Elementary Parallel Programming

R. Bader (LRZ)
G. Hager (RRZE)
V. Weinberg (LRZ)
Two Paradigms for Parallel Programming
Hardware Designs

- **Distributed Memory**
  - message passing
  - explicit programming required
  - Special design:
    - cache coherency protocol over interconnect
    - behaves like non-uniform shared memory

- **Shared Memory**
  - common address space for a number of CPUs
  - access efficiency may vary → SMP, (cc)NUMA
    (memory access time depends on the memory location relative to the processor)
  - many programming models
  - potentially easier to handle
  - hardware and OS support required
### Message Passing vs. Shared Memory: Programming Models

#### Distributed Memory
- **Same program on each processor/machine (SPMD) or Multiple programs with consistent communication structure (MPMD)**

- **Program written in a sequential language**
  - all variables process-local
  - no implicit knowledge of data on other processors

- **Data exchange between processes:**
  - send/receive messages via appropriate library
  - most tedious, but also the most flexible way of parallelization

- **Parallel library discussed here:**
  - Message Passing Interface, **MPI**

#### Shared Memory
- **Single Program on single machine**
  - UNIX Process splits off **threads**, mapped to CPUs for work distribution

- **Data**
  - may be process-global or thread-local
  - exchange of data not needed, or via suitable synchronization mechanisms

- **Programming models**
  - explicit threading (hard)
  - directive-based threading via **OpenMP** (easier)
  - automatic parallelization (very easy, but mostly not efficient)
Standards-based parallelism

- MPI standard
  - https://www.mpi-forum.org/docs/
- OpenMP standard
  - https://www.openmp.org/specifications/
Standards-based Parallelism

- **MPI standard**
  - MPI forum released version 2.2 in September 2009
  - MPI version 3.1 in June 2015
  - unified document („MPI1+2“)

- **OpenMP standard**
  - OpenMP 3.1 (July 2011) released by architecture review board (ARB)
    - feature update (tasking etc.)
  - OpenMP 4.0 (July 2013)
    - SIMD, affinity policies, accelerator support
  - OpenMP 4.5 (Nov 2015)
  - OpenMP 5.0 (Nov 2018)
    - two new tool interfaces, Multilevel memory systems

- **Base languages**
  - Fortran (77, 95)
  - C
  - C++ binding obsolescent → use C bindings

- **Base languages**
  - Fortran (77, 95)
  - C, C++ (Java is not a base language)

- **Resources:**
  - [http://www.mpi-forum.org](http://www.mpi-forum.org)
  - [http://www.openmp.org](http://www.openmp.org)
  - [http://www.compunity.org](http://www.compunity.org)
MPI Standard

Number of Pages in MPI Standard

1.0 (1994) 228
1.1 (1995) 231
1.3 (2008) 237
2.0 (1997) 362
2.1 (2008) 586
2.2 (2009) 623
3.0 (2012) 822
3.1 (2015) 836
Number of Pages in OpenMP Standard

- Fortran 1.0 (1997): 55
- Fortran 1.1 (1999): 76
- C/C++ 1.0 (1998): 77
- Fortran 2.0 (2000): 116
- C/C++ 2.0 (2002): 100
- 3.0 (2008): 318
- 3.1 (2011): 346
- 4.0 (2013) + Exam. 4.0.1 (2014): 226
- 4.0 (2013) + Exam. 4.0.2 (2015): 251
- 5.0 (2018): 666

Standard and Examples
Comparison of Number of Pages in Recent Standards

- MPI 3.1 (2015)
- OpenMP 4.0 (2013) + Exam. 4.0.2 (2015)
- OpenMP 5.0 (2018)
- OpenACC 2.6 (2017)
Compatibility with Sequential Execution

**MPI**

- Are semantics for sequential execution retained?
  - MPI environment required
  - not possible for some patterns

  **Do memory accesses occur in the same order?**
  - technically not possible
  - performance impact high

  **Are the same numeric results obtained for parallel execution?**
  - no associativity for model number operations
  - parallel execution reorders operations (may need to enforce order of e.g. reductions for reproducibility and/or numeric stability)

