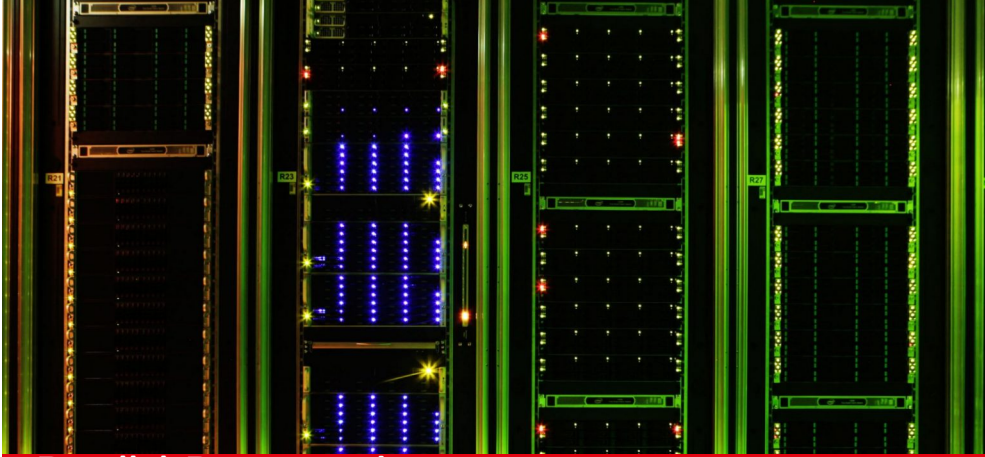


Parallel Programming

Single-core optimization, MPI, OpenMP, and hybrid programming

Nicolas Richart
Emmanuel Lanti

Course based on V. Keller's lecture notes

14^{th} - 18^{th} of November 2022

Advanced MPI

- Overview of more advanced functionalities
- Persistent communications
- Advanced collective communications
- Describing your own datatype
- Redefining communicators
- Associating a topology to a communicator
- Parallel I/O
- One sided communications

Persistent communications

- `MPI_Send_init` `MPI_Recv_init`, initialize the communication
- Same signature as non-blocking communications
- `MPI_Start`, `MPI_Startall` to start the communication
- Completion is checked the same way as for non-blocking

Persistent communications

- Replace the non-blocking communication in the Poisson code by persistent ones

Collective communications

V extension to `MPI_Gather`

Syntax

```
int MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, const int recvcounts[], const int displs[],
                MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- `recvcounts` is now an array, one entry per rank
- `displs` array of displacements defining where to place the i^{th} receive data
- receive different sizes per process
- receive in an array with strides

Semantic equivalent

```
1 // Every process
2 MPI_Send(sendbuf, sendcount, sendtype, root, /*...*/);
3
4 // On root process
5 for(i = 0; i < nb_process; ++i)
6   MPI_Recv(recvbuf+displs[j] * extent(recvtype), recvcounts[j], recvtype, i,
7            /*...*/);
```

Collective communications

V extension to `MPI_Scatter`

Syntax

```
1 int MPI_Scatterv(const void *sendbuf, const int sendcounts[],
2                  const int displs[], MPI_Datatype sendtype, void *recvbuf,
3                  int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- `sendcounts` is now an array, one entry per rank
- `displs` array of displacements defining where to place the i^{th} receive data
- receive different sizes
- receive in an array with strides

Semantic equivalent

```
1 // On root process
2 for(i = 0; i < nb_process; ++i)
3   MPI_Send(sendbuf+displs[i]*extent(sendtype), sendcounts[i], sendtype, i,
4            /*...*/)
5
6 // Every process
7 MPI_Recv(recvbuf, recvcount, recvtype, i, /*...*/).
```

