HPC Summer School 2018

Distributed-Memory Programming with MPI

Gang Liu and Hartmut Schmider
Centre for Advanced Computing (CAC)
{gang.liu, hartmut.schmider}@queensu.ca

Kingston

Outlines

- Introduction
- MPI basics
  - Programming environments
  - MPI predefined data types
  - Communications
  - User defined data types
  - Runtime environments
  - Some remarks
- Array distribution
- Sub-task distribution
- CAC bonus libraries
- References

Outlines

- Introduction
- MPI basics
    - Programming environments
    - MPI predefined data types
    - Communications
    - User defined data types
    - Runtime environments
    - Some remarks
- Array distribution
- Sub-task distribution
- CAC bonus libraries
- References

MPI

- Message Passing Interface
- System of subroutines/functions for communication between processes and facilities for such purpose in Fortran (90), C, and C++.
- Used for parallel computing on any combination of computers/clusters.

MPI Example 1 in Fortran

```
PROGRAM EXAMPLE01
USE MPI
INTEGER MYID,  TOTPS,  IERR
CALL MPI_INIT( IERR )
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYID, IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, TOTPS, IERR)
WRITE(*,*)"Hello from rank:",MYID," of total", &
                            TOTPS, " processes."
CALL MPI_FINALIZE(IERR)
END
```

MPI Example 1 in C

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

int main(int argc, char* argv[])
{  int myid, totps;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
   MPI_Comm_size(MPI_COMM_WORLD, &totps);
   printf (“Hi from rank: %d of %d processes.\n",
                                   myid, totps);
   MPI_Finalize();
}
```

MPI Example 1 in C++

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

int main()
{ MPI::Intracomm commall = MPI::COMM_WORLD;
  MPI::Init();
  int myid  = commall.Get_rank();
  int totps = commall.Get_size();
  printf (“Hi from rank: %d of %d processes.\n",
                                 myid, totps);
  MPI::Finalize();
}
```

Lab works

- Login to your account in CAC (login.cac.queensu.ca)
- `tar -xvf /global/project/workshop/mpi-lesson.tar`
- `[your_acount@caclogin02 ~]$`
- `salloc --reservation summer-school -A teaching -n 4 --mem 8g`
- `salloc: Granted job allocation …`
- `[your_acount@cac034 ~]$`

- Then you get 4 CPUs and 8GB memory to use exclusively for the lab today.

Lab work # 1

```
cd      mpi
cd      F90       (C,     CPP)
cd      f01       (c01,   cpp01)
cat     f01.f     (c01.c, cpp01.cpp)
mpif90  f01.f     (for FORTRAN) or
mpicc   c01.c     (for c)       or
mpicxx  cpp01.cpp (for C++)
mpirun  -np  4    ./a.out
```

Running Example 1

```
$ mpif90  f01.f
$ mpirun -np 9 ./a.out
  Hello from rank: 0  of total  9  processes.
  Hello from rank: 1  of total  9  processes.
  Hello from rank: 2  of total  9  processes.
  Hello from rank: 3  of total  9  processes.
  Hello from rank: 5  of total  9  processes.
  Hello from rank: 6  of total  9  processes.
  Hello from rank: 7  of total  9  processes.
  Hello from rank: 4  of total  9  processes.
  Hello from rank: 8  of total  9  processes.
```

Analyzing MPI Example 1

- How many source and executable code(s)?

  1 each

- How many WRITE(*,*) statement(s) in source code?

  1

- How many CPUs we asked?

  9

- How many outputs from the only one WRITE(*,*) ?

  9

In fact, 9 copies of the executable are run on 9 CPUs, like 9 complete independent codes running separately but simultaneously.

Process

Any set of instructions executed on a processor (CPU), in sequential/serial manner.

Any serial code run is a process. Any section of a serial code run is also a process, but the sections are run one after another.

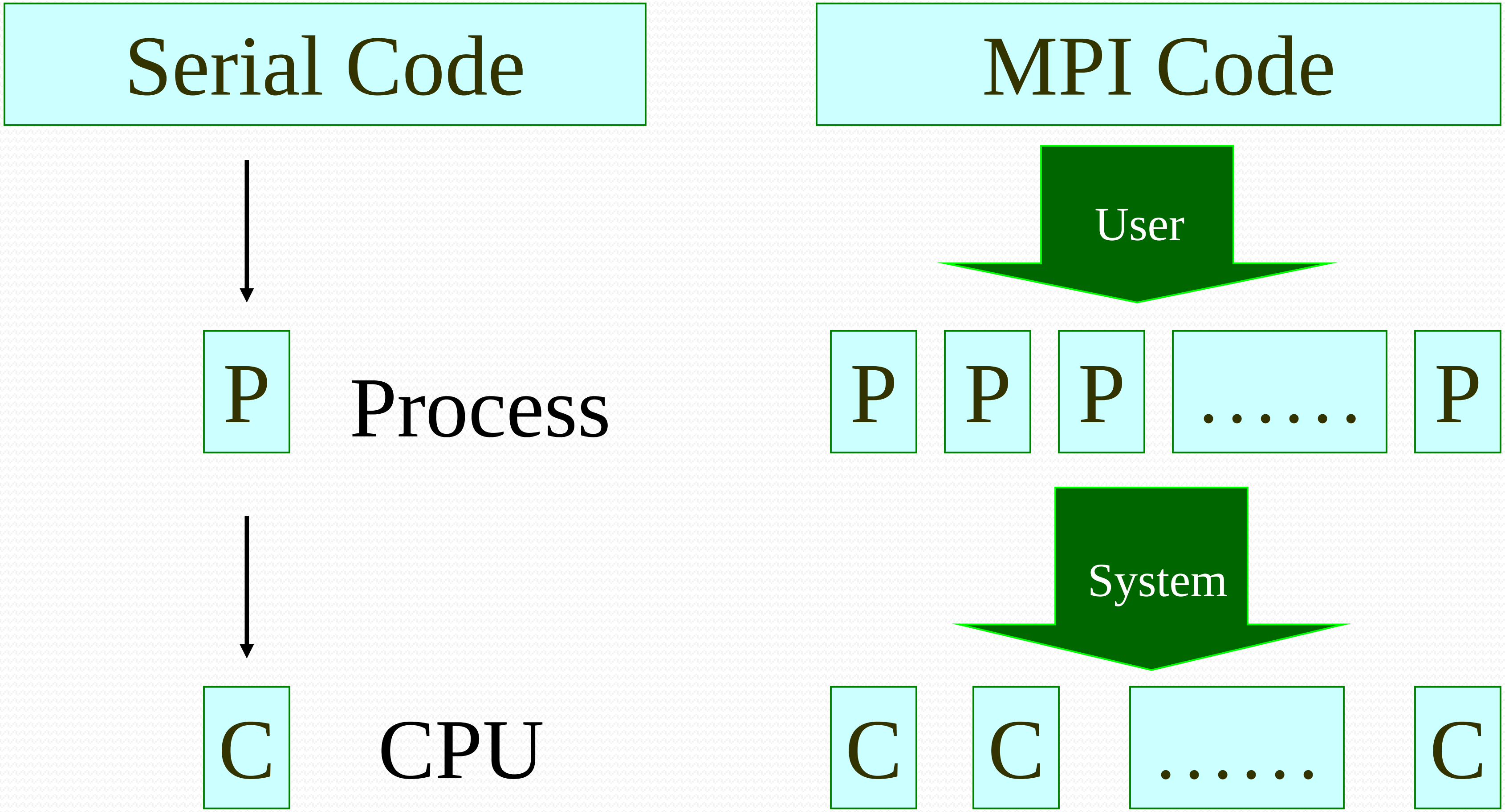
In MPI, a process usually means a full copy of the code being run, and many processes can be working at the same time.

When we submit an MPI job with the command

```
mpirun   –np    N    ./a.out
```

we are asking N processes to run a copy of the code each. Then the operating system allocates CPUs for all processes. MPI can not allocate CPUs directly.

A calculation job



Number of Processes

For efficiency, always choose the number of processes **smaller** than the number of available CPUs. This ensures that every process can get one CPU exclusively, i.e. is executed on a **dedicated** processor.

The first basic feature of MPI

An MPI code is usually run by a group of processes simultaneously.

Each process executes the code serially by iteself and independently on any other process, in principle.

“Ranks” for each process in MPI

0

1

2

…

Number of Processes -1

as each process identify itself with a unique number and thus performs some unique tasks.

MPI_COMM_RANK(…,RANK_or_MYID,…)

MPI_COMM_SIZE(…,Total_Number_of_Processes,…)

The RANK Numbers Outputted from Example 1

```
Hello from rank: 0  of total 9  processes.
Hello from rank: 1  of total 9  processes.
Hello from rank: 2  of total 9  processes.
Hello from rank: 3  of total 9  processes.
Hello from rank: 5  of total 9  processes.
Hello from rank: 6  of total 9  processes.
Hello from rank: 7  of total 9  processes.
Hello from rank: 4  of total 9  processes.
Hello from rank: 8  of total 9  processes.
```

The second basic feature of MPI

- ***Processes can identify themselves with the rank numbers and know all co-workers accurately.***

The Output from Example 1

```
Hello from rank: 0  of total 9  processes.
Hello from rank: 1  of total 9  processes.
Hello from rank: 2  of total 9  processes.
Hello from rank: 3  of total 9  processes.
Hello from rank: 5  of total 9  processes.
Hello from rank: 6  of total 9  processes.
Hello from rank: 7  of total 9  processes.
Hello from rank: 4  of total 9  processes.
Hello from rank: 8  of total 9  processes.
```

Lower rank does **not** imply earlier execution

The third basic feature of MPI

Any process always proceeds ahead immediately and run as quickly as possible.

No execution order among processes is reserved by default. None of the processes has a higher priority.

If such an order is really needed at certain points, it can be achieved by calling some MPI routines intentionally.

Analyzing MPI Example 1 Again

- How many source codes and executables?

  1

- How many times to declare the "MYID" variable in the code?

  1

- How many different values of the "MYID" variable outputted?

  9 from 9 processes

In fact: each process has its own independent copy of the "MYID" in its own memory space, i.e. memory is distributed.

Not in a one-car family, father drives work, mother shopping, son hockey , then daughter volleyball in sequence. But everyone has his/her own car, so a four-car family.

Analyzing MPI Example 1 Again

Although these “MYID” variables are named and referred to as the same way inside their own processes respectively, like the “first sons” but in different families, they are absolutely different individuals.

Shared vs Distributed memory

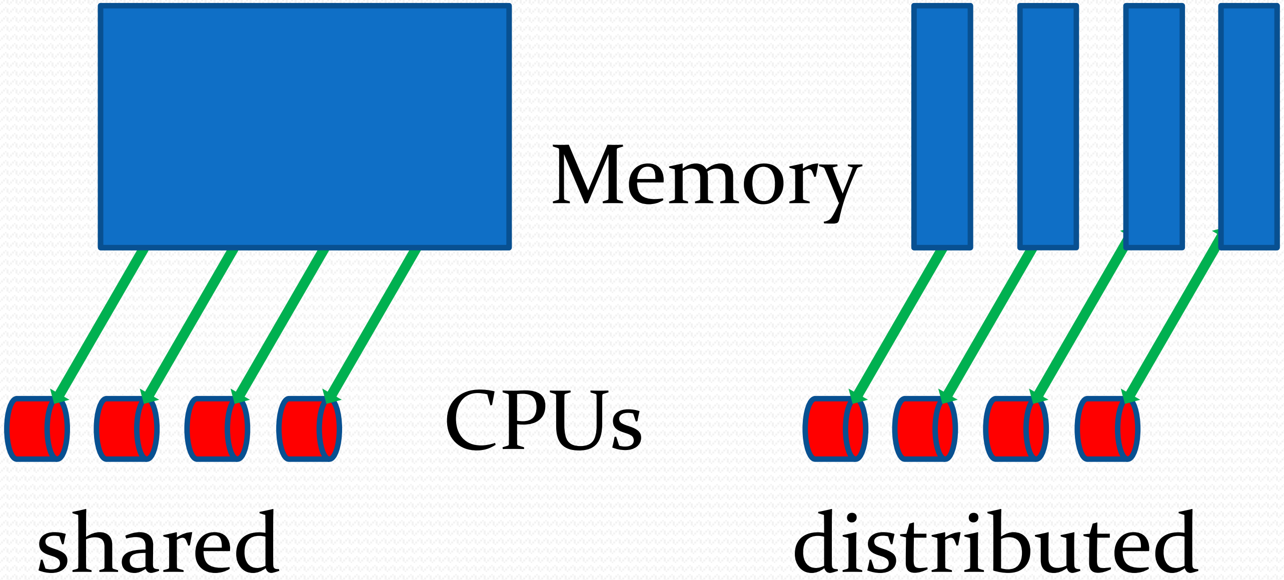
- Parallel computation means many processes are employed for computing at the same time on many CPUs to speed up.
- Each process must use some memory as working space.
- Then we are facing the choices of shared or distributed memory.

Shared vs Distributed memory

- If the same memory space can be accessed by some CPUs directly, it is shared;

- otherwise, if each CPU can only access its own exclusive memory space directly, the memory is distributed.

Shared vs Distributed memory



Memory

CPUs

shared

distributed

Shared vs Distributed memory

OpenMP can only work in physically shared memory machines.

MPI can work anywhere.

When MPI runs on physically shared-memory machines, the memory is used as distributed.

Shared vs Distributed memory

Repeatedly in one word, from MPI point of view, the memory is always distributed.

From OpenMP point of view, everything in MPI is private.

The fourth basic feature of MPI

- Whenever a process sees a variable or an array declaration , it allocates memory accordingly to have a copy of it, but in its own distributed memory space. Dynamically allocated ones the same.

- Then different processes have completely different pieces of physical memory for the variable/array, then can store the same or different values there independently.

- Each process can only access its own copy of them directly.

- The only way to get data from any other processes is MPI communications, except using external files. MPI is designed for such a purpose effectively and reliably.

MPI Example 1 in Fortran

```
PROGRAM EXAMPLE01
USE MPI
INTEGER MYID,  TOTPS,  IERR
CALL MPI_INIT( IERR )
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYID, IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, TOTPS, IERR)
WRITE(*,*)"Hello from rank:",MYID," of total", &
                            TOTPS, " processes."
CALL MPI_FINALIZE(IERR)
END
```

Example 1 in Fortran 90

```
PROGRAM EXAMPLE01
  USE   BASIC_MPI
  CALL  INITIALIZE_MPI()
  CALL  DEMO01()
  CALL  MPI_FINALIZE(IERR)
  STOP
END PROGRAM EXAMPLE01
```

Later, we will work here

Click for f01.f90

MPI Example 1 in C

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

int main(int argc, char* argv[])
{  int myid, totps;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
   MPI_Comm_size(MPI_COMM_WORLD, &totps);
   printf (“Hi from rank: %d of %d processes.\n",
                                 myid, totps);
   MPI_Finalize();
}
```

Restructured MPI Example 1 in C

```
void Demo01();

int main(int argc, char*argv[])
{
    initialmpi(&argc, &argv);
    Demo01();
    MPI_Finalize();
}
```

Later, we will work here

Click for c01n.c

MPI Example 1 in C++

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

int main()
{ MPI::Intracomm commall = MPI::COMM_WORLD;
  MPI::Init();
  int myid  = commall.Get_rank();
  int totps = commall.Get_size();
  printf (“Hi from rank: %d of %d processes.\n",
                                   myid, totps);
  MPI::Finalize();
}
```

Restructured MPI Example 1 in C++

```
void Demo01();
int main()
{
  initializempi();
  Demo01();
  MPI::Finalize();
}
```

Later, we will work here

Click for cpp01n.cpp

MPI Example 2

$$s = \sum_{i=0}^{m} \sqrt{i}$$

$$= \sqrt{0} + \sqrt{1} + \sqrt{2} + \cdots + \sqrt{m}$$

These square root computational sub-tasks will be distributed among all processes.

