Elements of OpenMP and MPI

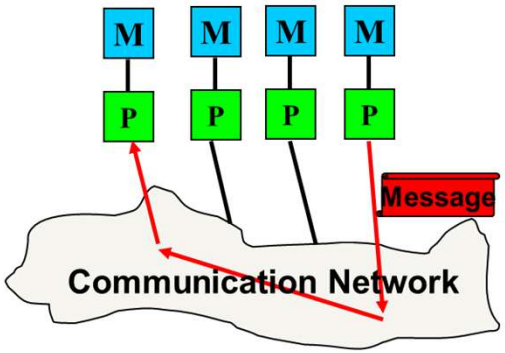
Reinhold Bader (LRZ)

Georg Hager (NHR@FAU)

Volker Weinberg (LRZ)

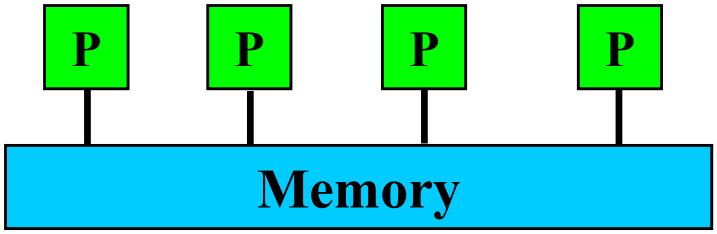
Two Paradigms for Parallel Programming

- 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



Two Paradigms for Parallel Programming

Distributed Memory

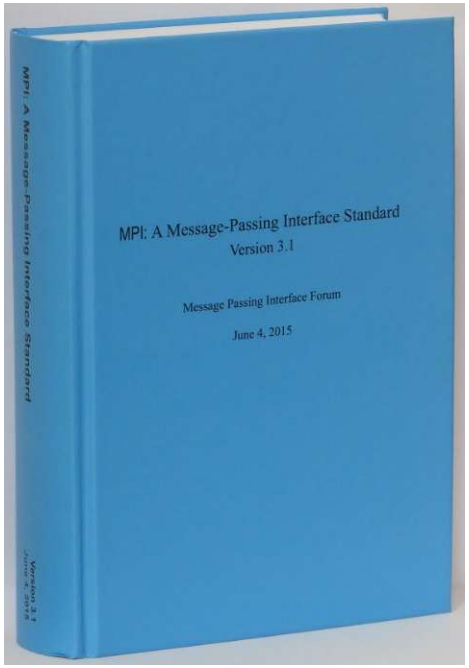
- **Same program on each processor/machine (SPMD) or**
  Multiple programs with consistent communication structure (MPMD)
- **Program written in a sequential language**
  - all variables process-local
  - no implicit knowledge of data on other processors
- **Data exchange between processes**
  - send/receive 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

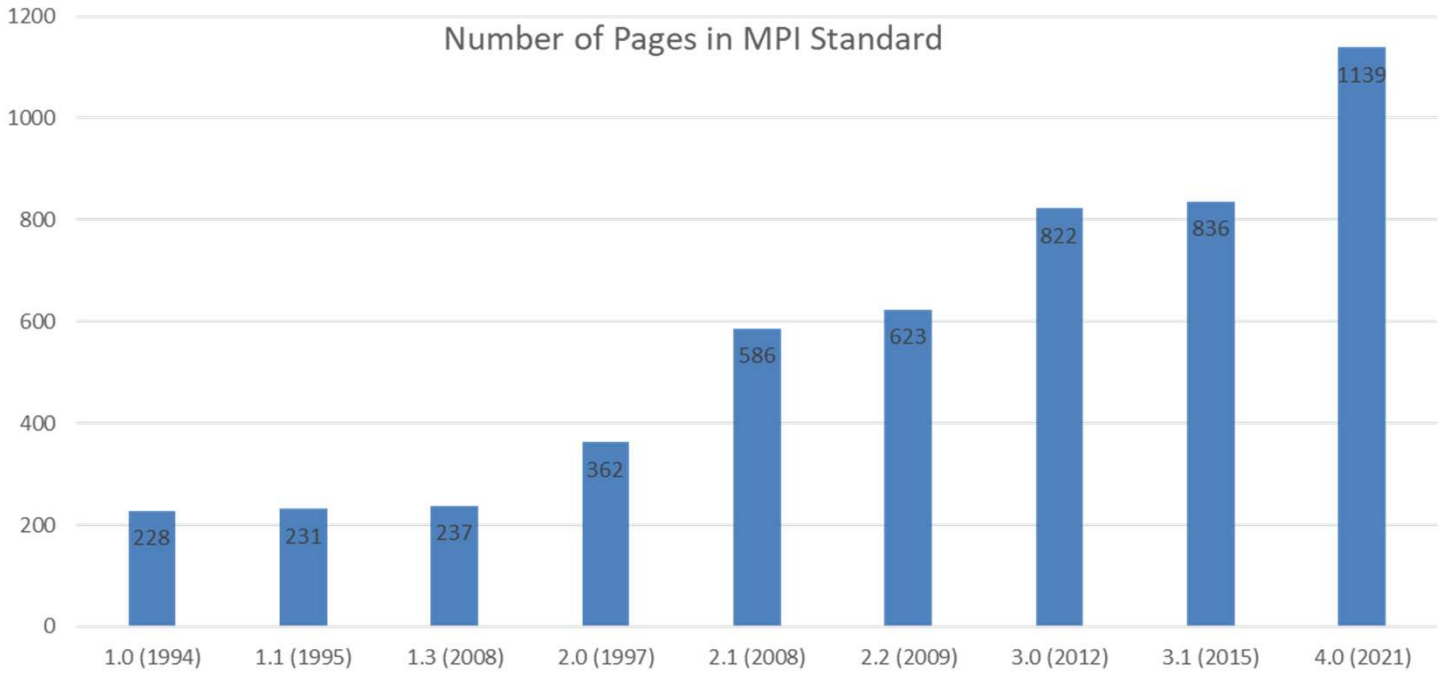
https://www.openmp.org/specifications/

Two Paradigms for Parallel Programming

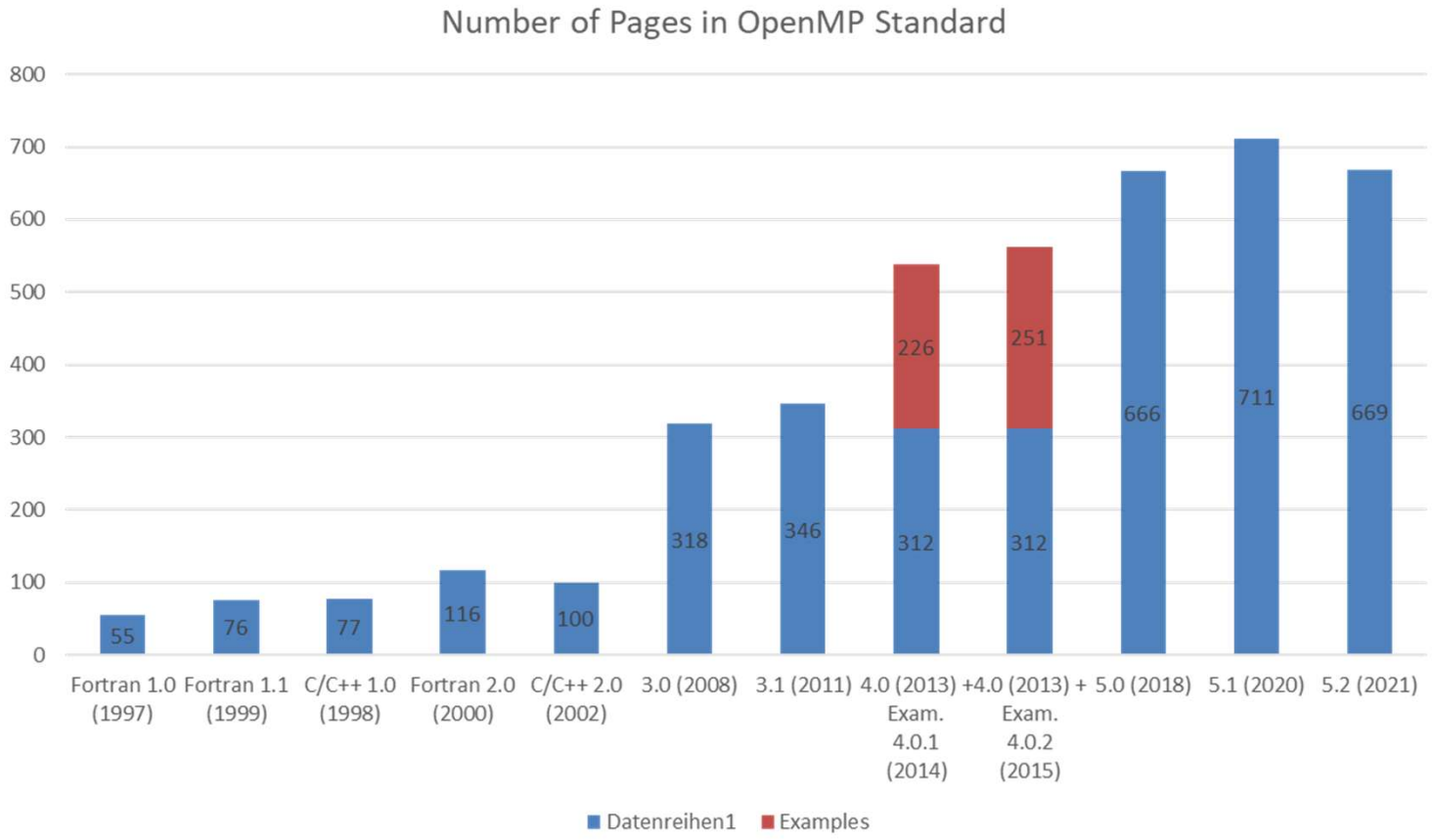
- **MPI Standard**
  - MPI version 1.0 in May 1994
  - MPI version 2.0 in July 1997
  - MPI version 3.0 in September 2012
  - MPI version 4.0 in June 2021
  - MPI version 4.1 in November 2023.
- **Base Languages**
  - Fortran
  - C
- **Resources**
  - http://www.mpi-forum.org