Non-blocking collective communications

- **I** variant of collective communications
- extra parameter `request`
- `MPI_Ibarrier`, `MPI_Ibcast`
- `MPI_Igather`, `MPI_Igatherv`, `MPI_Iscatter`, `MPI_Iscatterv`
- `MPI_Iallgather`, `MPI_Iallgatherv`, `MPI_Ialltoall`
- `MPI_Ireduce`, `MPI_Iallreduce`, `MPI_Iscan`, `MPI_Iexscan`

Persistent collective communications

- **_init** variant of collective communications
- extra parameter `request`
- `MPI_Barrier_init`, `MPI_Bcast_init`
- `MPI_Gather_init`, `MPI_Gatherv_init`, `MPI_Scatter_init`, `MPI_Scatterv_init`
- `MPI_Allgather_init`, `MPI_Allgatherv_init`, `MPI_Alltoall_init`
- `MPI_Reduce_init`, `MPI_Allreduce_init`, `MPI_Scan_init`, `MPI_Exscan_init`

Persistent collective

- Replace the the `MPI_Allreduce` by a persistent one

- `MPI_Datatype` opaque type containing a *Typemap*
  - $Typemap = \{(type_0, disp_0), \cdots, (type_{n-1}, disp_{n-1})\}$
  - sequence of basic datatypes
  - sequence of displacements (in bytes)
- **extent** is the span from the first byte to the last one, with alignment requirement

$$lb(Typemap) = \min_j(disp_j),$$

$$ub(Typemap) = \max_j(disp_j + \text{sizeof}(type_j)) + \epsilon, \text{ and}$$

$$extent(Typemap) = ub(Typemap) - lb(Typemap)$$

ϵ is there to account for alignment requirements

MPI datatype	C datatype
MPI_CHAR	char
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	signed long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
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 datatype	C datatype
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t
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 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 datatype	C datatype
`MPI_AINT`	`MPI_Aint`
`MPI_OFFSET`	`MPI_Offset`
`MPI_COUNT`	`MPI_Count`
`MPI_BYTE`	
`MPI_PACKED`	

Syntax

```
1 int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
2                         MPI_Datatype *newtype);
3
4 int MPI_Type_vector(int count, int blocklength, int stride,
5                     MPI_Datatype oldtype, MPI_Datatype *newtype);
```

- array of contiguous array or with strided blocks of same type
- `count` : number of repetition (blocks)
- `blocklength` : number of element per block
- `stride` : number of element between start of each block

- `MPI_Type_create_hvector` : same as `MPI_Type_vector` with `stride` expressed in bytes
- `MPI_Type_create_indexed_block` same as `MPI_Type_vector` with array of and `displacements`
- `MPI_Type_create_hindexed_block` : same as `MPI_Type_create_indexed_block` with `displacements` in bytes
- `MPI_Type_indexed` : same as `MPI_Type_create_indexed_block` with arrays of `blocklengths`
- `MPI_Type_create_hindexed` : same as `MPI_Type_indexed` with `displacements` in bytes

Syntax

```
int MPI_Type_create_struct(int count, const int array_of_blocklengths[],
                           const MPI_Aint array_of_displacements[],
                           const MPI_Datatype array_of_types[], MPI_Datatype *newtype)
```

- `count` : number of repetition (blocks)
- `array_of_blocklengths` : sizes per block
- `array_of_displacements` : displacements between blocks in bytes
- `array_of_types` : types contained in each blocks

Derived Datatypes

Usefull helper functions

- `MPI_Get_address` : get the address of a variable
- `MPI_Aint_diff` : get the difference between 2 addresses
- `MPI_Aint_add` : get the sum of 2 addresses
- `MPI_Type_size` : get the size of a datatype
- `MPI_Get_type_extent` : get the lower bound and the extent of a type
- `MPI_Type_create_resized` : reset the lower bound and the extent of a type

Derived Datatypes

Commit/free

Syntax

```
1 int MPI_Type_commit(MPI_Datatype *datatype);
2
3 int MPI_Type_free(MPI_Datatype *datatype);
```