MPI Example 2

```
PROGRAM EXAMPLE02
   USE    BASIC_MPI
   CALL   INITIALIZE_MPI()
   CALL   DEMO02()
   CALL   MPI_FINALIZE(IERR)
   STOP
END PROGRAM EXAMPLE02
```

Click for f02.f90   c02.c   cpp02.cpp

Running Example 2

- $ mpif90 f02.f90
- $ mpirun –np 3 ./a.out

```
How many terms?
24
RANK: 0  MYS= 28.242821379338707   M: 24
RANK: 2  MYS= 26.928063945678374   M: 24
RANK: 1  MYS= 25.462894950061063   M: 24
Total sum:             80.63378027507815
```

Lab work # 2

- Go to your account in CAC
- `cd       mpi`
- `cd       F90       (C,   CPP)`
- `cd       f02       (c02, cpp02)`
- `cat      f02.f90   (c02.c, cpp02.cpp)`
- `mpif90   f02.f90   (for FORTRAN) or`
- `mpicc    c02.c     (for c)`
- `mpicxx   cpp02.cpp (for C++)`
- `mpirun   -np 3   ./a.out`
- `time mpirun  -np 3 ./a.out`
- `echo 567`
- `echo 24 | time mpirun  -np 3 ./a.out`
- `echo 2000000000 | time mpirun  -np 1 ./a.out`
- `echo 2000000000 | time mpirun  -np 4 ./a.out`

Example 2 Shows

- Processes can communicate via MPI routines;
- The work load can be distributed among processes (by using rank and size numbers);
- The final results can be collected from the processes via MPI routines;
- MPI routines MPI_BCAST & MPI_REDUCE are powerful ones for communications.

The fifth basic feature of MPI

A usual code can be parallelized.

Parallelizability

- For a given computational task split into smaller ones, only if there is no data dependency among the sub-tasks, as in Example 2, the sub-tasks can be completed in parallel.
- Data dependency makes it impossible.
- A non-parallelizable example is solving an equation *iteratively*. Iteration steps cannot be parallelized due to data dependency. However it may still be possible to parallelize each step *internally*.
- In some seeming non-parallelizable cases, new parallel algorithm are possible. These are real challenges.
- Parallel libraries for many typical mathematical processing are available, then should be used.

Speedup and Scaling

- Speedup is the ratio between serial and parallel execution times:

$$S = T_1 / T_p$$

- If the speedup is equal to the number of processors in the parallel case, the program is said to *scale linearly*.
- *In most (but not* all) cases, the speedup will be *smaller* then the number of processors (sub-linear scaling).

Amdahl's Law

Amdahl's Law: as the speedup

$$S_P = \frac{T_{non-par} + T_{par}}{T_{non-par} + T_{par} / P} \leq \frac{1}{F},$$

even with an infinite number of processors, the speedup cannot exceed the above limit*, where F is the non-parallelized fraction.*

Worse for Speedup

- In shared memory parallelism, the more threads used, the more chance for memory conflicts.

- In MPI, the more processes employed, the more significant time for communication (overhead). Beyond a certain number of processors, performance becomes worse.

A brief History

- Standardization started in 1992 on a workshop on message passing in distributed-memory systems.
- A draft version was presented in late 1993 on a super-computing conference.
- Version 1.0 was released in the summer of 1994.
- *Version 2.0 was released in June 1997.*
- *Version 3.1 was released in June 2015.*

Why MPI?

- *Portability:* MPI runs on almost any hardware and OS. There are public-domain versions of it (MPICH, OPENMPI) available for any machine.
- Many parallel libraries in MPI developed already.
- *Ease of Use*: The MPI-1 standard includes about 120 functions, but with about 15 of them, well-working programs can be produced. Usually only private data are used and communications are explicitly managed.
- *Compatibility:* works with C and F77, and by extension with C++ and F90. Usage does not deviate too much from older systems, such as PVM.

Outlines

- Introduction
- MPI basics
  - Programming environments
  - MPI predefined data types
  - Communications
  - User defined data types
  - Runtime environments
  - Some remarks
- Array distribution
- Sub-task distribution
- CAC bonus libraries
- References

Outlines

- Introduction
- MPI basics
  - Programming environments
  - MPI predefined data types
  - Communications
  - User defined data types
  - Runtime environments
  - Some remarks
- Array distribution
- Sub-task distribution
- CAC bonus libraries
- References

MPI Header Files

USE MPI	Fortran
#include <mpi.h>	C/C++

Naming Conventions

In FORTRAN and C: MPI_*
In C++: MPI::*

Users are suggested not to use this form of names to avoid conflicts.

MPI_INIT

MPI_INIT(IERR)
int MPI_Init(int *argc, char ***argv)
void MPI::Init(int& argc, char**& argv)

Initializes MPI. Must be called *once, and only once* before any other MPI routine is called. IERR or the return value is an integer error code. NULL is a valid argument for *argc* and *argv*. In C++, the function can be called with no argument.

MPI_FINALIZE

MPI_FINALIZE(IERR)
int MPI_Finalize(void)
void MPI::Finalize()

Finalizes (closes) MPI. Must be called *once and only once* after the last MPI call. IERR or the return value is an integer error code. In C++ the function is called without arguments.

Communicator

A *communicator* is a group of processes that share a common communication system, so the processes inside can communicate.

Communicators must be specified in all MPI communications. Here communicators means intracommunicators. We will not talk about intercommunicators.

Communicator

A *communicator* can be split into smaller mutual-exclusive ones. A process may belong to many communicators simultaneously. Rank numbers (unique integers) are *communicator* specific, and always run from 0 contiguously in the positive direction inside a given *communicator* .

Communicator

The default communicator MPI_COMM_WORLD, includes *all* processes initiated. Usually it is enough for most communications.

MPI_COMM_SIZE

MPI_COMM_SIZE(COMM, ISIZE, IERR)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI::Comm::Get_size() const

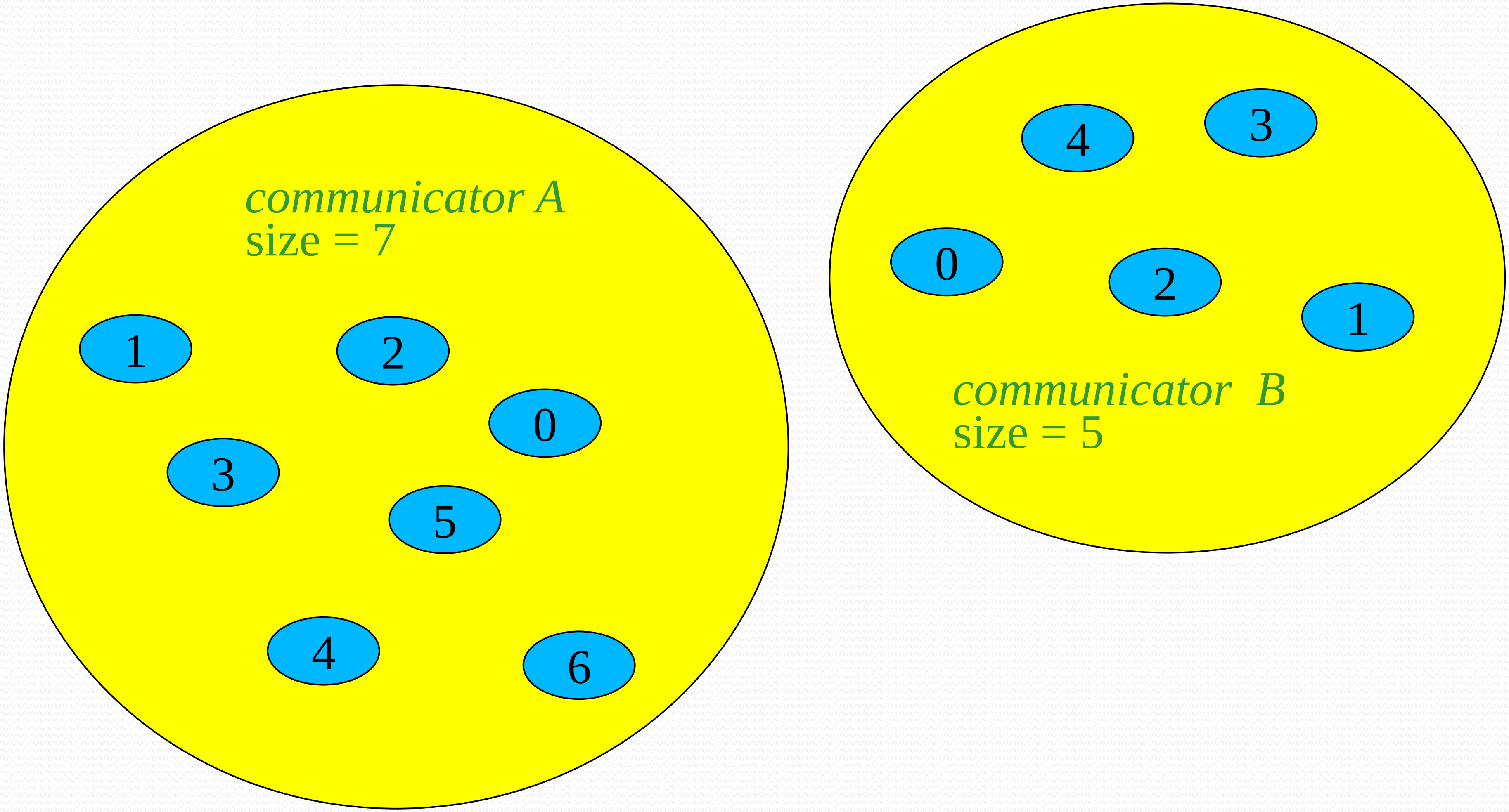
Returns the size of a communicator COMM as an integer (ISIZE, size, return value). *This routine is used to determine the number of available processes in a communicator.* Returns an error code (IERR, return value).

MPI_COMM_RANK

MPI_COMM_RANK(COMM, IRANK, IERR)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI::Comm::Get_rank() const

Returns the rank (internal number) as IRANK, rank or return value of the current process. *It is used to identify the process that calls it*. The rank ranges from *0* to *N-1* if *N* is the number of processes. COMM or comm denotes the communicator, and IERR is the usual integer error code.

Size and Rank


communicator A
size = 7

communicator B
size = 5

MPI_COMM_SPLIT

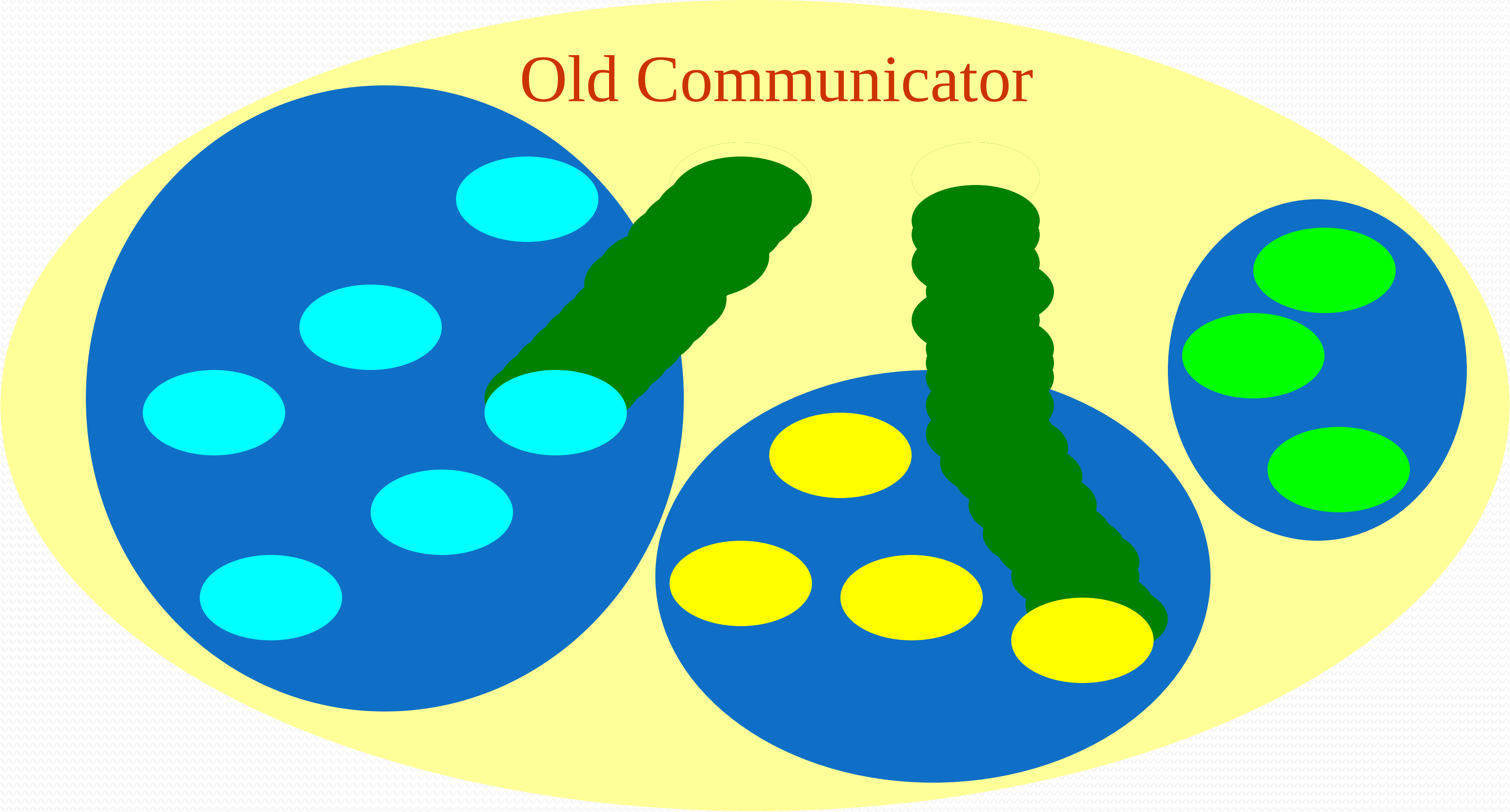
MPI_COMM_SPLIT(COMM,COLOR,KEY,NEWCOMM,IERR)

int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

MPI::Intracomm MPI::Intracomm::Split(int color, int key) const

This routine splits a communicator COMM (comm) into mutually exclusive communicators NEWCOMM (newcomm). Processes that have the same integer COLOR (color) will belong to the same new communicator. The integers KEY (key) are used to determine the order of ranks inside each new communicator.

MPI_COMM_SPLIT



Outlines

- Introduction
- MPI basics
    - Programming environments
    - MPI predefined data types
    - Communications
    - User defined data types
    - Runtime environments
    - Some remarks
- Array distribution
- Sub-task distribution
- CAC bonus libraries
- References

MPI Predefined Data Types

MPI provides its own data types. Most of them are compatible with Fortran, C, and C++ data types. Others provides more flexibility. For any data communication, data types must be specified in the form of MPI data types.

