UPPSALA
UNIVERSITET

Distributed memory parallelization using MPI

Part of course in Parallel Image Analysis

Elias Rudberg

elias.rudberg@it.uu.se

April 13, 2011

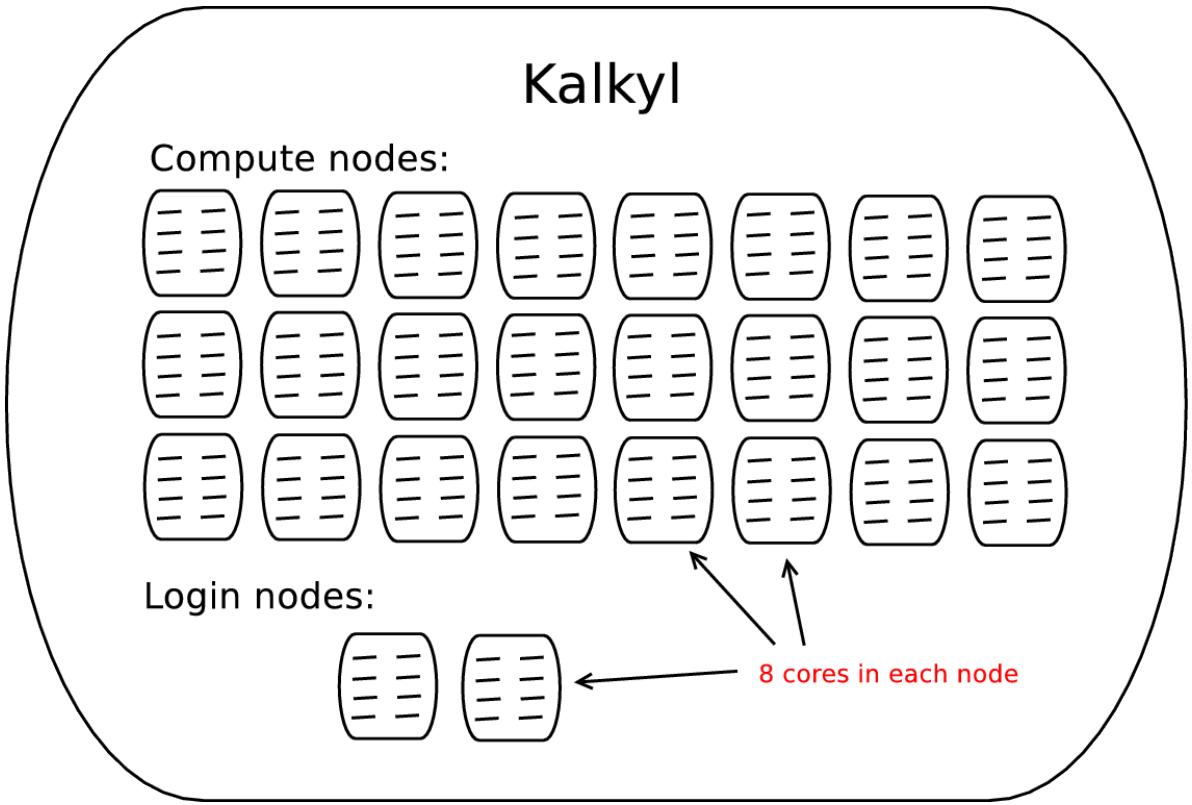
UPPSALA
UNIVERSITET

Outline

UPPSALA
UNIVERSITET

How does each compute node work?

Each compute node is a multi-core computer. On Kalkyl, each node has 8 cores.



Ways to use a compute cluster like Kalkyl

There are several different ways of using a compute cluster:

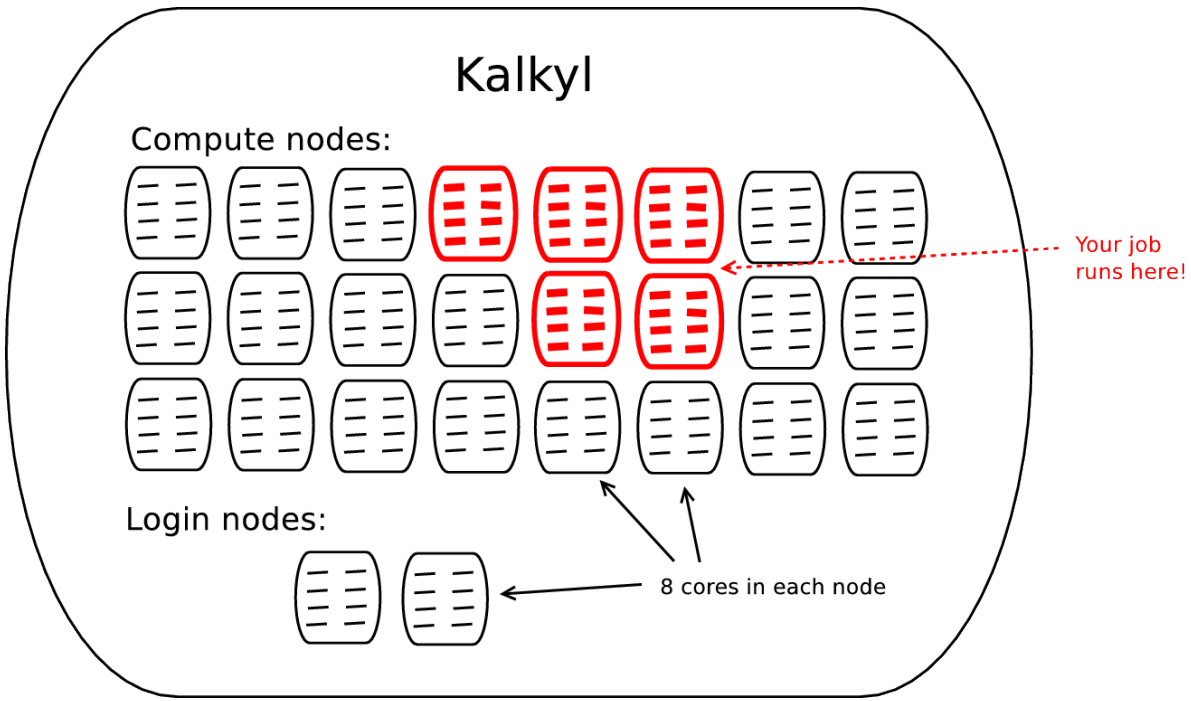
- Run serial program as a single-core job. Possibly run many such independent jobs in parallel. No parallel programming needed.
- Run threaded program as a single-node job. The same program then uses all 8 cores in one compute node. Possibly 8 times faster compared to single-core run.
- Run program using distributed memory parallelization using the message passing interface (MPI). The same program uses N compute nodes. Possibly $N \times 8$ times faster compared to single-core run.