- **OpenMP Standard**
  - OpenMP 1.0 in 1997 (Fortran) / 1998 (C, C++)
  - OpenMP 3.0 (May 2008)
    - tasking etc.
  - OpenMP 4.0 (July 2013)
    - SIMD, affinity policies, accelerator support
  - OpenMP 5.0 (Nov 2018)
    - two new tool interfaces, multilevel memory systems
  - OpenMP 6.0 (Nov 2024)
    - improvements in usability and fine grain control
- **Base Languages**
  - Fortran
  - C, C++
- **Resources**
  - http://www.openmp.org

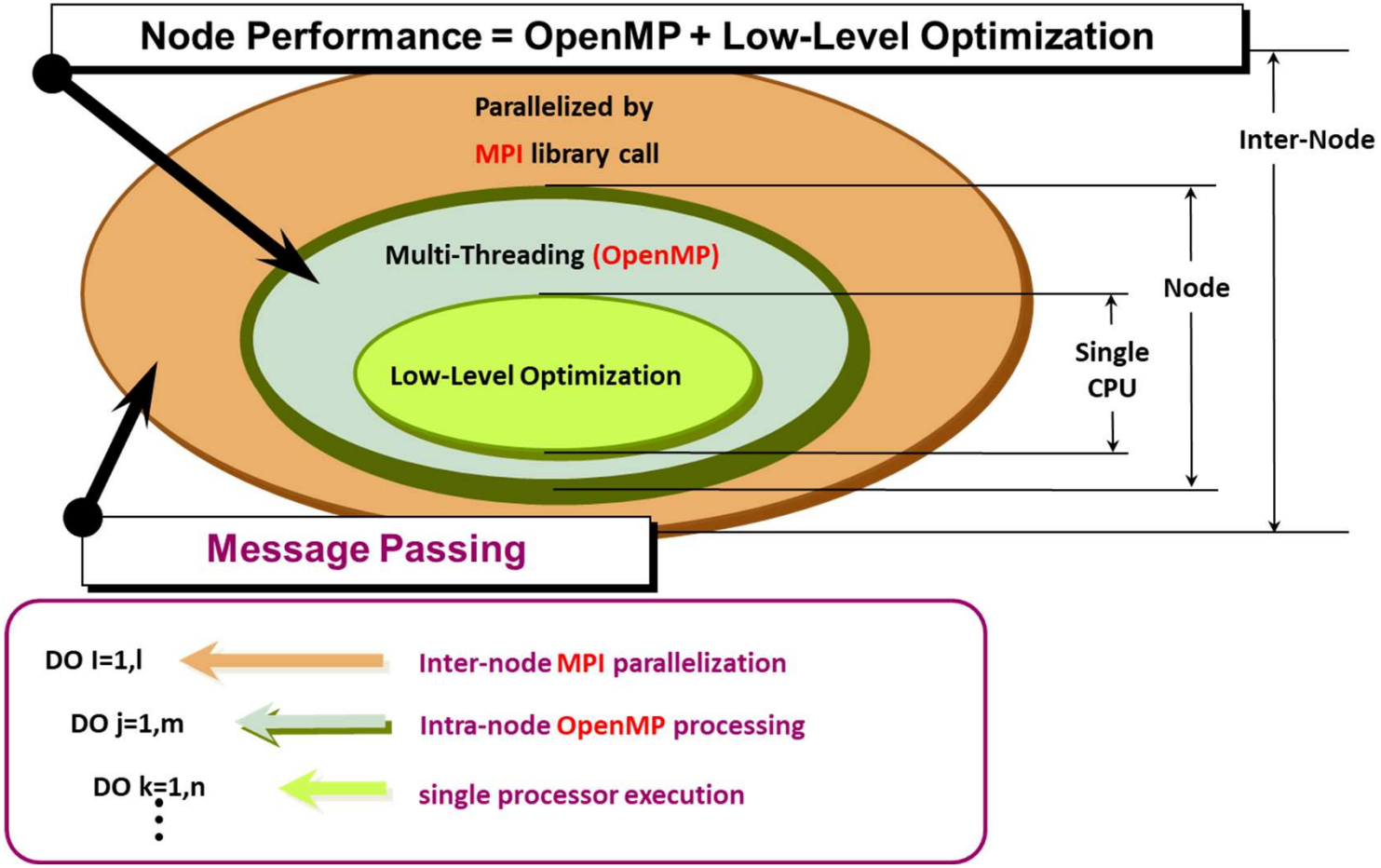
MPI Standard



OpenMP Standard



Typical Parallelization Hierarchy



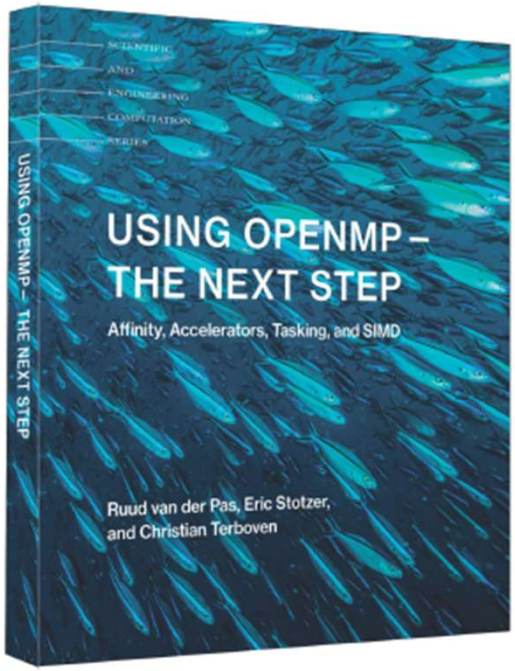
OpenMP

Principles of Directive Driven Shared Memory Parallelism

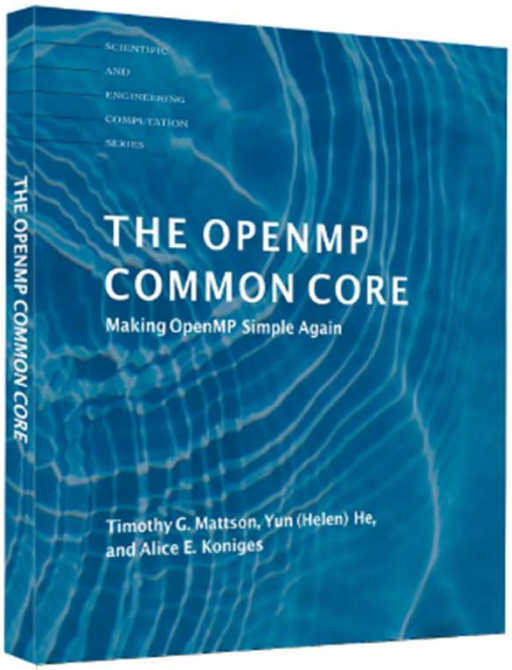
OpenMP Architecture Review Board (ARB)

The mission of the OpenMP ARB (Architecture Review Board) is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable.