MPI Predefined Data Types for FORTRAN

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	--

Examples of MPI Predefined Data Types for C

MPI_CHAR	signed char
MPI_SIGNED_CHAR	signed char
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t (MPI-2)
MPI_BYTE	--
MPI_PACKED	--

Examples of MPI Predefined Data Types for C++

MPI::CHAR	signed char
MPI::SIGNED_CHAR	signed char
MPI::SHORT	signed short
MPI::INT	signed int
MPI::LONG	signed long
MPI::UNSIGNED_CHAR	unsigned char
MPI::UNSIGNED_SHORT	unsigned short
MPI::UNSIGNED	unsigned int
MPI::UNSIGNED_LONG	unsigned long
MPI::FLOAT	float
MPI::DOUBLE	double
MPI::LONG_DOUBLE	long double
MPI::COMPLEX	complex<float>
MPI::DOUBLE_COMPLEX	complex<double>
MPI::LONG_DOUBLE_COMPLEX	complex<long double>

Other MPI Predefined Data Types for C++

MPI::WCHAR	wchar_t
MPI::BOOL	bool
MPI::INTEGER	(FORTRAN)
MPI::REAL	(FORTRAN)
MPI::DOUBLE_PRECISION	(FORTRAN)
MPI::LOGICAL	(FORTRAN)
MPI::CHARACTER	(FORTRAN)
MPI::F_COMPLEX	(FORTRAN)
MPI::F_DOUBLE_COMPLEX	(FORTRAN)
MPI::BYTE	--
MPI::PACKED	--

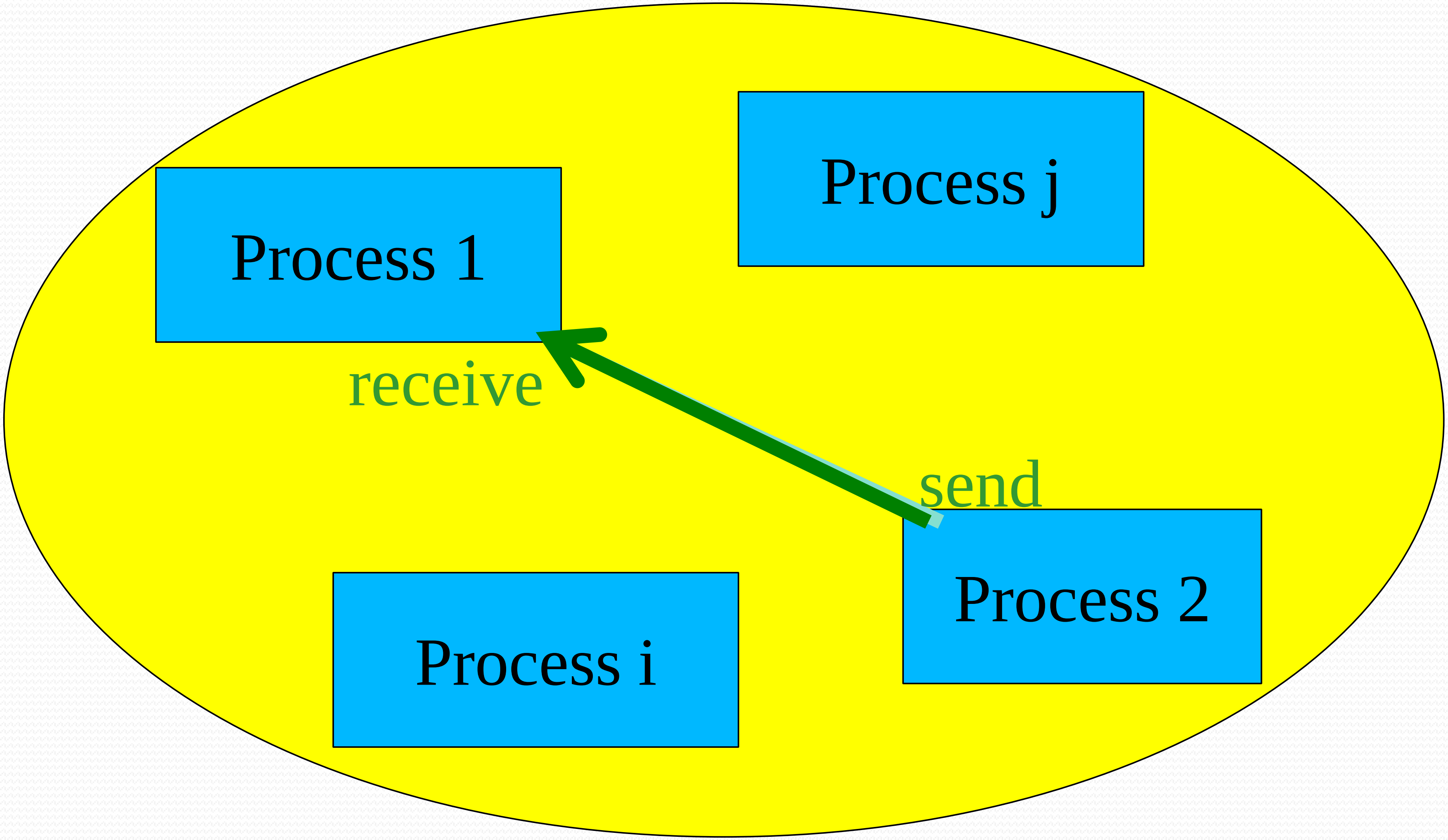
Outlines

- Introduction
- MPI basics
  - Programming environments
  - MPI predefined data types
  - Communications
  - User defined data types
  - Runtime environments
  - Some remarks
- Array distribution
- Sub-task distribution
- CAC bonus libraries
- References

Point-To-Point Communication

Point-To-Point Communication, the basic form of communication, is done between two processes. One *SEND*s data and the other *RECEIVE*s the data. The *SEND*ing needs to know the target (process) to send the data, the *RECEIVE*ing may expect a fixed source (process) or be open to any source for data coming from.

Point-To-Point Communication



Send and Receive Buffers

Variables/arrays to be sent or to be used to receive data in communications are called *send or receive buffers*. They can be any defined data types.

Blocking/Non-blocking Communications

Blocking means that a call to a communication routine *returns only when it is safe to use/re-use the buffer*.

Non-blocking means that the *communication operation has only be initiated when the call returns, not guaranteed finished*. Only when they are confirmed finished by calling checking MPI routines, *it is safe to use/re-use the buffer*. Then so-called *Request type objects* are used to label individual non-blocking communications for this purpose.

MPI_SEND (the generic name)

MPI_SEND(BUF, ICOUNT, TYPE, IDEST, ITAG, COMM, IERR)
int MPI_Send(void* buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)
void MPI::Comm::Send(const void* buf, int count, const MPI::Datatype& type, int dest, int tag) const

Sends ICOUNT (count) successive data entries of type TYPE (type) in buffer array BUF (buf) from the calling process to the process with rank IDEST (dest). The integer ITAG (tag) is used to identify this message. Valid values for tags are 0, 1, 2, …, UB>=32767. COMM (comm) is the communicator, IERROR the usual error code. This communication is blocking.

MPI_RECV

MPI_RECV(BUF,ICOUNT,TYPE,ISOURCE,ITAG,COMM,STATUS,IERR)
int MPI_Recv(void* buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status *status)
void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype& type, int source, int tag) const

Receives a message identified with ITAG (tag), ISOURCE (source), and COMM (comm). The received data are placed into buffer array BUF (buf) of ICOUNT (count) successive entries of type TYPE (type). STATUS (status) is an integer array (of MPI_STATUS_SIZE elements in FORTRAN) with status information about the message received (e.g. its actual length and source). The communication is blocking. It is often used together with MPI_SEND for communications.

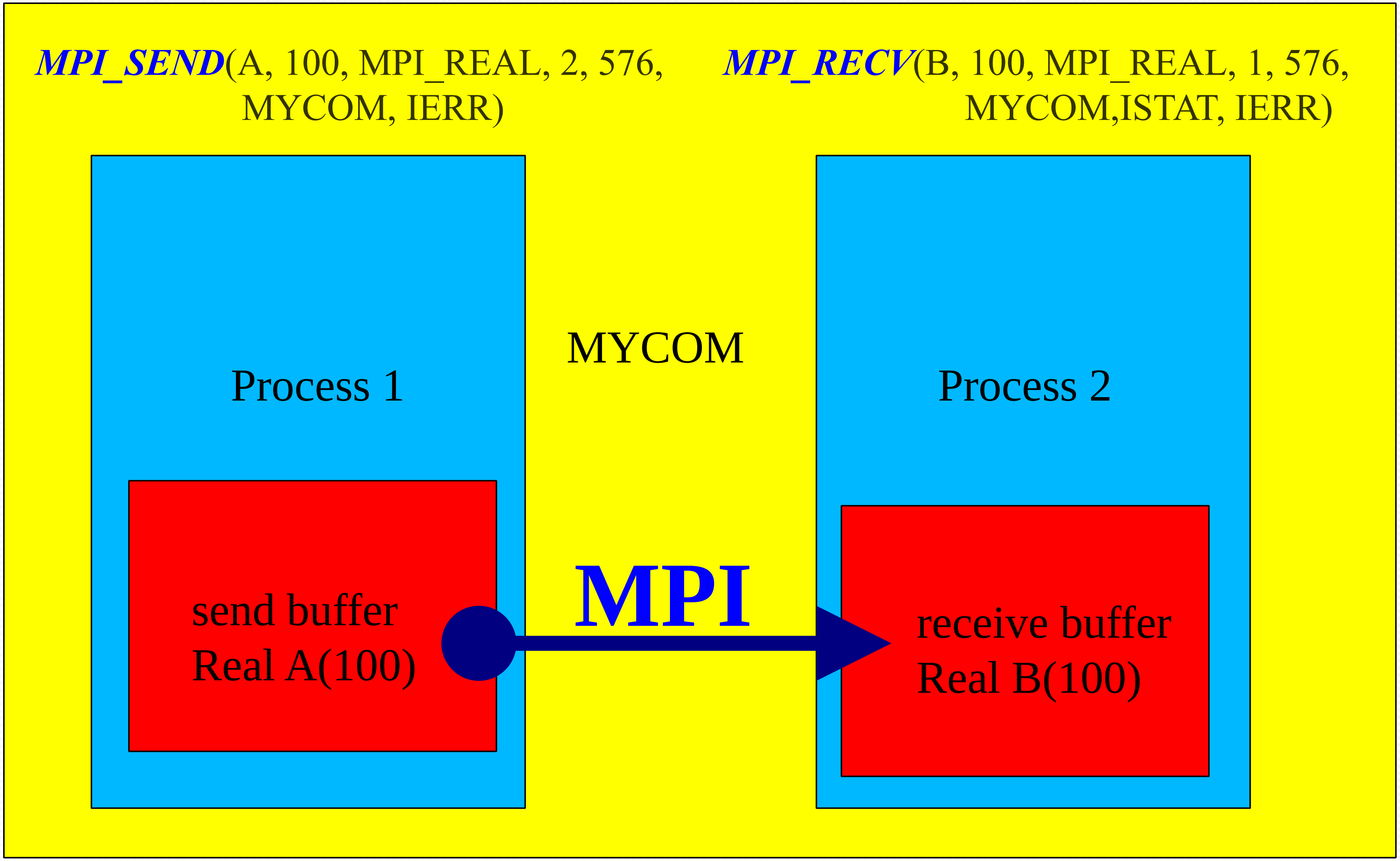
About MPI_RECV

- Note that MPI_RECV can accept messages from an *unspecified* source. For this, the wildcard value MPI_ANY_SOURCE (MPI::ANY_SOURCE in C++) is provided.

- If a distinction by tag is not required, the constant MPI_ANY_TAG (MPI::ANY_TAG in C++) can be used.

- Unspecified sources and tags can only be used by receives, not by sends.

MPI_Send/MPI_Recv



MPI_SEND(A, 100, MPI_REAL, 2, 576, MYCOM, IERR)

MPI_RECV(B, 100, MPI_REAL, 1, 576, MYCOM,ISTAT, IERR)

MPI_ISEND

MPI_ISEND(BUF, ICOUNT, TYPE, IDEST, ITAG, COMM, IREQ,IERR)
int MPI_ISend(void* buf, int count, MPI_Datatype type, int dest, int tag,
MPI_Comm comm, MPI_Request *req)
MPI::Request MPI::Comm::ISend(const void* buf, int count,
const MPI::Datatype& type, int dest, int tag) const

Nearly the same as MPI_SEND, but *non-blocking*. Calls to MPI_WAIT or MPI_TEST are usually needed for later checks if the communication is completed. For this purpose, the request integer IREQ, or object req is used.

MPI_IRECV

MPI_IRECV(BUF,ICOUNT,TYPE,ISOURCE,ITAG,COMM,IREQ,IERR)
int MPI_IRecv(void* buf, int count, MPI_Datatype type, int source, int tag,
MPI_Comm comm, MPI_Request request)
MPI::Request MPI::Comm::IRecv(void* buf, int count,
const MPI::Datatype& type, int source, int tag) const

Nearly the same as MPI_RECV, *but non-blocking*. MPI_WAIT or MPI_TEST is usually needed to check for completion. For this purpose the integer IREQ or the object req is used.

MPI_WAIT

MPI_WAIT(IREQ, ISTAT, IERR)
int MPI_Wait(MPI_Request *req, MPI_Status status)
void MPI::Request::Wait(MPI::Status& status)

Returns only when a non-blocking communication labelled by the request IREQ or req is completed. The request is usually returned by MPI_ISEND or MPI_IRECV.

MPI_TEST

MPI_TEST(IREQ, FLAG, ISTAT, IERR)
int MPI_Test(MPI_Request *req, int *flag, MPI_Status status)
bool MPI::Request::Test(MPI::Status& status)

Returns the logical FLAG (flag) as true if the non-blocking communication identified by IREQ (req) is completed, and as false otherwise. Request IREQ (req) is usually returned from MPI_ISEND or MPI_IRECV.

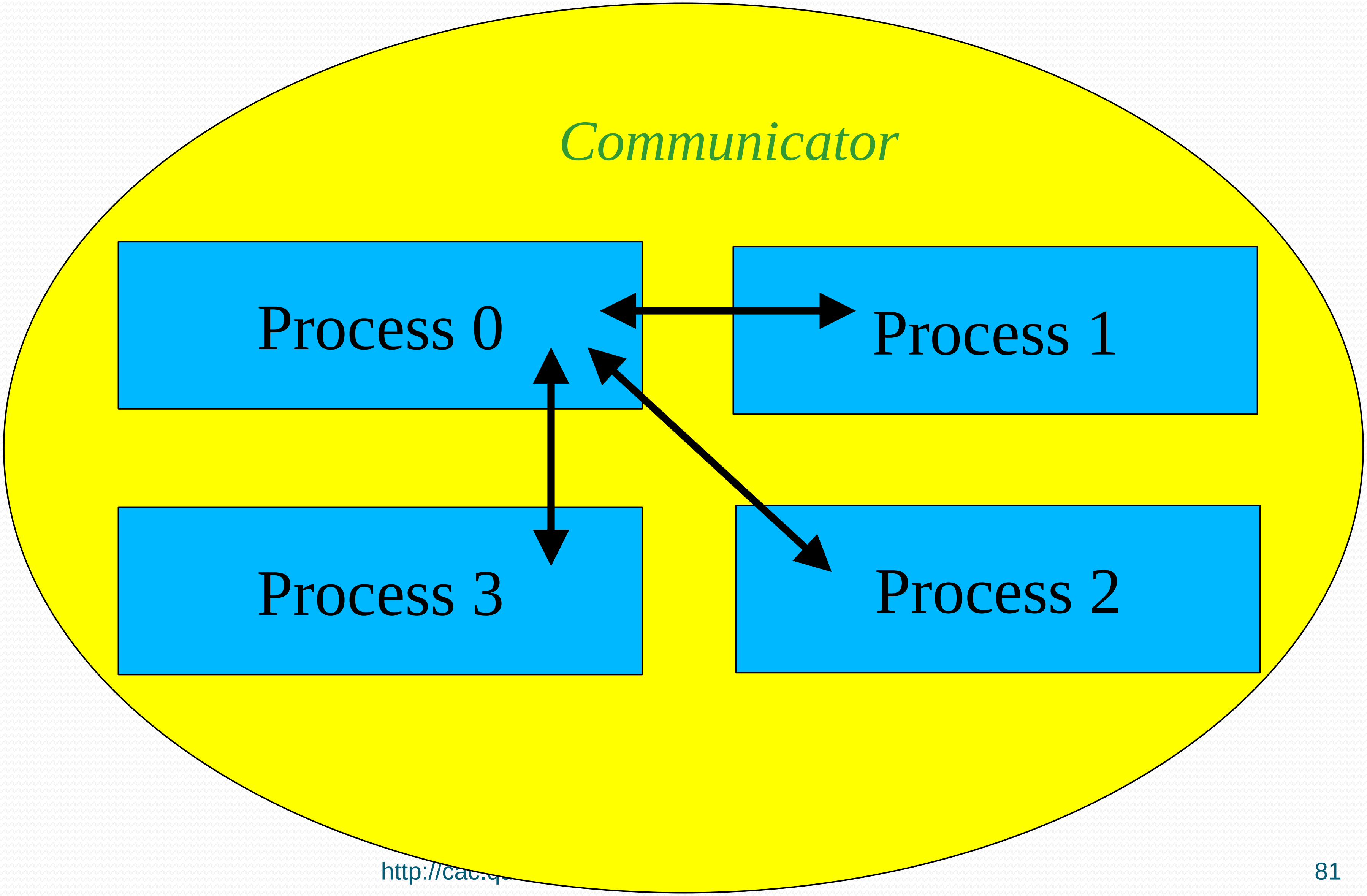
Collective Communications

Some communications and other operations involve *all processes* in a given communicator and are thus called *collective.* Examples are *Broadcast, Reduction* and *Barrier. Collective Communications* are often more efficient and easier to program than the *point-to-point* communications.