UPPSALA UNIVERSITET

Ways to use a compute cluster like Kalkyl

Multi-node job

A distributed memory parallelized (typically MPI) program may run on several compute nodes.



Message Passing Interface (MPI)

The Message Passing Interface (MPI) standard:

"MPI is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users."

Open standard available on web:

http://www.mcs.anl.gov/research/projects/mpi/

See also the *MPI Forum* web page:

http://www.mpi-forum.org/

Full MPI 2.2 specification available as PDF, 647 [!] pages:

http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

Development of MPI standard

- Version 1.0: May, 1994.
- Version 1.1: June, 1995.
- Version 1.2: July 18, 1997.
- Version 2.0: July 18, 1997.
- Version 1.3: May 30, 2008.
- Version 2.1: June 23, 2008.
- Version 2.2: September 4, 2009.

Different implementations of the MPI standard

- Open MPI (free software, available on Kalkyl, used in this course)
- MPICH (free software)
- Intel MPI (variant of MPICH)
- Scali MPI ("scampi")
- ...

In this course, we focus on OpenMPI.

UPPSALA
UNIVERSITET

Outline

Compiling an MPI program

To compile an MPI program, use compiler `mpicc` (for C) or `mpiCC` (for C++). For example:

```
mpicc mpitest1.c -o mpitest1
mpiCC mpi-cpp-test.cc -o mpi-cpp-test
```

The `mpicc` or `mpiCC` command typically invokes a standard compiler with parameters for MPI enabled, so the ordinary compiler flags can be applied as usual:

```
mpicc -Wall -O3 mpitest1.c -o mpitest1
```

If using a makefile, you typically modify the line specifying the compiler (instead of e.g. CC = gcc):

```
CC = mpicc
```

UPPSALA UNIVERSITET

Compiling MPI programs on Kalkyl at UPPMAX

To use the OpenMPI MPI implementation on Kalkyl, you need to load the module `openmpi`. Since the OpenMPI library is compiled differently for different compilers you need to specify a compiler. UPPMAX recommends that you load the compiler and the openmpi modules together on a single line, like this:

```
eliasr@kalkyl1 ~]$ module load gcc openmpi
mod: loaded OpenMPI 1.4.3, compiled with gcc4.4 (found in /opt/openmpi/1.4.3gcc4.4/)
[eliasr@kalkyl1 ~]$
```

Or if you want to use the Intel compiler:

```
[eliasr@kalkyl4 ~]$ module load intel openmpi
mod: loaded OpenMPI 1.4.3, compiled with intel11.1 (found in /opt/openmpi/1.4.3intel11.1/)
[eliasr@kalkyl4 ~]$
```

UPPSALA UNIVERSITET

Compiling MPI programs on Kalkyl at UPPMAX

The `mpicc` command behaves like a wrapper for the compiler OpenMPI was loaded with:

```
[eliasr@kalkyl4 ~]$ module load gcc openmpi
mod: loaded OpenMPI 1.4.3, compiled with gcc4.4 (found in /opt/openmpi/1.4.3gcc4.4/)
[eliasr@kalkyl4 ~]$ mpicc --version
gcc (GCC) 4.4.4
Copyright (C) 2010 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
[eliasr@kalkyl4 ~]$
```

```
[eliasr@kalkyl4 ~]$ module load intel openmpi
mod: loaded OpenMPI 1.4.3, compiled with intel11.1 (found in /opt/openmpi/1.4.3intel11.1/)
[eliasr@kalkyl4 ~]$ mpicc --version
icc (ICC) 12.0.2 20110112
Copyright (C) 1985-2011 Intel Corporation.  All rights reserved.
[eliasr@kalkyl4 ~]$
```

In this course we will mostly focus on using OpenMPI with gcc, so our standard way of loading the OpenMPI module will be like this:

```
module load gcc openmpi
```

A simple MPI program

Program eliastest.c:

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
  MPI_Init(0, 0);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int nProcsTot;
  MPI_Comm_size(MPI_COMM_WORLD, &nProcsTot);
  printf("rank: %d. nProcsTot: %d.\n", rank, nProcsTot);
  MPI_Finalize();
  return 0;
}
```

Uses four important MPI functions:

- `MPI_Init`: initialize MPI. Must be called before any other MPI function is called.
- `MPI_Comm_rank`: get rank (ID number) of current MPI process.
- `MPI_Comm_size`: get total number of MPI processes
- `MPI_Finalize`: Cleanup, should be called before program ends.

UPPSALA
UNIVERSITET

Running an MPI program

Compile (this generates executable file a.out):

```
[eliasr@kalky14 ~]$ mpicc eliastest.c
[eliasr@kalky14 ~]$
```

Run program using the `mpirun` utility using the flag `-np` to specify the number of MPI processes:

```
[eliasr@kalky14 ~]$ mpirun -np 5 a.out
rank: 2. nProcsTot: 5.
rank: 4. nProcsTot: 5.
rank: 0. nProcsTot: 5.
rank: 3. nProcsTot: 5.
rank: 1. nProcsTot: 5.
[eliasr@kalky14 ~]$
```

