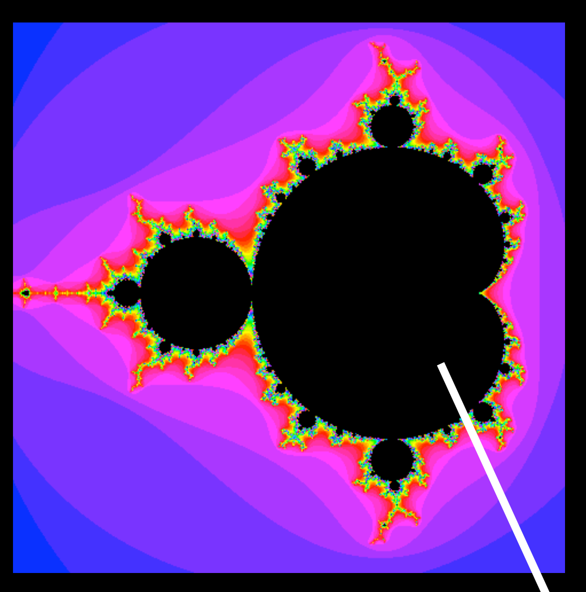
int main(int argc, char *argv[])
{
  int nnodes, nodeid;
  MPE_XGraph graph;
  char ckey;
  int ierr;
  MPE_Color my_color;

  MPI_Init(&argc, &argv);
  MPI_Comm_size( MPI_COMM_WORLD, &nnodes );
  MPI_Comm_rank( MPI_COMM_WORLD, &nodeid );

  MPE_Open_graphics( &graph, MPI_COMM_WORLD, NULL,
                            -1, -1, 400, 400, 0 );
  my_color = (MPE_Color) (nodeid + 1);
  if (nodeid == 0)
     ierr = MPE_Draw_string( graph, 187, 205, MPE_BLUE, "Hello" );
  ierr = MPE_Draw_circle( graph, 200, 200, 20+nodeid*5, my_color );
  ierr = MPE_Update( graph );

