More MPI

Hinnerk Stüben

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Universität Bremen
13–18 September 2025

Contents

Placement of data in memory

Alignment

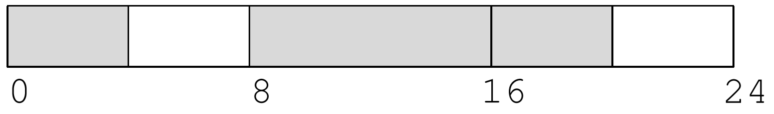
alignment at word boundaries:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    struct {float f; double d; int i;} s;

    printf("offset f = %d\n", (int) ((void*) &(s.f) - (void*) &s));
    printf("offset d = %d\n", (int) ((void*) &(s.d) - (void*) &s));
    printf("offset i = %d\n", (int) ((void*) &(s.i) - (void*) &s));
    printf("size   s = %d\n", (int) sizeof(s));
}
```

output:

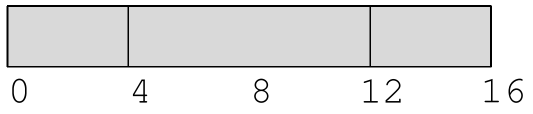
Sun:

```
offset f = 0
offset d = 8
offset i = 16
size   s = 24
```



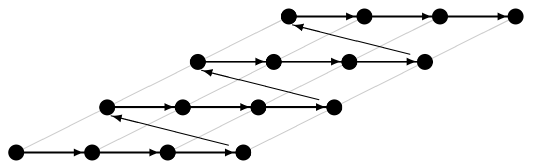
Linux-PC:

```
offset f = 0
offset d = 4
offset i = 12
size   s = 16
```

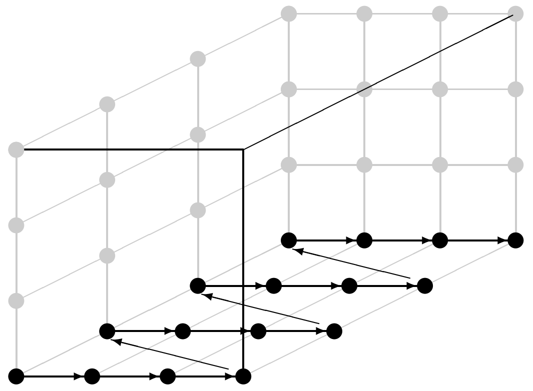


Multi-dimensional arrays

Fortran

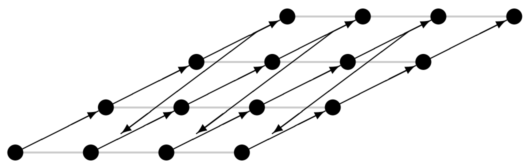


`real(8) :: v(Nx, Ny)`

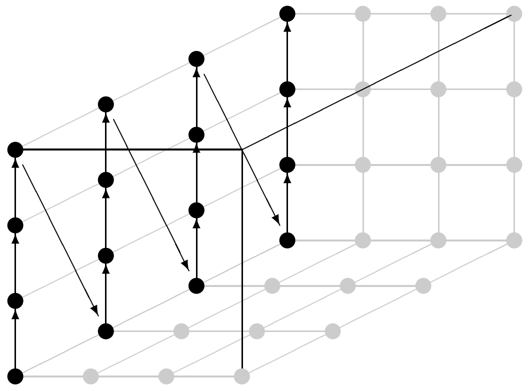


`real(8) :: v(Nx, Ny, Nz)`

C



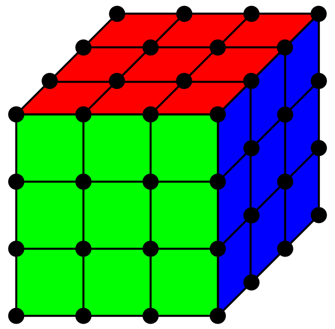
`double v[Nx][Ny];`

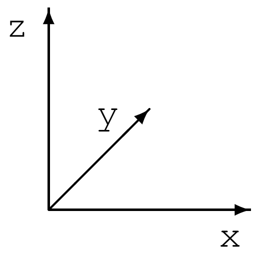


`double v[Nx][Ny][Nz];`

Storage of surface values

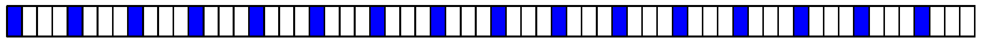
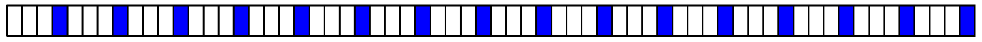
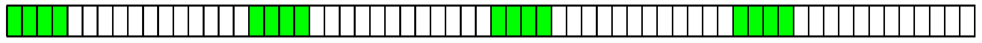
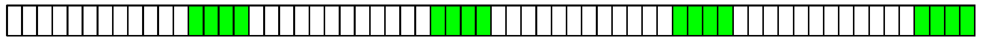
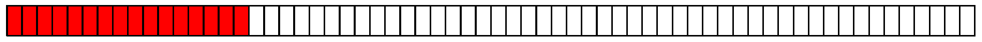
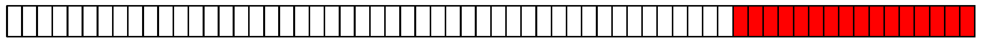
example: 4^3 cube



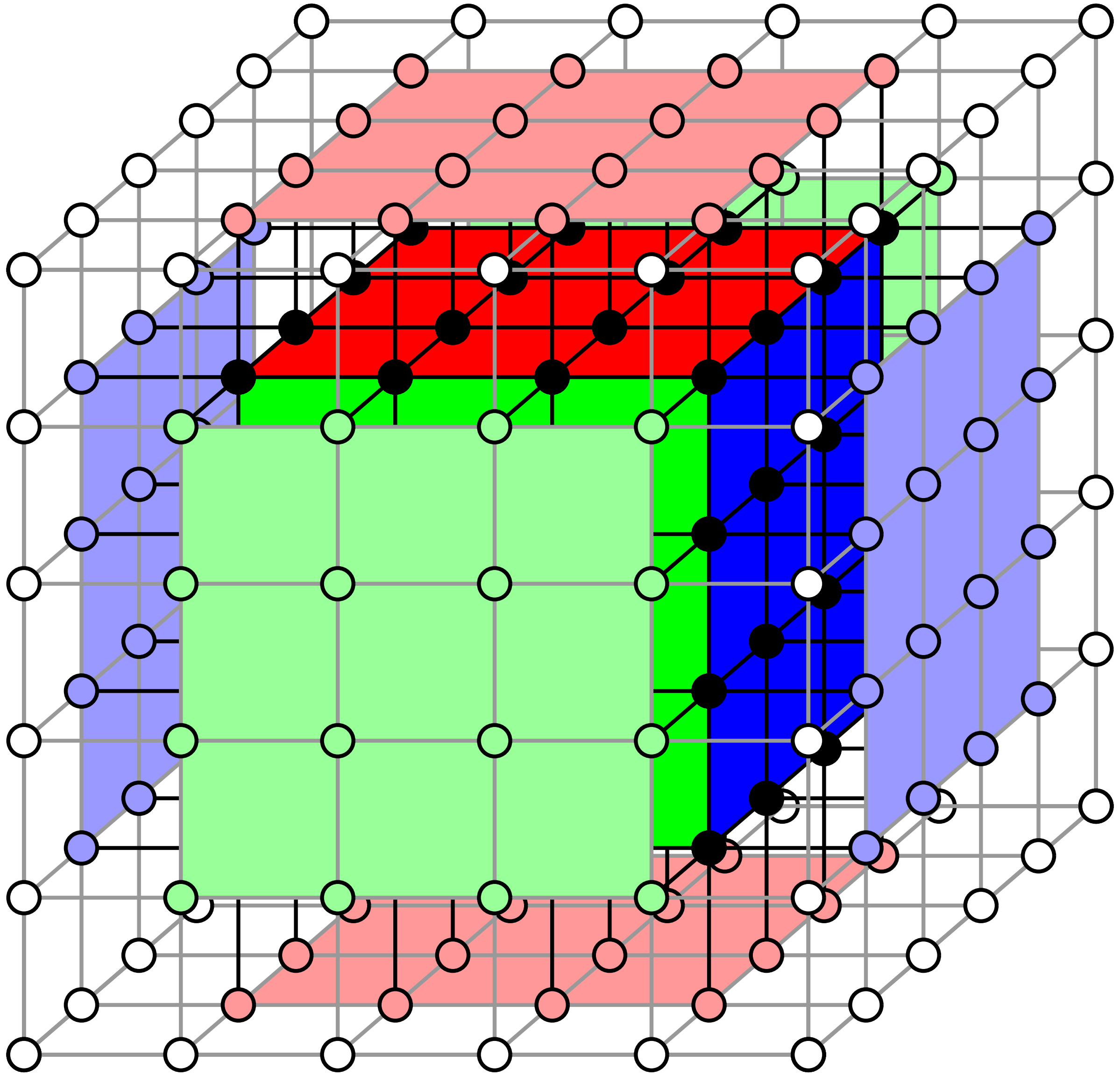