```
[eliasr@kalky14 ~]$ mpirun -np 3 a.out
rank: 1. nProcsTot: 3.
rank: 2. nProcsTot: 3.
rank: 0. nProcsTot: 3.
[eliasr@kalky14 ~]$
```

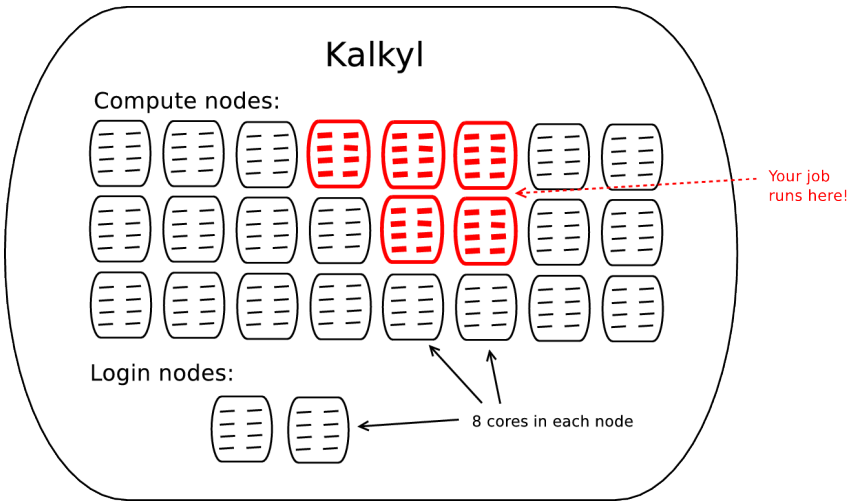
Note that output from `printf` statements on different processes can appear in any order.

UPPSALA
UNIVERSITET

Where is the executable?

An MPI program runs as several separate processes that may run on different compute nodes.

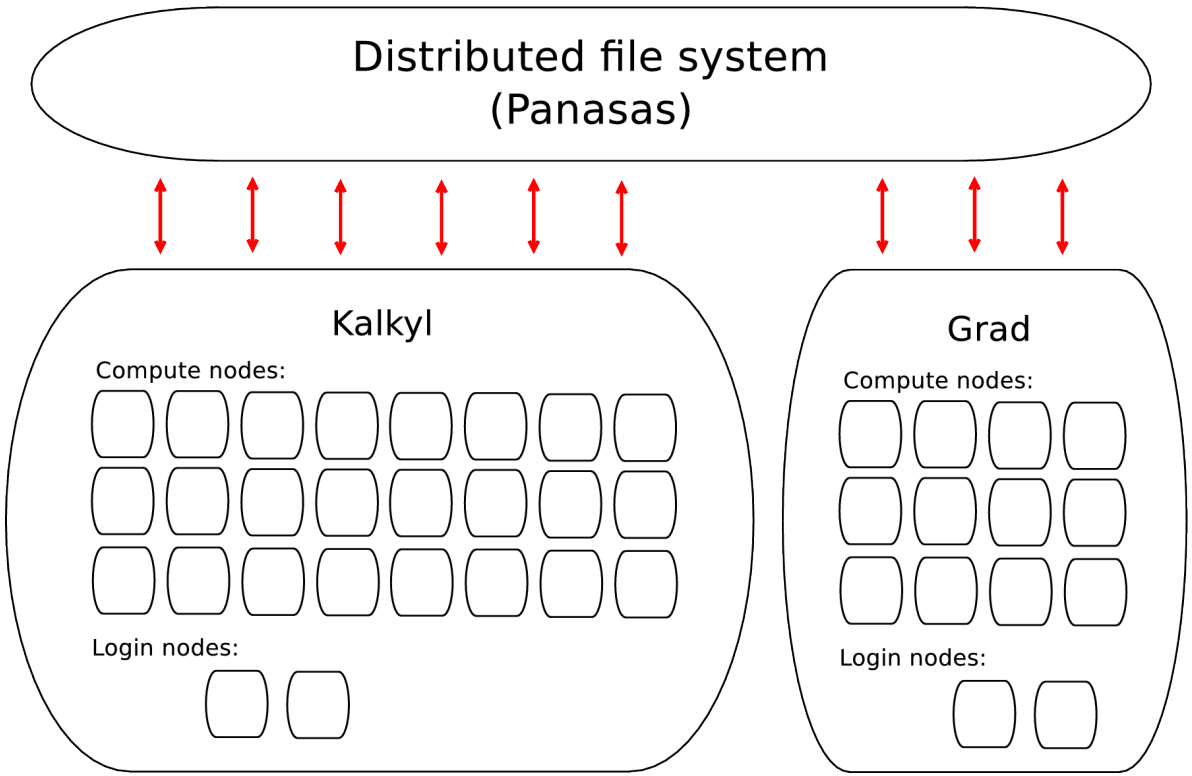
The executable file must be accessible from each compute node. This is typically solved automatically when using a distributed file system.



UPPSALA
UNIVERSITET

Distributed file systems

- Same file system accessible from all compute nodes.
- Local /scratch directory on each node can provide faster file operations, useful for temporary files.



UPPSALA UNIVERSITET

Where is the executable?

If running from a local directory e.g. /scratch you must make sure that the executable is copied to local /scratch on each compute node before running the program.

```
[eliasr@q34 scratch]$ mpirun -np 10 a.out
--------------------------------------------------------------------------
mpirun was unable to launch the specified application as it could not find an executable:

Executable: a.out
Node: q35

while attempting to start process rank 8.
--------------------------------------------------------------------------
[eliasr@q34 scratch]$
```

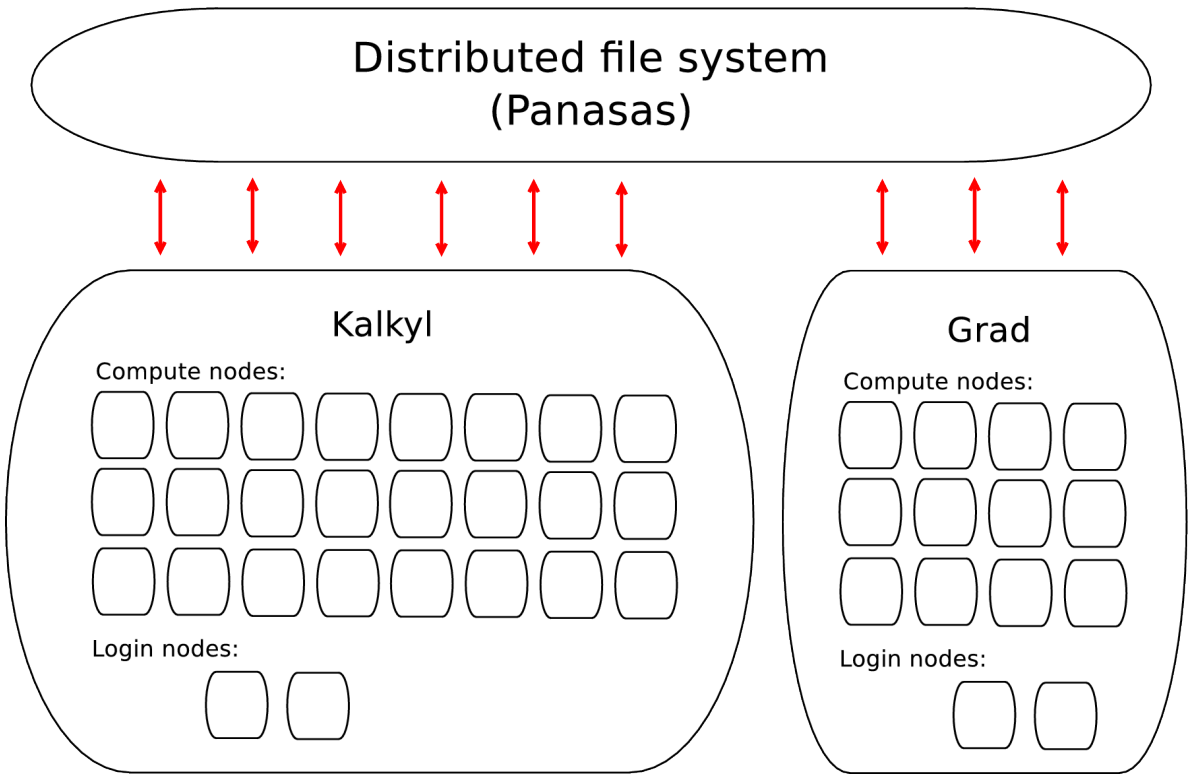
In this case, after copying the executable to /scratch on compute node q35, it works:

```
[eliasr@q34 scratch]$ scp a.out eliasr@q35:/scratch/
a.out
[eliasr@q34 scratch]$ mpirun -np 10 a.out
rank: 8. nProcsTot: 10.
rank: 9. nProcsTot: 10.
rank: 0. nProcsTot: 10.
rank: 7. nProcsTot: 10.
[...]
```

UPPSALA
UNIVERSITET

Where is the executable?

Easiest way to make sure the executable file is accessible from all compute nodes is to run it from a directory in the distributed file system. For example, somewhere under your home directory.



Where will the executable run?

Two ways of controlling how and where an MPI executable will run:

- Parameters to the queueing system
- Parameters to `mpirun`

UPPSALA UNIVERSITET

Where will the executable run?

Parameters to the queueing system

Most important SLURM parameters for MPI jobs: `-n` (number of processes) and `-N` (number of compute nodes).

Example of job script:

```
#! /bin/bash -l
#SBATCH -p node -n 32 -t 7-00:00:00
#SBATCH -A p2010999 -J elixir_B
module load gcc openmpi
mpirun elixir B_gamma.txt
```

The parameter `-n 32` means that you want to be able to run 32 MPI processes. On Kalkyl the default is to run 8 processes per compute node, so with `-n 32` four compute nodes will be allocated to your job.

UPPSALA UNIVERSITET

Where will the executable run?

Parameters to the queueing system

Example of job script using `-n` and `-N` together:

```
#! /bin/bash -l
#SBATCH -p node -N 4 -n 16 -t 7-00:00:00
#SBATCH -A p2010999 -J elixir_B
module load gcc openmpi
mpirun elixir B_gamma.txt
```

Here, `-N 4` means that you want access to 4 compute nodes. When combined with `-n` the processes will be distributed over those compute nodes, in this case four processes per node. Combining `-n` and `-N` can be useful in some cases:

- If you program needs much memory, so that 8 copies of the program do not fit on one node.
- If you use threading to exploit the cores on each node you may want to specify e.g. `-N 4 -n 4` to run only one copy of the program on each node.

Where will the executable run?

Parameters to mpirun

Simplest case: use `mpirun` without any other parameters than the executable file to run:

```
mpirun elixir
```

Then, `mpirun` will use information from the queueing system to determine how many copies of the program to run, and where to run them.

Use `-np` to explicitly tell `mpirun` to start a given number of copies of the program:

```
mpirun -np 12 elixir
```

Where will the executable run?

Parameters to mpirun

By default, `mpirun` will start 8 copies of the program on the first available node, then start copies on the next available node, etc. To distribute processes evenly among the available compute nodes, use `-loadbalance`:

```
mpirun -np 12 -loadbalance elixir
```

Many more options exist; see `man mpirun` for details.

Where will the executable run?

Parameters to mpirun

Any arguments given after the program filename will be treated as arguments to the program. Example:

```
mpirun -np 12 elixir infile 14 88
```

In this case, `mpirun` wil pass on the parameters `infile 14 88` as input parameters to the program `elixir`.

Parameters accessible through `argc` and `argv` parameters to `main()`.

How to compile and run MPI programs

Summary

Load openmpi module together with compiler module:

```
module load gcc openmpi
```

Compile using `mpicc` (for C) or `mpiCC` (for C++):

```
mpicc -Wall -O3 mpitest1.c -o mpitest1
```

For debugging, run program using `mpirun -np` directly at command prompt:

```
mpirun -np 5 mpitest1
```

For longer runs on several nodes, use queueing system parameters:

```
#! /bin/bash -l
#SBATCH -p node -n 32 -t 01:00:00
#SBATCH -A p2010999 -J elixir_B
module load gcc openmpi
mpirun mpitest1
```

UPPSALA
UNIVERSITET

Outline

MPI Programming

An MPI program uses MPI function calls to communicate.

The MPI standard includes:

- Point-to-point communication (functions)
- Datatypes
- Collective operations (functions)
- Environmental Management and inquiry (functions)
- Parallel file I/O
- ...

In C, MPI functions usually have names of the following form: `MPI_Class_action` or `MPI_Class_action_subset`.

UPPSALA
UNIVERSITET

Init and Finalize

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
  MPI_Init(0, 0);
  /* ...  */
  MPI_Finalize();
  return 0;
}
```

- `MPI_Init`: initialize MPI. Must be called before any other MPI function is called.
- `MPI_Finalize`: Cleanup, should be called before program ends.