Collective communications are always blocking ones and should be called by every process in the given communicator.

The following routines are collective.

Collective Communication



Communicator

Process 0

Process 1

Process 3

Process 2

MPI_BARRIER

MPI_BARRIER(COMM, IERR)
int MPI_Barrier(MPI_Comm comm)
void MPI::Comm::Barrier() const=0

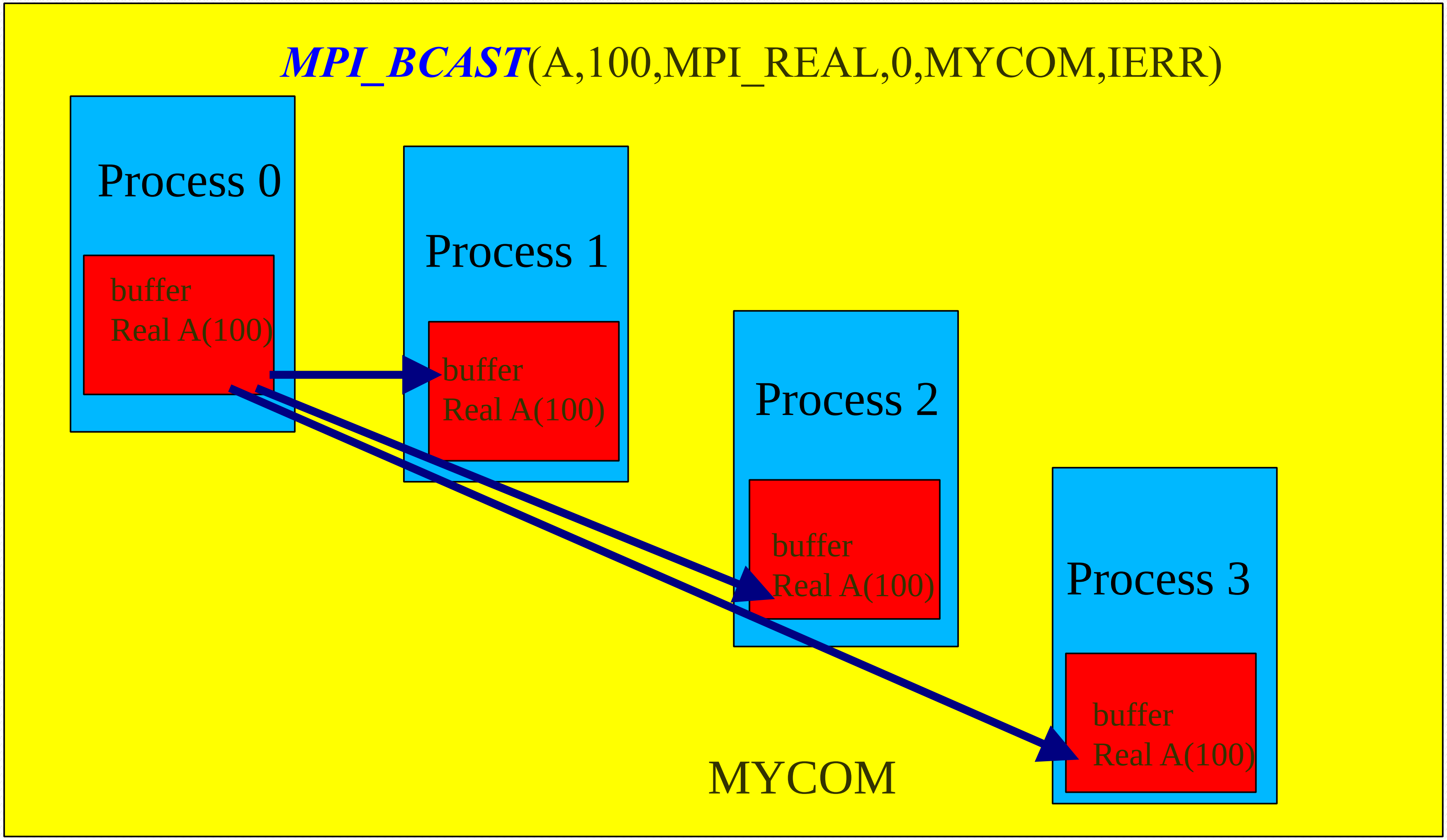
Blocks the process until all members of the communicator COMM or comm have reached here. *This routine is used to synchronize all processes in a communicator.*

MPI_BCAST

MPI_BCAST(BUF,ICOUNT,TYPE,IROOT,COMM,IERR)
int MPI_Bcast(void* buf, int count, MPI_Datatype type,
int root, MPI_Comm comm)
void MPI::Comm::Bcast(void* buf, int count,
const MPI::Datatype& type, int root) const=0

"Broadcasts" BUF (buf) of ICOUNT (count) values of type TYPE (type) from the process with rank IROOT (root) to all other processes. MPI_BCAST *is used to disseminate information among all processes in the communicator.*

MPI_BCAST



MPI_BCAST(A,100,MPI_REAL,0,MYCOM,IERR)

MPI_REDUCE

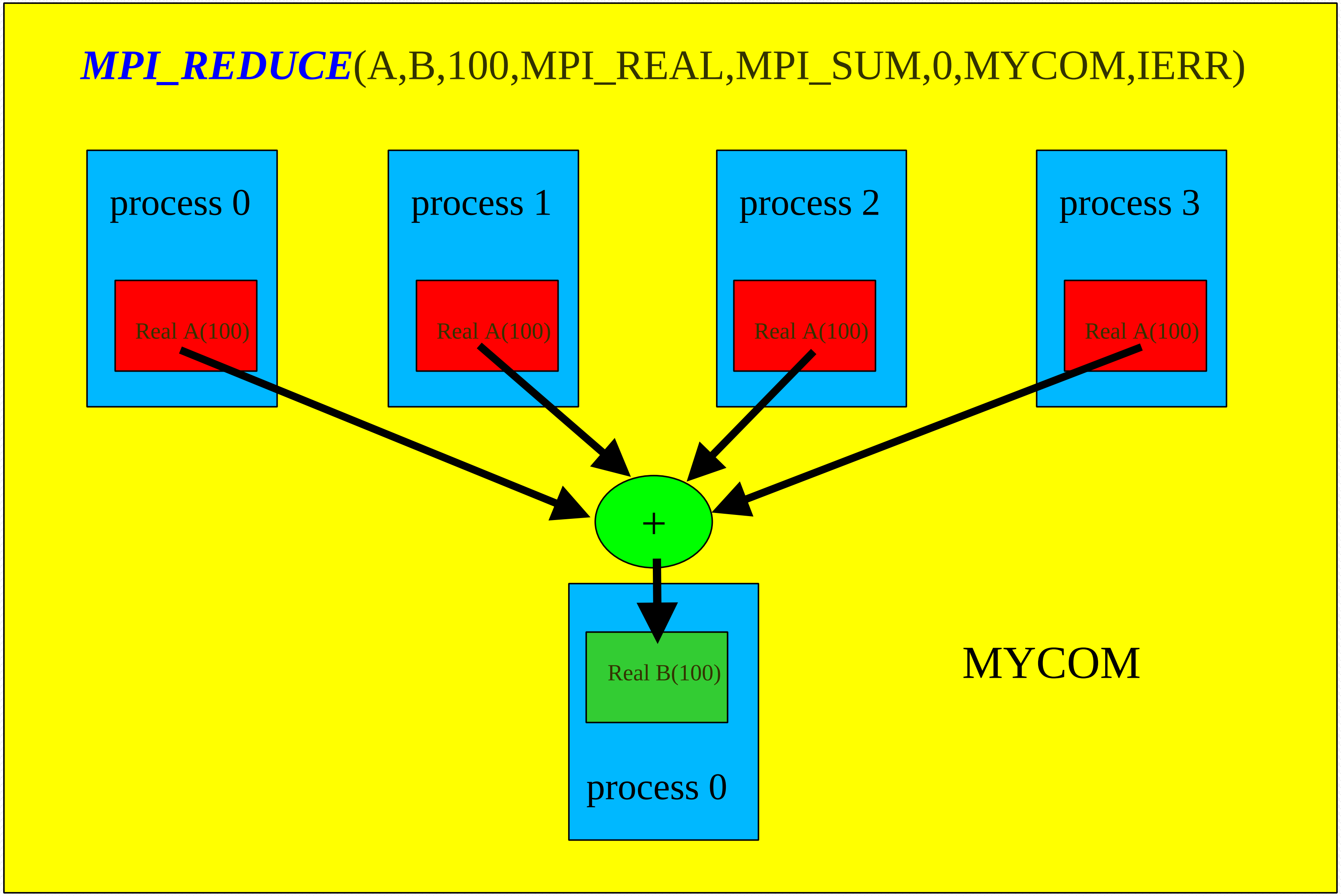
MPI_REDUCE(SBUF,RBUF,ICOUNT,TYPE,OP,IROOT,COMM,IERR)

int MPI_Reduce(void* sbuf, void* rbuf, int count, MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm)

void MPI::Comm::Reduce(const void* sbuf, void* rbuf, int count, const MPI::Datatype& type, const MPI::Op& op, int root) const=0

MPI_REDUCE takes ICOUNT (count) data of type TYPE (type) that are stored in SBUF (sbuf) on all processes in COMM (comm) and reduces all the corresponding elements via operation OP (op), then stores the result into the corresponding element of RBUF (rbuf) on the process with rank IROOT (root). Possible operations are MPI_MAX (maximum), MPI_MIN (minimum), MPI_SUM (sum), MPI_PROD (product), etc.

MPI_REDUCE



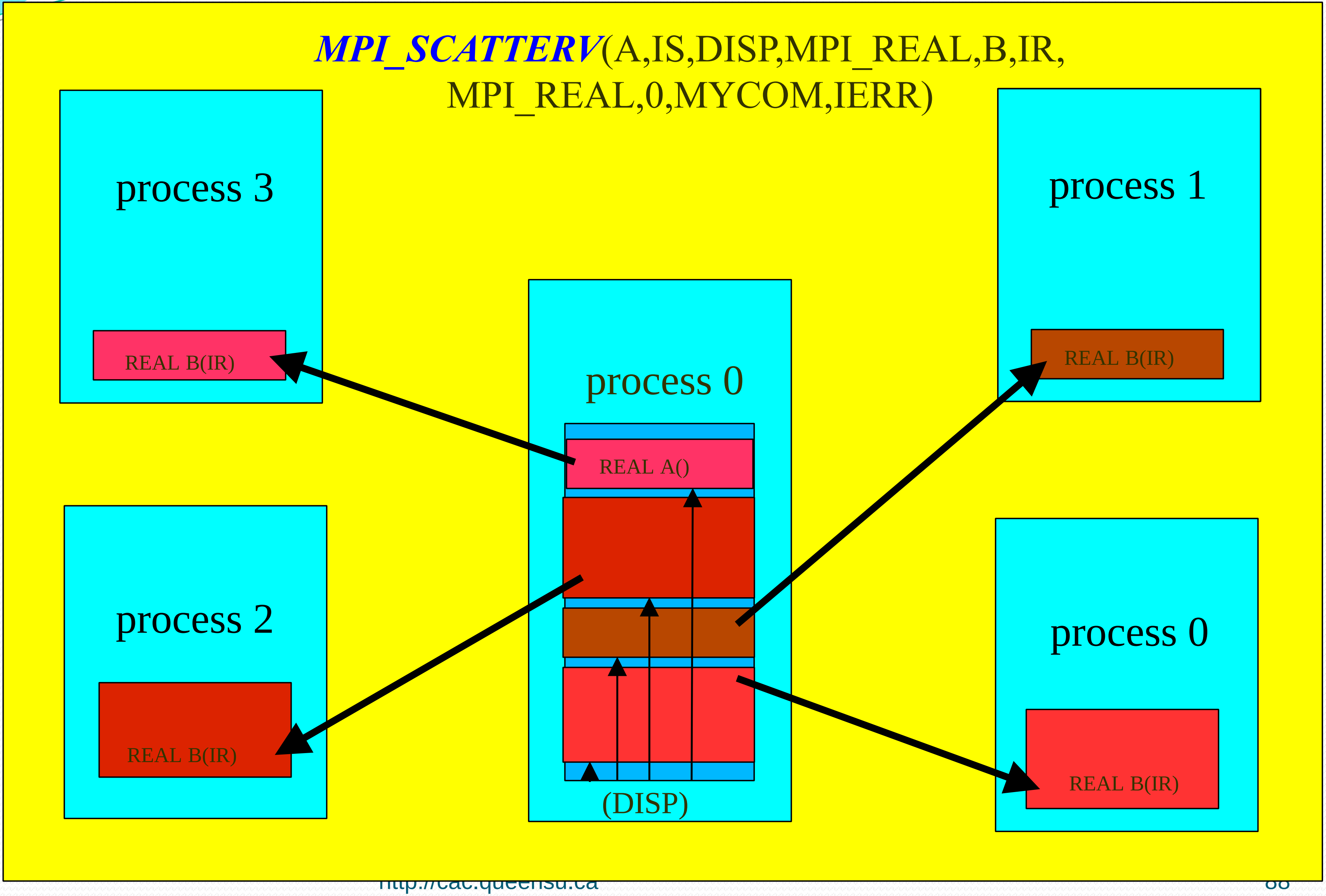
MPI_REDUCE(A,B,100,MPI_REAL,MPI_SUM,0,MYCOM,IERR)

MPI_SCATTERV

MPI_SCATTERV(SBUF,IS,DISP,TS,RBUF,IR,TR, IROOT,COMM,IERR)
MPI_Scatterv(void* sbuf, int *is, int *disp, MPI_Datatype ts, void* rbuf, int ir, MPI_Datatpe tr, int root, MPI_Comm comm)
void MPI::Comm::Scatterv(const void* sbuf, const int is[], const int disp[], const MPI::Datatype& ts, void* rbuf, int ir, const MPI::Datatype& tr, int root) const=0

To scatter SBUF (sbuf) of type TS (ts) in rank IROOT (root) to all processes in the COMM (comm).The integer arrays DISP (disp) and IS (is) are used to specify from which entry and the total number of entries to be scattered to each process, in the order of ranks. For a specific calling process, the received data will be placed into RBUF (rbuf) of integer IR (ir) entries of type TR (tr) .