Recent Books about OpenMP

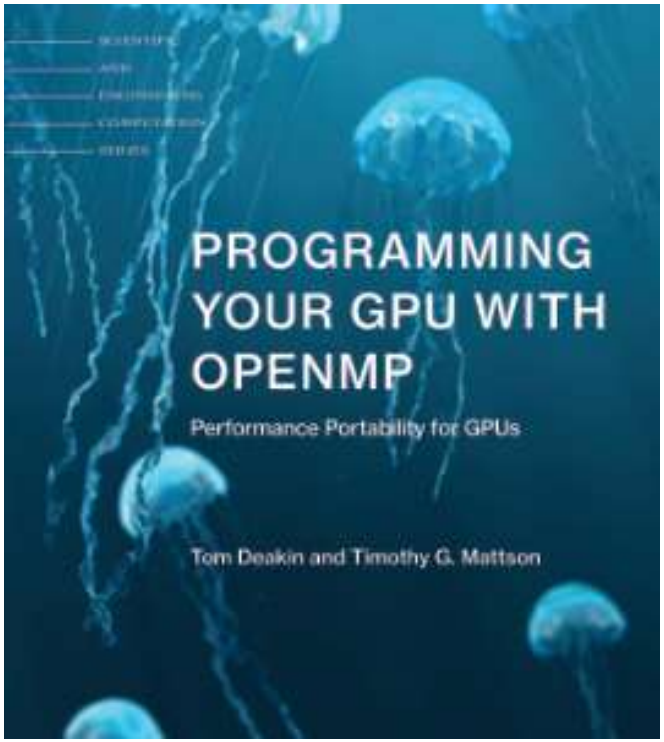


Covers all of the
OpenMP 4.5 features, 2017



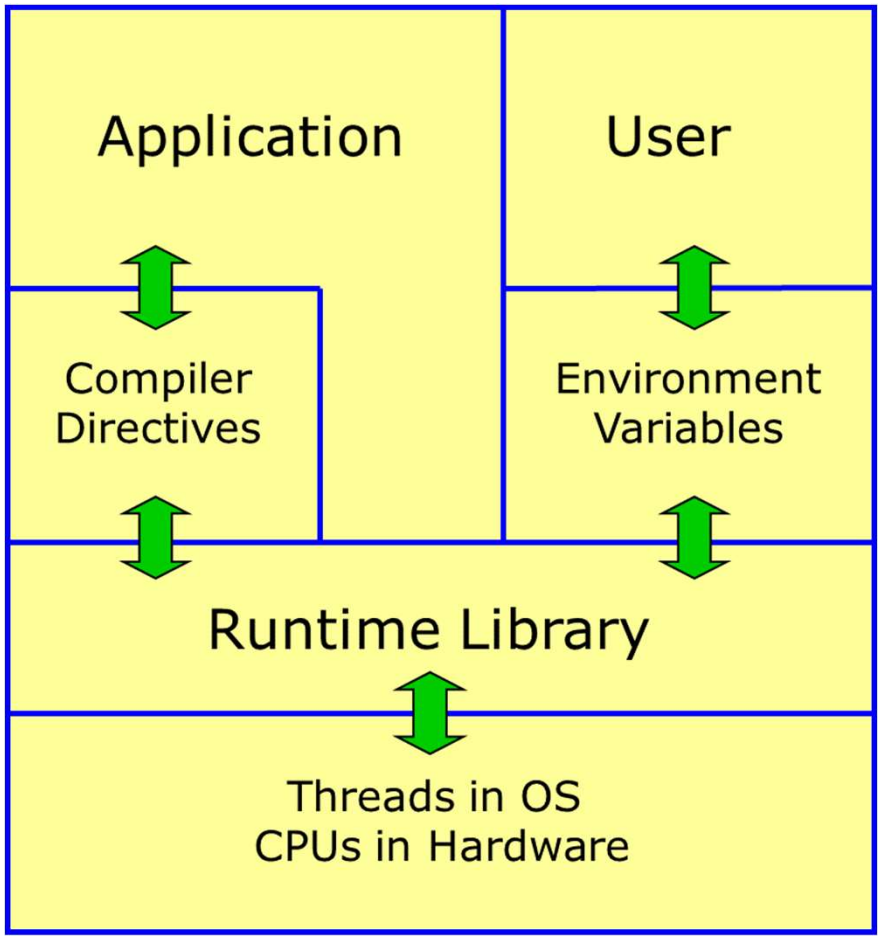
Introduces the
OpenMP Common Core, 2019

Recent Books about OpenMP



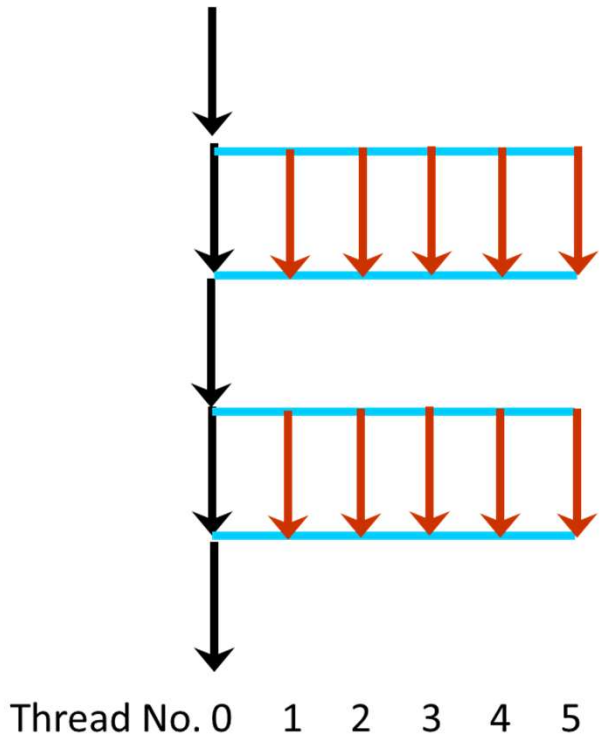
Covers all about Accelerator Programming, 2023

Two Paradigms for Parallel Programming



- **Operating system view:**
  - parallel work done by **threads**
- **Programmer's view:**
  - **directives**: comment lines in code, e.g.
    - `!$omp parallel`
    - `#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`

Two Paradigms for Parallel Programming



- Program start: only initial thread (formerly known as master thread) runs
- Parallel region: team of worker threads is generated (“fork”)
- Threads synchronize when leaving parallel region (“join”)
- Only initial thread 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

Parallel region: Simplest Program Example: Fortran

```
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
  - 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)

Parallel region: Simplest Program Example: C/C++

```
#include <stdio.h>
#include <omp.h>
int nthr, myth;
int main(int arc, char *argv[])
{
#pragma omp parallel private(myth)
  {
    #pragma omp single
    nthr = omp_get_num_threads();

    myth = omp_get_thread_num();
    printf("Hello from %i of %i\n", myth, nthr);
  }
}
```

- **Parallel region directive:**
  - enclosed code executed by **all** threads
  - may include subprogram calls („dynamic region“)
- **Special function calls:**
  - Include file <omp.h>
  - 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 Fortran (e.g. with Intel compiler):

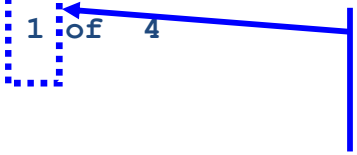
```
ifx -qopenmp –o hello.exe hello.f90
```

Compile C (e.g. with Intel compiler):

```
icx -qopenmp –o hello.exe hello.f90
```

Run:

```
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
```



ordering not reproducible

Compile for serial run (e.g. with Intel compiler):

```
ifx -qopenmp-stubs –o hello.exe hello.f90
```

- special switch for „stub library“

- **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

OpenMP Fortran Syntax

- Specifications:
  - Fortran 77 style

```
include "omp_lib.h"
```

  - Fortran 90 module (**preferred**)

```
use omp_lib
```

- Directives:
  - fixed form source:

```
C$OMP <directive> [<clause [(<args>)]>, …]
```

sentinel starting in column 1,
also : ***$OMP, !$OMP**

  free form source (preferred):

```
!$OMP <directive> [<clause [(<args>)]>, …]
```

- Conditional compilation:

```
 myid = 0
!$ myid = omp_get_thread_num()
```

  - In fixed form also sentinels *$, c$
- Continuation line:

```
!$OMP <directive> &
!$OMP <clause>
```

OpenMP C/C++ Syntax

- Include file:

```
#include <omp.h>
```

- Preprocessor directive: uses pragma feature

```
#pragma omp <directive> [clause …]
```

- Conditional compilation: OpenMP switch sets preprocessor macro

```
#ifdef _OPENMP
  … /* do something */
#endif
```

- Continuation line:

```
#pragma omp directive \
   clause
```

OpenMP Syntax: Remarks on Clauses

- 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++**
- **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()`

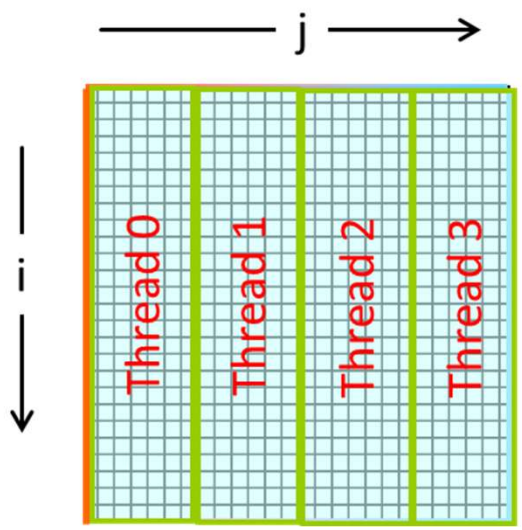
- **Block structure example:**
  - C version of simplest program

```
#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 */
}
```

Work Sharing in OpenMP (1): Fortran

- **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
```

j-loop is sliced

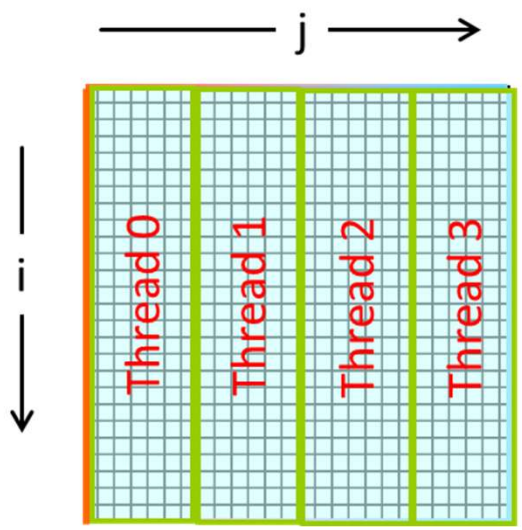
synchronization between threads

further parallel execution

  - iteration space of **directly nested loop** is sliced

Work Sharing in OpenMP (1): C/C++

- **Making parallel regions useful …**
  - divide up work between threads
- **Example:**
  - working on an array processed by a nested loop structure



  - iteration space of **directly nested loop** is sliced

```
float a[ndim][ndim];

int main(int arc, char *argv[])
{
#pragma omp parallel
  {
#pragma omp for
    for(int j=0;j<ndim;j++) {
      for(int i=0;i<ndim;i++) {
        a[i][j]= …;
      }
    }
  …
  }
}
```

j-loop is sliced

synchronization between threads

further parallel execution

Work Sharing in OpenMP (2)

- **Synchronization behaviour:**
  - all threads (by default) **wait for completion** at the end of the work sharing region („barrier“)
  - following references and definitions using 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` / `#pragma omp for`**
  - will be discussed in advanced OpenMP talk

- **Fortran syntax:**

```
!$omp do [clause]
do ...
   ...   // loop body
end do
```

- **C/C++ syntax:**

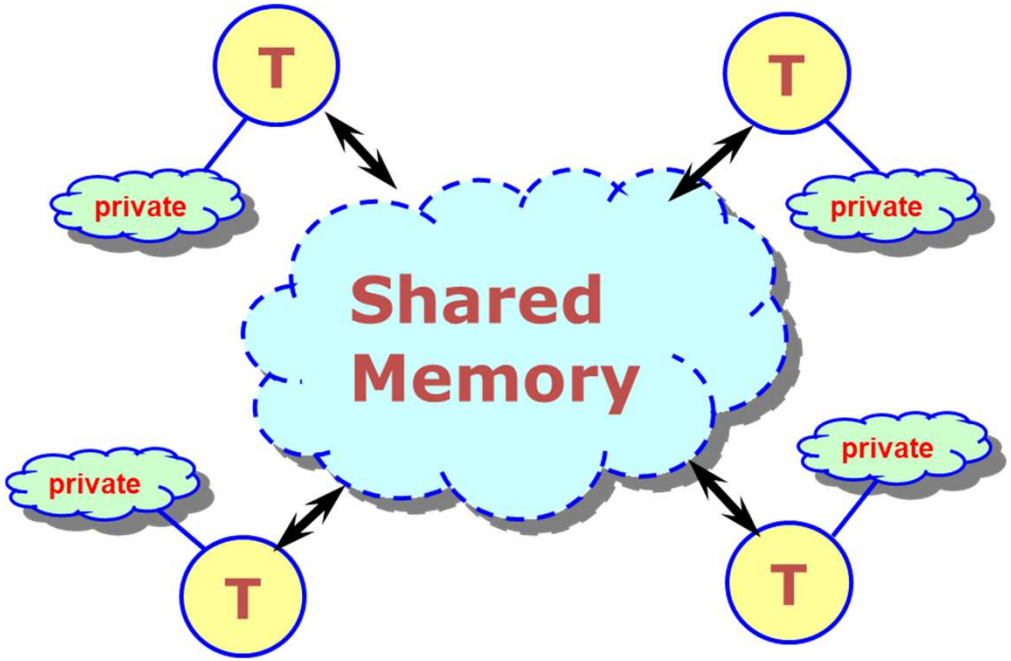
```
#pragma omp for [clause]
for ( ... )  {
      ...   // loop body
}
```

- **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

Memory Model

- **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
  - private data – each per-thread copy only available to thread that owns it
- **Data transfer** transparent to programmer
- **Synchronization** takes place (is mostly implicit)
- **threadprivate variables**
  - see advanced OpenMP talk

Data-Sharing Attributes

- By default most variables are shared
  - local variables outside the scope of construct
  - static/global (C/C++) or save/common (Fortran) variables

```
int s = 1;

#pragma omp parallel
{
  int p = omp_get_thread_num();
  printf("s=%d p=%d\n", s, p);
}
```

- Except
  - variables* defined inside the construct are private
    - i.e. declared inside `{}`-block or `BLOCK/END BLOCK`
  - variables* local to functions/routines called from within the region are private
  - loop iteration variables of worksharing loops are private

* non-static (C/C++) or without save attribute (Fortran)

Data-Sharing Attribute Clauses

- Clauses for explicitly specifying how a variable should be treated
  - supported by several directives, e.g., `parallel`, `do`/`for`, `single`, `sections`, `task`, ...
- Clauses:
  - `shared(var1, var2, …)`
  - `private(var1, var2, …)`
  - private + special operation
    - `firstprivate(var1, var2, …)`
    - `lastprivate`, for `do`/`for` construct
- Change default:
  - Fortran `default(shared|private|firstprivate|none)`
  - C/C++: `default(shared|none)`
  - best practice: `default(none)`
    - every variable referenced must appear in a `shared`/`private`/… clause
    - avoids incorrect assumptions about shared/private

Scoping: Second-Simplest Example: Fortran

- Summation inside a loop

```
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

!$omp critical
  stot = stot + s
!$omp end critical

!$omp end parallel
```

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

- 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)

Scoping: Second-Simplest Example: C/C++

- Summation inside a loop

```
 float s, stot;
 stot = 0.;
#pragma omp parallel private(s)
  {
    s = 0.;
#pragma omp for
    for(int i=0;i<ndim;i++) {
      … // workload
      s = s + … ;
    }
#pragma omp critical
    {
      stot = stot + s;
    }
  }
```

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

- 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)