```
x  =  0  ..  Nx−1
y  =  0  ..  Ny−1
z  =  0  ..  Nz−1
```

where surfaces are stored in memory

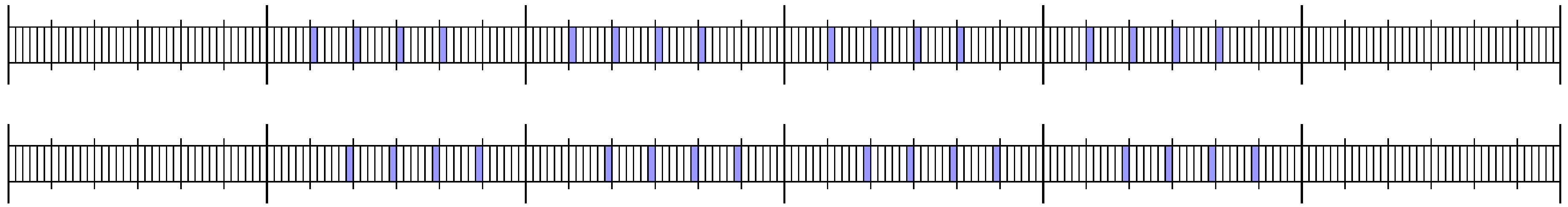
surface		*count*	*block*	*stride*	*offset*
`x ≡ xMin`		`Ny*Nz`	`1`	`Nx`	`0`
`x ≡ xMax`		`Ny*Nz`	`1`	`Nx`	`Nx−1`
`y ≡ yMin`		`Nz`	`Nx`	`Nx*Ny`	`0`
`y ≡ yMax`		`Nz`	`Nx`	`Nx*Ny`	`Nx*(Ny−1)`
`z ≡ zMin`		`1`	`Nx*Ny`	`Nx*Ny*Nz`	`0`
`z ≡ zMax`		`1`	`Nx*Ny`	`Nx*Ny*Nz`	`Nx*Ny*(Nz−1)`

Example – regular mesh with boundary



Storage of surface values

- points on the light blue surface are stored with *two* strides



- implementation → `./demos/mpi/surface.f90`

One-sided communication

One-sided communication

- other names
  - remote memory access (RMA)
  - remote direct memory access (RDMA)
- underlying computer architecture
  - distributed memory with global address space
- typical communication calls
  - *get* and *put*

    in contrast to *send* and *recv* in MPI-1

Motivation

- completeness
  - a programming model natural for NUMA machines

    OpenMP is natural for shared memory

    MPI-1 is natural for distributed memory
- performance
  - if there is hardware support for *remote direct memory access (RDMA)*, *put* is the fastest communication call
- expressivity
  - example: global gather operation: `A = B(`*map*`)` see slide 12

Programming models implementing one-sided communication

- SHMEM

  originally: *shared memory* library

  now: Symmetric Hierarchical MEMory

  - Cray SHMEM (1993)
  - OpenSHMEM (2012) → `www.openshmem.org`

- MPI-2 (1997) → `mpi-forum.org`

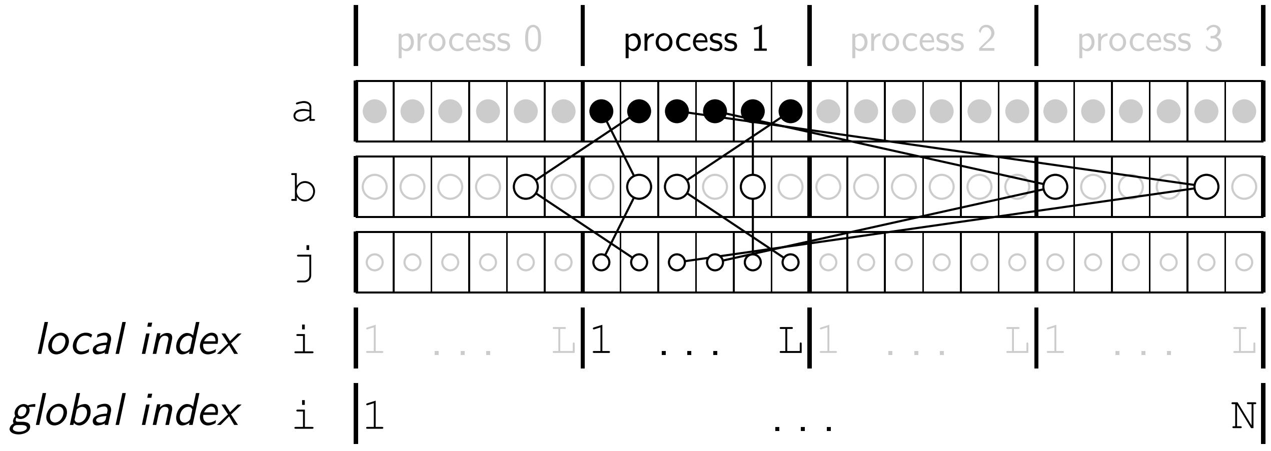
- Global Address Programming Interface (GPI) (Fraunhofer Institute, 2005)

  → `www.gpi-site.com/gpi2`

Discussion: global gather operation

- How can the *gather loop* be parallelised with MPI-1?

```
for i := 1 to N do
   a[i] := b[j[i]]
```



- Data movements are indicated for one process only.
- The other processes will perform the same kind of operations at the same time.

More use cases

- global counters
  - *master-worker* can be implemented without *master* process

    → `./demos/mpi/master-worker-shmem.f90`

    → `./demos/mpi/master-worker-shmem.c`

    → `./demos/mpi/master-worker-mpi2.f90`

    → `./demos/mpi/master-worker-mpi3.f90`

- global arrays
  - *Global Arrays (GA) toolkit* → `http://www.emsl.pnl.gov/docs/global`

Example of a global array: `output_parallel_shmem.f90`

```fortran
subroutine output_parallel(v, Lx, Ly)

  use module_decomp
  implicit none

  integer, intent(in) :: Lx, Ly
  real(8), intent(in) :: v(0:Lx + 1, 0:Ly + 1)
  real(8)             :: vv(0:decomp%Nx + 1, 0:decomp%Ny + 1)
  integer             :: x, y, x_local, y_local, home_of_xy

  call shmem_barrier_all()

  if (decomp%my_rank == 0) then

     do y = 0, decomp%Ny + 1
        do x = 0, decomp%Nx + 1
           call global2local(x, y, x_local, y_local, home_of_xy)

           call shmem_get64(vv(x, y), v(x_local, y_local), 1, home_of_xy)
        enddo
     enddo

     call output(vv, decomp%Nx, decomp%Ny)
  endif
end
```

Example of a global array: `output_parallel_shmem.c`

```
# include <stdio.h>
# include <mpp/shmem.h>
# include "laplace.h"
# include "decomp.h"

void output_parallel(field v, int Lx, int Ly)
{
  field      vv = field_alloc(decomp.Ny, decomp.Nx);
  int        x, y, x_local, y_local, home_of_xy;

  shmem_barrier_all();

  if (decomp.my_rank == 0) {

      for (y = 0; y <= decomp.Ny + 1; y++) {
          for (x = 0; x <= decomp.Nx + 1; x++) {
              global2local(x, y, &x_local, &y_local, &home_of_xy);

              shmem_get64(&vv[y][x], &v[y_local][x_local], 1, home_of_xy);
          }
      }
      output(vv, decomp.Nx, decomp.Ny);
  }
  field_free(vv);
}
```

Cray SHMEM

Cray SHMEM – overview

- terms

SHMEM	MPI
PE (processing element)	process
`num_pes`	`size`
`my_pe`	`rank`

- concepts
  - symmetric heap
  - explicit synchronisation

Cray SHMEM – one-sided data access

- routines*
  - `shmem_get` – remote read
  - `shmem_put` – remote write

- properties
  - `shmem_get` and `shmem_put` are asynchronous operations
  - `shmem_put` is non-blocking

- synchronisation !
  - `barrier` – global barrier

* On 1990s Crays two routines were enough, because integers and floating point numbers were both 64-bit.

Cray SHMEM – basic Fortran routines

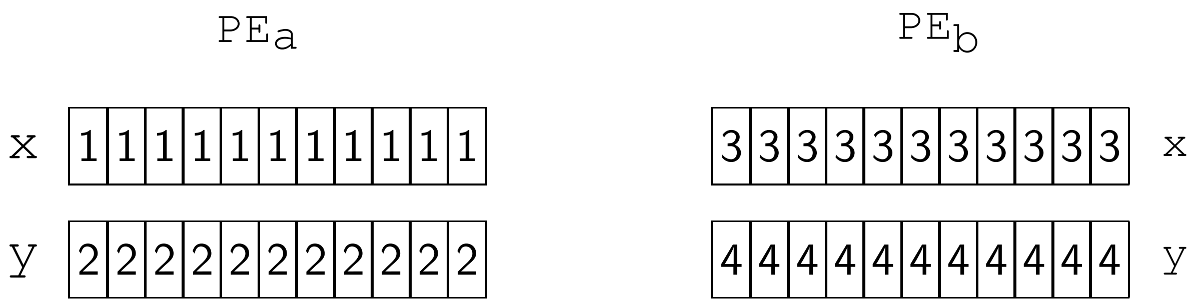
- `n = num_pes()`
- `i = my_pe()`
- `call shmem_get(target, source, n_words, pe)`
- `call shmem_put(target, source, n_words, pe)`
- `call barrier()`

Cray SHMEM – basic C/C++ routines

- `#include <mpp/shmem.h>`
- `int _num_pes (void);`
- `int _my_pe (void);`
- `void shmem_get(void *target, const void *source, size_t len, int pe);`
- `void shmem_put(void *target, const void *source, size_t len, int pe);`
- `void barrier(void);`

Cray SHMEM – `shmem_get` and `shmem_put` functionality

- initial situation:

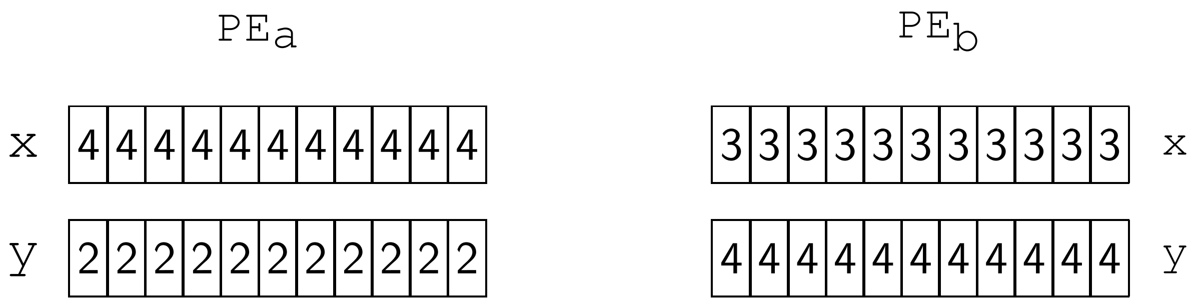


- SHMEM call:

PEa calls **or** PEb calls

`shmem_get(x, y, N, PEb)` `shmem_put(x, y, N, PEa)`

- result:



Symmetric data objects (I)

Local address calculation example:

- address of an element of a 2-dimensional array in C

```
double a[N][M];
int i, j;
intptr_t address1, address2;
intptr_t base_address, offset;

address1 = (intptr_t) &a[i][j];

base_address = (intptr_t) a;
offset = (i * M + j) * sizeof(double);

address2 = base_address + offset;
```

→ `address1` and `address2` are identical

Symmetric data objects (II)

Remote address calculations:

- with *symmetric* data objects all remote address calculations can be performed locally

- symmetric data objects have on all PEs the same
  - base address
  - structure (types, sub-types, dimensions)

- objects have the same base address on all PEs if they are
  - in *static* memory
    Fortran: `SAVE` or `COMMON`
    C: `static`
  - allocated from the *symmetric* heap, the following calls must be strictly SPMD!
    Fortran: `allocate` → `shpalloc`
    C: `malloc` → `shmalloc`

Allocating symmetric objects in Fortran

- `shpalloc` allocates 32-bit words via *Cray pointers*