Rank

Each process has a unique ID number called the *rank* of the process.

The rank numbers range from 0 to $(n-1)$ where n is the number of processes.

Each process can determine its own rank. The rank is used as address when sending messages between processes.

Typically, the programmer lets processes do different things depending on their rank. For example, in a master/slave setting it is common to let the process with rank 0 be the master.

UPPSALA UNIVERSITET

Determining the rank of the current process

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
  MPI_Init(0, 0);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int nProcsTot;
  MPI_Comm_size(MPI_COMM_WORLD, &nProcsTot);
  printf("rank: %d. nProcsTot: %d.\n", rank, nProcsTot);
  MPI_Finalize();
  return 0;
}
```

- `MPI_Comm_rank`: get rank (ID number) of current MPI process.
- `MPI_Comm_size`: get total number of MPI processes

UPPSALA UNIVERSITET

Sending a message

Definition of `MPI_Send` in MPI specification:

```
MPI_SEND(buf, count, datatype, dest, tag, comm)
IN   buf       initial address of send buffer (choice)
IN   count     number of elements in send buffer (non-negative integer)
IN   datatype  datatype of each send buffer element (handle)
IN   dest      rank of destination (integer)
IN   tag       message tag (integer)
IN   comm      communicator (handle)
```

C syntax:

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
```

Example of how to use `MPI_Send`:

```
/* Send message containing two integer numbers to rank 4. */
int intListToSend[2] = {33, 77};
int messageLength = 2;
int destRank = 4;
int tag = 66;
MPI_Send(intListToSend, messageLength, MPI_INTEGER, destRank, tag, MPI_COMM_WORLD);
```

UPPSALA
UNIVERSITET

Receiving a message

Definition of MPI_Recv in MPI specification:

```
MPI_RECV (buf, count, datatype, source, tag, comm, status)
OUT  buf        initial address of receive buffer (choice)
IN   count      number of elements in receive buffer (non-negative integer)
IN   datatype   datatype of each receive buffer element (handle)
IN   source     rank of source or MPI_ANY_SOURCE (integer)
IN   tag        message tag or MPI_ANY_TAG (integer)
IN   comm       communicator (handle)
OUT  status     status object (Status)
```

C syntax:

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

Example of how to use MPI_Recv:

```
/* Receive message from rank 7. */
int intListToReceive[2] = {0, 0};
int messageLength = 2;
int sourceRank = 7;
int tag = 66;
MPI_Status status;
MPI_Recv(intListToReceive, messageLength, MPI_INTEGER, sourceRank, tag, MPI_COMM_WORLD, &status);
```

Communication example

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
  MPI_Init((void *)0, (void *)0);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if(rank == 0) {
    /* This is rank 0. */
    /* Send message containing two integer numbers to rank 1. */
    int intListToSend[2] = {3, 7};
    int messageLength = 2;
    int destRank = 1;
    int tag = 66;
    MPI_Send(intListToSend, messageLength, MPI_INTEGER, destRank, tag, MPI_COMM_WORLD);
  }
  else {
    /* This is rank 1. */
    /* Receive message from rank 0. */
    int intListToReceive[2] = {0, 0};
    int messageLength = 2;
    int sourceRank = 0;
    int tag = 66;
    MPI_Status status;
    MPI_Recv(intListToReceive, messageLength, MPI_INTEGER, sourceRank, tag, MPI_COMM_WORLD, &status);
    printf("Received integers: %d %d\n", intListToReceive[0], intListToReceive[1]);
  }
  MPI_Finalize();
  exit(0);
}
```

Error codes

Almost all MPI functions have return type `int`, with the returned value being an error code if something went wrong, or `MPI_SUCCESS` on success.

Example of checking for error code:

```
/* This is rank 0. */
/* Send message containing two integer numbers to rank 1. */
int intListToSend[2] = {3, 7};
int messageLength = 2;
int destRank = 2, tag = 66;
if(MPI_Send(intListToSend, messageLength,
            MPI_INTEGER, destRank, tag, MPI_COMM_WORLD) != MPI_SUCCESS)
{
  printf("Error in MPI_Send when sending from rank 0.\n");
  return -1;
}
```

Error codes

The default error handler (`MPI_ERRORS_ARE_FATAL`) means that the program is anyway aborted if any MPI call fails. So, as long as the default error handler is used there is no need to check for error codes.

Example of what happens when an MPI call fails:

```
[eliasr@kalkyl4 example2]$ mpirun -np 2 mpitest2
Starting program. This is printf output from rank 0. Total number of processes: 2.
[kalkyl4.uppmax.uu.se:6465] *** An error occurred in MPI_Send
[kalkyl4.uppmax.uu.se:6465] *** on communicator MPI_COMM_WORLD
[kalkyl4.uppmax.uu.se:6465] *** MPI_ERR_RANK: invalid rank
[kalkyl4.uppmax.uu.se:6465] *** MPI_ERRORS_ARE_FATAL (your MPI job will now abort)
--------------------------------------------------------------------------
mpirun has exited due to process rank 0 with PID 6465 on
node kalkyl4.uppmax.uu.se exiting without calling "finalize". This may
have caused other processes in the application to be
terminated by signals sent by mpirun (as reported here).
--------------------------------------------------------------------------
[eliasr@kalkyl4 example2]$
```

In this case, the error `MPI_ERR_RANK: invalid rank` resulted because the program called `MPI_Send` with a destination rank that was outside the allowed range.

Data types

Many MPI data types available:

```
MPI_INTEGER
MPI_CHARACTER
MPI_CHAR
MPI_SIGNED_CHAR
MPI_UNSIGNED_CHAR
MPI_UINT8_T
MPI_FLOAT
MPI_DOUBLE
...
```

However, in plain send/receive calls what is sent is really just a buffer with data, so if you like you can use e.g. `MPI_CHARACTER` for the data and cast pointer to the proper type.

Message destination/source rank

The `dest` and `source` arguments to `MPI_Send` and `MPI_Recv` are used to specify the rank of the destination/source process:

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);

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

For `MPI_Send` calls, a `dest` rank must always be given.

For `MPI_Recv` calls, the special `source` value `MPI_ANY_SOURCE` may be specified.

Message tags

Both `MPI_Send` and `MPI_Recv` require a `tag` argument:

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);

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

For `MPI_Send` calls, a `tag` must always be given.

For `MPI_Recv` calls, the special `tag` value `MPI_ANY_TAG` may be specified.

Communicators

Many MPI functions require a `communicator` argument:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);

int MPI_Send(void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);

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

Using different communicators allows independent communication layers, useful for example if working with parallel libraries.

In many cases, the standard communicator `MPI_COMM_WORLD` can be used for all calls.

The status object

When using `MPI_Recv` the last argument is the `status`:

```
MPI_Status status;
MPI_Recv(buf, size, MPI_INTEGER,
         sourceRank, tag, MPI_COMM_WORLD, &status);
```

The `status` structure has three fields that can be accessed directly:

```
status.MPI_ERROR
status.MPI_TAG
status.MPI_SOURCE
```

The `MPI_ERROR` field can be used to check for error codes.

The `MPI_TAG` and `MPI_SOURCE` can be used to check the tag and source, respectively. Example:

```
MPI_Status status;
MPI_Recv(buf, size, MPI_INTEGER, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
printf("Message with tag %d received from rank %d\n", status.MPI_TAG, status.MPI_SOURCE);
```

UPPSALA
UNIVERSITET

Probing for messages

Definition of `MPI_Probe` in MPI specification:

```
MPI_PROBE(source, tag, comm, status)
IN   source   rank of source or MPI_ANY_SOURCE (integer)
IN   tag      message tag or MPI_ANY_TAG (integer)
IN   comm     communicator (handle)
OUT  status   status object (Status)
```

C syntax:

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);
```

Example of how to use `MPI_Probe` if messages of unknown length are to be received:

```
MPI_Status status;
MPI_Probe(sourceRank, tag, MPI_COMM_WORLD, &status);
int msgSize;
MPI_Get_count(&status, MPI_INTEGER, &msgSize);
int* buf = (int*)malloc(msgSize*sizeof(int));
MPI_Recv(buf, msgSize, MPI_INTEGER, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
```

Blocking vs nonblocking operations

So far only *blocking* operations.

Next: *nonblocking* operations.

UPPSALA UNIVERSITET

Blocking vs nonblocking operations

The MPI communication functions discussed so far are all *blocking* operations:

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);

int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status);

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);
```

Corresponding *nonblocking* operations also exist:

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm, MPI_Request *request);

int MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm, MPI_Request *request);

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);
```

Nonblocking functions always return immediately; the program can continue doing other things, and check the communication result later.

Nonblocking send

Definition of `MPI_Isend` in MPI specification:

```
MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
IN   buf initial address of send buffer (choice)
IN   count number of elements in send buffer (non-negative integer)
IN   datatype datatype of each send buffer element (handle)
IN   dest rank of destination (integer)
IN   tag message tag (integer)
IN   comm communicator (handle)
OUT  request communication request (handle)
```

C syntax:

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm, MPI_Request *request);
```

Example of how to use `MPI_Isend`:

```
MPI_Request request;
MPI_Isend(buf, size, MPI_INTEGER, destRank, tag, MPI_COMM_WORLD, &request);
/* Possibly do other stuff here. */
MPI_Status status;
MPI_Wait(&request, &status);
```

UPPSALA
UNIVERSITET

Nonblocking receive

Definition of `MPI_Irecv` in MPI specification:

```
MPI_IRECV (buf, count, datatype, source, tag, comm, request)
OUT  buf         initial address of receive buffer (choice)
IN   count       number of elements in receive buffer (non-negative integer)
IN   datatype    datatype of each receive buffer element (handle)
IN   source      rank of source or MPI_ANY_SOURCE (integer)
IN   tag         message tag or MPI_ANY_TAG (integer)
IN   comm        communicator (handle)
OUT  request     communication request (handle)
```

C syntax:

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm, MPI_Request *request);
```

Example of how to use `MPI_Irecv`:

```
MPI_Request request;
MPI_Irecv(buf, msgSize, MPI_INTEGER, source, tag, MPI_COMM_WORLD, &request);
/* Possibly do other stuff here. */
MPI_Status status;
MPI_Wait(&request, &status);
```

Wait functions

When using nonblocking MPI calls, wait functions can be used to wait until communication operation(s) have finished:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);

int MPI_Waitall(int count, MPI_Request *array_of_requests,
                MPI_Status *array_of_statuses);

int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,
                MPI_Status *status);

int MPI_Waitsome(int incount, MPI_Request *array_of_requests,
                 int *outcount, int *array_of_indices,
                 MPI_Status *array_of_statuses);
```

UPPSALA
UNIVERSITET

Test functions

Test functions can be used to check if nonblocking communication operation(s) have finished:

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);

int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
                MPI_Status *array_of_statuses);

int MPI_Testany(int count, MPI_Request *array_of_requests, int *index,
                int *flag, MPI_Status *status);

int MPI_Testsome(int incount, MPI_Request *array_of_requests,
                 int *outcount, int *array_of_indices,
                 MPI_Status *array_of_statuses)
```

Cancelling nonblocking calls

The function `MPI_Cancel` can be used to cancel a previously initiated nonblocking communication call:

```
int MPI_Cancel(MPI_Request *request);
```

For example, if a nonblocking receive call was made but no message arrives and the program needs to continue anyway, the request should be cancelled. Otherwise there will be resource leaks (memory leaks and other).