Private Variables – Masking: Fortran

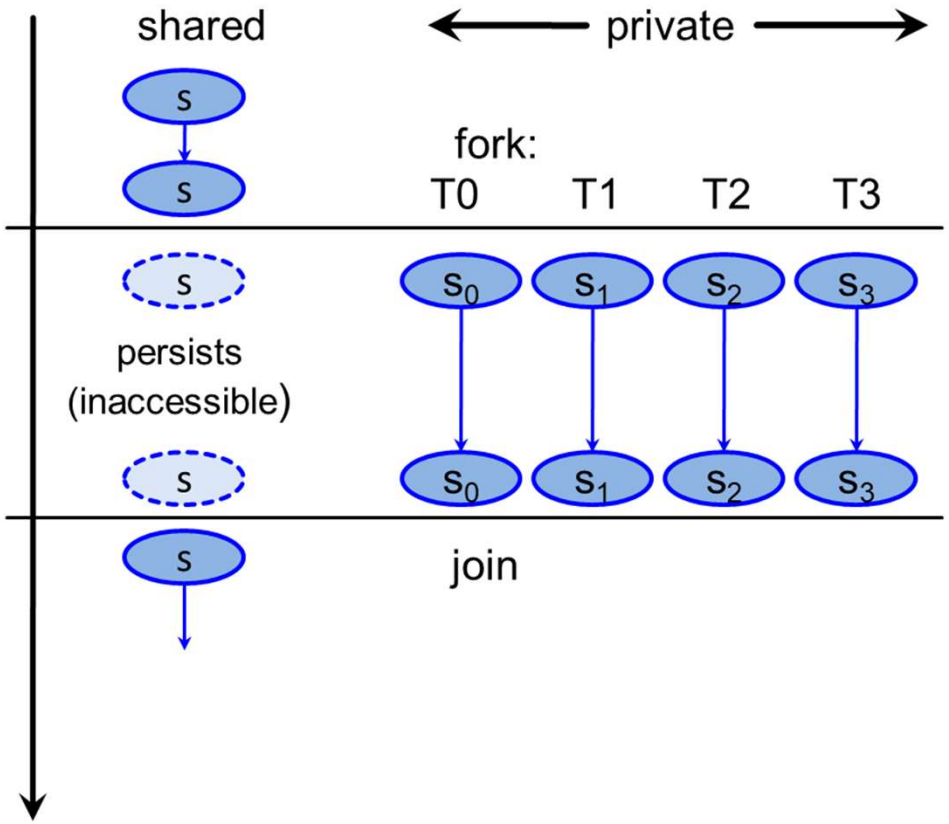
```
real :: s

  s = …

!$omp parallel private(s)
  s = …
  … = … + s
!$omp end parallel

  … = … + s
```

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



Private Variables – Masking: C/C++

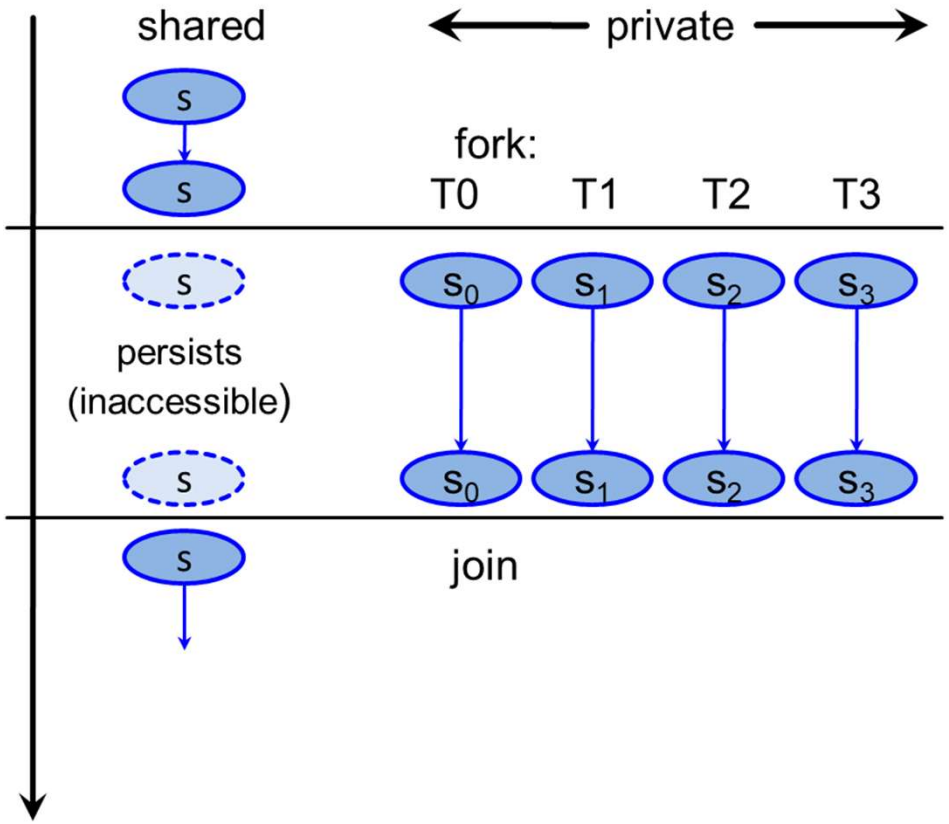
```
float s;

  s = … ;

#pragma omp parallel private(s)
{
  s = … ;
  … = … + s;
}

  … = … + s;
```

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



The firstprivate Clause: Fortran

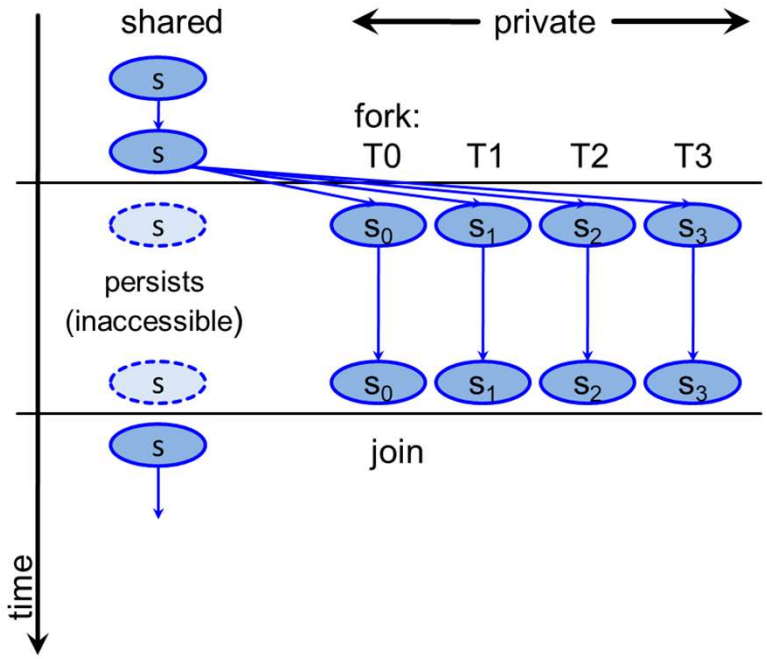
```
real :: s

  s = …

!$omp parallel 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.

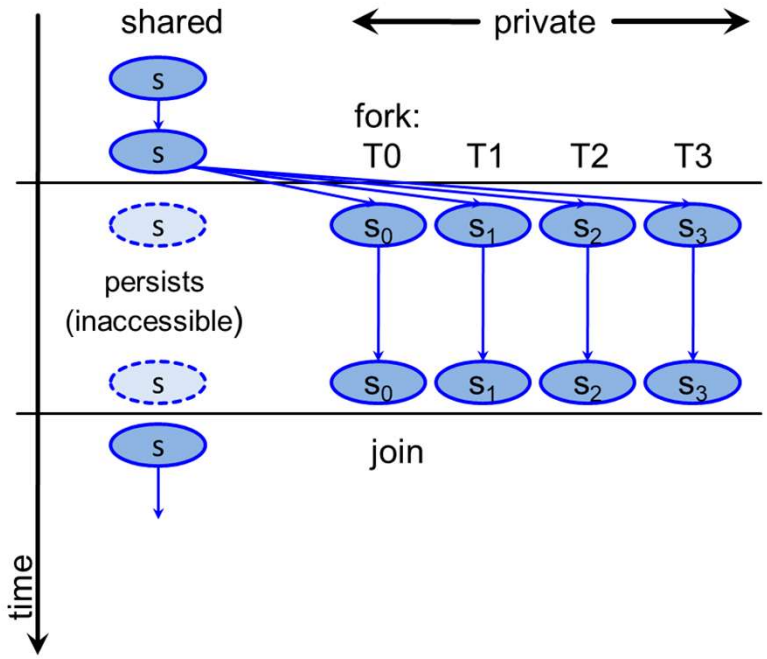
The firstprivate Clause: C/C++

```
float s;

  s = … ;

