Erlangen Regional Computing Center

UNIVERSITÄT GREIFSWALD
Wissen lockt. Seit 1456

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Winter term 2020/2021

Parallel Programming with OpenMP and MPI

Dr. Georg Hager
Erlangen Regional Computing Center (RRZE) at Friedrich-Alexander-Universität Erlangen-Nürnberg
Institute of Physics, Universität Greifswald

Lecture 8: Introduction to the Message Passing Interface

High Performance Computing

Outline of course

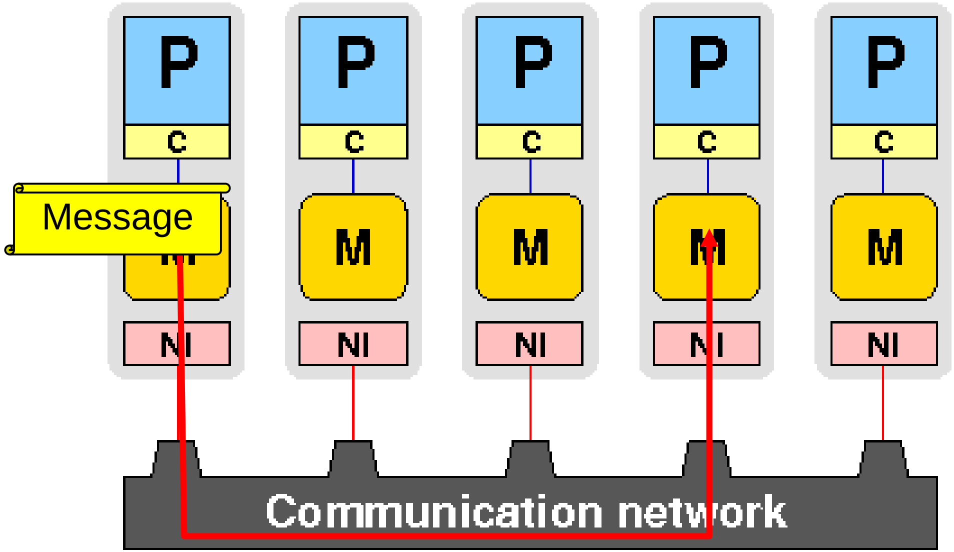
- Basics of parallel computer architecture
- Basics of parallel computing
- Introduction to shared-memory programming with OpenMP
- OpenMP performance issues
- Introduction to the Message Passing Interface (MPI)
- Advanced MPI
- MPI performance issues
- Hybrid MPI+OpenMP programming

The message passing paradigm

Distributed-memory architecture:

Each process(or) can only access its dedicated address space.

No global shared address space



Data exchange and communication between processes is done by explicitly passing messages through a communication network

Message passing library:

- Should be flexible, efficient and portable
- Hide communication hardware and software layers from application developer

The message passing paradigm

- Widely accepted standard in HPC / numerical simulation:
  Message Passing Interface (MPI)

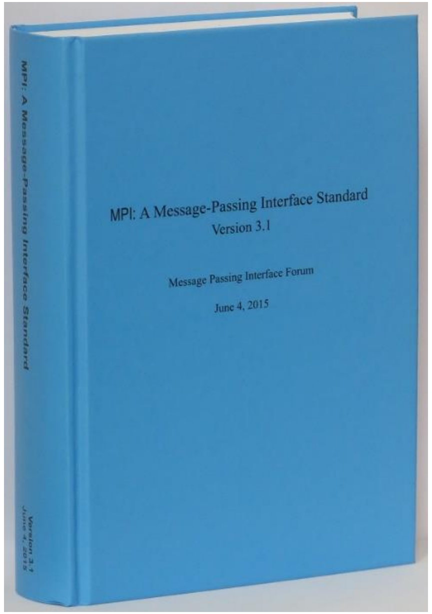
- Process-based approach: All variables are local!
- Same program on each processor/machine (SPMD)

- The program is written in a sequential language (Fortran/C[++])

- Data exchange between processes: Send/receive messages via MPI library calls
  - No automatic workload distribution