- new datatypes should be committed before being usable in communications
- committed types need to be freed once not used anymore

Derived Datatypes

Example

mpi/datatypes.cc

```
13 struct Test_t {
14   double d[2];
15   int i;
16 };
17
18 std::vector<Test_t> foo(100);
19
20 std::array<int, 2> block_lengths = {2, 1};
21 std::array<MPI_Aint, 2> displacements;
22 std::array<MPI_Datatype, 2> old_types = {MPI_DOUBLE, MPI_INT};
23
24 MPI_Aint addr0, addr1;
25 MPI_Get_address(&foo[0], &addr0);
26 MPI_Get_address(&foo[0].d[0], &displacements[0]);
27 MPI_Get_address(&foo[0].i, &displacements[1]);
28
29 displacements[0] = MPI_Aint_diff(displacements[0], addr0);
30 displacements[1] = MPI_Aint_diff(displacements[1], addr0);
31
32 MPI_Datatype mpi_test_t, mpi_test_vector_t;
33
34 MPI_Type_create_struct(2, block_lengths.data(), displacements.data(),
35                        old_types.data(), &mpi_test_t);
36
37 MPI_Get_address(&foo[1], &addr1);
38 addr1 = MPI_Aint_diff(addr1, addr0);
39
40 MPI_Type_create_resized(mpi_test_t, 0, addr1, &mpi_test_vector_t);
41 MPI_Type_commit(&mpi_test_vector_t);
```

Derived Datatypes

Send lines in poisson code

- Create a `MPI_Datatype line_t` representing a line of data
- Exchange data of type `line_t` instead of `MPI_FLOAT`

Pack/Unpack

Pack

Syntax

```
1 int MPI_Pack(const void *inbuf, int incount, MPI_Datatype datatype,
2               void *outbuf, int outsize, int *position, MPI_Comm comm);
```

- `inbuf`, `incount`, `datatype` correspond to the description of data to pack
- `outbuf`, `outsize` description of the buffer where to pack
- `position` current position in the packing buffer

Pack/Unpack

Unpack

Syntax

```
int MPI_Unpack(const void *inbuf, int insize, int *position, void *outbuf,
               int outcount, MPI_Datatype datatype, MPI_Comm comm);
```

- `inbuf`, `incount`, description of the buffer from which to unpack
- `position` current position in the unpacking buffer
- `outbuf`, `outsize`, and `datatype` correspond to the description of data to unpack

Pack/Unpack

Example

mpi/pack_unpack.cc

```
26
27 if (rank == 0) {
28   a = 0xcafe;
29   MPI_Pack(&a, 1, MPI_INT, buf.data(), buf.size(), &position, MPI_COMM_WORLD);
30   MPI_Pack(d, 10, MPI_DOUBLE, buf.data(), buf.size(), &position,
31            MPI_COMM_WORLD);
32   MPI_Send(buf.data(), position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
33 } else if (rank == 1) {
34   MPI_Recv(buf.data(), buf.size(), MPI_PACKED, 0, 0, MPI_COMM_WORLD, &status);
35   MPI_Unpack(buf.data(), buf.size(), &position, &a, 1, MPI_INT,
36              MPI_COMM_WORLD);
37   MPI_Unpack(buf.data(), buf.size(), &position, d, 10, MPI_DOUBLE,
38              MPI_COMM_WORLD);
39 }
```

Groups and Communicators

- a **communicator**:
  - Encapsulate a **context**, a **group**, a **virtual topology** and **attributes**
  - Two kinds **intra-communicator** and **inter-communicator**
- a **group**:
  - ordered set of processes
  - each process has an unique ID (rank within the group) and can belong to several different groups
  - a group can be used to create a new communicator