```
real(8) :: x(n)
pointer (p_x, x)

call shpalloc(p_x, 2 * n, ierr, 1)

x(1) = ...
```

Synchronisation (I)

- synchronisation for `shmem_get`

```
call barrier()          wait for completion of remote write operations
call shmem_get(...)
call barrier()          wait for completion of local read operations
                        by remote shmem_gets
```

- safe programming
  - the first barrier is necessary

    remote data must be available
  - the second barrier is sufficient

    local data must not be overwritten while being read by other processes

Synchronisation (II)

- synchronisation for `shmem_put`

```
call barrier()          do not overwrite other processes' data
call shmem_put(...)
call barrier()          wait for data from other processes
```

- safe programming
  - the second barrier is necessary

    data from remote processes must be available
  - the first barrier is sufficient

    remote data must not be overwritten while being read by other processes

OpenSHMEM

OpenSHMEM functions

- OpenSHMEM has introduced more functions, incomplete list:

Cray SHMEM	OpenSHMEM
–	`shmem_init`
`num_pes`	`shmem_n_pes`
`my_pe`	`shmem_my_pe`
`shmem_get`	`shmem_get32`
	`shmem_get64`
	`shmem_get128`
	`shmem_`*`type`*`_get`
`shmem_put`	*correspondingly*
`barrier`	`shmem_barrier_all`
	`shmem_barrier`

- *`type`*s are names for basic data types

  Fortran: `integer`, `real`, `double`, . . .
  C: `int`, `float`, `double`, . . .

OpenSHMEM implementations

- Cray MPI

  load SHMEM environment: `module load cray-shmem`
  compile as usual: `cc`, `ftn` (works with Cray, GNU and Intel compilers)
  run as usual: `aprun`

- Open-MPI

  compile with: `oshcc`, `oshfort`
  run with: `mpirun`

Global sum – Fortran

- see `laplace-shmem-f90/global_sum.f90` and `laplace-shmem-c/global_sum.c`

```
real(8) function global_sum(local_sum)

    implicit none
    include 'mpp/shmem.fh'
    real(8), intent(in) :: local_sum
    integer, external :: shmem_n_pes  ! not in mpp/shmem.fh of Open-MPI
    integer, save :: psync(SHMEM_REDUCE_SYNC_SIZE)
    data             psync / SHMEM_REDUCE_SYNC_SIZE * SHMEM_SYNC_VALUE /
    real(8), save :: pwrk(SHMEM_REDUCE_MIN_WRKDATA_SIZE)
    real(8), save :: source, dest

    source = local_sum
    call shmem_real8_sum_to_all(dest, source, 1, 0, 0, &
                                shmem_n_pes(), pwrk, psync)
    global_sum = dest
end
```

Global sum – C

```
# include <mpp/shmem.h>

double global_sum(double local_sum)
{
    static long pSync[_SHMEM_REDUCE_SYNC_SIZE];
    static double pWrk [_SHMEM_REDUCE_MIN_WRKDATA_SIZE];
    static int not_initialised = 1;
    static double source, dest;
    int i;

    if (not_initialised) {
        for (i = 0; i < _SHMEM_REDUCE_SYNC_SIZE; i++)
            pSync[i] = _SHMEM_SYNC_VALUE;
        shmem_barrier_all();
        not_initialised = 0;
    }
    source = local_sum;
    shmem_double_sum_to_all(&dest, &source, 1, 0, 0,
                            shmem_n_pes(), pWrk, pSync);
    return dest;
}
```

SHMEM vs. MPI-1

- MPI ist more general
  - communicators
  - datatypes
- in SHMEM *point-to-point* calls are simpler
- in MPI *collective* calls are simpler

Realistic usage scenario for SHMEM

SHMEM can be used

- in combination with MPI

```
shmem_n_pes() ≡ size(MPI_COMM_WORLD)
shmem_my_pe() ≡ rank(MPI_COMM_WORLD)
```

- to implement performance or implementation critical parts

Example for SHMEM plus MPI (works with Cray MPI and Open-MPI)

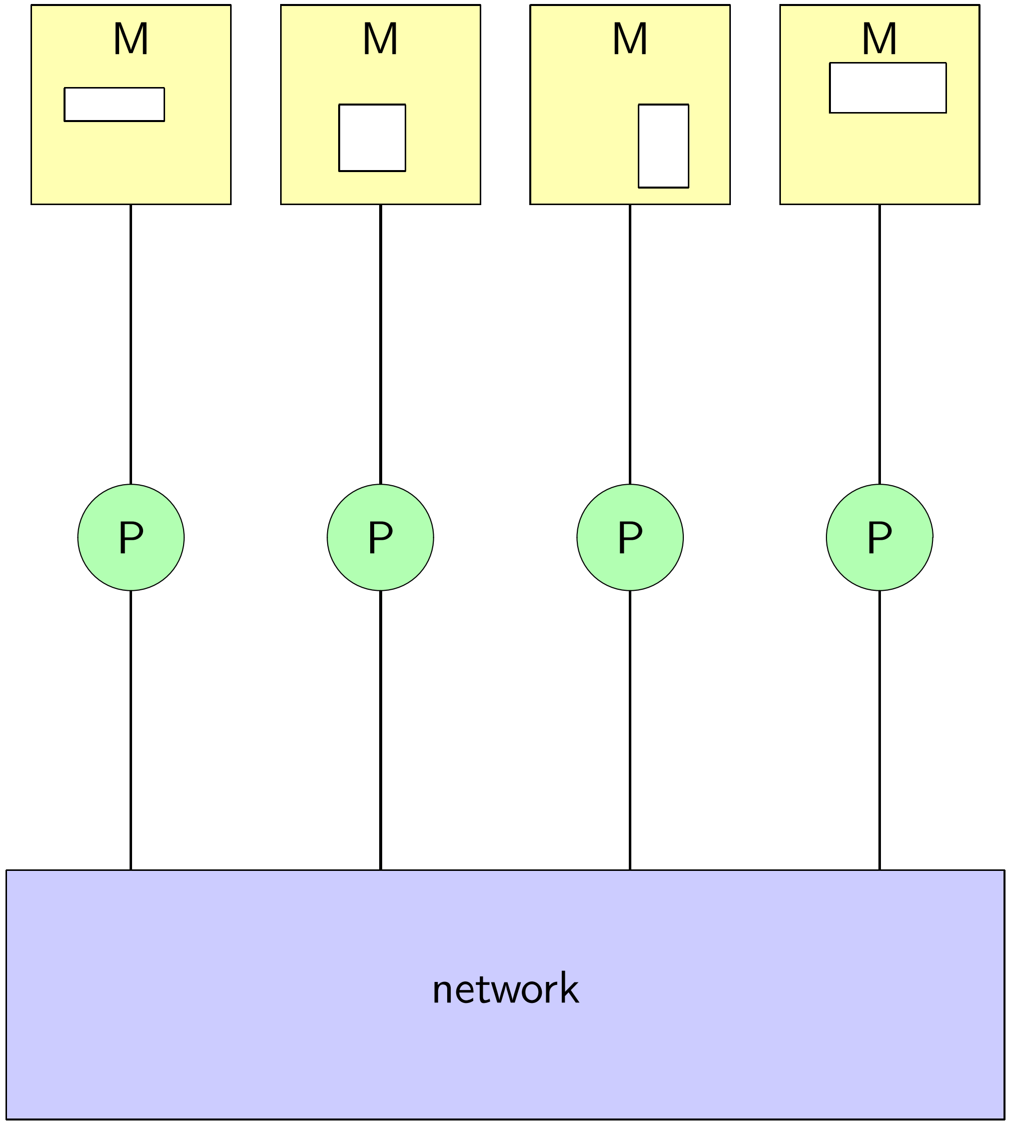
```
./demos/mpi/shmempi.f90
./demos/mpi/shmempi.c
```

One-sided communication with MPI-2

Memory windows (I)

- memory window
  - consecutive region in local memory made accessible to other processes
  - several processes can perform read/write operations *simultaneously*
  - creation (*collective* call)
    ```
    int MPI_Win_create(void* base, MPI_Aint size, int disp_unit,
                       MPI_Info info, MPI_Comm comm, MPI_Win *win);
    ```
  - destruction (*collective* call)
    ```
    int MPI_Win_free(MPI_Win *win);
    ```

Memory windows (II)



Memory windows (III)

- parameters

`base`	base address
`size`	size in bytes
`disp_unit`	*displacement unit* i.e. unit of address calculations in bytes examples: `1`, `sizeof(double)`
`info`	hint on optimisation examples: `MPI_INFO_NULL`, `no_locks`
`comm`	communicator
`win`	returned value (handle)

- Within communicator `comm` each process provides a window with *individual* size (which can be zero).

Memory windows (IV)

- in order to allow for performance improvements (implementation dependent) a special memory allocation routine is provided:

```
MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void* base);

MPI_Free_mem(void* base);
```

$\rightarrow$ `base` is input to `MPI_Win_create`

MPI_Put()

```
int MPI_Put(void*         local_addr,        // "send buffer"
            int           local_count,
            MPI_Datatype  local_datatype,
            int           rank,
            MPI_Aint      remote_disp,
            int           remote_count,
            MPI_Datatype  remote_datatype,   // locally defined
            MPI_Win       win
           );

// "receive buffer":
//      remote_addr = remote_window_base + remote_disp * disp_unit
```

MPI_Get()

```
int MPI_Get(void*        local_addr,       // "receive buffer"
            int          local_count,
            MPI_Datatype local_datatype,
            int          rank,
            MPI_Aint     remote_disp,
            int          remote_count,
            MPI_Datatype remote_datatype,  // locally defined
            MPI_Win      win
           );

// "send buffer":
//     remote_addr = remote_window_base + remote_disp * disp_unit
```

MPI_Accumulate()

```
int MPI_Accumulate(
            void*        local_addr,       // "send buffer"
            int          local_count,
            MPI_Datatype local_datatype,
            int          rank,
            MPI_Aint     remote_disp,
            int          remote_count,
            MPI_Datatype remote_datatype,  // locally defined
            MPI_Op       op,               // like in MPI_Reduce()
            MPI_Win      win
           );

// update at:
//     remote_addr = remote_window_base + remote_disp * disp_unit
// additional operation: MPI_REPLACE
```

Synchronisation (I)

- `Put`, `Get` and `Accumulate` must occur in the same *access epoch*.
- An epoch starts and ends with a synchronisation call. If it returns all data movement is completed.
- A typical (collective) synchronisation routine is `MPI_Win_fence()`.
  It corresponds to the barrier in Cray *shmem*.