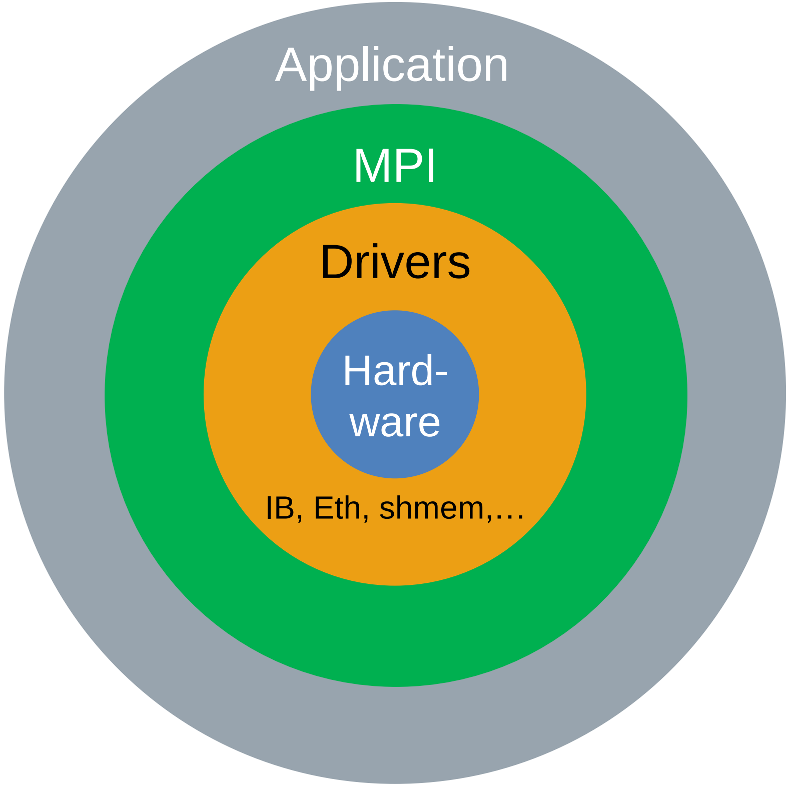
The MPI standard

- MPI forum – defines MPI standard / library subroutine interfaces
- Latest standard: MPI 3.1 (2015), 868 pages
  - MPI 4.0 under development
- Members (approx. 60) of MPI standard forum
  - Application developers
  - Research institutes & computing centers
  - Manufacturers of supercomputers & software designers
- Successful free implementations (MPICH, mvapich, OpenMPI) and vendor libraries (Intel, Cray, HP,…)
- Documents: **http://www.mpi-forum.org/**



MPI goals and scope

- Portability is main goal: architecture- and hardware-independent code
- Fortran and C interfaces (C++ deprecated)
- Features for supporting parallel libraries
- Support for heterogeneous environments (e.g., clusters with compute nodes of different architectures)



Erlangen Regional Computing Center

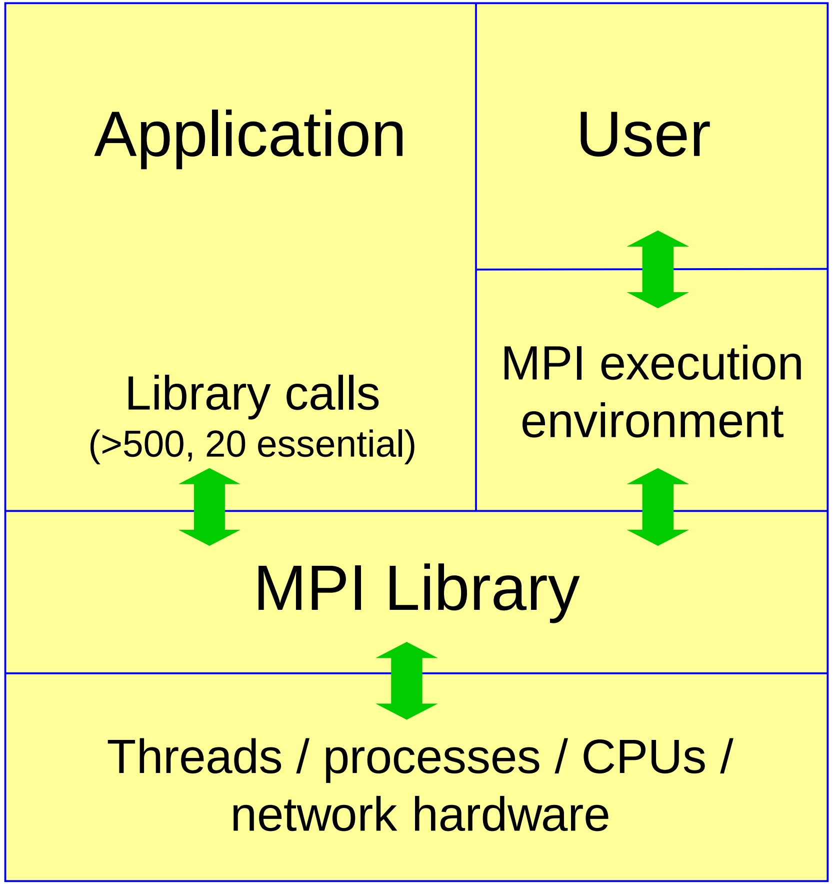
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