#pragma omp parallel firstprivate(s)
{
  … = … + s;
}

  … = … + s;
```



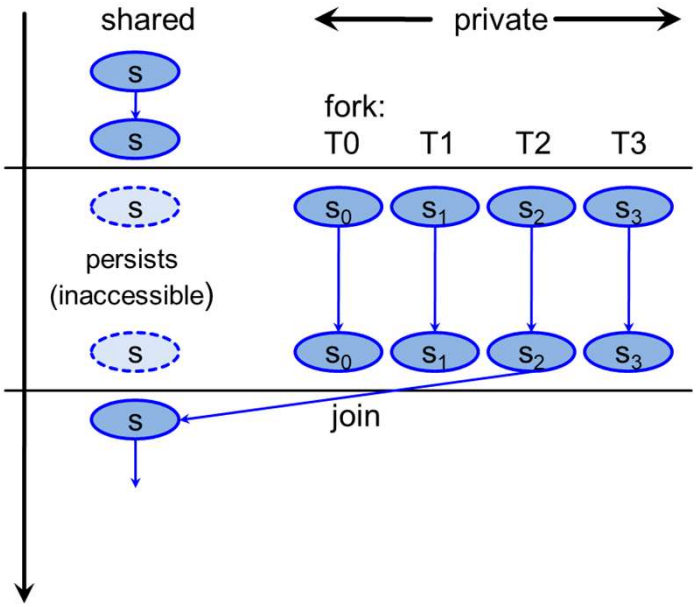
- **Extension of private:**
  - value of master copy is transferred to private variables
  - **restrictions:** not a pointer, not assumed shape, not a subobject, master copy not itself private etc.

The lastprivate Clause: Fortran

```
real :: s

  s = …
!$omp parallel
!$omp do lastprivate(s)
  do i = …
     s = …
  end do
!$omp end do
!$omp end parallel
  … = … + s
```

on work sharing directive



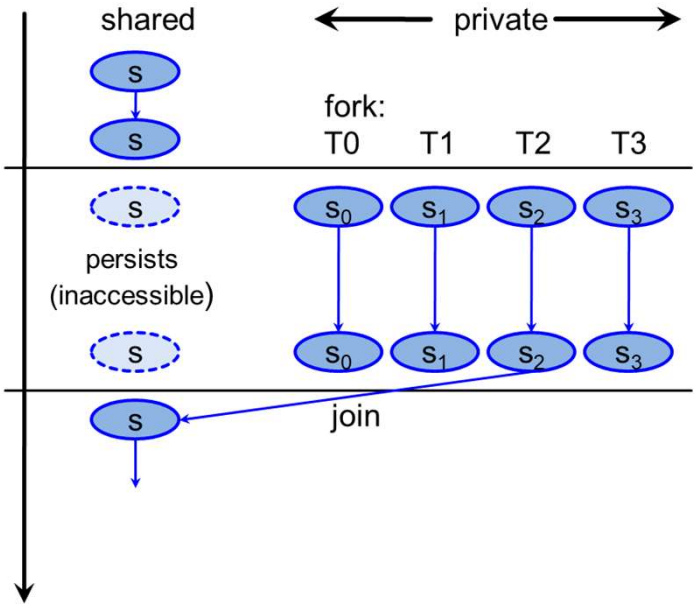
- **Extension of private:**
  - additional semantics for work sharing
  - value from thread which executes last iteration of loop is transferred back to master copy (which must be allocated if it is a dynamic entity)
  - restrictions similar to `firstprivate`

The lastprivate Clause: C/C++

```
float s;

  s = …;
#pragma omp parallel
  {
#pragma omp for lastprivate(s)
    for(int i=0; i<ndim; i++) {
      s = …;
    }
  }
 … = … + s;
```

on work sharing directive



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

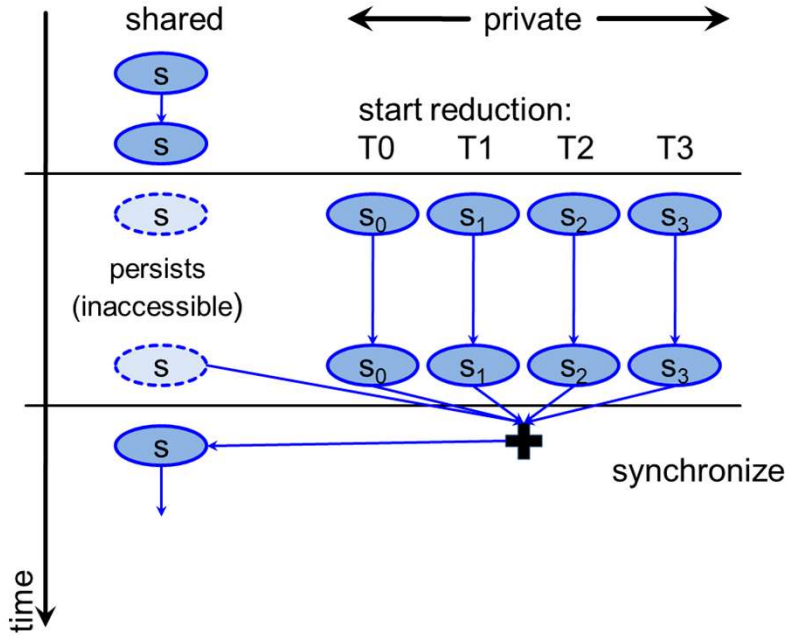
- **Extension of private:**
  - additional semantics for work sharing
  - value from thread which executes last iteration of loop is transferred back to master copy (which must be allocated if it is a dynamic entity)
  - restrictions similar to `firstprivate`

Reduction Operations (1): Fortran

```
real :: s

!$omp parallel
!$omp do reduction(+:s)
  do i = …
    …
      s = s + …
  end do
!$omp end do
  … = … * s
!$omp end parallel
```

s is still shared here



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

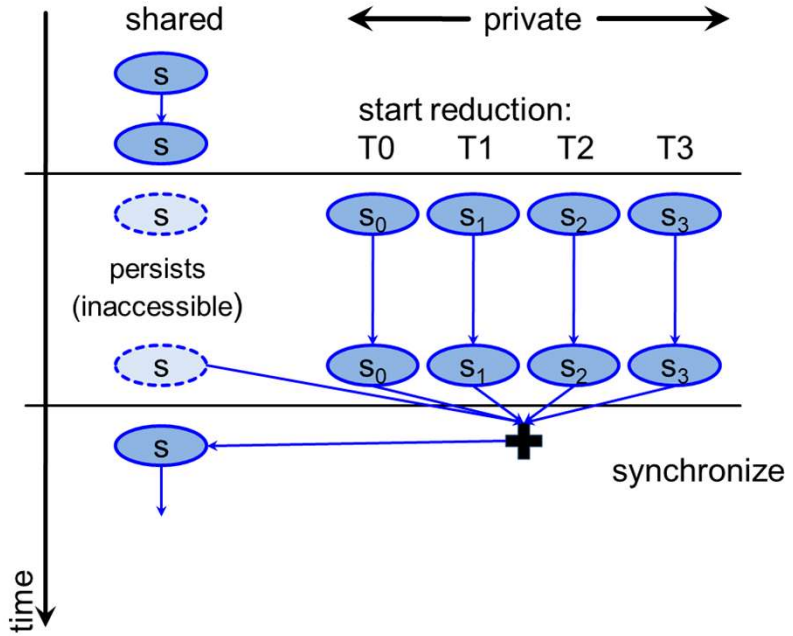
- **At synchronization point:**
  - reduction operation is performed
  - result is transferred to master copy
  - restrictions similar to **`firstprivate`**

Reduction Operations (1): C/C++

```
 float s;

#pragma omp parallel
  {
#pragma omp for reduction(+:s)
    for(int i=0;i<ndim;i++) {
      ...;
      s = s + …;
    }
     … = … * s;
  }
```

s is still shared here



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): Fortran

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

Operation	Initial Value
+	0
-	0
*	1
.and.	.true.
.or.	.false.
.eqv.	.true.
.neqv.	.false.
MAX	min(type)
MIN	max(type)
IAND	all bits set
IEOR	0
IOR	0

- **Consistency required**
  - operation specified in clause vs. update statement
- **Multiple reductions:**
  - multiple scalars, or an array:

```
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)
```

Reduction Operations (2): C/C++

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

Operation	Initial Value
+	0
-	0
*	1
&	~ 0
|	0
^	0
&&	1
||	0
max	min(type)
min	max(type)

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

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