  ```
  int MPI_Win_fence(int assert, MPI_Win win);
  ```

  `assert` ist an optimisation parameter which can be zero.
- remark
  - `shmem_get` is blocking (data can be used when the routine returns)
  - `MPI_Get` is non-blocking (data can only be used after synchronisation!)

Synchronisation (II)

- additional synchronisation routines
  - `MPI_Win_start()`
  - `MPI_Win_complete()`
  - `MPI_Win_post()`
  - `MPI_Win_wait()`
  - `MPI_Win_test()`

Synchronisation – locks

- routines for *locking* memory regions
  locking = protection against modification by other processes
  - `MPI_Win_lock()`
  - `MPI_Win_unlock()`

→ OpenMP

- other advanced topics:
  - data cache coherency
  - thread safety

Exercise 5

Exercise 5 – one-sided communication with Get and Put

- Switching between `MPI_Get` and `MPI_Put`.

→ `./exercises/mpi/exercise5`

Typical bugs in parallel programs

Typical bugs in parallel programs

- deadlock
- livelock
- race condition
- incorrect memory access

Deadlock

- standstill, processes are in a waiting state
  - a group of processes is waiting for an event that can only be initiated from within this group

- examples
  - two processes trying to send synchronously to each other
  - synchronous send in a closed chain
  - unequal number of barriers in a process group
  - two processes trying to use buffered send if both system buffers are full and hold only data for the corresponding other process

Livelock

- no progress is made although processes are in a busy state
- similar to an infinite loop in a sequential program, but more than one process is involved

Race condition (I)

- A situation in which two or more processes are processing the same data (at least one process is modifying that data) and data access is random in time.
- An indication for a race condition is that results differ from run to run for identical input.
- It can also happen that the program sometimes completes successfully and sometimes crashes — again for (more or less) identical input.

Race condition (II)

- The bug might lie dormant until there is a change in the environment leading to a change of the run-time behaviour of the program, like
  - change to another computer,
  - change of hardware of the usual computer,
    e.g. a broken processor or communication link resulting in different routing of messages,
  - change of the software environment,
    compiler, compiler version, run-time library,
  - using extremely different input than usual,
    e.g. for a very small problem.
- The erroneous behaviour often disappears after compiling with debug option or after inserting `printf`s for debugging.

Race conditions – examples

- MPI programs
  - overwriting the send buffer used in `Isend`
  - overwriting the receive buffer used in `Irecv`
  - using the same buffer for sending and receiving in `Sendrecv`
  - race conditions are more likely if `MPI_ANY_SOURCE` or `MPI_ANY_TAG` are used
- OpenMP programs
  - missing `private` declaration
  - undetected data dependences

Incorrect memory access

- MPI
  - receiving from or sending to a wrong process
  - mismatch in sequence of send and receive calls
  - inconsistent tags
- OpenMP
  - memory bound violation resulting in overwriting data from another thread

Parallel I/O

Parallel I/O

- hardware architectures
  - each node has local disks (does scale)
  - all nodes can access a global disk system (convenient)
  - (global) disks can be accessed from dedicated I/O nodes
- in a parallel program I/O can be performed by
  - a single process
  - all processes
  - a group of processes

Parallel I/O – typical scenarios (I)

- reading parameters
  - all processes read the same (small) file
  - one process reads the parameter file and broadcasts the parameters read
- writing a log file
  - a single process writes
- processing dump or scratch files
  - each process reads/writes its data

Parallel I/O – typical scenarios (II)

- writing data for later processing on a workstation
  - a single process gathers and writes all data
  - all process write to the same file in a synchronised way
- debugging / tracing
  - each process writes to its own file

Several processes writing to the same file

- write operations to a sequential file are sequential
  - data from more than one process appears in *random* order
  - records may be split
  - short records are likely to remain complete
  - records are not automatically labelled with the *rank*
- direct access files (explicit positioning with `lseek()`)
  - processes can write to disjoint regions of the file
- in general a *parallel* file system is needed

Several processes reading the same file

- sequential file, e.g. *stdin*
  - it would be random which pieces of data processes get
- direct access file
  - each process can position an individual file pointer and read its portion

Working with several files in different environments

- global file system
  - convenient: every process can access every file
  - limited bandwidth

- local disks
  - files might have to be distributed (after pre-processing) or collected (for post-processing)

Parallel I/O – design goals

- portability / compatibility with work-flow

  → it is a good idea to understand how data is stored in files at byte level
  → data storage a *long term* issue

- independence of the number of processes used

- adaptivity to different architectures (global vs. local file systems)

- total size of data (capacity)

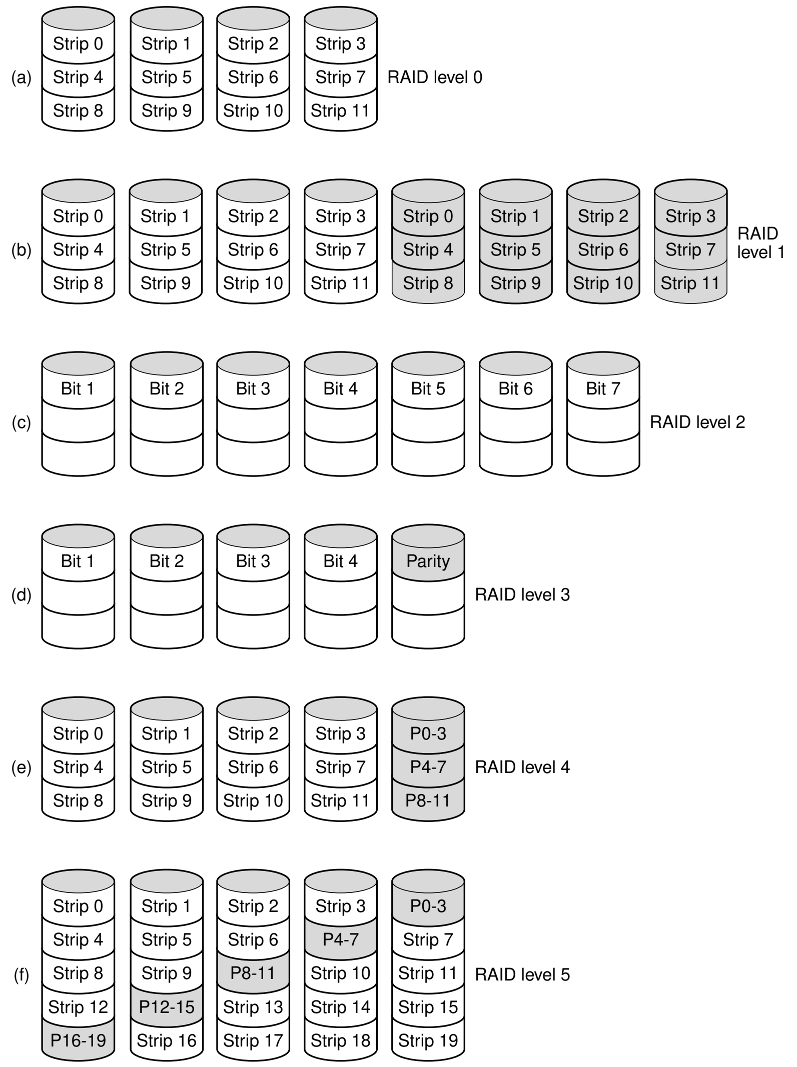
- performance (bandwidth)

Parallel I/O with MPI-2

- access to non-consecutive regions (in memory and file)
- collective I/O operations
- individual and shared file pointers
- non-blocking / asynchronous I/O
- portable and adjustable representations of numbers

I/O Hardware and Lustre

RAID – Redundant Array of Inexpensive Disks



source: A.S. Tanenbaum, *Computerarchitektur*, `http://www.cs.vu.nl/˜ast/books/sco4/sco4-eps.zip`

Lustre

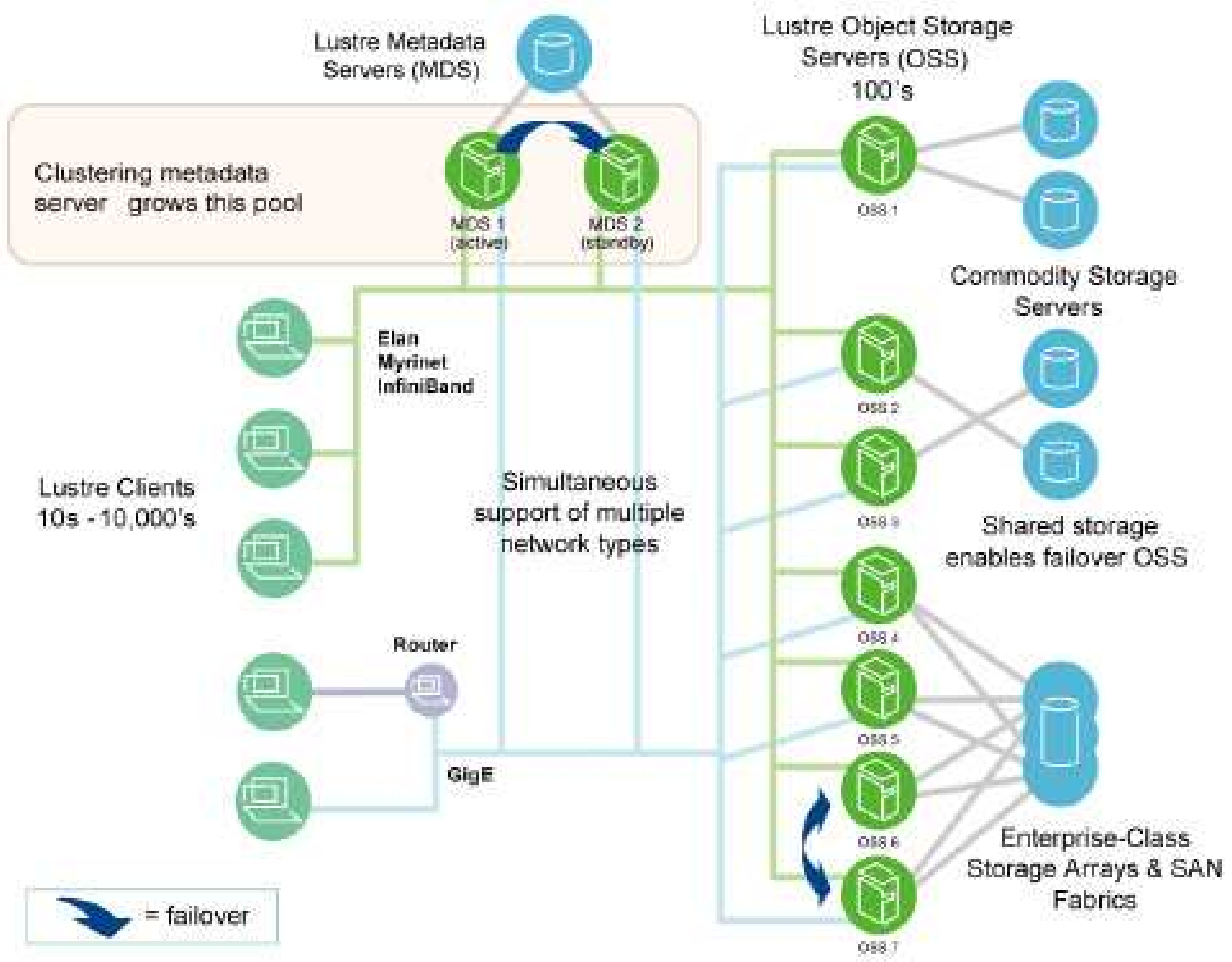
- Lustre is an open source cluster file system
- originally developed by *Cluster File Systems*, continued by *Sun*, *Oracle* and since 2010 by several communities
- quite widespread in supercomputer installations

Lustre – components

- metadata server (MDS)
- object storage server (OSS)
- object storage target (OST)
- client
- network

→ important for efficient use: *striping*

Lustre – architecture



Lustre Metadata Servers (MDS)
Lustre Object Storage Servers (OSS) 100's
Clustering metadata server grows this pool
MDS 1 (active)
MDS 2 (standby)
OSS 1
Commodity Storage Servers
Elan
Myrinet
InfiniBand
OSS 2
OSS 3
Lustre Clients 10s - 10,000's
Simultaneous support of multiple network types
Shared storage enables failover OSS
OSS 4
Router
OSS 5
GigE
OSS 6
Enterprise-Class Storage Arrays & SAN Fabrics
= failover
OSS 7

source: http://dlc.sun.com/pdf/820-3681/820-3681.pdf

I/O in highly parallelised programs

- a single process performs I/O
  - no parallelism $\rightarrow$ no scaling
- all processes perform I/O
  - very many system calls $\rightarrow$ danger of I/O system overload or even crash
- some processes perform I/O
  - granularity of parallel I/O processes is given by the parallelism of the global disk/file system
  - programs should be adjustable accordingly

Binary I/O in Fortran and C

Binary I/O in Fortran and C

- example
  - output in the sequential version of the Laplace program
- new routine
  - `output_binary()`
- programs for testing: binary file → ASCII
  - `iotest.c` / `iotest.f90`
  - `iotest2.c` / `iotest2.f90`
- sources