MPI in a nutshell

The beginner's MPI toolbox

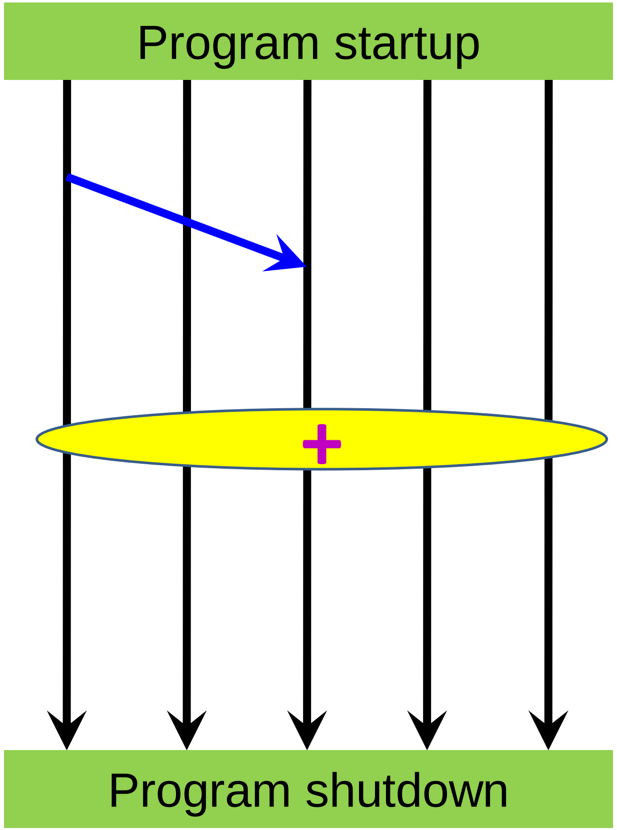
High Performance Computing

Architecture



- Operating system view:
  - Running processes
- Developer's view: Library routines for
  - coordination
  - communication
  - synchronization
- User's view: MPI execution environment provides
  - resource allocation
  - parallel program startup
  - other (implementation-dependent) behavior

Parallel execution in MPI



- Processes run throughout program execution
- MPI startup mechanism:
  - launches tasks/processes
  - establishes communication context ("communicator")
- MPI Point-to-point communication:
  - between pairs of tasks/processes
- MPI Collective communication:
  - between all processes or a subgroup
  - barrier, reductions, scatter/gather
- Clean shutdown by MPI

C and Fortran interfaces for MPI

- Required header files:
  - C: #include <mpi.h>
  - Fortran: include 'mpif.h'
  - Fortran90: use mpi / use mpi_f08
- Bindings:
  - C: error = MPI_Xxxx(...);
  - Fortran: call MPI_XXXX(...,ierror)
  - MPI constants (global/common): All upper case in C
- Arrays:
  - C: indexed from 0
  - Fortran: indexed from 1

MPI error handling

- C routines
  - return an `int` — may be ignored
- Fortran MPI routines
  - `ierror` argument — cannot be omitted!