MPI SCATTERV



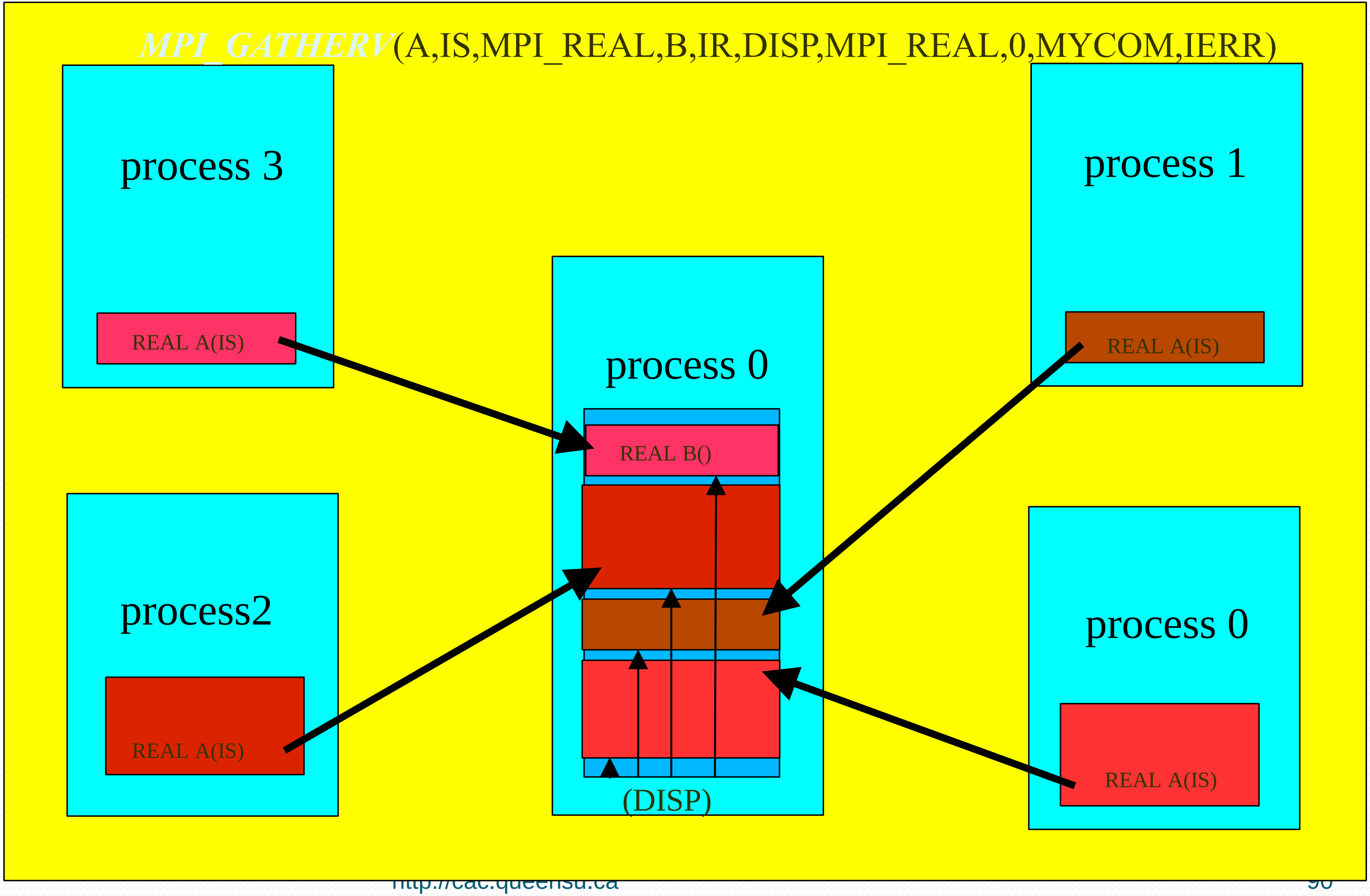
MPI_SCATTERV(A,IS,DISP,MPI_REAL,B,IR,
MPI_REAL,0,MYCOM,IERR)

MPI_GATHERV

```
MPI_GATHERV(SBUF,IS,TS,RBUF,IR,DISP,TR,IROOT,COMM,IERR)
MPI_Gatherv(void* sbuf, int is, MPI_Datatype ts, void* rbuf, int *ir,
int *disp, MPI_Datatpe tr, int root, MPI_Comm comm)
void MPI::Comm::Gatherv(const void* sbuf, int is, const MPI::Datatype& ts,
void* rbuf, const int ir[], const int disp[], const MPI::Datatype& tr, int root)
const=0
```

To gather SBUF (sbuf) of integer IS (is) entries of type TS (ts) from a specific calling process. These data in all processes of the COMM (comm) will be gathered and placed into RBUF (rbuf) of type TR (tr) in rank IROOT (root). The integer arrays DISP (disp) and IR (ir) are used to specify from which entry and the total number of entries to be placed into RBUF (rbuf), in the order of ranks for elements.

MPI_GATHERV



MPI_GATHERV(A,IS,MPI_REAL,B,IR,DISP,MPI_REAL,0,MYCOM,IERR)

Outlines

- Introduction
- MPI basics
    - Programming environments
    - MPI predefined data types
    - Communications
    - User defined data types
    - Runtime environments
    - Some remarks
- Array distribution
- Sub-task distribution
- CAC bonus libraries
- References

User-Defined Data Types

- Users often define new data types based on predefined ones in their code (Fortran 90 and C/C++), and like to transfer them with MPI.

- However MPI never reads the code, then knows nothing about such User-Defined Data Types (UDDT).

- Users should inform MPI the details by redefining them through calling MPI routines. Then they are called MPI UDDT or still UDDT for short.

MPI UDDT

As a matter of fact, MPI UDDTs are not simply a redefinition of the regular UDDTs, but much wider/deeper, then much more powerful.

MPI UDDTs can be used to send or receive any related and completely un-related data all together in the whole local memory space.

This means data defined as of an MPI UDDT but never defined in any regular UDDT in the normal code can also be transferred together.

MPI UDDT

- Four steps: to define, to commit, to use the same way as predefined data types, and to delete after used.

- Committed MPI UDDTs can be used as predefined types in further MPI UDDT definitions.

MPI_GET_ADDRESS

MPI_GET_ADDRESS(DATAPOINT,ADDRESS,IERROR)
int MPI_Get_address(void *datapoint, MPI_Aint *address)
MPI::Aint MPI::Get_address (void* datapoint)

Finds the absolute byte ADDRESS of a "memory location", i.e., a DATAPOINT. This call is commonly used to compute the true offset of a data point inside a structure, e.g. to load the IDISP array in a MPI_TYPE_CREATE_STRUCT call.

MPI_TYPE_CREATE_RESIZED

MPI_TYPE_CREATE_RESIZED(TOLD, LOW, EXT, TNEW, IERROR)

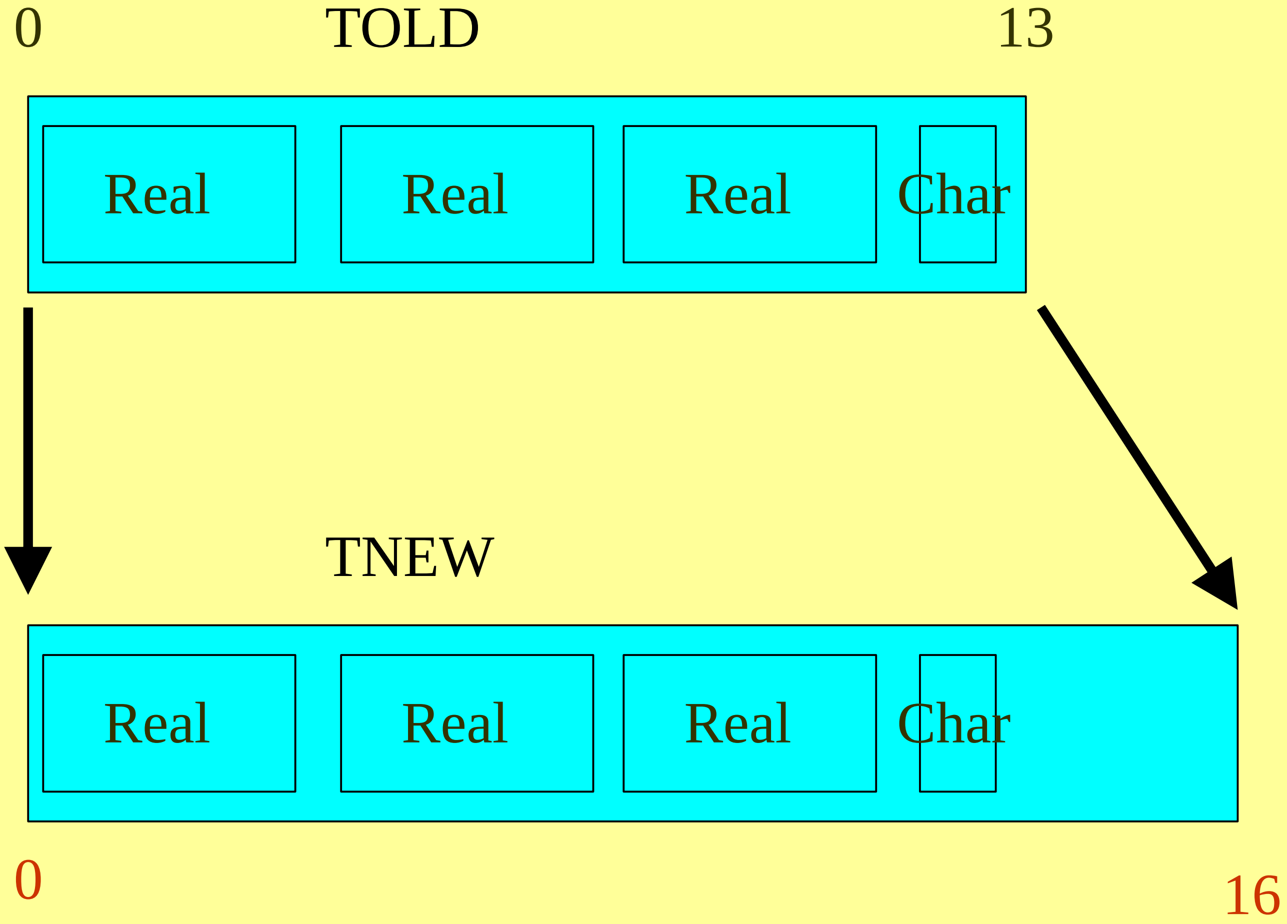
MPI_Type_create_resized(MPI_Datatype told, MPI_Aint low, MPI_Aint ext, MPI_Datatype *tnew)

MPI::Datatype MPI::Datatype::Resized (const MPI::Aint low, const MPI::Aint ext) const

Creates a new data type TNEW identical to a pre-existing one TOLD but with reset boundaries. The lower boundary is set to LOW and the upper boundary is set to LOW+EXT. Commonly used to adapt an MPI_DATATYPE in size to an existing datatype in case of padding.

MPI_TYPE_CREATE_RESIZED

MPI_TYPE_CREATE_RESIZED(TOLD,0,16,TNEW,IERROR)



MPI_TYPE_CREATE_STRUCT

MPI_TYPE_CREATE_STRUCT(ICOUNT, LBLOCK, IDISP, TYPES, TNEW, IERROR)

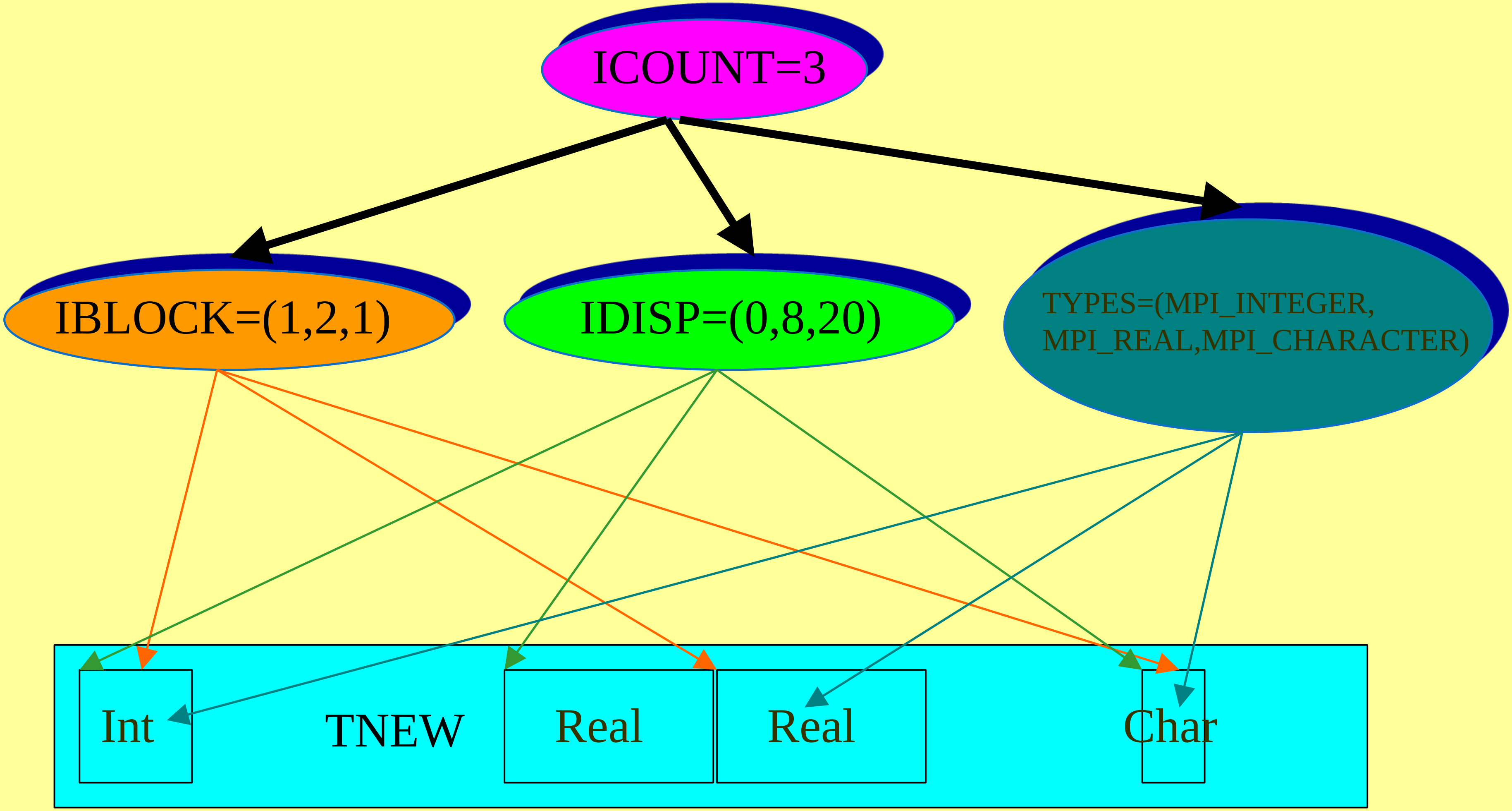
MPI_Type_create_struct(int icount, int *lblock, MPI_Aint *idisp, MPI_Datatype *types, MPI_Datatype *tnew)

static MPI::Datatype MPI::Datatype::Create_struct (int icount, const int lblock[], const MPI::Aint idisp[], const MPI::Datatype types[])

Creates a new data type TNEW by concatenating ICOUNT blocks of changing types specified in array TYPES with lengths specified in array LBLOCK. Among each other, these blocks may not be contiguous in memory. The onsets are specified in array IDISP.

MPI_TYPE_CREATE_STRUCT

MPI_TYPE_CREATE_STRUCT(3,LBLOCK,IDISP,TYPES,TNEW,IERROR)



ICOUNT=3

IBLOCK=(1,2,1)

IDISP=(0,8,20)

TYPES=(MPI_INTEGER,
MPI_REAL,MPI_CHARACTER)

Int

TNEW

Real

Real

Char

MPI_TYPE_COMMIT

MPI_TYPE_COMMIT(TYPE,IERROR)
int MPI_Type_Commit(MPI_Datatype type)
void MPI::Datatype::Commit ()

Commits a new data type TYPE and makes it ready for use. Must be called before first use.

MPI_TYPE_FREE

MPI_TYPE_FREE(TYPE, IERROR)
MPI_Type_free(MPI_Datatype type)
void MPI::Datatype::Free ()

Releases the objects associated with a data type TYPE. Should be called when TYPE is not used anymore. Datatypes that depend on the freed one are not affected.