  MPI_Barrier(MPI_COMM_WORLD);
  ierr = MPE_Close_graphics( &graph );
  MPI_Finalize();
}
```

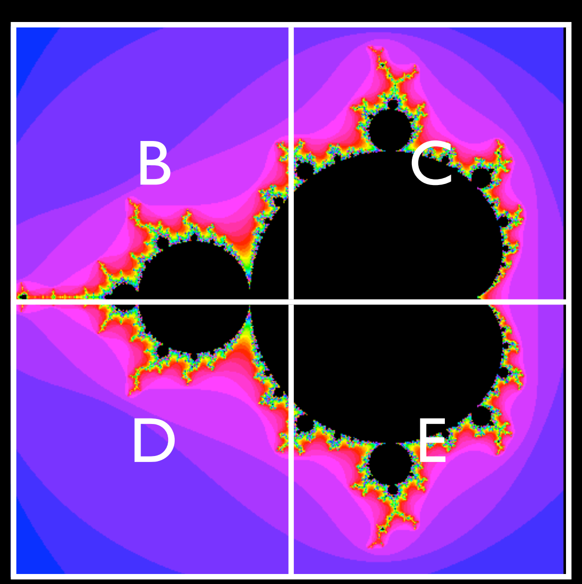
Self-scheduling master-slave example



- z = x + iy
- repeat

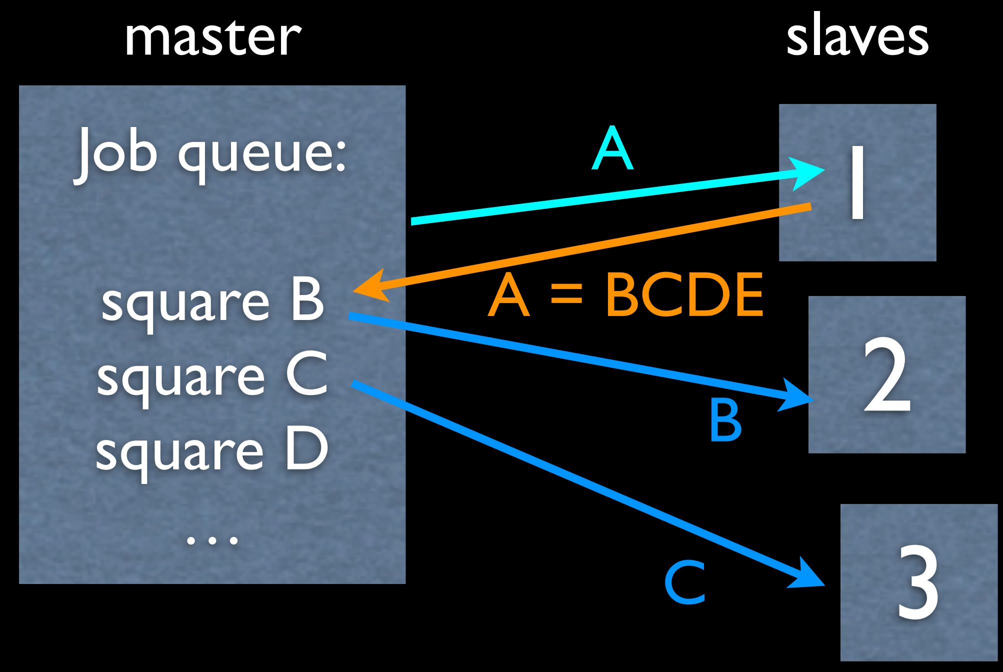
  $$f_c(z) = z^2 + c$$

  If f remains bounded, it is in M,
  otherwise color is a (optional) measure for divergence
- “Easy to parallelize”
  since each pixel’s color can be computed independently
- BUT: load imbalances!

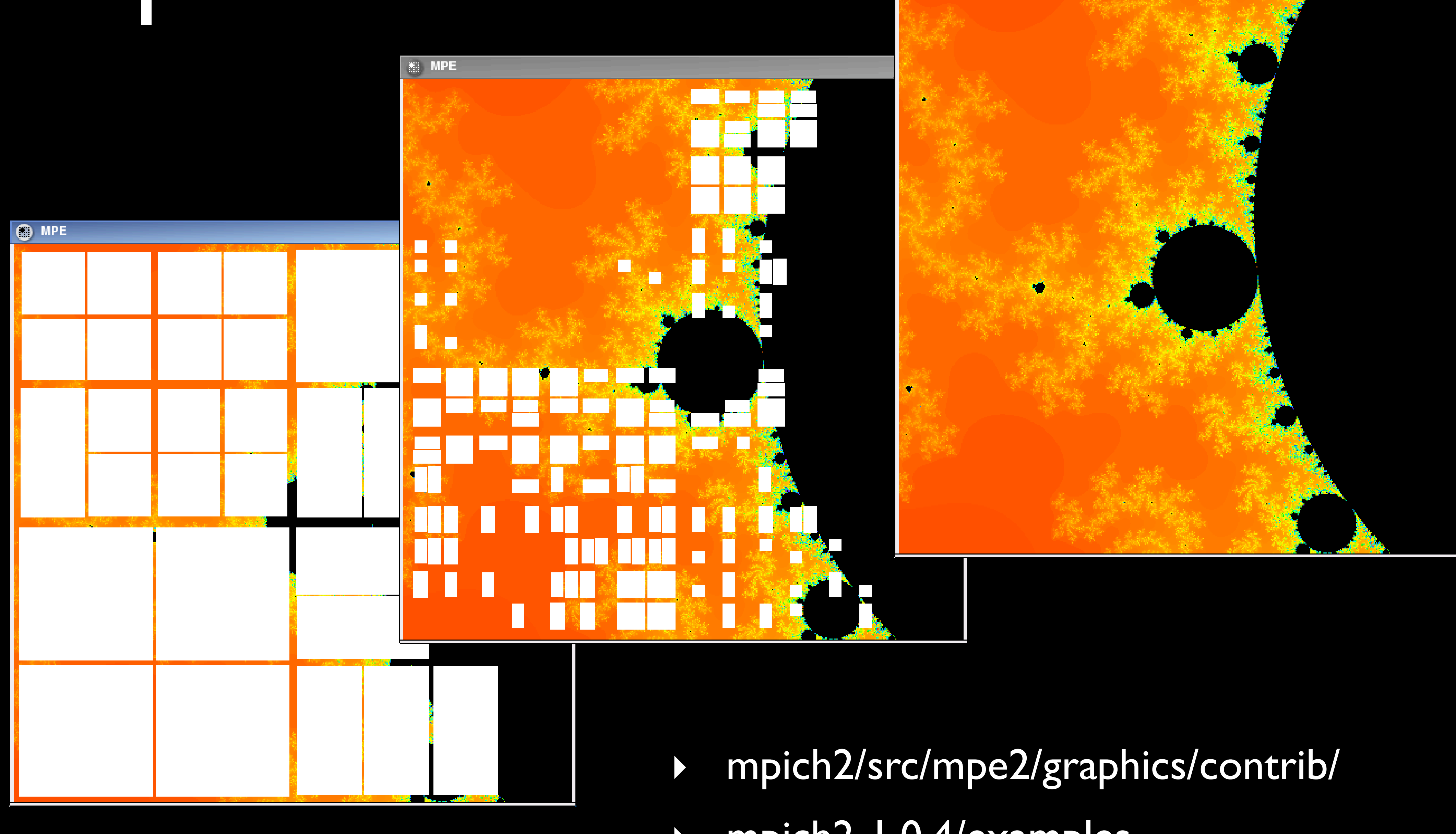


Self-scheduling master-slave example

- accelerate computation & balance load:
  - if border of any square is made up of same color, all pixels inside have that color, too
  - start: put whole area (A) into queue
  - master sends queue job to a free slave
    - slave computes boundary
    - if same color => fill & display
    - at first different color => subdivide into 4 subsquares, return them to master, but carry on with boundary. Display boundary when finished.



Self-scheduling master-slave example



- mpich2/src/mpe2/graphics/contrib/
- mpich2-1.0.4/examples
- lam-7.1.2/examples
- http://www-unix.mcs.anl.gov/mpi/usingmpi