- Return value `MPI_SUCCESS`
  - Indicates that all is fine

- Default: Abort parallel computation in case of other return values
  - but can also define error handlers (not covered here)

Initialization and finalization

- Details of MPI startup are implementation defined
- First call in MPI program: initialization of parallel machine

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

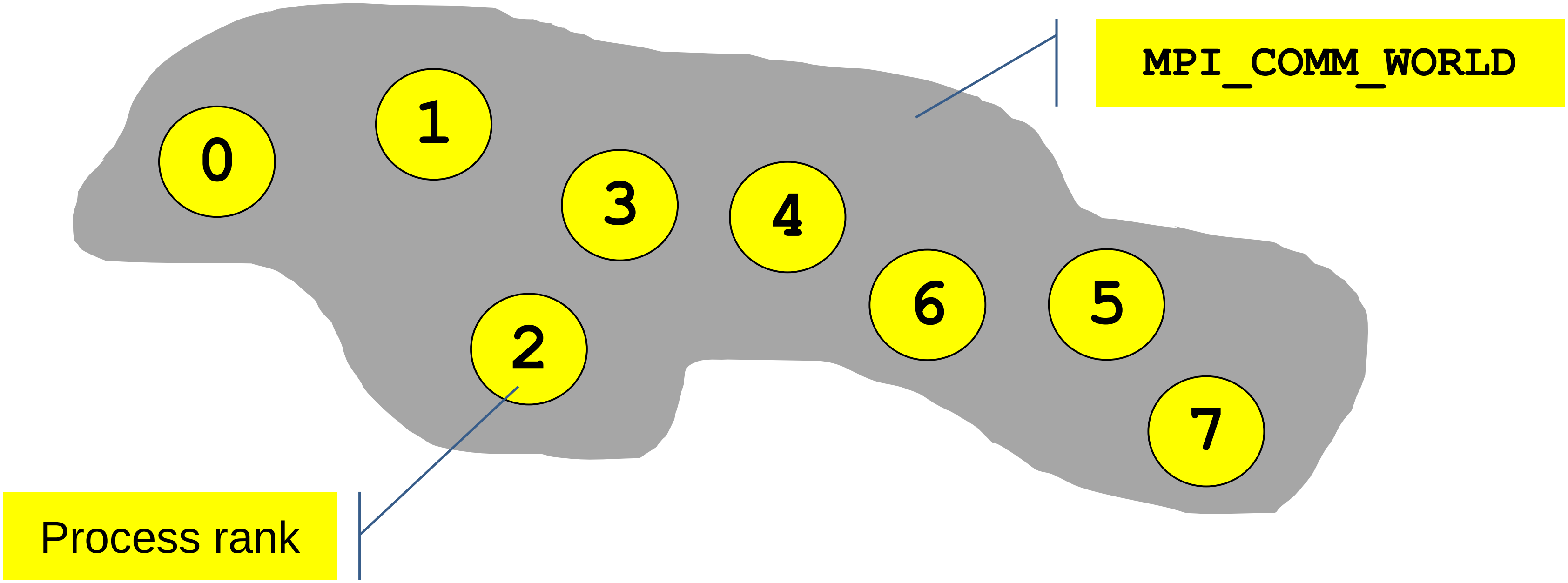
- Last call: clean shutdown of parallel machine

```
int MPI_Finalize();
```

  Only “master” process is guaranteed to continue after finalize
- Stdout/stderr of each MPI process
  - usually redirected to console where program was started
  - many options possible, depending on implementation

World communicator and rank

- `MPI_Init()` defines “communicator” `MPI_COMM_WORLD` comprising all processes



Communicator and rank

- Communicator defines a set of processes (`MPI_COMM_WORLD`: all)
- The rank identifies each process within a communicator
  - Obtain rank:

    ```
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ```

  - `rank =` 0,1,2,…, (number of processes in communicator – 1)
  - One process may have different ranks if it belongs to different communicators
- Obtain number of processes in communicator:

```
int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

MPI “Hello World!” in C

```
#include <mpi.h>

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

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

  printf(“Hello World! I am %d of %d\n”, rank, size);