A Simple Example

- In Fortran
- In C
- In C++

Outlines

- Introduction
- MPI basics
    - Programming environments
    - MPI predefined data types
    - Communications
    - User defined data types
    - Runtime environments
    - Some remarks
- Array distribution
- Sub-task distribution
- CAC bonus libraries
- References

Compiling and execution in our cluster

To compile :

```
mpif90      files.f90
mpicc        files.c
mpicxx      files.cpp
```

To run :

```
mpirun  –np  N  executable
```

where N is the number of processes.

SLURM

In CAC (HPCVL), all production jobs must be submitted to SLURM, then to cluster.

One way is as before: salloc ...

The other way is as:

1, a script file should be edited, e.g. *ajob*

2, submitting it: *sbatch ajob*

3, monitoring: squeue –u THE_USER

4, submitted jobs can be deleted: scancel job#

https://cac.queensu.ca/wiki/index.php/SLURM

Script example for SLURM

```
#!/bin/bash
#SBATCH --job-name=My_MPI_job
#SBATCH --mail-type=ALL
#SBATCH --mail-user=joe.user@email.ca
#SBATCH --output=STD.out
#SBATCH --error=STD.err
#SBATCH --nodes=1
#SBATCH --ntasks=8
#SBATCH --cpus-per-task=1
#SBATCH --time=0-0:30:00
#SBATCH --mem=20GB
mpirun -np $SLURM_NTASKS ./mpi_program
```

Outlines

- Introduction
- MPI basics
    - Programming environments
    - MPI predefined data types
    - Communications
    - User defined data types
    - Runtime environments
    - Some remarks
- Array distribution
- Sub-task distribution
- CAC bonus libraries
- References

Parallel Principles

- Try to parallel heavy computations as much as possible.
- Distribute sub-tasks among processes as evenly as possible, to reduce waiting time.
- Reduce or combine communications as much as possible, as eventually they become the performance bottleneck.
- If possible, repeat some quick calculations across processes to avoid communications for them.
- Parallelize out-most loop rather than inner ones to reduce communications, if nested loops parallelizable.

About MPI I/O

- In MPI-1, each process handles I/O completely separately, therefore, processes will NOT cooperate. Results are unpredictable when multiple processes write into one same file.
- Simple solution: One process does all I/O, all others communicate with it for necessary information (see examples).
- In MPI-2, parallel I/O is available (beyond the scope of this course, and not necessary in most cases).

Steps for parallelizing a serial code

- Make sure the serial code in a reasonable status.
- Introduce MPI into the code (header file, initializing, rank, size, and finalizing).
- Properly handle I/Os (let one process read in all input data, broadcast them immediately, and do all output operations).
- Profile the code to determine which sections should be parallelized.
- Choose parallel method and parallelize the above sections (new algorithm might be needed) .
- Furthermore, distribute big arrays to save memory if possible.
- Repeat the above last three steps till satisfaction in performance and memory requirement.

A simple tip

In order to parallelize the following many nested very limited loops:

```
loop1 from 1 to n1
    loop2 from 1 to n2
        ...
           loopm from 1 to nm
                independent_jobs(loop_indexed)
           end loopm
        ...
    end loop2
end loop1
```

A simple tip

Save loop indexes to array MMM (as an example):

```
count=0
loop1 from 1 to n1
    loop2 from 1 to n2
        …
            loopm from 1 to nm
                count=count+1
                save_all_loop_indexes_to_MMM(count)
            end loopm
        …
    end loop2
end loop1
```

A simple tip

Then the same computation can be done with the following one loop, which should be parallelized more efficiently:

```
loop from 1 to count
        the_independent_jobs(MMM(loop))
end loop
```

Outlines

- Introduction
- MPI basics
  - Programming environments
  - MPI predefined data types
  - Communications
  - User defined data types
  - Runtime environments
  - Some remarks
- Array distribution
- Sub-task distribution
- CAC bonus libraries
- References

Arrays in memory

Memory is the place we place our data

In serial code, we may completely forget any details about how an array is managed in memory.

However, in MPI code, there are a few respects about arrays in memory which we should pay attention to, either for running the code much faster or even for ensuring the code running correctly.

A mathematical array

From now on, let us consider the following mathematical expression of an array of M rows and N columns (M-by-N, with both row and column indexes starting from 1):

$$\mathbf{A} = \begin{pmatrix} A(1,1) & A(1,2) & \cdots & A(1,N) \\ A(2,1) & A(2,2) & \cdots & A(2,N) \\ \cdots & \cdots & \cdots & \\ A(M,1) & A(M,2) & \cdots & A(M,N) \end{pmatrix}$$

where the elements are

A(i,j) with i = 1,2,…, M and j = 1, 2, …, N.

Programming on an array

The array can be stored in any way,

as long as accessed accordingly.

$$\mathbf{A} = \begin{pmatrix} A(1,1) & A(1,2) & \cdots & A(1,N) \\ A(2,1) & A(2,2) & \cdots & A(2,N) \\ \cdots & \cdots & \cdots & \\ A(M,1) & A(M,2) & \cdots & A(M,N) \end{pmatrix}$$

The usual ways are

in FORTRAN

```
REAL*8 :: FA(M,N)
…
FA(I,J)=A(I,J)
```

normal way

in C/C++

```
float ca[M][N];
…
ca[i][j]=A(i+1,j+1)
```

or

```
REAL*8 :: FA(N,M)
…
FA(I,J)=A(J,I)
```

transposed way

```
float ca[N][M];
…
ca[i][j]=A(j+1,i+1)
```

based on further considerations.

Sequence in memory

The next element/data in memory of the element

```
REAL*8 :: FA(M,N)
FA(I,J)
```

```
float ca[M][N];
ca[i][j]
```

is always

```
FA(I+1,J)
```

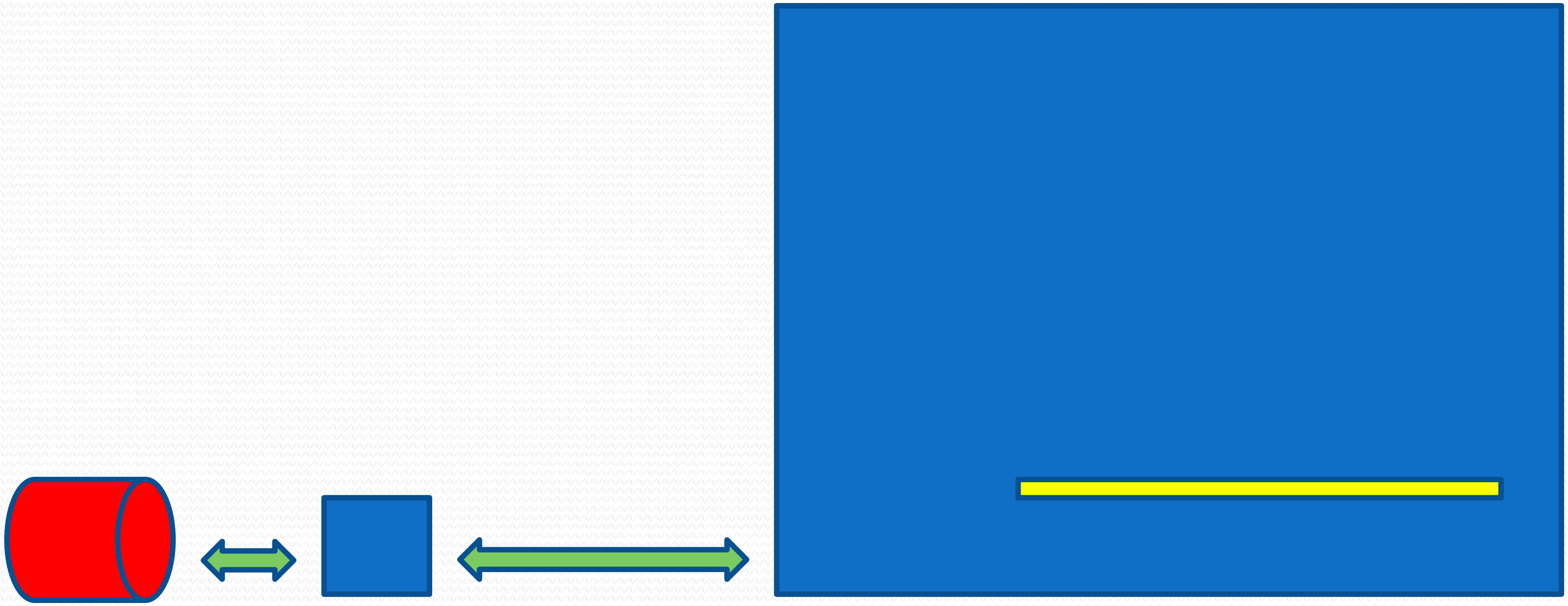
```
ca[i][j+1]
```

in FORTRAN

in C/C++

if existing.

A sketch of a computer structure



CPU

cache, faster
limited in size

main memory (RAM)
huge in size, slow

For a piece of code, accessing elements of an array

in FORTRAN

```
DO I = 1, M
   DO J = 1, N
     FA(I,J) = …
     …
   END DO
END DO
```

in C/C++

```
for(i=0; i<M; i++){
    for(j=0; j<N; j++){
        ca[j][i] = … ;
        …
        }
    }
```

is usually much slower in performance than:

For a piece of code, accessing elements of an array

in FORTRAN

```
DO J = 1, N
   DO I = 1, M
      FA(I,J) = …
      …
   END DO
END DO
```

in C/C++

```
for(j=0; j<N; j++){
      for(i=0; i<M; i++){
            ca[j][i] = … ;
            …
            }
      }
```

when the order of the I and J loops reversed, accessing elements in memory sequence.

The reason is that memory has different levels with different sizes and speeds. The data in consecutive memory will automatically flow together in any case, then more efficient if used in sequence immediately.

To send many-element data with MPI

You inform MPI the first element (e.g. an array element or point), total number of elements to be sent, and the data type.

Then, MPI will get the first element, the next element, the next next element, …, till all the required number of elements in memory based on the length of the data type, then send them.

Then the data to be sent should be prepared in such a sequence in memory.

To send the red elements of the array

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

FORTRAN (transposed)

```
FA(I,J)=A(2,2)
FA(I+1,J)=A(2,3)
FA(I+2,J)=A(2,4)
```

C/C++ (normal)

```
ca[i][j]=A(2,2)
ca[i][j+1]=A(2,3)
ca[i][j+2]=A(2,4)
```

to make sure the data to be sent in sequential memory location and send from (if not using MPI UDDT)

```
FA(I,J)
```

```
ca[i][j]
```

To send the red elements of the array

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

FORTRAN (normal)

```
FA(I,J)=A(1,3)
FA(I+1,J)=A(2,3)
FA(I+2,J)=A(3,3)
```

C/C++ (transposed)

```
ca[i][j]=A(1,3)
ca[i][j+1]=A(2,3)
ca[i][j+2]=A(3,3)
```

to make sure the data to be sent in sequential memory and send from (if not using MPI UDDT)

```
FA(I,J)
```

```
ca[i][j]
```

To choose normal or transposed ways in array coding, we need to consider how they will be transferred in MPI routines. If never being transferred or only broadcast as a whole in MPI, the performance should be considered when accessed by CPUs.

It is quite often that one-dimensional arrays in C/C++ code are dynamically allocated but employed as two-dimensional mathematical arrays. In such a case, we still have the choice of normal and transposed ways to store the two-dimensional array data.

Programming for an array

$$\mathbf{A} = \begin{pmatrix} A(1,1) & A(1,2) & \cdots & A(1,N) \\ A(2,1) & A(2,2) & \cdots & A(2,N) \\ \cdots & \cdots & \cdots & \\ A(M,1) & A(M,2) & \cdots & A(M,N) \end{pmatrix}$$

For M-row by N-column array $\mathbf{A}$ in C/C++

```
float* ca;
ca = (float *) malloc(M*N*sizeof(float));
/* the above in C and the next in C++ */
ca = (float *) new float[M*N];
```

Normal way

```
ca(i*N+j)=A(i+1,j+1)
```

Transposed way

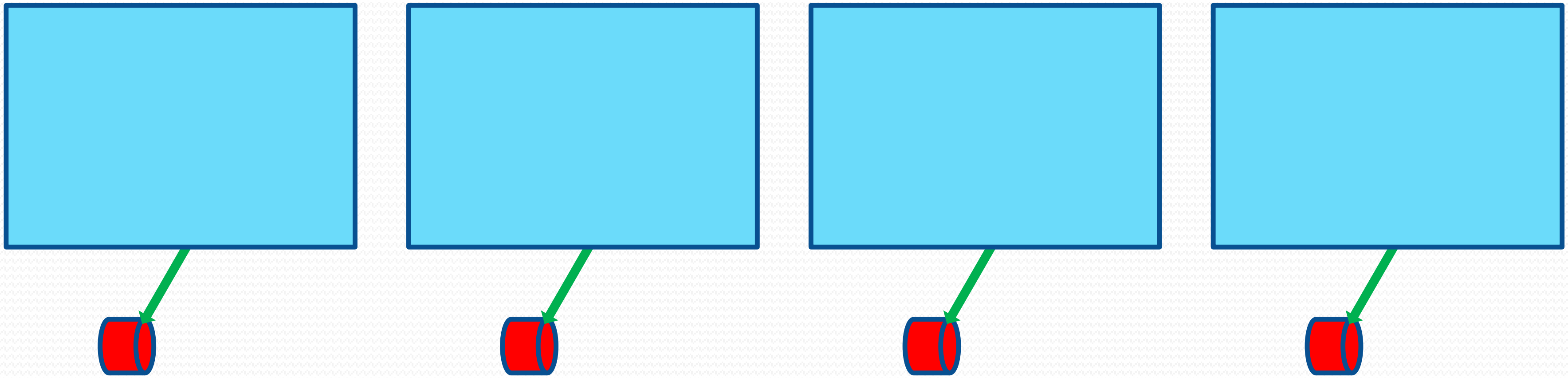
```
ca[i+j*M]=A(i+1,j+1)
```

Memory is distributed across processes in MPI

Under this big background, we further have a choice to duplicate or distribute arrays in MPI code.