**OpenMP**

- yes, due to directive concept
- program may choose not to

**Are memory accesses occur in the same order?**
- no, due to relaxed memory consistency

- same as for MPI
- same as for MPI
Typical Parallelization Hierarchy

Node Performance = OpenMP + Low-Level Optimization

Parallelized by MPI library call

Multi-Threading (OpenMP)

Low-Level Optimization

Message Passing

DO \( i=1,l \) \quad \text{Inter-node MPI parallelization}

DO \( j=1,m \) \quad \text{Intra-node OpenMP processing}

DO \( k=1,n \) \quad \text{single processor execution}
OpenMP

Principles of
directive driven
shared memory parallelism
OpenMP Architecture

- **Operating system view:**
  - parallel work done by *threads*

- **Programmer’s view:**
  - *directives*: comment lines in code, e.g.
    - `#pragma omp parallel`
  - library routines, e.g.
    - `omp_get_num_threads()`
    - `omp_get_thread_num()`
    - `omp_get_max_threads()`

- **User’s view:**
  - *environment variables*
    - determine:
      - resource allocation
      - scheduling strategies
      - and other (implementation-dependent) behaviour, e.g.
        - `OMP_NUM_THREADS`
        - `OMP_SCHEDULE`
        - `OMP_NESTED`
OpenMP: Parallel execution

- **Program start**: only master thread runs

- **Parallel region**: team of worker threads is generated ("fork")

- Threads **synchronize** when leaving parallel region ("join")

- Only master executes sequential part (worker threads persist, but are inactive)

- **Task** and **data** distribution possible via directives

- **Nesting of parallel regions**: allowed, but level of support implementation dependent

- Usually optimal: one thread per processor core other resource mappings are allowed/possible
program hello
  use omp_lib
  implicit none
  integer :: nthr, myth

!$omp parallel private(myth)

!$omp single
  nthr = omp_get_num_threads()
!$omp end single

  myth = omp_get_thread_num()

  write(*,*) `Hello from `,myth, &
  & `of `, nthr

!$omp end parallel

end program hello

- **Parallel region directive:**
  - enclosed code executed by all threads ("lexical construct")
  - may include subprogram calls ("dynamic region")

- **Special function calls:**
  - module omp_lib provides interface
  - here: get number of threads and index of executing thread

- **Data scoping:**
  - uses a clause on the directive
  - myth thread-local: private
  - nthr process-global: shared

(Will be discussed in more detail later)
Compiling and running an OpenMP program

Compile:
```f90
f90 -openmp -o hello.exe hello.f90
```

Run:
```bash
export OMP_NUM_THREADS=4
./hello.exe
```

```
Hello from 0 of 4
Hello from 2 of 4
Hello from 3 of 4
Hello from 1 of 4
```

- **Special compiler switch**
  - activates OpenMP directives
  - generates threaded code
  - further suboptions may be available
  - each compiler has something different here

- **OpenMP environment**
  - defines runtime behaviour
  - here: number of threads used

- **Serial functionality of program**
  - (dis)order of output

Compile for serial run:
```f90
f90 -o hello.exe hello.f90
```

- may require special switch for „stub library“ and module file
OpenMP Fortran Syntax

**Specifications:**
- Fortran 77 style
  ```fortran
  include "omp_lib.h"
  ```
- Fortran 90 module *(preferred)*
  ```fortran
  use omp_lib
  ```

**Directives:**
- fixed form source:
  ```fortran
  !$OMP <directive> [clause [(<args>)]], ...
  ```
  sentinel starting in column 1, also: `$OMP, $omp`
- free form source *(preferred)*:
  ```fortran
  !$OMP <directive> [clause [(<args>)]], ...
  ```

**Conditional compilation:**
- In fixed form also sentinels `$, c$
  ```fortran
  myid = 0
  !$myid = omp_get_thread_num()
  ```