  MPI_Finalize();
}
```

Never forget that these are pointers to the original varables!

Communicator required for (almost) all MPI calls

Compiling and running the code

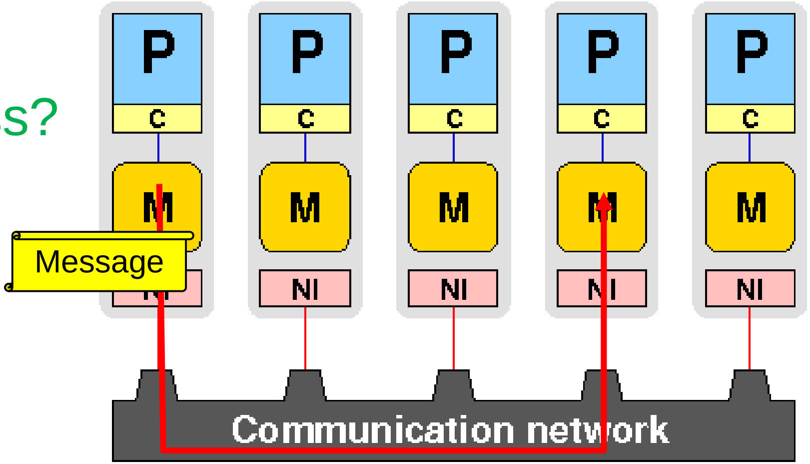
- Compiling/linking
  - Headers and libs must be found by compiler
  - Most implementations provide wrapper scripts, e.g.,
    - `mpif77` / `mpif90`
    - `mpicc` / `mpiCC`
  - Behave like normal compilers/linkers
- Running
  - Details are implementation specific
  - Startup wrappers: `mpirun`, `mpiexec`, `aprun`, `poe`

```
$ mpiCC -o hello hello.cc
$ mpirun -np 4 ./hello
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
```

- Details are implementation specific
  - Where/how are processes started?
  - Can I set the process-core affinity?
  - Where does the output go?
  - Do I need a shared file system?

Point-to-point communication: message envelope

- Which process is sending the message?
- Where is the data on the sending process?
- What kind of data is being sent?
- How much data is there?



- Which processes are receiving the message?
- Where should the data be left on the receiving process?
- How much data is the receiving process prepared to accept?

- Sender and receiver must pass their information to MPI separately

MPI point-to-point communication

- Processes communicate by sending and receiving messages
- MPI message: array of elements of a particular type

- Data types
  - Basic
  - MPI derived types

Predefined data types in MPI (selection)

MPI type	C type
MPI_CHAR	signed char
MPI_INT	signed int
MPI_LONG	signed long
MPI_LONG_LONG_INT	signed long long int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT32_T	int32_t
MPI_UINT64_T	int64_t
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_C_BOOL	_Bool
MPI_C_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_BYTE	N/A

8 binary digits

Data type matching: Same type in send and receive call required

Support for heterogeneous systems: automatic data type conversion

A similar list exists for Fortran, of course

MPI blocking point-to-point communication

- Point-to-point: one sender, one receiver
  - Identified by rank

- Blocking: After the MPI call returns,
  - the source process can safely modify the send buffer
  - the receive buffer (on the destination process) contains the entire message.

  - This is not the “standard” definition of “blocking”

Standard blocking send

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

`buf`	address of send buffer
`count`	# of elements
`datatype`	MPI data type
`dest`	destination rank
`tag`	message tag
`comm`	communicator

At completion

- Send buffer can be reused as you see fit
- Status of destination is unknown – the message could be anywhere

Standard blocking receive

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

buf	address of receive buffer
count	# of elements that fit into receive buffer
datatype	MPI data type
source	sending process rank
tag	message tag
comm	communicator
status	address of status object

At completion

- Message has been received successfully
- Message length, and probably the tag and the sender, are still unknown

Source and tag wildcards

- **`MPI_Recv`** accepts wildcards for the **`source`** and **`tag`** arguments:
  `MPI_ANY_SOURCE`, **`MPI_ANY_TAG`**

- Actual source and tag values are available in the status object:

```
MPI_Status s;
MPI_Recv(buf, count, datatype, MPI_ANY_SOURCE,
         MPI_ANY_TAG, MPI_COMM_WORLD, &s);
printf("Received from rank %d with tag %d\n",
        s.MPI_SOURCE, s.MPI_TAG);
```

Received message length