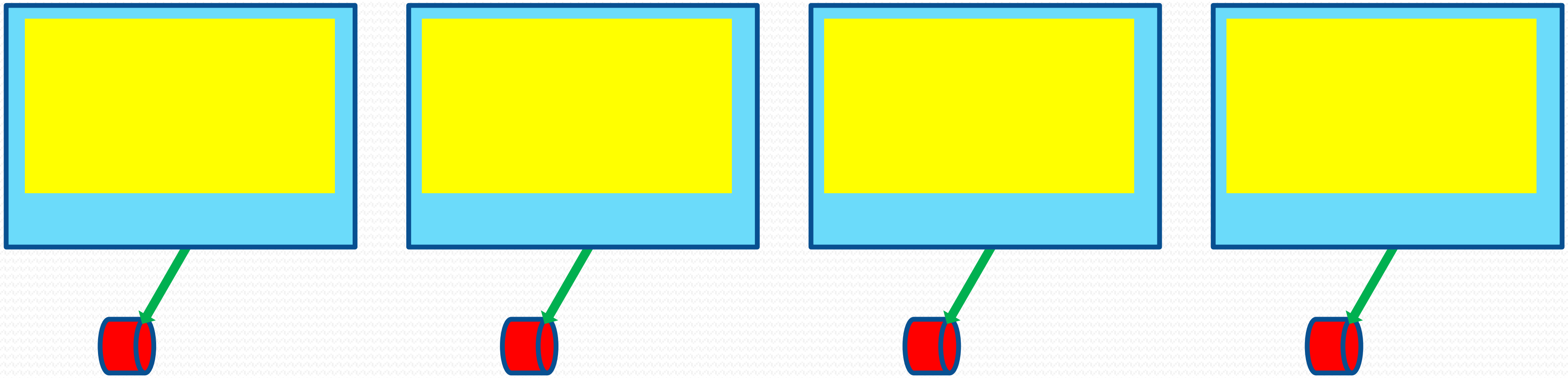
Array duplicated

$$\mathbf{A} = \begin{pmatrix} A(1,1) & A(1,2) & \cdots & A(1,N) \\ A(2,1) & A(2,2) & \cdots & A(2,N) \\ \cdots & \cdots & \cdots & \\ A(M,1) & A(M,2) & \cdots & A(M,N) \end{pmatrix}$$



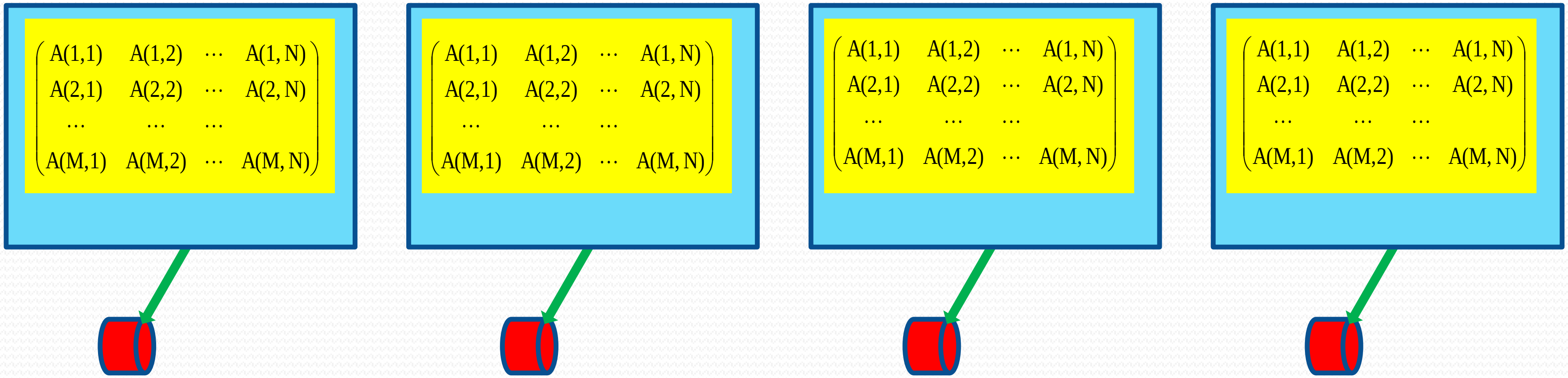
Array duplicated

$$\mathbf{A} = \begin{pmatrix} A(1,1) & A(1,2) & \cdots & A(1,N) \\ A(2,1) & A(2,2) & \cdots & A(2,N) \\ \cdots & \cdots & \cdots & \\ A(M,1) & A(M,2) & \cdots & A(M,N) \end{pmatrix}$$



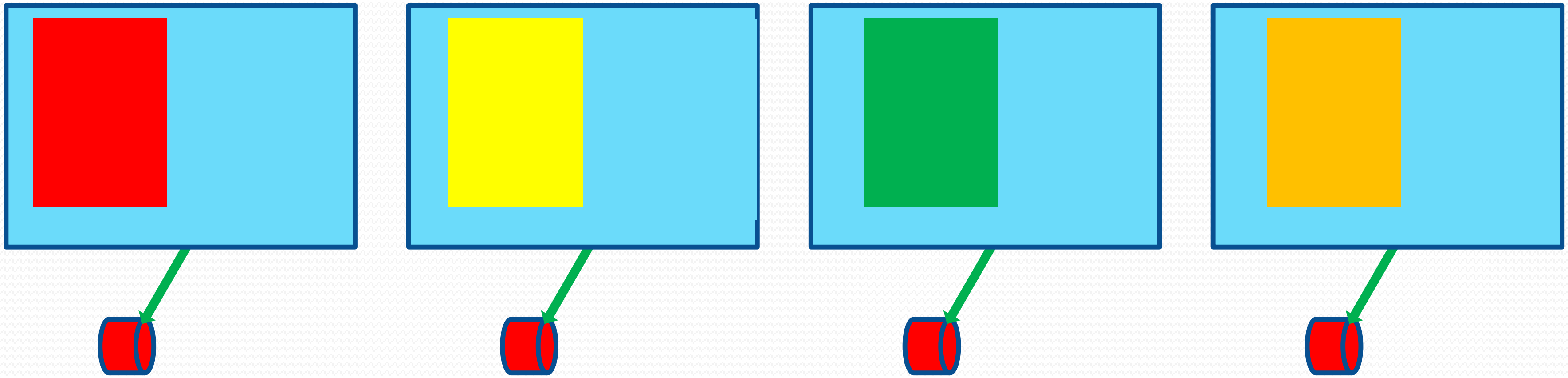
Array duplicated

$$\mathbf{A} = \begin{pmatrix} A(1,1) & A(1,2) & \cdots & A(1,N) \\ A(2,1) & A(2,2) & \cdots & A(2,N) \\ \cdots & \cdots & \cdots & \\ A(M,1) & A(M,2) & \cdots & A(M,N) \end{pmatrix}$$

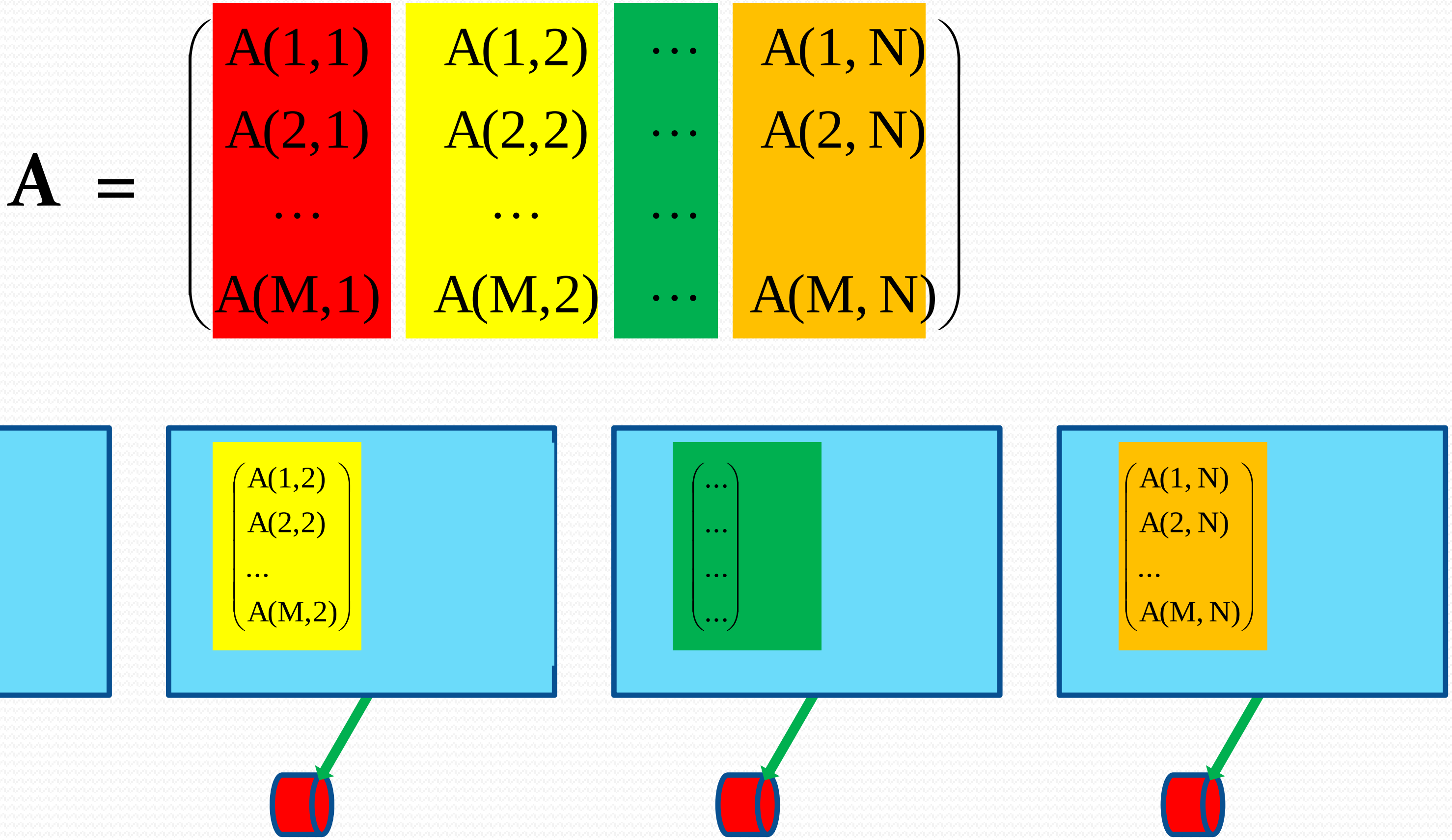


Array distributed

$$\mathbf{A} = \begin{pmatrix} A(1,1) & A(1,2) & \cdots & A(1,N) \\ A(2,1) & A(2,2) & \cdots & A(2,N) \\ \cdots & \cdots & \cdots & \\ A(M,1) & A(M,2) & \cdots & A(M,N) \end{pmatrix}$$



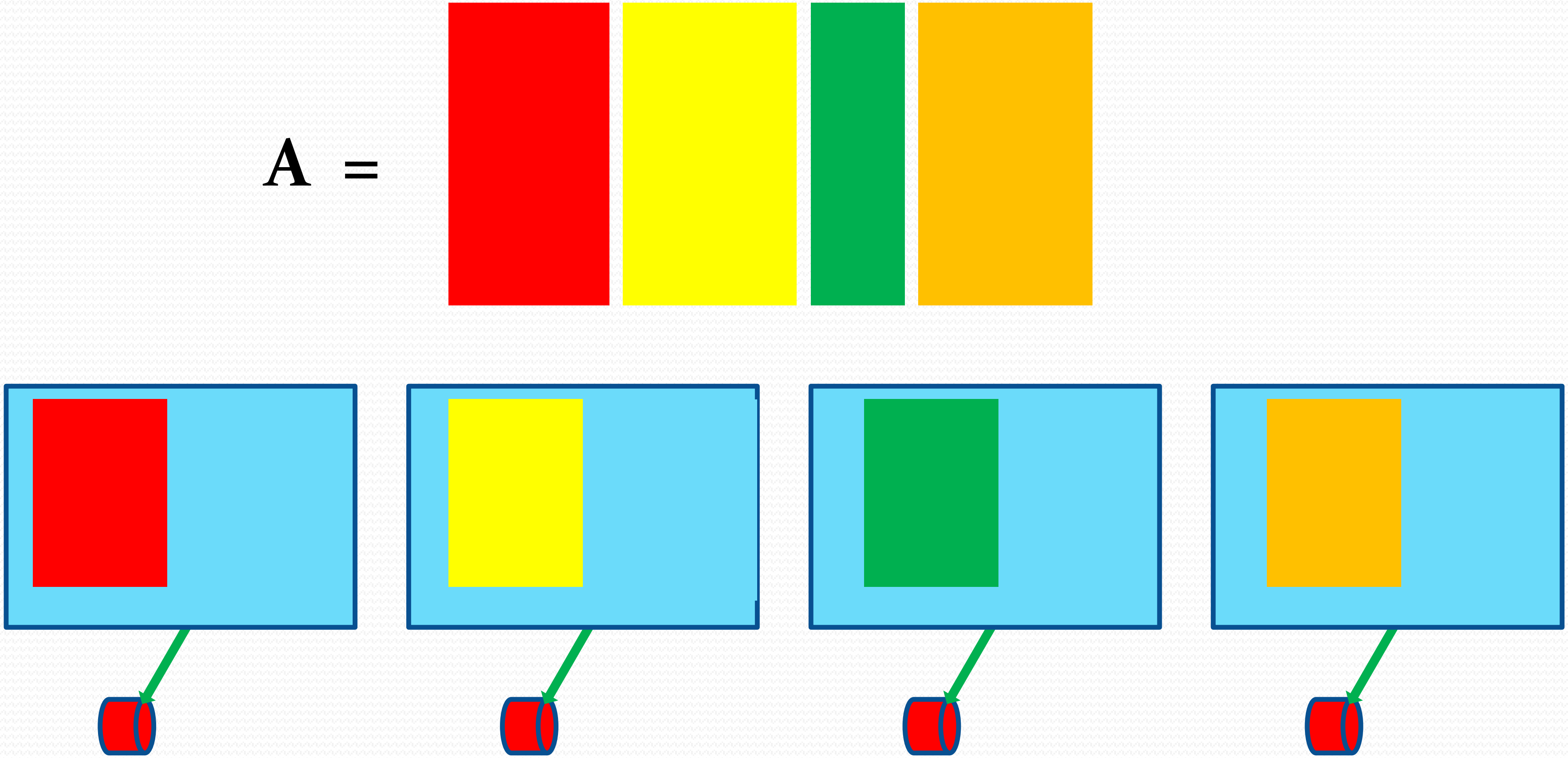
Array distributed



$$\mathbf{A} = \begin{pmatrix} A(1,1) & A(1,2) & \cdots & A(1,N) \\ A(2,1) & A(2,2) & \cdots & A(2,N) \\ \cdots & \cdots & \cdots & \\ A(M,1) & A(M,2) & \cdots & A(M,N) \end{pmatrix}$$

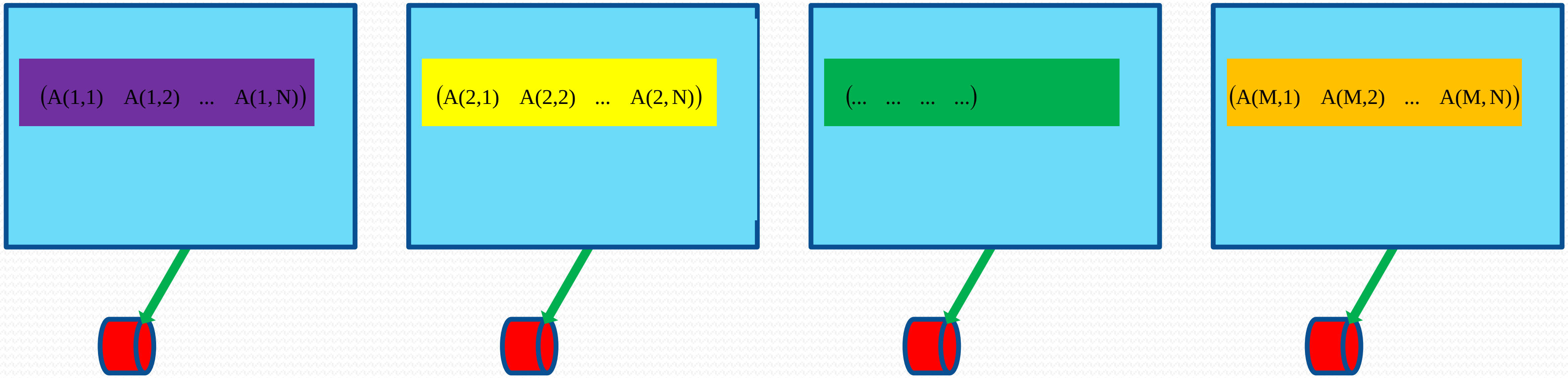
$$\begin{pmatrix} A(1,1) \\ A(2,1) \\ \ldots \\ A(M,1) \end{pmatrix} \quad \begin{pmatrix} A(1,2) \\ A(2,2) \\ \ldots \\ A(M,2) \end{pmatrix} \quad \begin{pmatrix} \ldots \\ \ldots \\ \ldots \\ \ldots \end{pmatrix} \quad \begin{pmatrix} A(1,N) \\ A(2,N) \\ \ldots \\ A(M,N) \end{pmatrix}$$

Array distributed



Array distributed

$$\mathbf{A} = \begin{pmatrix} A(1,1) & A(1,2) & \cdots & A(1,N) \\ A(2,1) & A(2,2) & \cdots & A(2,N) \\ \cdots & \cdots & \cdots & \\ A(M,1) & A(M,2) & \cdots & A(M,N) \end{pmatrix}$$