**Continuation line:**
- ```fortran
  !$OMP <directive> &
  !$OMP <clause>
  ```
OpenMP C/C++ Syntax

- Include file:
  ```c
#include <omp.h>
  ```

- Preprocessor directive: uses pragma feature
  ```c
#pragma omp <directive> [clause ...]
  ```

- Conditional compilation: OpenMP switch sets preprocessor macro
  ```c
#ifdef _OPENMP

... /* do something */
#endif
  ```

- Continuation line:
  ```c
#pragma omp directive \
  clause
  ```
Many (but not all) OpenMP directives support clauses
• more than one may appear on a given directive

Clauses specify additional information associated with the directive
• modification of directive’s semantics

“Simplest example” from above:
• private(...) appears as clause to the parallel directive

The specific clause(s) that can be used depend on the directive
OpenMP Syntax: Structured block

- Defined by braces in C/C++
- If explicitly specified in Fortran:
  - Code between begin/end of an OpenMP construct must be a complete, valid Fortran block
- Single point of entry:
  - No **GOTO** into block (Fortran), no **setjmp()** to entry point (C)
- Single point of exit:
  - **RETURN, GOTO, EXIT** outside block are prohibited (Fortran)
  - **longjmp()** and **throw()** must not violate entry/exit rules (C, C++)
  - **exception**: termination via **STOP** or **exit()**

Block structure example:
- C version of simplest program

```c
#include <omp.h>

int main() {
    int numth = 1;
    #pragma omp parallel
    {
        int myth = 0; /* private */
        #ifdef _OPENMP
        #pragma omp single
        numth = omp_get_num_threads();
        /* block above: one statement */
        myth = omp_get_thread_num();
        #endif
        printf("Hello from %i of %i\n",
               myth,numth);
    } /* end parallel */
}
```
Making parallel regions useful …

- divide up work between threads

Example:

- working on an array processed by a nested loop structure

```
real :: a(ndim, ndim)
...
!$omp parallel
!$omp do
do j=1, ndim
  do i=1, ndim
  a(i, j) = ...
  end do
end do
!$omp end do
!$omp end parallel
```

- iteration space of **directly nested loop** is sliced
### Synchronization behaviour:
- all threads (by default) **wait for completion** at the end of the work sharing region („barrier“)
- following references and definitions to an array element by other threads are therefore OK.

### Slicing of iteration space:
- „loop scheduling“
- default behaviour is implementation dependent
- usually as equal as possible chunks of largest possible size

### Additional clauses on !$OMP DO
- will be discussed in another talk

### C/C++ syntax:
```
#pragma omp for [clause]
for ( ... ) {
    ... // loop body
}
```

### Restrictions on loop structure:
- trip count must be **computable** at entry to loop  
  **disallowed**: C style loops modifying the loop variable, or otherwise violating the requirement, Fortran do while loop without loop control;
- loop body with single entry and single exit point
Two kinds of memory exist in OpenMP

- **Threads access globally shared memory**
- **Data can be shared or private**
  - shared data – one instance of an entity available to all threads (in principle)
  - private data – each per-thread copy only available to thread that owns it
- **Data transfer** transparent to programmer
- **Synchronization** takes place (is mostly implicit)
Defaults for data scoping

- All variables defined or referenced in a parallel region are shared
  - including global variables or variables in a COMMON block

Exceptions:
1. local variables of subroutines invoked (or blocks entered) inside a parallel region (unless they have the SAVE attribute)
2. loop variables of workshared loops and loops nested inside a workshared loop

which are (and always should be) private
- exceptions immutable if default is changed

Notes:
- many local variables/automatic arrays → default setting for thread specific stack size may be too small
- specifying SAVE and use of global variables:
  usually not a good idea
  code often not thread-safe

Changing defaults:
- default clause (for some directives)

```fortran
!$OMP <directive> default(private)
!$OMP <directive> default(none)
!$OMP <directive> default(shared)
!$OMP <directive> default(firstprivate)
```