Freeing MPI communication requests

Use `MPI_Request_free` to deallocate a request object without waiting for the associated communication to complete.

```
int MPI_Request_free(MPI_Request *request);
```

Note that `MPI_Request_free` does not cancel the communication; it only frees the request object.

UPPSALA UNIVERSITET

Using lists of MPI communication requests

Example of using a list of request objects to deal with several ongoing nonblocking communication calls simultaneously:

```
/* Make two nonblocking receive calls. */
MPI_Request requestList[2];
char recvBuf1[bufsz1];
char recvBuf2[bufsz2];
MPI_Irecv(recvBuf1, bufsz1, MPI_CHARACTER, MPI_ANY_SOURCE, tag1, MPI_COMM_WORLD, &requestList[0]);
MPI_Irecv(recvBuf2, bufsz2, MPI_CHARACTER, MPI_ANY_SOURCE, tag2, MPI_COMM_WORLD, &requestList[1]);
/* Wait for any of the two receive calls to finish. */
MPI_Status status;
int index;
MPI_Waitany(2, requestList, &index, &status);
if(index == 0) {
  /* First call finished. Handle result. */
}
else {
  /* Second call finished. Handle result. */
}
```

Note that if the other receive call is not expected to complete, that request should be cancelled using `MPI_Cancel` to avoid resource leaks.

UPPSALA
UNIVERSITET

Outline

Point-to-point vs Collective operations

So far only *point-to-point* operations.

Next: *collective* operations.

Point-to-point vs Collective operations

In a *point-to-point* operation a message is sent from one sender process to one receiver process.

In *collective* operations, more than two processes can be involved.

Examples of collective operations:

- Barrier – synchronize across all processes
- Broadcast – send same data from one process to all others
- Gather – gather data from all processes to one
- Scatter – scatter data from one process to all others
- Reduce – global reductions such as sum, max, min across all processes
- ...

Collective operations

Collective operations illustrated on page 132 in MPI specification PDF: broadcast, scatter, gather, allgather, complete exchange.

Why use collective operations?

Collective operations may seem superfluous since all such communication can be achieved using (many) point-to-point communication calls.

Two main reasons for using collective operations:

- Make the code simpler
- Make the program run faster

A good MPI implementation can perform collective operations much faster than corresponding repeated point-to-point calls.

Collective operations

Broadcast

Definition of `MPI_Bcast` in MPI specification:

```
MPI_BCAST( buffer, count, datatype, root, comm )
INOUT   buffer    starting address of buffer (choice)
IN      count     number of entries in buffer (non-negative integer)
IN      datatype  data type of buffer (handle)
IN      root      rank of broadcast root (integer)
IN      comm      communicator (handle)
```

C syntax:

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

Example of how to use `MPI_Bcast`:

```
/* This example sends 100 ints from process 0 to all others. */
int array[100];
int root = 0;
/* ... fill array with data ... */
MPI_Bcast( array, 100, MPI_INT, root, MPI_COMM_WORLD);
```

Collective operations

Gather

Definition of `MPI_Gather` in MPI specification:

```
MPI_GATHER( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
IN   sendbuf     starting address of send buffer (choice)
IN   sendcount   number of elements in send buffer (non-negative integer)
IN   sendtype    data type of send buffer elements (handle)
OUT  recvbuf     address of receive buffer (choice, significant only at root)
IN   recvcount   number of elements for any single receive (non-negative integer, significant only at root
IN   recvtype    data type of recv buffer elements (significant only at root) (handle)
IN   root        rank of receiving process (integer)
IN   comm communicator (handle)
```

C syntax:

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
               void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
               MPI_Comm comm);
```

Example of how to use `MPI_Gather`:

```
/* Gather 100 ints from every process to root. */
int gsize,sendarray[100];
int root, *rbuf;
MPI_Comm_size( MPI_COMM_WORLD, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, MPI_COMM_WORLD);
```

Collective operations

Gather

See illustrations of Gather operations in MPI specification PDF, pages 143 and 144.

Collective operations

Scatter

Definition of `MPI_Scatter` in MPI specification:

```
MPI_SCATTER( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
IN   sendbuf     address of send buffer (choice, significant only at root)
IN   sendcount   number of elements sent to each process (non-negative integer, significant only at root)
IN   sendtype    data type of send buffer elements (significant only at root) (handle)
OUT  recvbuf     address of receive buffer (choice)
IN   recvcount   number of elements in receive buffer (non-negative integer)
IN   recvtype    data type of receive buffer elements (handle)
IN   root        rank of sending process (integer)
IN   comm        communicator (handle)
```

C syntax:

```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                MPI_Comm comm);
```

Example of how to use `MPI_Scatter`:

```
/* Scatter sets of 100 ints from the root to each process in the group. */
int gsize, *sendbuf, root, rbuf[100];
MPI_Comm_size( MPI_COMM_WORLD, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
/* ... put data in sendbuf here ... */
MPI_Scatter( sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, MPI_COMM_WORLD);
```

Collective operations

Scatter

See illustrations of Scatter operations in MPI specification PDF, pages 152 and 153.

Collective operations

Reduce

Definition of `MPI_Reduce` in MPI specification:

```
MPI_REDUCE( sendbuf, recvbuf, count, datatype, op, root, comm)
IN   sendbuf    address of send buffer (choice)
OUT  recvbuf    address of receive buffer (choice, significant only at root)
IN   count      number of elements in send buffer (non-negative integer)
IN   datatype   data type of elements of send buffer (handle)
IN   op         reduce operation (handle)
IN   root       rank of root process (integer) IN comm communicator (handle)
```

C syntax:

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
```

Example of how to use `MPI_Reduce`:

```
double sum = 0;
/* ... Compute local sum, sum += xxx */
double c; /* result (at node zero) */
! global sum
MPI_Reduce(&sum, &c, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
/* Now global sum is stored in c at process with rank 0. */
```

Collective operations

Synchronization

Collective operations are blocking (no associated request object).