```
./laplace-code/laplace-io-c.tar
./laplace-code/laplace-io-f90.tar
```

Parallel I/O with MPI
– MPI-IO –

Introduction

- problem
  - several processes (and/or threads) are writing to the same file
- solution of MPI-2 (← concepts from MPI-1)
  - ‘simple’ I/O
  - reading and writing to non-consecutive regions (← datatypes)
  - collective I/O operations
  - non-blocking I/O operations
- MPI-IO is designed for unformatted/binary data files
- MPI-IO resembles binary I/O in C
  - position file pointer *(seek)*
  - write block of data *(write)*

Analogies to `Send` and `Recv` for writing with MPI-IO

- the file takes over the role of a receiving process,
  the processes writing can be viewed as sending processes

- `MPI_File_write`
  - corresponds to `MPI_Send`

- `MPI_File_set_view`
  - corresponds to `MPI_Recv`
  - the *displacement* corresponds to the address of the receive buffer

- sender/receiver roles are exchanged for reading with MPI-IO

MPI_File_open() and MPI_File_close()

```
int MPI_File_open(MPI_Comm comm,
                  char *filename,
                  int accessmode,    // bitwise flags (see example)
                  MPI_Info info,
                  MPI_File *fh);     // Fortran: integer
                                     // file handle
int MPI_File_close(MPI_File *fh);
```

example:

```
    MPI_File testfile;
    ...
    MPI_File_open(MPI_COMM_WORLD,
                 "test.dat",
                 MPI_MODE_CREATE + MPI_MODE_WRONLY,
                 MPI_INFO_NULL,
                 &testfile);
    ...
    MPI_File_close(&testfile);
```

MPI_File_seek() – position file pointer

```
int MPI_File_seek(MPI_File fh,
                  MPI_Offset offset, // Fortran: integer(MPI_OFFSET_KIND)
                                     // file offset in bytes ...
                  int whence);       // ... starting from position:
                                     // MPI_SEEK_SET → beginning
                                     // MPI_SEEK_CUR → current position
                                     // MPI_SEEK_END → end
```

MPI_File_read() and MPI_File_write()

Read/write of a *consecutive* block in a *file* at the position set with `MPI_File_seek()`. As specified by `datatype` data can be *non-consecutive* in *memory*.

```
int MPI_File_read(MPI_File fh,
                  void *buf,
                  int count,
                  MPI_Datatype datatype,
                  MPI_Status *status);
```

```
int MPI_File_write(MPI_File fh,
                  void *buf,
                  int count,
                  MPI_Datatype datatype,
                  MPI_Status *status);
```

MPI_File_read_at() and MPI_File_write_at()

Positioning and read/write operation in a combined step.

These routines are *thread-safe*.

```
int MPI_File_read_at(MPI_File fh,
                     MPI_Offset offset, //Fortran: integer(MPI_OFFSET_KIND)
                     void *buf,
                     int count,
                     MPI_Datatype datatype,
                     MPI_Status *status);
```

```
int MPI_File_write_at(MPI_File fh,
                     MPI_Offset offset, //Fortran: integer(MPI_OFFSET_KIND)
                     void *buf,
                     int count,
                     MPI_Datatype datatype,
                     MPI_Status *status);
```

MPI_File_set_view()

I/O to *non-consecutive* regions in a *file*.

```
int MPI_File_set_view(
             MPI_File fh,

             MPI_Offset disp,          // Fortran: integer(MPI_OFFSET_KIND)
                                       // offset/skipped block at beginning

             MPI_Datatype etype,       // "elementary datatype"
                                       // size of an elementary data access

             MPI_Datatype filetype,    // visible data region of fh

             char *datarep,            // data/number representation
                                       // "native"   → no data conversion
                                       // "external" → portable data format

             MPI_Info info);
```

Collective I/O operations

All processes read/write.

```
int MPI_File_read_all(MPI_File fh,
                      void *buf,
                      int count,
                      MPI_Datatype datatype,
                      MPI_Status *status);
```

```
int MPI_File_write_all(MPI_File fh,
                      void *buf,
                      int count,
                      MPI_Datatype datatype,
                      MPI_Status *status)
```

Collective I/O operations can use *aggregation*.

Non-blocking (asynchronous) I/O operations

```
MPI_File_iread                MPI_File_iwrite
MPI_File_iread_at             MPI_File_iwrite_at
MPI_Wait                      MPI_Wait                // known from MPI-1