- default(private) and default(firstprivate) (since 3.0) only in Fortran

export OMP_STACKSIZE=100M
Scoping: second-simplest example

- **Summation inside a loop**

```fortran
real :: s, stot
stot = 0.0
!$omp parallel private(s)
s = 0.0
!$omp do
do i=1, ndim
    ... ! workload ???
    s = s + ...
end do
!$omp end do
!$omp end parallel
```

- require thread-individual variable for partial sum calculated on each thread
- **but**: private copies of variables are *undefined* at entry to, and become *undefined* at exit of the parallel region
- **therefore**: collect partial sums to a *shared* variable defined after the worksharing region
- updates to shared variable must be specially protected:
  - use a *critical region*
  - only one thread at a time may execute (mutual exclusion)

(performance impact due to explicit synchronization)

**Note**: large workload inside loop improves threaded performance

© 2010-2019 LRZ and RRZE
Private variables - Masking

- **Masking relevant for**
  - privatized variables defined in scope outside the parallel region

```fortran
real :: s
s = ...
!$omp parallel private(s)
s = ...
... = ... + s
!$omp end parallel
... = ... + s
```

- Masking works by **persists** (inaccessible) for privatized variables defined in scope outside the parallel region.

Diagram:
- Shared variables `s` are accessible to all threads.
- Private variables `s` are accessible only within the parallel region.
- Fork/Join phases illustrate thread execution and variable accessibility over time.
The firstprivate clause

```
real :: s
    s = ...
!$omp parallel &
!$omp firstprivate(s)
    ... = ... + s
!$omp end parallel
    ... = ... + s
```

- **Extension of private:**
  - value of master copy is transferred to private variables
  - **restrictions:** not a pointer, not assumed shape, not a subobject, master copy not itself private etc.
The lastprivate clause

- **When to use?**
  - as little as possible
  - legacy code

- **Extension of private:**
  - additional semantics for work sharing
  - value from thread which executes last iteration of loop is transferred back to master copy (which must be allocated if it is a dynamic entity)
  - restrictions similar to `firstprivate`

```fortran
real :: s
s = ...
$omp parallel
$omp do lastprivate(s)
do i = ...
  s = ...
end do
$omp end do
$omp end parallel
... = ... + s
```
Reduction operations (1)

```fortran
real :: s

!$omp parallel
!$omp do reduction(+:s)
    do i = ...
        ...
        s = s + ...
    end do
!$omp end do
!$omp end parallel

Note: this improves on the summation example (no explicit critical region needed)
```

At synchronization point:
- reduction operation is performed
- result is transferred to master copy
- restrictions similar to `firstprivate`
Reduction operations (2)

- **Initial value of reduction variable**
  - depends on operation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>.and.</td>
<td>.true.</td>
</tr>
<tr>
<td>.or.</td>
<td>.false.</td>
</tr>
<tr>
<td>.eqv.</td>
<td>.true.</td>
</tr>
<tr>
<td>.neqv.</td>
<td>.false.</td>
</tr>
<tr>
<td>MAX</td>
<td>-HUGE(X)</td>
</tr>
<tr>
<td>MIN</td>
<td>HUGE(X)</td>
</tr>
<tr>
<td>IAND</td>
<td>all bits set</td>
</tr>
<tr>
<td>IEOR</td>
<td>0</td>
</tr>
<tr>
<td>IOR</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Consistency required**
  - operation specified in clause vs. update statement

- **Multiple reductions**:
  - multiple scalars, or an array:

```c
real :: x, y, z
$OMP do reduction(+:x, y, z)
real :: a(n)
$OMP do reduction(*:a)
$OMP do reduction(+:x, y) &
$OMP reduction(*:z)
```
MPI

Principles of message passing on distributed memory architectures
MPI Architecture

- Operating system view:
  - parallel work done by tasks

- Programmer’s view:
  - library routines for coordination, communication, synchronization

- User’s view:
  - MPI execution environment provides resource allocation, startup method, and other (implementation-dependent) behaviour
MPI Parallel execution