Even though all processes are involved in a collective communication operation, they may not be synchronized.

For example, a broadcast operation may be implemented in such a way that the `MPI_Bcast` function call at the sender process returns before other processes have made the `MPI_Bcast` call.

Avoid deadlocks

Be careful to avoid deadlocks.

Example of erroneous program:

```
switch(rank) {
  case 0:
    MPI_Bcast(buf1, count, type, 0, comm);
    MPI_Send(buf2, count, type, 1, tag, comm);
    break;
  case 1:
    MPI_Recv(buf2, count, type, 0, tag, comm, status);
    MPI_Bcast(buf1, count, type, 0, comm);
    break;
}
```

This program may deadlock. The broadcast call on process zero *may* block until process one executes the matching broadcast call, so that the send is not executed.

UPPSALA
UNIVERSITET

Outline

Combining MPI with threading

If you need to use threads in your MPI program, you need to initilize MPI using `MPI_Init_thread` instead of `MPI_Init`.

Definition of `MPI_Init_thread` in MPI specification:

```
MPI_INIT_THREAD(required, provided)
IN    required    desired level of thread support (integer)
OUT   provided    provided level of thread support (integer)
```

C syntax:

```
int MPI_Init_thread(int *argc, char *((*argv)[]), int required, int *provided);
```

Example of how to use `MPI_Init_thread`:

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
  int providedThreadSupport;
  MPI_Init_thread(0, 0, MPI_THREAD_MULTIPLE, &providedThreadSupport);
  if(providedThreadSupport != MPI_THREAD_MULTIPLE) {
    printf("Error: MPI_THREAD_MULTIPLE not supported.\n");
    return -1;
  }
  printf("OK, MPI_THREAD_MULTIPLE supported.\n");
  /* ... Threaded MPI program here ... */
```

UPPSALA
UNIVERSITET

Combining MPI with threading

Available thread support levels for `MPI_Init_thread`:

- `MPI_THREAD_SINGLE`: Only one thread allowed.
- `MPI_THREAD_FUNNELED`: Only main thread makes MPI calls. (for the definition of main thread, see `MPI_Is_thread_main` function.
- `MPI_THREAD_SERIALIZED`: Different threads make MPI calls, but not simultaneously.
- `MPI_THREAD_MULTIPLE`: Multiple threads may call MPI, with no restrictions.

If using `MPI_THREAD_FUNNELED`, this function can be used to check which thread is the main thread allowed to make MPI calls:

```
int MPI_Is_thread_main(int *flag);
```

UPPSALA
UNIVERSITET

Determining where a process is running

The function `MPI_Get_processor_name` can be used to get a text string describing the node where the MPI process is running:

```
char name[MPI_MAX_PROCESSOR_NAME];
int nameLen;
MPI_Get_processor_name(name, &nameLen);
printf("This is MPI process with rank %3d running on node '%s'\n", rank, name);
```

Creating MPI processes dynamically

The function `MPI_Comm_spawn` can be used to dynamically create new MPI processes.

```
#include <mpi.h>
#include <iostream>
int main(int argc, char* argv[]) {
  int threading_level_required = MPI_THREAD_MULTIPLE;
  int threading_level_provided;
  MPI_Init_thread(0, 0, threading_level_required, &threading_level_provided);
  int n_workers = 5;
  MPI_Comm comm_to_workers;
  char worker_program[100];
  strcpy(worker_program, "worker");
  MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, n_workers,  MPI_INFO_NULL,
                 0, MPI_COMM_SELF, &comm_to_workers, MPI_ERRCODES_IGNORE);
  /* Now the worker processes have been created. */
  /* This "mother" process can communicate with workers using the communicator comm_to_workers. */
```

UPPSALA
UNIVERSITET

Outline

Master/slave example

Example of master/slave MPI program:

```
int rank, nProcsTot;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nProcsTot);
if(rank == 0) {
  /* This is the "master" process. */
  /* ... Master performs some startup work here ... */
  /* Master decides what work should be done by each slave process. */
  for(int i = 1; i < nProcsTot; i++) {
    /* Prepare work data to be sent to process with rank i. */
    MPI_Send(... send data to rank i ...);
  }
  /* Now data has been send to slave processes. */
  /* Maybe master performs some work itself, or maybe it just waits for slaves to finish. */
  for(int i = 1; i < nProcsTot; i++) {
    /* Receive results from process with rank i. */
    MPI_Recv(... receive data from rank i ...);
  }
  /* Now master puts together the pieces received from the slaves. */
  /* Master produces final result, maybe writes it to file. */
}
else
{
  /* This is a "slave" process. Simply wait for data to arrive. */
  MPI_Recv(... receive data from rank 0 ...);
  /* Slave uses received data to perform some work. */
  /* Produce partial result to be returned to master. */
  MPI_Send(... send result data back to rank 0 ...);
}
```

Static division of work

Example:

```
int rank, nProcsTot;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nProcsTot);
/* Do some initialization, make sure each process has the part of the data it is responsible for. */
for(int iter = 0; iter < nIters; iter++)
{
  /* Each iteration begins with some communication between processes,  */
  /* to send and receive data between neighbors.                       */
  /* Send data to neighbors. */
  MPI_Isend(... send data to neighbor 1 ... )
  MPI_Isend(... send data to neighbor 2 ... )
  /* Receive data from neighbors. */
  MPI_Irecv(... receive data from neighbor 1 ... )
  MPI_Irecv(... receive data from neighbor 2 ... )
  /* Now wait for all 4 communication requests to finish. */
  MPI_Waitall(...);
  /* OK, communication done!
  /* Now do the actual work for this iteration. */
  /* The same code executes for all processes, each of them working on its own data. */
}
/* Now the final result is computed, but the data is spread out over all processes. */
/* Maybe some filalization here to collect the final result, maybe write it to file. */
```