```
int MPI_Get_count(const MPI_Status *status,
                  MPI_Datatype datatype, int *count)
```

`status`	address of status object
`datatype`	MPI data type
`count`	address of element count variable

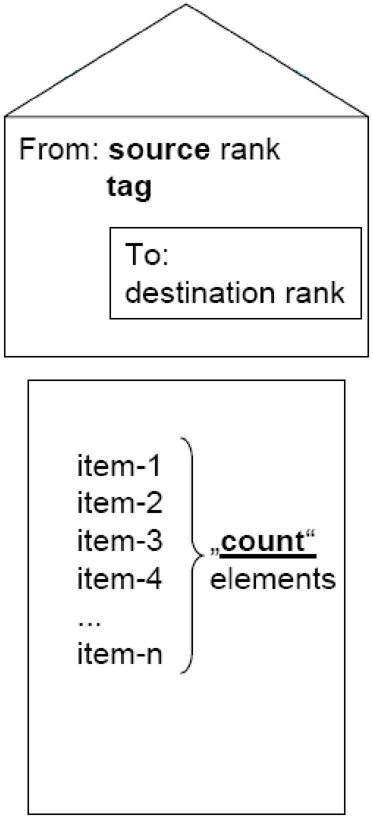
- Determines number of elements received

```
int count;
MPI_Get_count(&s, MPI_DOUBLE, &count);
```

Requirements for poit-to-point communication

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 (or **MPI_ANY_TAG** for receiver)
- message data types must match
- receiver's buffer must be large enough



Beginner's MPI toolbox

- Basic point-to-point communication and support functions:
  - `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 many items have I received?
  - `MPI_Finalize()` finish off

- Send/receive buffer may safely be reused after the call has completed
- `MPI_Send()` must have a specific target/tag, `MPI_Recv()` does not
- So far no explicit synchronization!

Example: parallel integration in MPI

```
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// integration limits
double a=0., b=2., res=0., tmp;
// limits for "me"
mya = a + rank * (b-a)/size;
myb = mya + (b-a)/size;
// integrate f(x) over my own chunk
psum = integrate(mya,myb);
```

Task: calculate $\int_a^b f(x)dx$ using (existing) function **integrate(x,y)**

- Split up interval [a,b] into equal disjoint chunks
- Compute partial results in parallel
- Collect global sum at rank 0

```
// rank 0 collects partial results
if(0==rank) {
  res = psum; // local result
  for(int i=1; i<size; ++i) {
    MPI_Recv(tmp,          // receive buffer
             1,            // array length
             MPI_DOUBLE,   // data type
             i,            // rank of source
             0,            // tag (unused here)
             MPI_COMM_WORLD,
             &status);    //status object
    res += tmp;
  }
  printf("Result: %.15lf\n", res);
} else { // ranks != 0 send results to rank 0
  MPI_Send(psum,       // send buffer
           1,          // message length
           MPI_DOUBLE,// data type
           0,          // rank of destination
           0,          // tag (unused here)
           MPI_COMM_WORLD);
}
```

Remarks on parallel integration example

- Gathering results from processes is a very common task in MPI – there are more efficient and elegant ways to do this (see later).
- This is a reduction operation (summation). There are more efficient and elegant ways to do this (see later).
- The "master" process waits for one receive operation to be completed before the next one is initiated. There are more efficient ways... You guessed it!
- "Master-worker" schemes are quite common in MPI programming but scalability to high process counts may be limited.
- Error checking is rarely done in MPI programs – debuggers are often more efficient if something goes wrong.
- Every process has its own **`res`** variable, but only the master process actually uses it → it's typical for MPI codes to use more memory than actually needed.

Some useful MPI calls

- `double MPI_Wtime();`
  Returns current time stamp
- `double MPI_Wtick();`
  Returns resolution of timer

- `int MPI_Abort(MPI_Comm comm, int errorcode);`
  - "Best effort" attempt to abort all tasks in communicator, deliver error code to calling environment
  - This is a last resort; if possible, shut down the program via `MPI_Finalize()`

Summary of beginner's MPI toolbox

- Starting up and shutting down the "parallel program" with **`MPI_Init()`** and **`MPI_Finalize()`**
- MPI task ("process") identified by rank (**`MPI_Comm_rank()`**)
- Number of MPI tasks: **`MPI_Comm_size()`**
- Startup process is very implementation dependent
- Simple, blocking point-to-point communication with **`MPI_Send()`** and **`MPI_Recv()`**
  - "Blocking" == buffer can be reused as soon as call returns
- Message matching
- Timing functions