- **Tasks run throughout program execution**
  - all variables are local

- **Startup phase**: establishes communication context ("communicator") among all tasks

  - **Point-to-point data transfer**:
    - usually between pairs of tasks
    - usually coordinated
    - may be blocking or non-blocking
    - explicit synchronization is needed for non-blocking

  - **Collective communication**:
    - between all tasks or a subgroup of tasks
    - MPI 2 blocking-only (→ MPI 3)
    - reductions, scatter/gather operations

- **Clean shutdown**
MPI 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(argument,...,ierror)`
  - MPI constants (global/common): All 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!

© 2010-2019 LRZ and RRZE
MPI Error handling

- **C MPI routines**
  - return an `int` — may be ignored

- **Fortran MPI routines**
  - `ierror` argument — cannot be omitted!

- **Return value** `MPI_SUCCESS`
  - indicates that all went ok

- **Default:**
  - abort parallel computation in case of other return values
  - but can also define error handlers (not dealt with in this course)
Initialization and finalization (1)

- Each processor must start/terminate an MPI process
  - Usually handled automatically
  - More than one process per processor is mostly possible

- First call in MPI program: initialization of parallel machine
  
  ```
  call MPI_INIT(ierr)
  ```

- Last call: clean shutdown of parallel machine
  
  ```
  call MPI_FINALIZE(ierr)
  ```

- Only process with rank 0 (see later) is guaranteed to return from `MPI_FINALIZE`

- Stdout/stderr of each MPI process
  - usually redirected to console where program was started
  - many options possible, depending on implementation
Frequent source of errors: **MPI_Init() in C**

**C binding:**

```c
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:

```c
void init_all(int *argc, char***argv) {
  MPI_Init(argc, argv);
  ...
}
```

Depending on implementation, mistakes at this point might even go unnoticed until code is ported.
Communicator and rank (1)

- **MPI_INIT** defines "communicator" **MPI_COMM_WORLD**:

  - **MPI_COMM_WORLD** defines the processes that belong to the parallel machine
  - other communicators (subsets) are possible
  - rank labels processes inside a communicator
Communicator and rank (2)

- The rank identifies each process within a communicator (e.g. `MPI_COMM_WORLD`):
  - obtain rank:
    ```
    integer rank, ierror
    call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
    ```
  - `rank = 0,1,2,…, (number of MPI tasks – 1)`

- Obtain number of MPI tasks in communicator:
  ```
  integer size, ierror
  call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
  ```
Communicator and rank (3)

- **MPI_COMM_WORLD** is
  - effectively an **MPI-global** variable and required as argument for nearly all MPI calls

- **rank**
  - is target label for MPI messages
  - can drive user-defined directives what each process should do:

```plaintext
if (rank == 0) then
  ... ! *** do work for rank 0 ***
else
  ... ! *** do work for other ranks ***
end if
```
A very simple MPI program

program hello
  use mpi
  implicit none

  integer :: rank, size, ierror

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

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

  call MPI_FINALIZE(ierr)
end program
Compiling and running MPI code

- **Compile time:**
  - include files or module information file needed

- **Link time:**
  - MPI library required

- **Most implementations**
  - provide `mpif77`, `mpif90`, `mpicc` and `mpiCC` wrappers
  - not standardized, so variations must be expected e.g., with Intel-MPI (`mpiifort`, `mpiicc` etc.)
  - **Startup facilities**
    - `mpirun` (legacy)
    - `mpiexec`
    - site and implementation dependent

- **Compile:**
  - `mpif90 -o hello hello.f90`

- **Run on 4 processors:**
  - `mpirun -np 4 ./hello` or `mpiexec -n 4 ./hello`

- **Output:**
  - `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`
  - order undefined
MPI process communication

- Communication between two processes: Sending / Receiving of MPI-Messages

- MPI-Message:
  
  Array of elements of a particular MPI datatype

- MPI data types:
  
  - basic data types
  - derived data types
## Basic Fortran and C data types

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>FORTRAN datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MPI datatype</th>
<th>C datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR / MPI_SHORT</td>
<td>signed char / short</td>
</tr>
<tr>
<td>MPI_INT / MPI_LONG</td>
<td>signed int / long</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR / ...</td>
<td>unsigned char / ...</td>
</tr>
<tr>
<td>MPI_FLOAT / MPI_DOUBLE</td>
<td>float / double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
MPI data types cont’d

- **MPI_BYTE**: Eight binary digits
  - hack value, do not use

- **MPI_PACKED**: can implement new data types → however, it is more flexible to use ...

- Derived data types: Built at run time from basic data types
  - a topic in the advanced talk

- Data type **matching**: Same MPI data type in SEND and RECEIVE call
  - type must match on both ends in order for the communication to take place

- **Support for heterogeneous systems/clusters**
  - implementation-dependent
  - automatic data type conversion between systems of differing architecture may be needed
Point-to-point communication

- Communication between exactly two processes within the communicator.
  - Identification of source and destination via the rank within the communicator!
  - Blocking: MPI call returns after completion of the corresponding send/receive operation.
Blocking Standard Send: MPI_Send

- **Fortran syntax:**
  ```fortran
  call MPI_SEND (buf, count, datatype, dest, tag, &
                  comm, ierror)
  ```

  - **buf:** starting address of data buffer 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 **dest** is not defined – message may or may not have been received after return!

- **Send buffer may be reused after MPI_SEND returns**
Example: send array of 10 integers to task no. 5

```fortran
integer count, dest, tag
integer, allocatable :: field(:)
...
count=10; dest=5; tag=0
allocate(field(count)); ! initialize field
call MPI_SEND(field, count, MPI_INTEGER, dest, tag, &
                MPI_COMM_WORLD, ierror)
```

Source and destination may coincide
- beware potential deadlocks!
Blocking standard receive: MPI_Recv

- **MPI_RECV**: 1. receive data
  2. complete

- **Fortran syntax:**
  
  ```fortran
  call MPI_RECV (buf, count, datatype, source, tag, comm, &
                 status, ierror)
  ```

  - **buf**  
    size of buffer **must** be $\geq$ 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
    (is a complex object - see next slide)
Handling status information

- **MPI status provides additional information about the message**
  - size, source, tag, error code – may not be otherwise known if wildcards are used
  - can also use MPI_STATUS_IGNORE in some contexts

- **MPI_status in Fortran**
  - Array of integers of size MPI_STATUS_SIZE
  - index values for query: MPI_SOURCE, MPI_TAG, MPI_ERROR

  ```fortran
  integer :: &
  status(MPI_STATUS_SIZE)
  ```

- **Inquiring message length needs an additional MPI call:**
  ```fortran
  call MPI_GET_COUNT(status, datatype, count, ierror)
  ```
  - count is output argument
  - datatype must be the same datatype used in the MPI call that produced the status variable

- **MPI_status in C/C++**
  - Structure of type MPI_Status
  - hand a reference to MPI_Recv
  - component names for query:
    status.MPI_SOURCE, status.MPI_TAG, status.MPI_ERROR
MPI_Recv Example

- Example: receive array of REALs from any source

```fortran
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)

- Obtain number of actually received items:

call MPI_GET_COUNT(status, datatype, count, ierror)
```
For a communication to succeed:

- sender must specify a valid destination.
- receiver must specify a valid source rank (or \texttt{MPI\_ANY\_SOURCE}).
- communicator must be the same (e.g., \texttt{MPI\_COMM\_WORLD}).
- tags must match.
- message datatypes must match.
- receiver's buffer must be large enough (otherwise result is undefined!)
Summary of basic MPI API calls

- 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 many items 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** must have a specific target/tag, **MPI_RECV** does not
Write a parallel program in which a master process collects some data (e.g., numbers to sum up) from the others