```
float x, y, z;
#pragma omp for reduction(+:x, y, z)
```

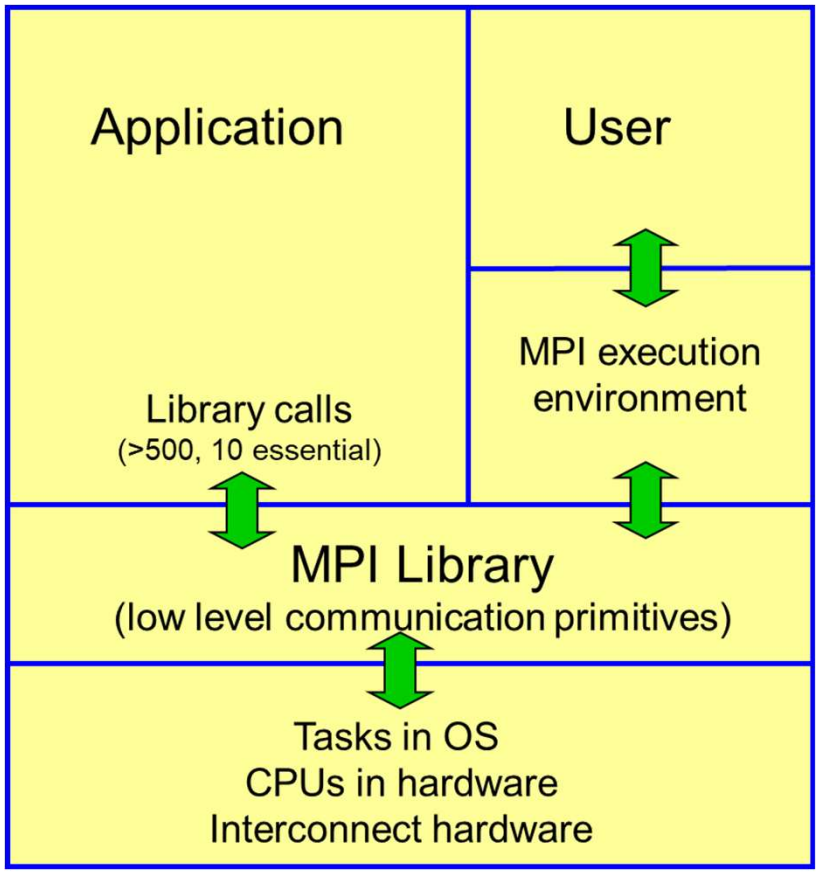

```
float a[n];
#pragma omp for reduction(*:a[0:n])
```

```
#pragma omp for reduction (+:a[0:n]) \
    reduction (*:b[0:n],c[0:n])
```

MPI

Principles of Message Passing on Distributed Memory Architectures

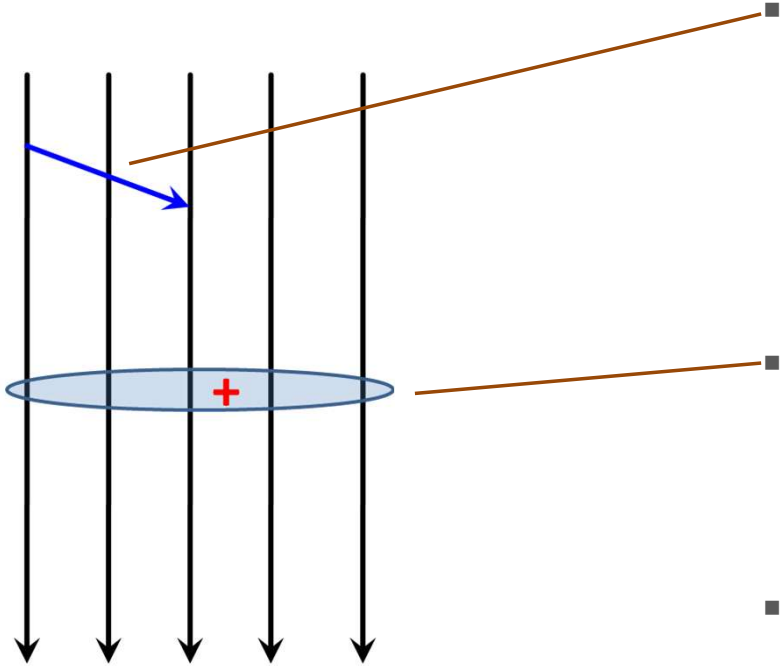
MPI Architecture



- **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

MPI Error Handling

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

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**
  - Fortran: `call MPI_INIT(ierror)`
  - C: `MPI_Init(&argc, &argv);`
- **Last call: clean shutdown of parallel machine**
  - Fortran: `call MPI_FINALIZE(ierror)`
  - C: `MPI_Finalize();`
- 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

Initialization and Finalization (2)

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

  C binding:

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

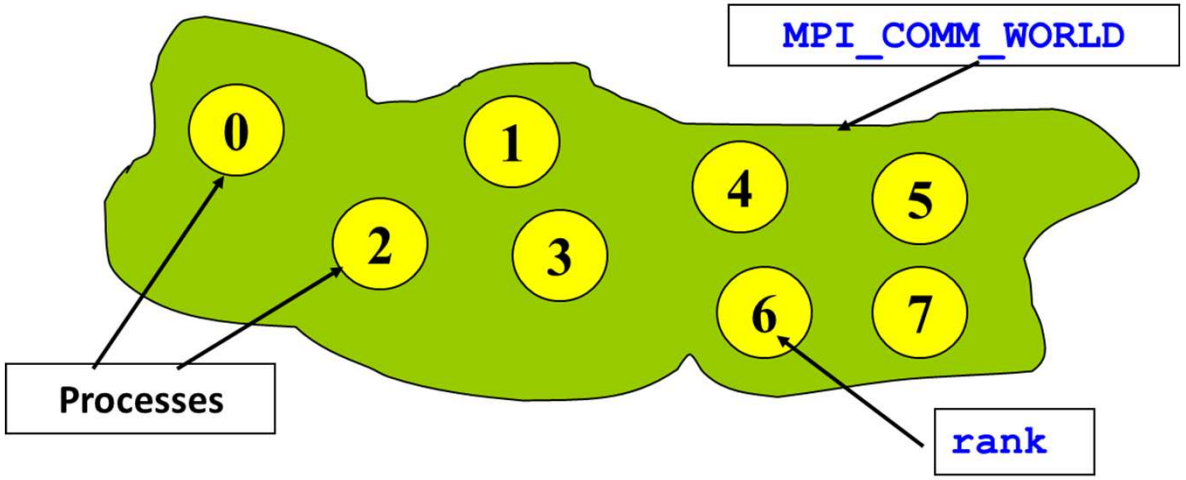
- **If `MPI_Init()` is called in a function** (bad idea anyway), this function must have pointers to the original data:

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

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

  - obtain rank in C:

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

  - **rank** = 0, 1, 2, … , (number of MPI tasks – 1)
- Obtain number of MPI tasks in communicator:
  - in Fortran:

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

  - in C:

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

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:

Fortran

```
if (rank == 0) then
  ... ! do work for rank 0
else
  ... ! do work for other ranks
end if
```

C

```
if (rank == 0){
  ... // do work for rank 0
}
else {
  ... // do work for other ranks ***
}
```

A Very Simple MPI Program: Fortran

```
program hello
 use mpi
 implicit none

 integer :: rank, size, ierror

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

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

 call MPI_FINALIZE(ierror)
end program
```

A Very Simple MPI Program: C/C++

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

int main(int 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 %i of %i\n", rank, size);

  MPI_Finalize();
}
```

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 (**`mpiifx`**, **`mpiicx`** etc.)
- **Startup facilities**
  - **`mpirun`** (legacy)
  - **`mpiexec`**
  - site and implementation dependent

- **Compile**:
  - Fortran: **`mpiifx -o hello hello.f90`**
  - C: **`mpiicx -o hello hello.c`**
- **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

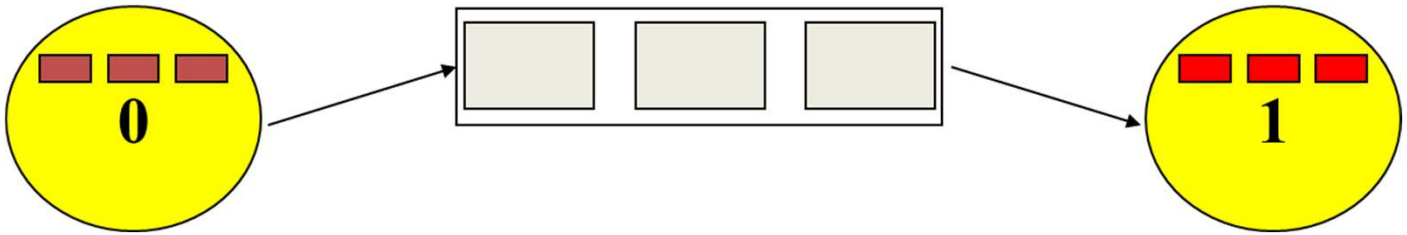
MPI Process Communication

- **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

Most important basic data types:

Fortran

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

C

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

Basic Fortran and C Data Types in MPI 4.1

Table 3.1: Predefined MPI datatypes corresponding to Fortran datatypes

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

Table 3.2: Predefined MPI datatypes corresponding to C datatypes

MPI datatype	C datatype
MPI_CHAR	char (treated as printable character)
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	signed long long int
MPI_LONG_LONG (as a synonym)	signed long long int
MPI_SIGNED_CHAR	signed char (treated as integral value)
MPI_UNSIGNED_CHAR	unsigned char (treated as integral value)
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t (defined in <stddef.h>) (treated as printable character)
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX (as a synonym)	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	
MPI_PACKED	

Table 3.3: Predefined MPI datatypes corresponding to both C and Fortran datatypes

MPI datatype	C datatype	Fortran datatype
MPI_AINT	MPI_Aint	INTEGER(KIND=MPI_ADDRESS_KIND)
MPI_OFFSET	MPI_Offset	INTEGER(KIND=MPI_OFFSET_KIND)
MPI_COUNT	MPI_Count	INTEGER(KIND=MPI_COUNT_KIND)

Table 3.4: Predefined MPI datatypes corresponding to C++ datatypes

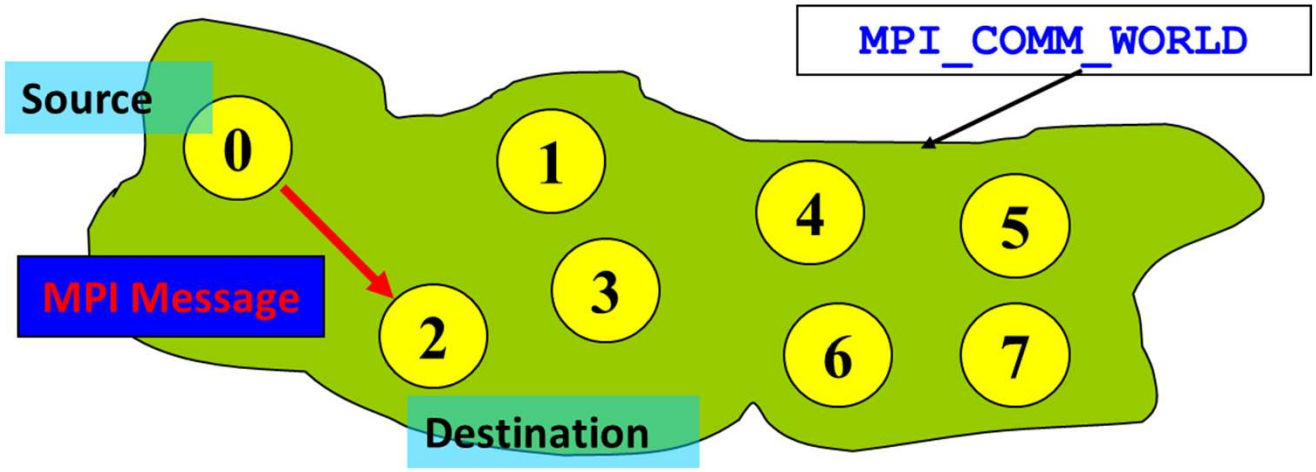
MPI datatype	C++ datatype
MPI_CXX_BOOL	bool
MPI_CXX_FLOAT_COMPLEX	std::complex<float>
MPI_CXX_DOUBLE_COMPLEX	std::complex<double>
MPI_CXX_LONG_DOUBLE_COMPLEX	std::complex<long double>

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
- **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: `call MPI_SEND (buf, count, datatype, dest, tag, comm, ierror)`
- `C: MPI_Send (&buf, count, datatype, dest, tag, comm)`

  - `buf / &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 (Fortran-only)

- 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

MPI_Send Example

- Example:
  send array of 10 integers to task no. 5

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

```
int count, dest, tag;
int *field;
…
 count=10; dest=5; tag=0;
 field = (int*)malloc(count*sizeof(int));
 MPI_Send(field, count, MPI_INT,
          dest, tag, MPI_COMM_WORLD);
```

- **Source and destination may coincide**
  - beware potential deadlocks!

Blocking Standard Receive: MPI_Recv

- **`MPI_Recv`**: 1. receive data
  2. complete
- Fortran: `call MPI_RECV (buf, count, datatype, source, tag, comm, status, ierror)`
- C: `MPI_Recv(&buf, count, datatype, source, tag, comm, &status)`

  - **`buf`** size of buffer **must** be ≥ size of message
  - **`count`** **maximum** number of elements to receive
  - **`source, tag`** **wildcards** may be used (**`MPI_ANY_SOURCE`**, **`MPI_ANY_TAG`**)
  - **`status`** information from the message that was received
    (**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**

```
integer :: status(MPI_STATUS_SIZE)
```

  - Array of integers of size MPI_STATUS_SIZE
  - index values for query: MPI_SOURCE, MPI_TAG, MPI_ERROR

- **MPI_status in C/C++**

```
MPI_Status status;
```

  - Structure of type MPI_Status
  - hand a reference to MPI_Recv
  - component names for query: status.MPI_SOURCE, status.MPI_TAG, status.MPI_ERROR

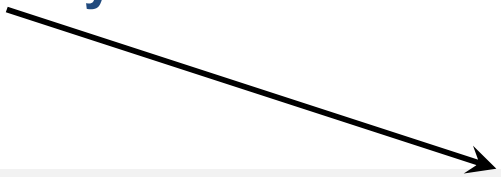
- Inquiring message length needs an additional MPI call:
  - Fortran: `call MPI_GET_COUNT(status, datatype, count, ierror)`
  - C: `MPI_Get_count(&status, datatype, &count);`
    - count is output argument
    - datatype must be the same datatype used in the MPI call that produced the status variable

MPI_Recv Example: Fortran

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

```
integer count, countrecv, 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, MPI_REAL, countrecv, ierror)`

MPI_Recv Example: C/C++

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

```
int count, countrecv;
MPI_Status status;
field = (float *)malloc(count*sizeof(float));
…
MPI_Recv(field, count, MPI_FLOAT, MPI_ANY_SOURCE,
   MPI_ANY_TAG, MPI_COMM_WORLD, &status);
printf("Received from %i with tag %i count: %i \n",
 status.MPI_SOURCE, status.MPI_TAG)
```

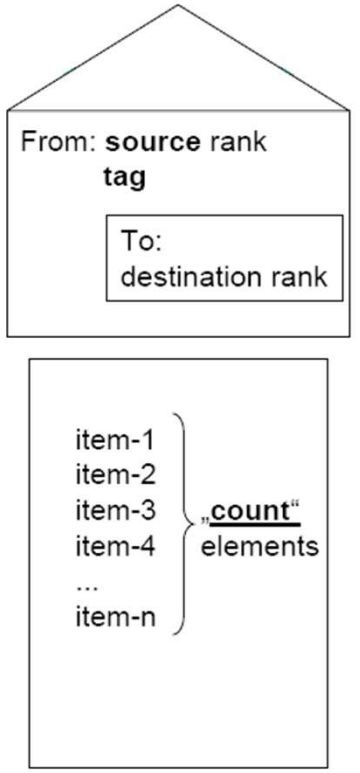
-

```
MPI_Get_count(&status, MPI_FLOAT, &countrecv);
```

Requirements for Point-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.
- 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

First Complete MPI Example in Fortran

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

```
program collect
  use mpi
  implicit none
  integer :: i,size,rank,ierror, &
             status(MPI_STATUS_SIZE)
  integer :: number,sum
  call MPI_INIT(ierror)
  call MPI_COMM_RANK (MPI_COMM_WORLD,&
             rank,ierror)
 if(rank.eq.0) then
    sum=0
    call MPI_COMM_SIZE(MPI_COMM_WORLD,&
                       size,ierror)
   do i=1,size-1
     call MPI_RECV(number,1, &
      MPI_INTEGER, MPI_ANY_SOURCE, &
      MPI_ANY_TAG, MPI_COMM_WORLD, &
       status, ierror)
      sum=sum+number
   enddo
  write(*,*) 'The sum is ',sum
 else
  call MPI_SEND(rank,1,MPI_INTEGER, &
        0, 0, MPI_COMM_WORLD, ierror)
 endif
 call MPI_FINALIZE(ierror)
end program
```

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

```
#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_SOU
    RCE ,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();
}
```

First Complete MPI Example

Remarks:

- **gathering results** from processes is a very common task in MPI – there are more efficient ways to do this (see advanced talk).
- this is a **reduction operation** (summation). There are more efficient ways to do this (see advanced talk).
- 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