MPI_File_read_all_begin       MPI_File_write_all_begin
MPI_File_read_all_end         MPI_File_write_all_end
```

Exercise 6

Exercise 6 – MPI I/O

Using ‘simple’ write in MPI I/O.

- Parallel output with `MPI_File_write` in the Laplace example.

$\rightarrow$ `./exercises/mpi/exercise6`

Hybrid parallelisation with MPI and OpenMP

Hybrid programming in general

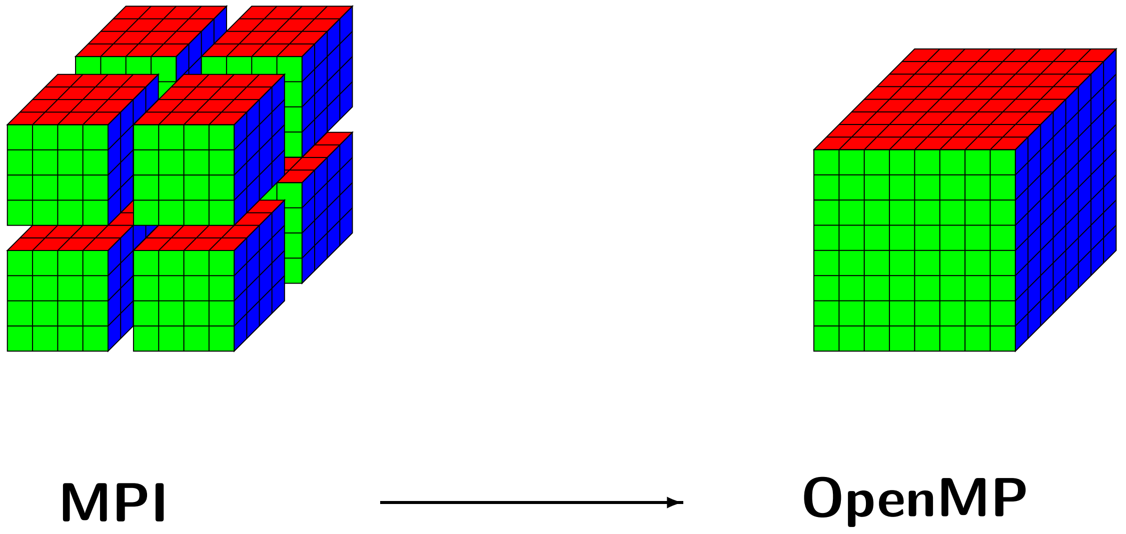
- using more than one programming model
- goal: optimal utilisation of coupled hardware
- examples
  - standard processor (CPU) and graphics card (GPU)
  - cluster of shared memory nodes

Hybrid parallelisation with MPI and OpenMP

- programming of hybrid parallel computers (clusters of shared memory nodes)
- *shared memory* parallelisation inside the nodes with OpenMP
- *message passing* between nodes with MPI
- goals
  - better performance
  - improved scaling
- example
  - astrophysics – *gamma ray bursts* [C. D. Ott et al., 15th ACM Mardi Gras conference]
    `http://www.cct.lsu.edu/˜eschnett/doc/BatonRouge-2008-MardiGras-GammaRayBursts.pdf`

Parallelisation at node level

- example
  - 3-dim. Laplace equation
  - shared memory node with 8 cores



Possible advantages of the hybrid approach

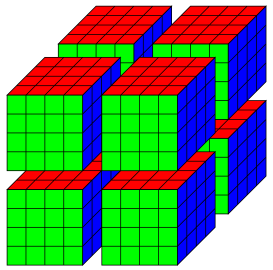
- node level
  - no MPI communication
  - no halo regions $\rightarrow$ reduced memory consumption
  - better utilisation of data caches
  - larger domains $\rightarrow$ possibly more balanced load
  - fewer MPI processes $\rightarrow$ less MPI internal memory usage
- overall program
  - reduction of communication overhead

Reduction of communication overhead

example: 3-dim. Laplace equation

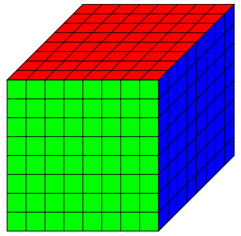
- assumption 1: the mesh is decomposed into cubes of size L^3

  surface $= 6L^2$
  volume $= L^3$
  surface / volume $= 6/L$



- assumption 2: given are SMP nodes with 8 CPUs, the cubes of size L^3 are placed in such a way that they form a cube of size $(2L)^3$

  surface (per node) $= 6(2L)^2$
  volume $= 8L^3$
  surface / volume $= 3/L$



$\Rightarrow$ By hybrid programming (MPI + OpenMP) data traffic is reduced by a factor of 2.

Possible disadvantages of the hybrid approach

- OpenMP overhead
  - barriers / synchronisation
  - *false sharing*
  - sequential parts
  - if only outer loops are parallelised the resulting domain decomposition might lead to less efficient program execution
  - inefficient memory access on NUMA architectures
- in general a few cores can saturate the MPI network