```fortran
program collect
  use mpi
  implicit none
  integer :: i,size,rank,ierror, &
       status(MPI_STATUS_SIZE)
  integer :: number,sum
  call MPI_INIT(ierr)
  call MPI_COMM_RANK (MPI_COMM_WORLD, &
        rank,ierror)
  call MPI_COMM_RANK (MPI_COMM_WORLD, &
        rank,ierror)
  call MPI_SEND(rank,1,MPI_INTEGER, &
        0, 0, MPI_COMM_WORLD, ierror)
endif
end program
```

```fortran
if(rank.eq.0) then
  sum=0
  call MPI_COMM_SIZE(MPI_COMM_WORLD,&
       size,ierror)
  do i=1,size-1
    call MPI_RECV(number,1, &
      MPI_INTEGER, MPI_ANY_SOURCE, &
      MPI_ANY_TAG, MPI_COMM_WORLD, &
      status, ierr)
    sum=sum+number
  enddo
  write(*,*) 'The sum is ',sum
else
  call MPI_SEND(rank,1,MPI_INTEGER, &
        0, 0, MPI_COMM_WORLD, ierror)
endif
```

```fortran
call MPI_FINALIZE(ierr)
```
First complete MPI example in C

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

```c
#include <mpi.h>

int main(int argc, char *argv[]) {
    int i, size, rank;
    int sum, number;
    MPI_Status status;

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

    if(rank==0){
        sum=0;
        MPI_Comm_size(MPI_COMM_WORLD,&size);
        for(i=0;i<size-1;i++) {
            MPI_Recv(&number,1,MPI_INT,MPI_ANY_SOURCE ,MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            printf("Got number: %i\n", number);
            sum+=number;
        }
        printf("The sum is %i\n", sum);
    }
    else {
        MPI_Send(&rank,1,MPI_INT, 0,
                 0,MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```
Remarks:

- gathering results from processes is a very common task in MPI – there are more efficient ways to do this (see later).
- this is a reduction operation (summation). There are more efficient ways to do this (see the following slides).
- 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
- error checking is rarely done in MPI programs – debuggers are often more efficient if something goes wrong
- every process has its own sum variable, but only the master process actually uses it
Collective Communication: Introduction

Collective communication always involves every process in the specified communicator

- **Features:**
  - all tasks 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
  - type-matching conditions more strict than in point-to-point
  - sent message **must** fill receive buffer (count is exact)
  - no tags

- **Can be manually constructed from point-to-point communications**
  - however, collective calls allow optimized internal implementations e.g., tree based algorithms
A one-to-many communication
Every process receives one copy of the message from a root process

Syntax:

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

(root may be 0, but there is no "default" root process)
Collective Communication: Reduction Operations

- Combine data from several processes to produce a single result.
Global Operations: Introduction

Compute a result which involves data distributed across all processes of a group

- Example: global maximum of a variable: max( var[rank] )
- MPI provides 12 predefined operations (see next slide)
- Definition of user defined operations (not treated here): 
  \texttt{MPI\_OP\_CREATE} & \texttt{MPI\_OP\_FREE}
- MPI assumes that the operations are associative!
  (floating point operations may be not exactly associative because of rounding errors)
## Global Operations: Predefined Operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
<th>Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
<td>MPI_BAND</td>
<td>Bit-AND</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
<td>MPI_BOR</td>
<td>Bit-OR</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical XOR</td>
<td>MPI_BXOR</td>
<td>Bit-XOR</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum+ Position</td>
<td>MPI_MINLOC</td>
<td>Minimum+ Position</td>
</tr>
</tbody>
</table>

Position: also returns the rank of the (first) process, where the MAX or MIN was found
Global Operations: Syntax

- Results stored on root process:
  
  call 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 are desired to be stored on all processes:

- MPI_ALLREDUCE: No root argument
  - combination of MPI_REDUCE and MPI_BCAST
Global Operations: Example

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

Process Data distribution


$\texttt{MPI\_REDUCE(...,e,4,MPI\_INT, MPI\_MAX,0,...)}$