- duplicating or splitting an existing one `MPI_Comm_dup`, `MPI_Comm_split`
- creating communicator from a group `MPI_Comm_create`, `MPI_Comm_create_group`
- need to create groups
  - from a communicator `MPI_Comm_group`
  - boolean operations `MPI_Group_union`, `MPI_Group_intersection`, `MPI_Group_difference`
  - specifying ranks `MPI_Group_incl`, `MPI_Group_excl`
- destroy created objects `MPI_Comm_free`, `MPI_Group_free`

Virtual Topologies

- potential performance gain by mapping process to hardware
- helps for program readability
- types of topologies: Cartesian, Graph, Distributed Graph
- collective communication on neighborhoods

Syntax

```
1 int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],
2                     const int periods[], int reorder, MPI_Comm *comm_cart);
```

- create a communicator with cartesian information
- convenient functions:
  - `MPI_Dims_create` helps creating balanced distribution of process
  - `MPI_Cart_shift` helps determining neighboors
  - `MPI_Cart_rank` get the rank based on coordinates
  - `MPI_Cart_coords` get coordinates based on rank

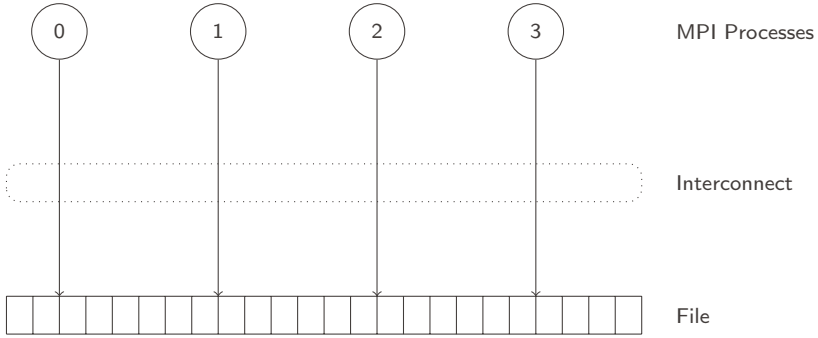
- `MPI_Neighbor_allgather` assuming we are on process with rank i, gather data from all rank j if edge (j, i) exists and send same data to all j where edge (i, j) exists
- `MPI_Neighbor_alltoall` compare to allgather, sends different data to all j process
- vector variant are available **v**
- immediate variant are available **I**
- persistent variant are available **_init**
- `MPI_Neighbor_alltoall` as one in all flavors the **w**, different datatypes are echanged with all neighbors

Virtual topology

- Rewrite the parallelism using a cartesian communicator
- Use neighbor collective communications

Parallel I/O overview

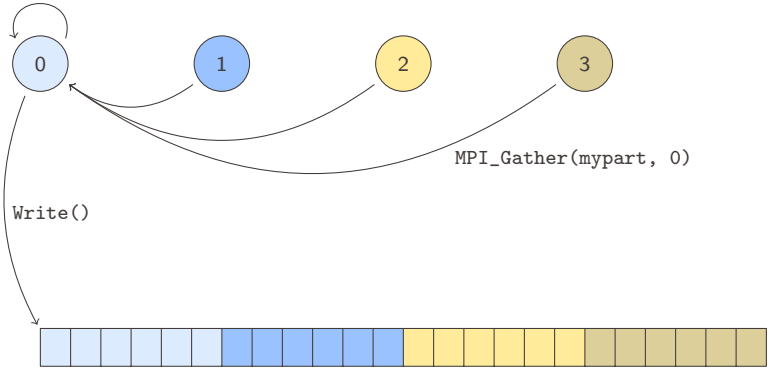
- I/O is often (if not always) the main bottleneck in a parallel application
- MPI provides a mechanism to read/write in parallel