  - what are the other threads/cores doing meanwhile?

Thread-safety

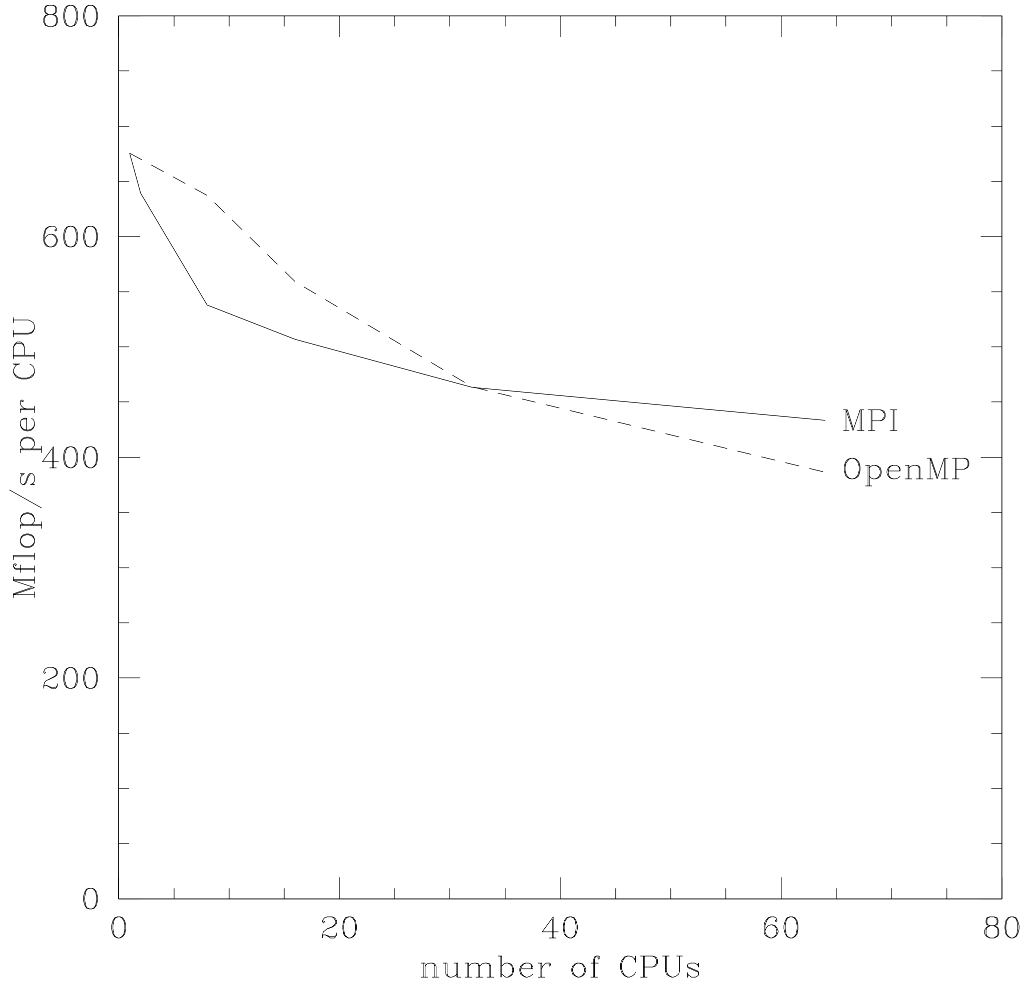
- in an OpenMP program the whole code and libraries must be thread-safe
- MPI-1 has a thread-safe design
  - exceptions: `MPI_Probe`, `MPI_Iprobe`
- however, it is not guaranteed that an MPI-1 *implementation* is thread-safe
  - hybrid programming is possible with MPI-1 alone
  - MPI-2 offers additional functionality

Hybrid programming with MPI-1

- to be on the safe side
  - only the *master-thread* calls MPI
  - MPI is only called in sequential regions

Hybrid programming – example using MPI-1

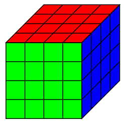
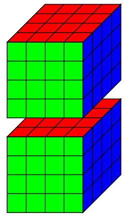
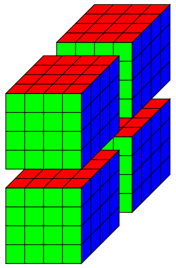
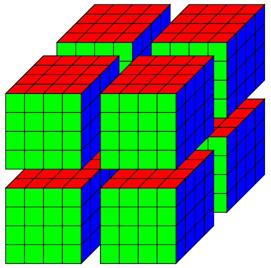
- BQCD on Hitachi SR8000 (8 CPUs (cores) per node, weak scaling: 8^4 lattice per CPU)



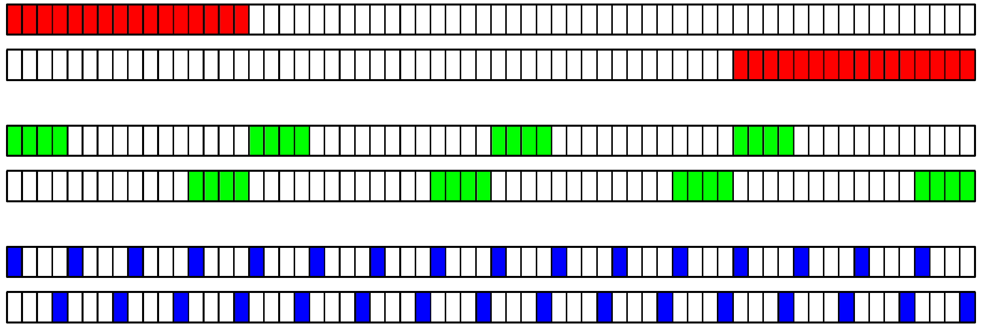
$\leftarrow$ pure MPI

$\leftarrow$ OpenMP / hybrid

Hybrid programming – example using MPI-1

nodes	1	2	4	8
CPUs	8	16	32	64
lattice	$8\times16\times16\times16$	$8\times16\times16\times32$	$8\times16\times32\times32$	$8\times32\times32\times32$
geometry				

storage sequence

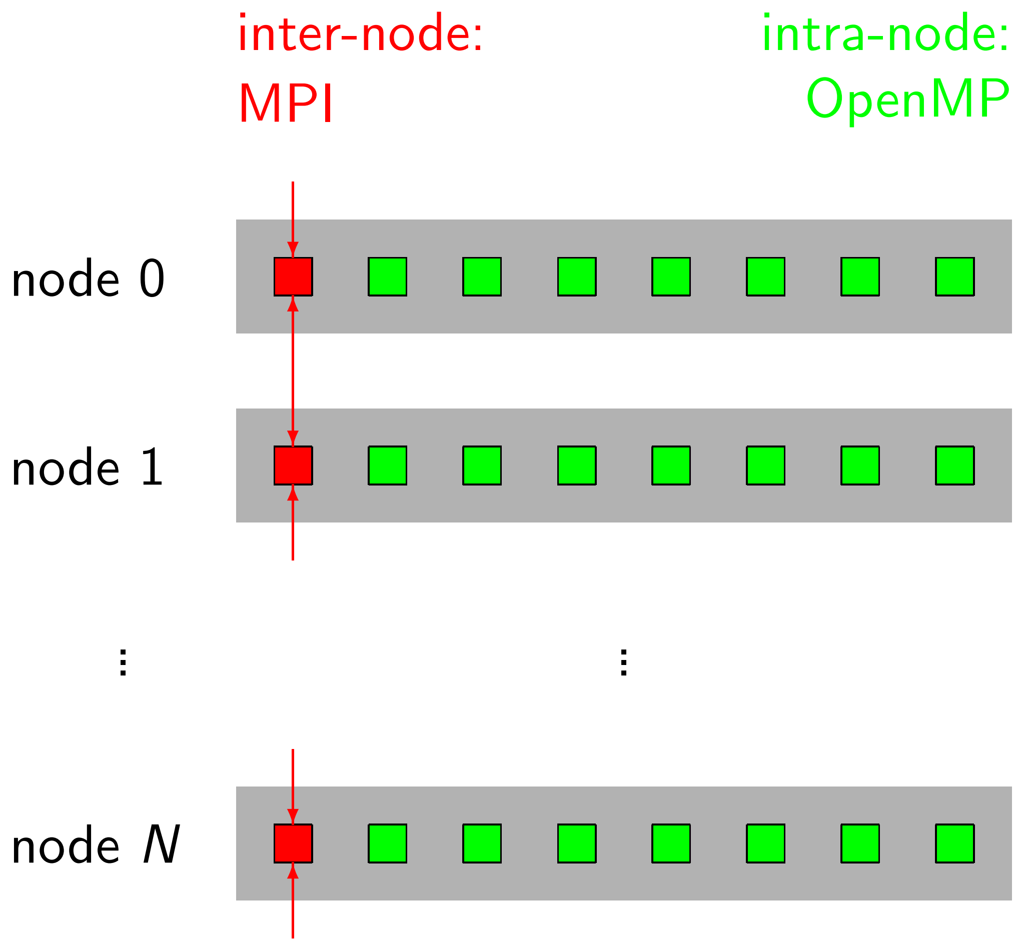


Hybrid programming – example using MPI-1

- discussion
  - at the node level the (pure) OpenMP program is considerably faster
  - on 8 nodes the pure MPI version is faster,
    obviously the network cannot be fully used by a single process
    (this implementation of MPI is not thread-safe $\rightarrow$ per node only one MPI process/thread is possible)

- enhancement
  - now: communication and computation alternate
  - new: communication and computation overlap

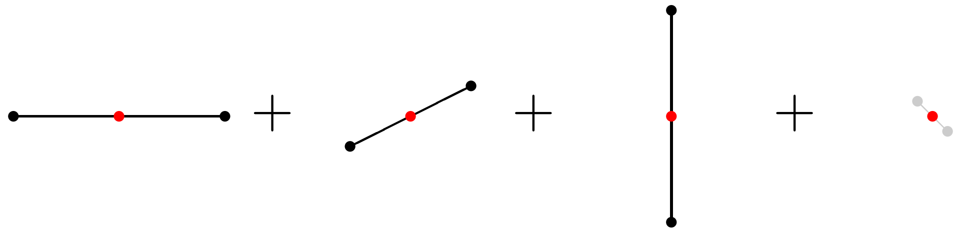
Overlapping communication and computation – principle



Overlapping communication and computation – requirements

- the program must have an appropriate level of parallelism

- parallelisation approach

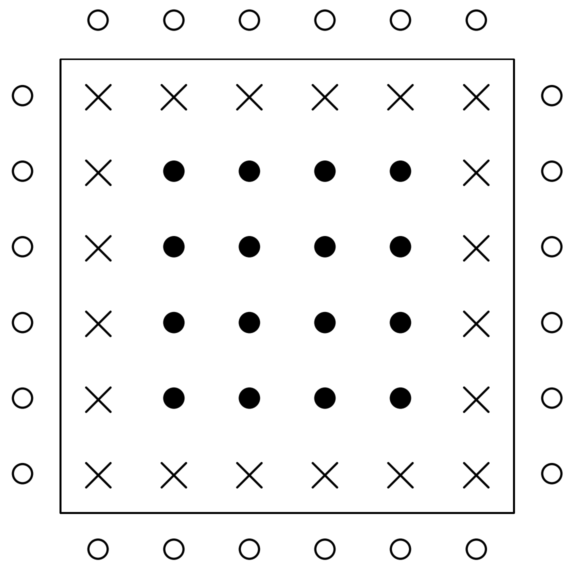
operations per lattice site = 

overlapping is implemented by a *pipeline*

	communicate direction	compute direction
step 1	y	x
step 2	z	y
step 3	t	z
step 4		t

Overlapping communication and computation – approach #2

- local domain:



	compute	communicate
step 1	●	○
step 2	×	

Overlapping communication and computation – implementation

- one stage of the pipeline

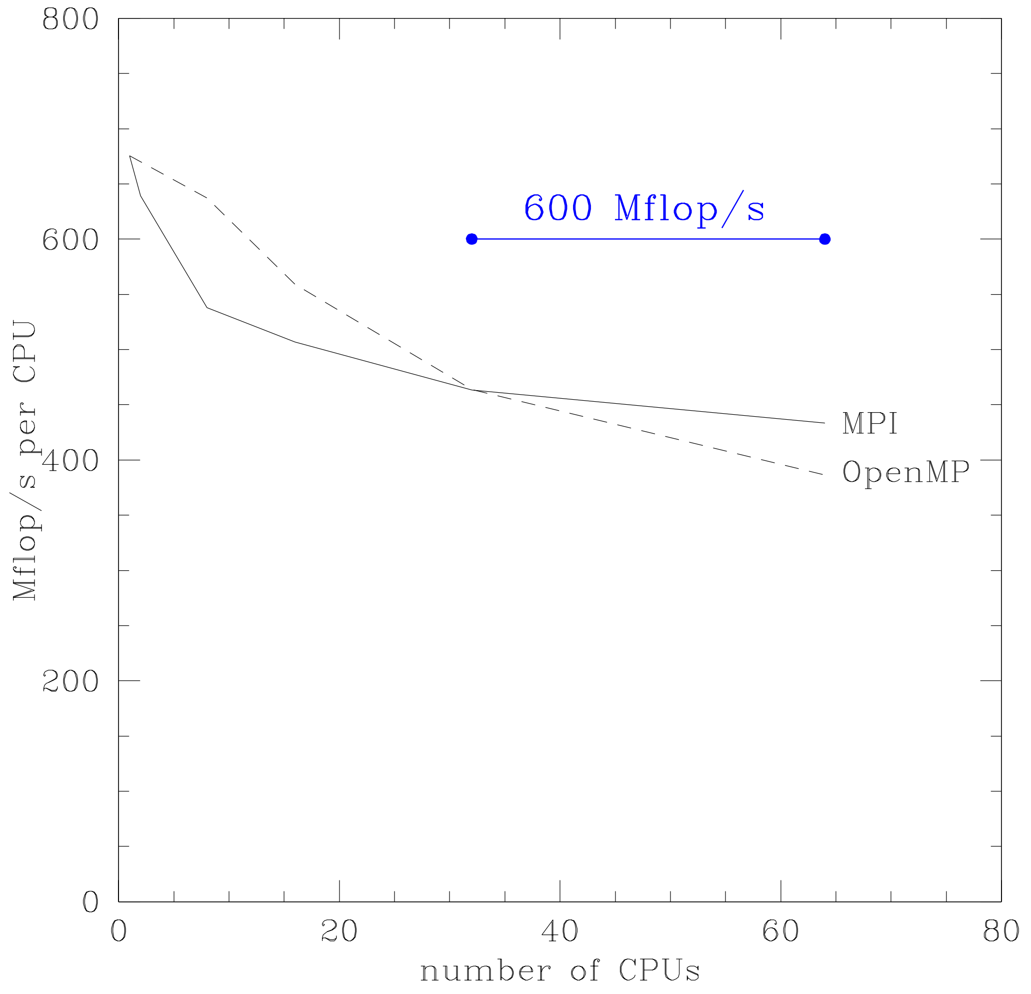
```
!$omp parallel private(i, thread ...)
thread = omp_get_thread_num()

if (thread == 0) then           !  master CPUs communicate
   call mpi_sendrecv(...)
else                            !  other CPUs compute
   do i = i_start(thread), i_end(thread)
    ...
   enddo
endif
!$omp barrier                   !  synchronisation !!

...
!$omp end parallel
```

Overlapping communication and computation – implementation

- calculation of `i_start` and `i_end`

```
n = omp_get_max_threads() - 1
chunk = size / n
rest = size - (chunk * n)

i_start(1) = 1
i_end(1) = chunk
if (rest >= 1) i_end(1) = i_end(1) + 1

do i = 2, n
   i_start(i) = i_end(i - 1) + 1
   i_end(i)   = i_end(i - 1) + chunk
   if (rest >= i) i_end(i) = i_end(i) + 1
enddo
```

Overlapping communication and computation – result



weak scaling: 8^4 lattice per CPU (core)

Hybrid programming with MPI-2

- MPI-2 supports multi-threaded programming
- supports all kinds of threads (e.g. *pthreads*).
- new initialisation:

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

  ```
  mpi_init_thread(required, provided)
  ```

- `required` is the desired thread level support,
  `provided` returns the available thread support
- `MPI_Finalize` is unchanged

master thread vs. main thread

- in OpenMP the *master thread* is always active including sequential regions
  - its thread number is `0`
- in MPI the *main thread* is the thread that makes **all** MPI calls
  - its thread number can be chosen but has to remain fixed

Levels of thread support in MPI-2

`required` and `provided` take the following values (increasing constant `integer`s):

- `MPI_THREAD_SINGLE`

  only one thread is allowed (no other threads will execute)

- `MPI_THREAD_FUNNELED`

  multiple threads are allowed, but only the *main-thread* may call MPI

- `MPI_THREAD_SERIALIZED`

  only one thread at a time may call MPI (to be made sure by the programmer)

- `MPI_THREAD_MULTIPLE`

  no restrictions

Multi-threaded MPI – performance

- single-threaded MPI (see example with MPI-1)
  - one communication thread cannot fully utilise the network
  - for larger numbers of nodes overlapping communication and computation becomes necessary
- multi-threaded MPI
  - several or all threads can communicate
  - hopefully, communication and computation do not have to overlap
  - some threads can communicate while the others compute
    (loads of communication and computation can balanced)

Multi-threaded MPI – implementation

- two processes communicate and run at least two communication threads each
  - the order of messages is no longer guaranteed
  - messages need additional identification (e.g. by *tags*)
  - alternatively one could create communicators for each pair of threads

The mapping problem – mapping problems to machines

- MPI program
  - find a matching domain decomposition
  - map ranks $\leftrightarrow$ nodes
- OpenMP program
  - map threads $\leftrightarrow$ cores
- hybrid program
  - split into MPI and OpenMP
- in any case
  - consider node architecture and network topology

Obtaining node architecture information

- standard command

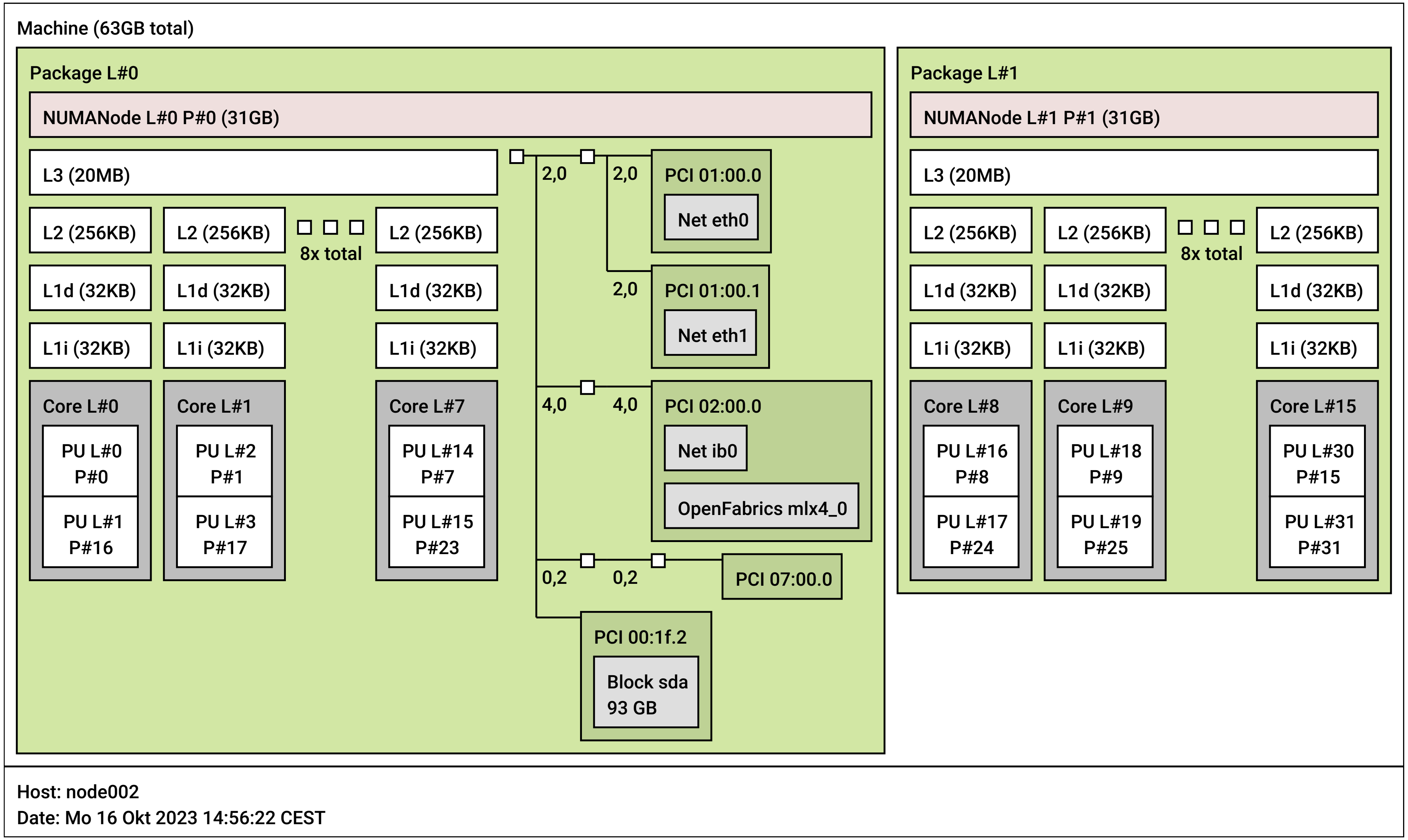
```
lscpu
lscpu -e
```

- command from the *hwloc* package

```
lstopo
lstopo --output-format ascii
```

$\longrightarrow$ demo

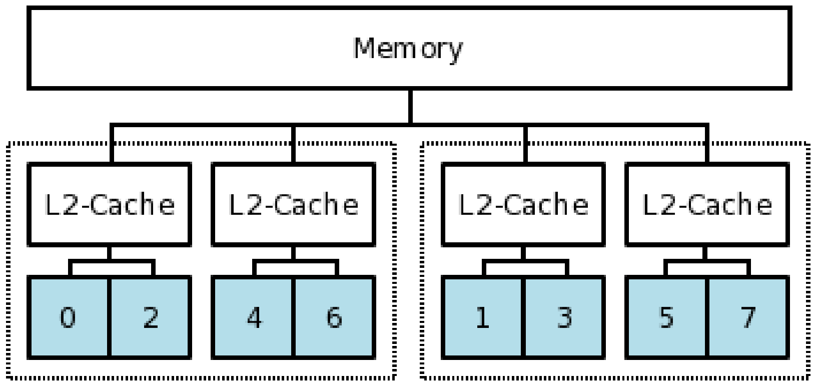
Example output from `lstopo`



Example 2 – architecture of HLRN-II nodes

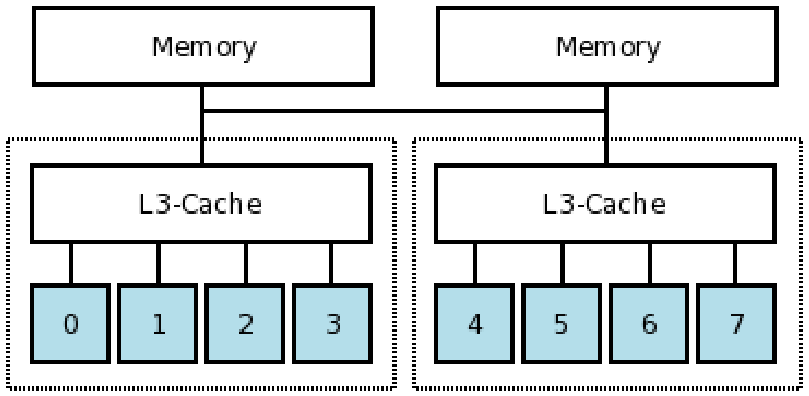
ICE-1

2008: 2 × *Harpertown*



ICE-2

2009: 2 × *Gainestown*



→ `https://www.hlrn.de/home/view/System2/SgiHardware`

Typical splittings between MPI and OpenMP

- 1 MPI process at a hardware level:
  - per node
  - per socket
  - per *NUMA domain*
  - per shared data cache

- number of OpenMP threads per MPI process =

  number of cores at the chosen hardware level

NUMA domains

- current Intel processors
  - have 1 *NUMA domain* per *socket*
- current AMD processors
  - have multiple *NUMA domains* per *socket*
  - keywords
    Core Complex Dies (CCDs)
    Multi-Chip Module (MCM)

Process placement and memory affinity

- problem
  - processes and threads can move from core to core
  - as a consequence memory affinity gets lost
- MPI programs
  - can be *pinned* to cores
- OpenMP programs
  - it becomes interesting, where a thread is placed
- hybrid programs
  - placement and pinning of processes and threads

Placement and pinning of processes and threads

- system calls

```
sched_setaffinity()
sched_getaffinity()
```

Process placement with Open-MPI

- automatic binding (starting with version 1.8)
- options of `mpirun`

```
--bind-to socket
--bind-to core
--bind-to numa
--bind-to ...
--bind-to none

--report-bindings
```

- explicit process placement and binding (since version 4)

```
--bind-to cpu-list:ordered --cpu-list cpulist
```

Process placement with Intel MPI

- automatic binding
- environment variables

`I_MPI_PIN_PROCESSOR_LIST=0,2,4,8`	user defined bindings
`I_MPI_PIN_PROCESSOR_LIST=0-3,8-11`	
`I_MPI_DEBUG=5`	report bindings

Thread placement

- generic OpenMP environment variables

```
OMP_BIND
OMP_PLACES
```

- environment variable for OpenMP programs compiled with GNU or Intel

```
GOMP_CPU_AFFINITY='0-15'
GOMP_CPU_AFFINITY='0 2 4 6 8'
GOMP_CPU_AFFINITY='0 3 1-2 4-15:2'
```

- environment variable for Intel OpenMP

```
KMP_AFFINITY
```

Pinning processes and threads of hybrid programs (I)

- . . . is tricky, see examples on the following slides:
  - running on a node with 2×8 cores
  - note that CPUs 3, 7, 11 and 15 are not used

- programs that print *cpusets* can be found in

```
./demos/cpuset/
```

$\longrightarrow$ demo

Pinning processes and threads of hybrid programs (II)

- GNU compiler and Open-MPI 4.1.1

```
$ OMP_NUM_THREADS=3 OMP_PLACES=cores \
  mpirun -np 4 --map-by numa:PE=4 --rank-by core \
  ./print-cpuset-hybrid | sort
node=node002 rank=0/4 thread=0/3: Cpus_allowed_list:    0,16
node=node002 rank=0/4 thread=1/3: Cpus_allowed_list:    1,17
node=node002 rank=0/4 thread=2/3: Cpus_allowed_list:    2,18
node=node002 rank=1/4 thread=0/3: Cpus_allowed_list:    4,20
node=node002 rank=1/4 thread=1/3: Cpus_allowed_list:    5,21
node=node002 rank=1/4 thread=2/3: Cpus_allowed_list:    6,22
node=node002 rank=2/4 thread=0/3: Cpus_allowed_list:    8,24
node=node002 rank=2/4 thread=1/3: Cpus_allowed_list:    9,25
node=node002 rank=2/4 thread=2/3: Cpus_allowed_list:    10,26
node=node002 rank=3/4 thread=0/3: Cpus_allowed_list:    12,28
node=node002 rank=3/4 thread=1/3: Cpus_allowed_list:    13,29
node=node002 rank=3/4 thread=2/3: Cpus_allowed_list:    14,30
```

Pinning processes and threads of hybrid programs (III)

- Intel compiler and Intel MPI

```
$ OMP_NUM_THREADS=3 \
  I_MPI_PIN_DOMAIN=auto \
  KMP_AFFINITY=granularity=fine,compact,2,0 \
  mpirun -np 4 ./print-cpuset-hybrid | sort
node=node002 rank=0/4 thread=0/3: Cpus_allowed_list:    0
node=node002 rank=0/4 thread=1/3: Cpus_allowed_list:    1
node=node002 rank=0/4 thread=2/3: Cpus_allowed_list:    2
node=node002 rank=1/4 thread=0/3: Cpus_allowed_list:    4
node=node002 rank=1/4 thread=1/3: Cpus_allowed_list:    5
node=node002 rank=1/4 thread=2/3: Cpus_allowed_list:    6
node=node002 rank=2/4 thread=0/3: Cpus_allowed_list:    8
node=node002 rank=2/4 thread=1/3: Cpus_allowed_list:    9
node=node002 rank=2/4 thread=2/3: Cpus_allowed_list:    10
node=node002 rank=3/4 thread=0/3: Cpus_allowed_list:    12
node=node002 rank=3/4 thread=1/3: Cpus_allowed_list:    13
node=node002 rank=3/4 thread=2/3: Cpus_allowed_list:    14
```

Exercise 7

Exercise 7 – hybrid programming

Back to the (programming) roots:

- Write a hybrid *hello world* program.
- Run the program in several modes:
  - single process, single *thread*
  - pure MPI
  - pure OpenMP
  - hybrid

→ `./exercises/mpi/exercise7`