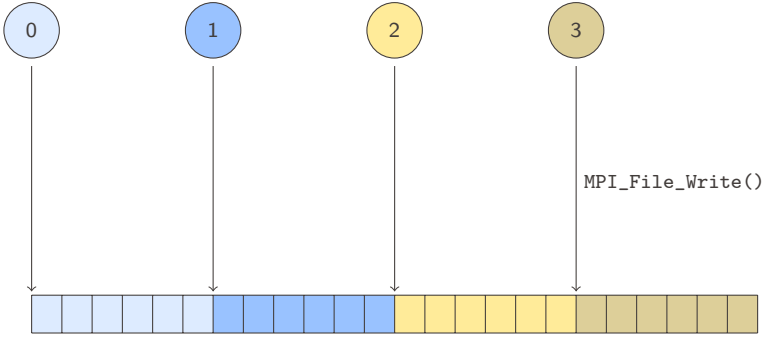
Introducing remarks

- MPI IO API works on your desktop/laptop
- Most of the large HPC systems have a **parallel file system** (like GPFS, Lustre, *etc.*)
- If the file is distributed smartly on a parallel file system: performance increases
- MPI IO offers a high-level API to access a distributed file (no needs to implement complex POSIX calls)
- **does not work with ASCII files**
- Most of the standard file format support MPI IO (*e.g.* HDF5, NetCDF, *etc.*.)

Poisson so far



Poisson ideal



Open/Close a file in parallel

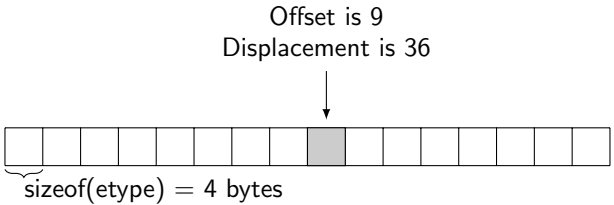
Syntax

```
1 int MPI_File_open(MPI_Comm comm, const char *filename, int amode,
2                   MPI_Info info, MPI_File *fh);
3
4 int MPI_File_close(MPI_File *fh);
```

- `comm` : the communicator that contains the writing/reading MPI processes
- `filename` : a file name
- `amode` : file access mode, `MPI_MODE_RDONLY` , `MPI_MODE_WRONLY` , `MPI_MODE_RDWR` , `MPI_MODE_CREATE` , *e.t.c.*
- `info` : file info object (`MPI_INFO_NULL` is a valid info)
- `fh` : file handle

Collective calls !!

- **etype** is the elementary type of the data of the parallel accessed file
- **offset** is a position in the file in term of multiple of etypes
- **displacement** of a position within the file is the number of bytes from the beginning of the file



Syntax

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

int MPI_File_write_at(MPI_File fh, MPI_Offset offset, const void *buf,
                      int count, MPI_Datatype datatype, MPI_Status *status);
```

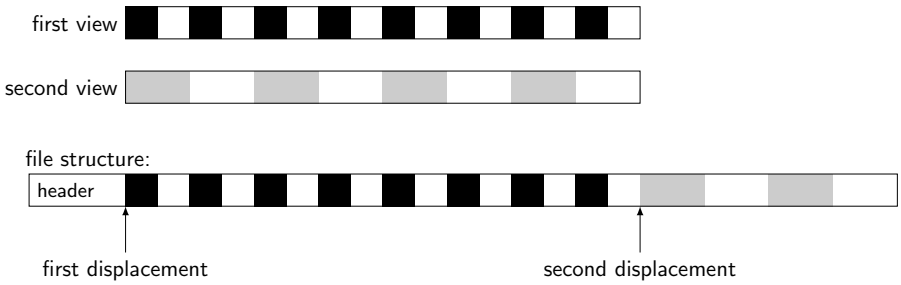
- Can be used from a single (or group) of processes
- `offset` must be specified in the `buf` buffer
- `count` elements of type `datatype` are written

Syntax

```
1 int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
2                       MPI_Datatype filetype, const char *datarep, MPI_Info info);
3
4 int MPI_File_get_view(MPI_File fh, MPI_Offset *disp, MPI_Datatype *etype,
5                       MPI_Datatype *filetype, char *datarep);
```

- initially, each process view the file as a linear byte stream and each process views data in its own native representation
- `disp` is the displacement (defines the beginning of the data of the file that belongs to the process) in byte
- `etype` is the unit of data access and positioning
- `filetype` is a single `etype` of a multiple of it

Setting up a view



(source : MPI 2.2 specifications)

Parallel IO

Simple independent read/write without offset

Syntax

```
1 int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
2                   MPI_Status *status);
3
4 int MPI_File_write(MPI_File fh, const void *buf, int count,
5                    MPI_Datatype datatype, MPI_Status *status);
```

Syntax

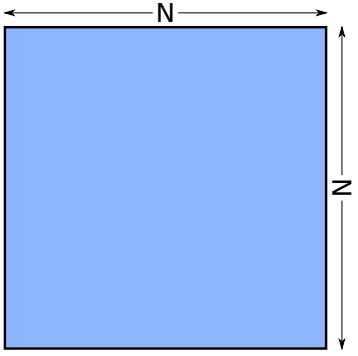
```
int MPI_File_write_all(MPI_File fh, const void *buf, int count,
                       MPI_Datatype datatype, MPI_Status *status);

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

What we did not view

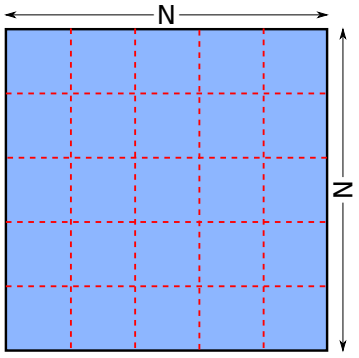
- One Sided communications
  - `MPI_Put`, `MPI_Get`
  - `MPI_Win_*`
  - shared memory
- Process management
  - `MPI_Comm_spawn`
  - Communications on inter-communicators

Parallelization of the poisson code



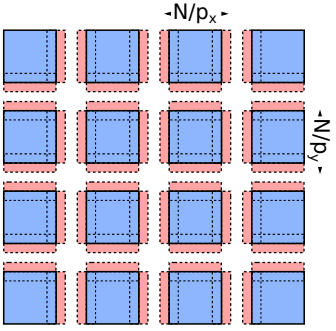
- Parallelize the Poisson 2D problem using the Messages Passing Interface (MPI)

Parallelization of the poisson code



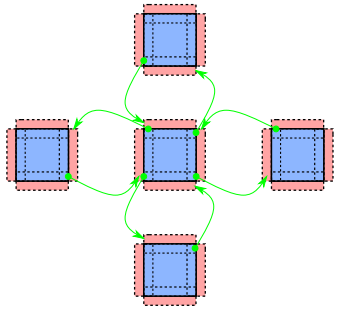
- This time, we want to make a 2D domain decomposition using Cartesian topology
- Use **MPI_Dims_create** and **MPI_Cart_create** to create a Cratesian topology

Parallelization of the poisson code



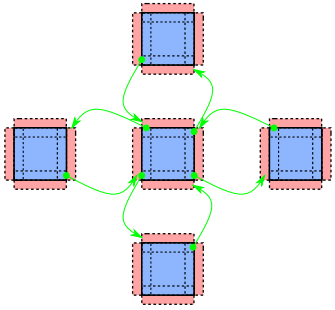
- The p processes are split into (p_x, p_y) to make the Cartesian grid
- Each domain has size $(N/p_x, N/p_y)$ (1 per process)
- Use **MPI_Cart_shift** to find the neighboring domains

Parallelization of the poisson code



- Adding *ghost* lines before and after
- Use the *ghost* lines to receive the missing local data
- You will need to define a new *matrix column* datatype and update the *matrix line* datatype

Parallelization of the poisson code



- Use the **MPI_neighbor_alltoallw** routine
- You can use the number of iteration as a check
- Remove the `dump()` function to start