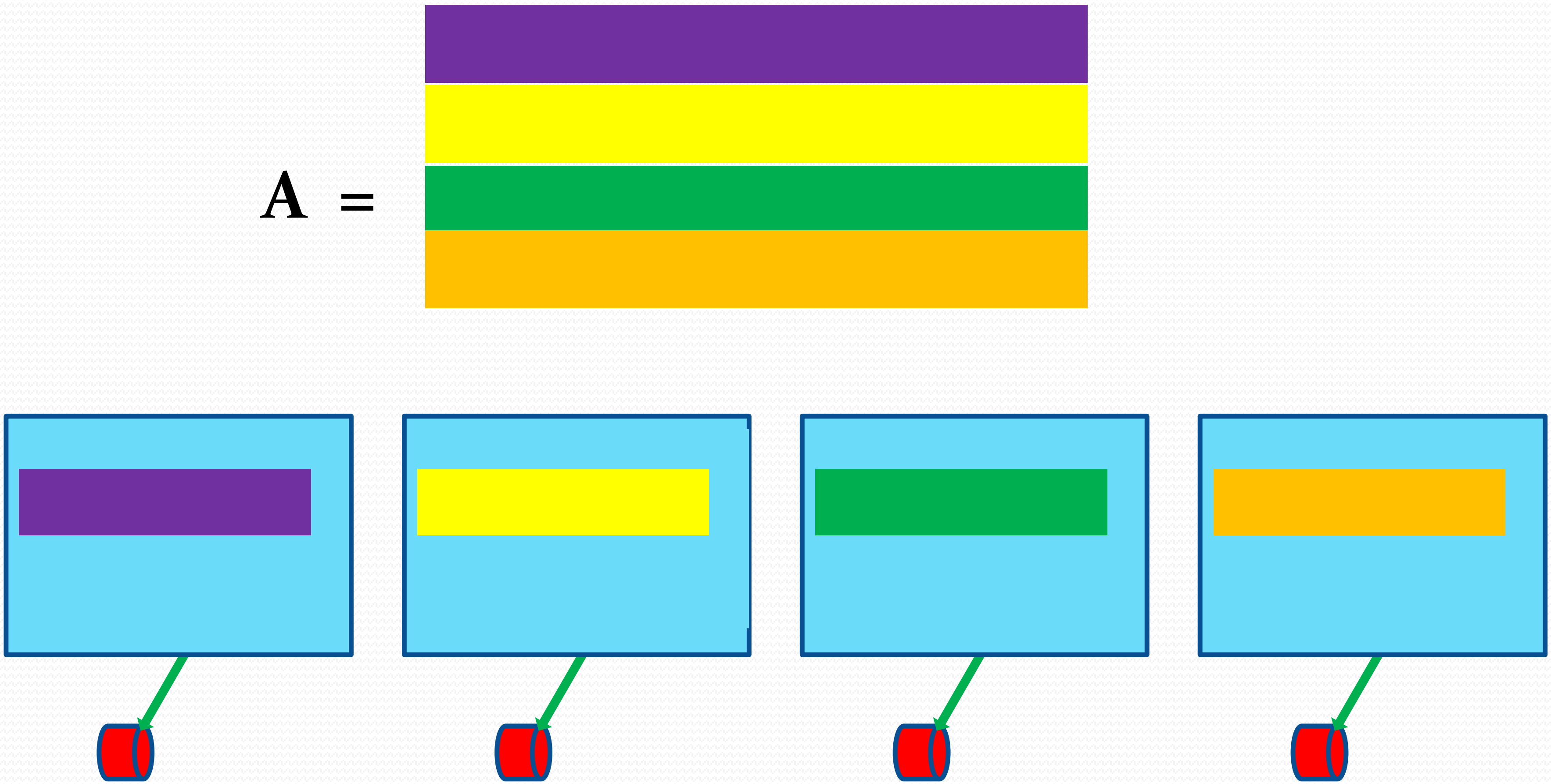
$(A(1,1) \quad A(1,2) \quad \ldots \quad A(1,N))$

$(A(2,1) \quad A(2,2) \quad \ldots \quad A(2,N))$

$(\ldots \quad \ldots \quad \ldots \quad \ldots)$

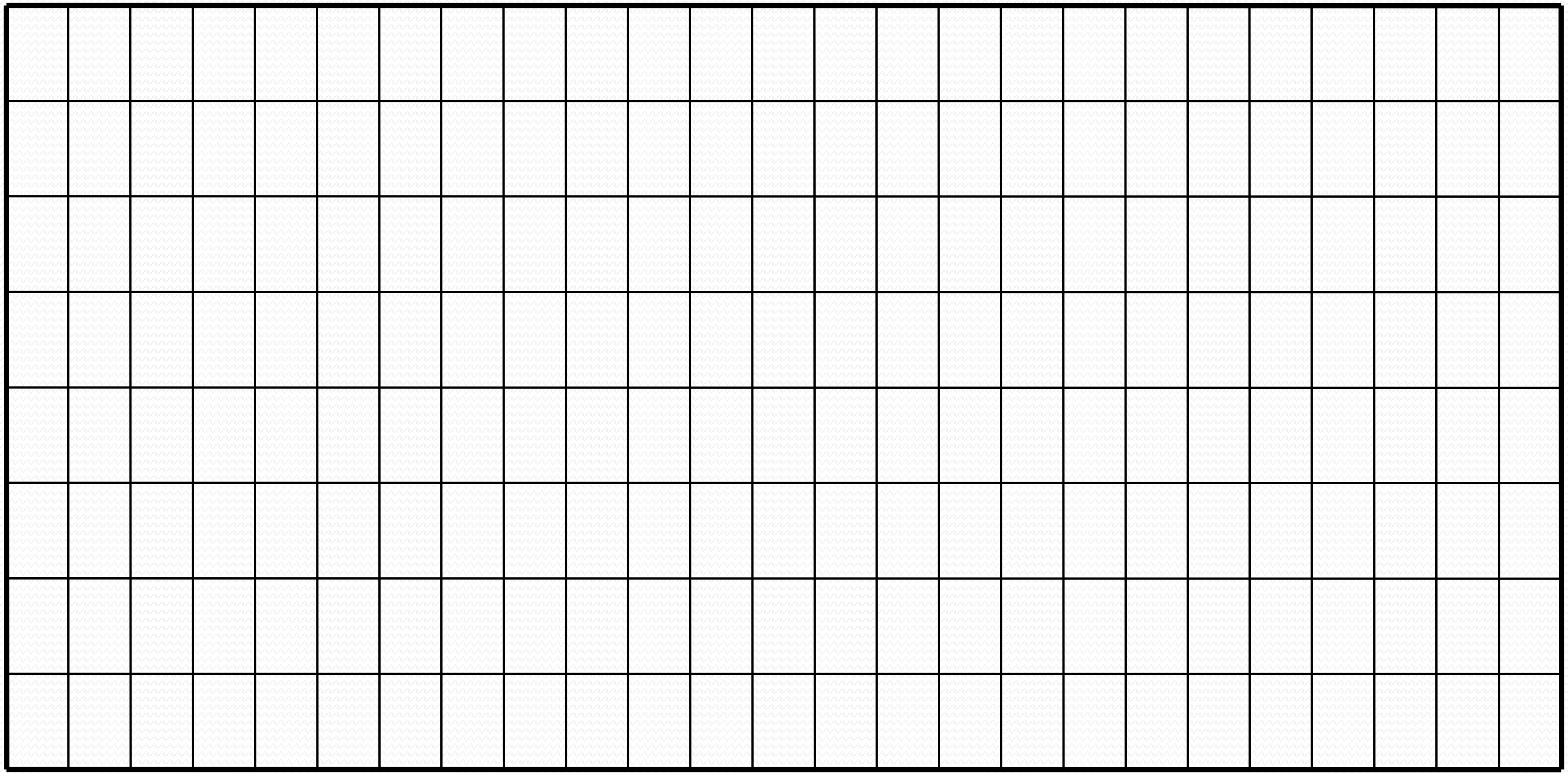
$(A(M,1) \quad A(M,2) \quad \ldots \quad A(M,N))$

Array distributed



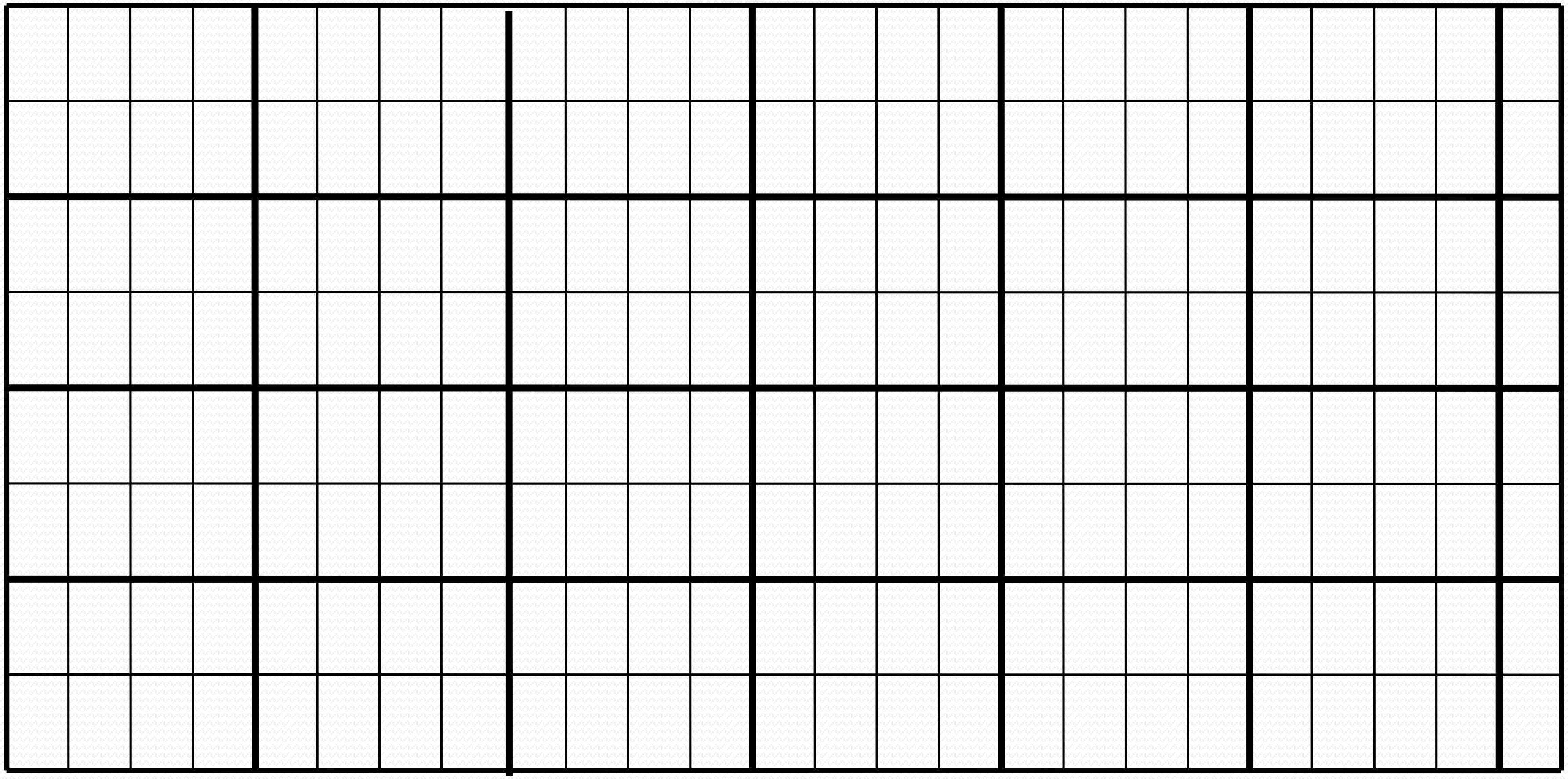
Round-robin distribution of two-dimensional arrays

A two-dimensional 8X25 array



With block sizes of 2X4,

the array is split into 4X7 blocks



2D Grid of Processes

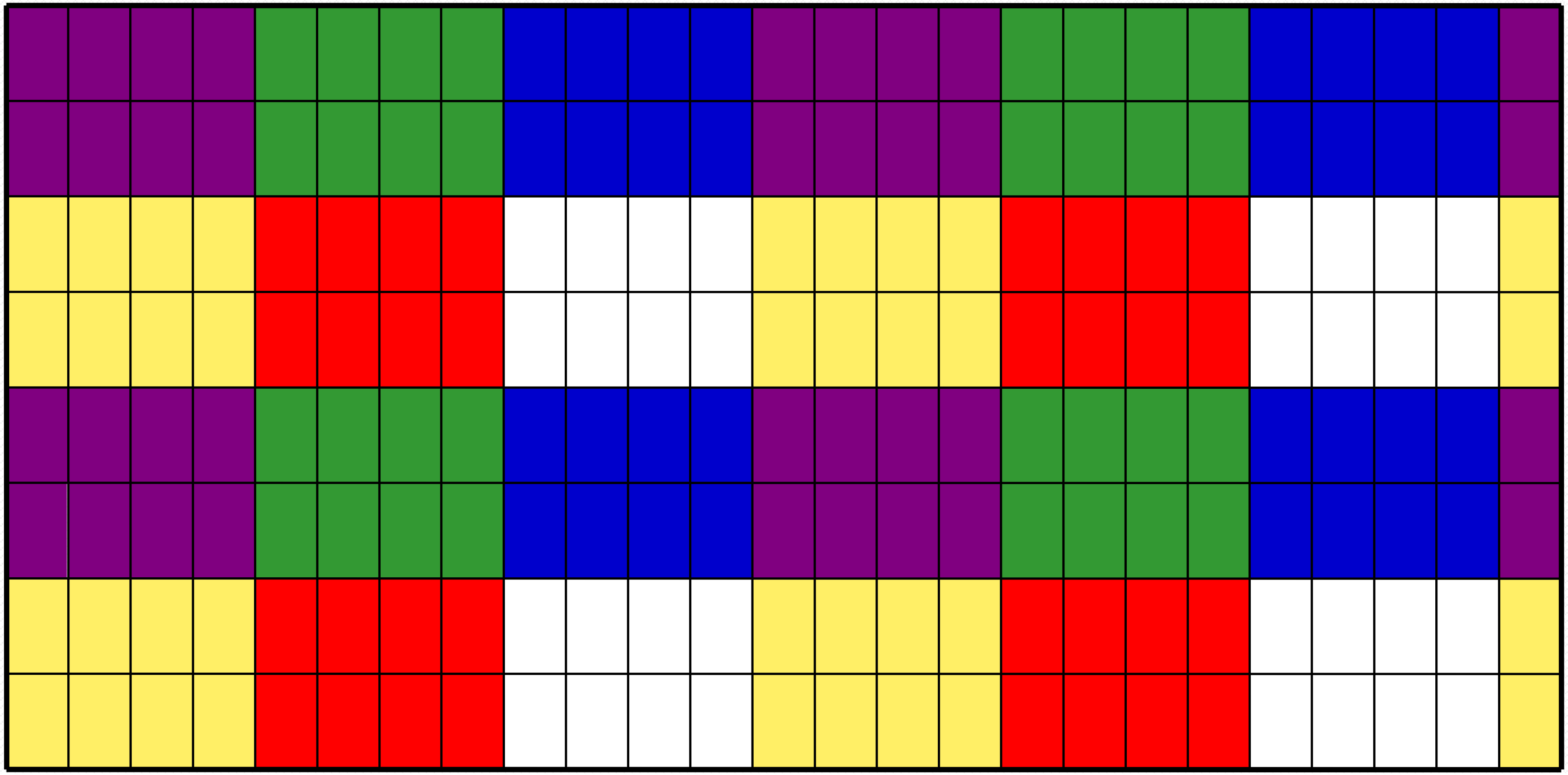
Suppose we have 2X3=6 processes with ranks 0, 1, 2, 3, 4, and 5. The table below shows the rank and row and column numbers of the grid of processors as

rank (row, column)

0(0,0)	1(0,1)	2(0,2)
3(1,0)	4(1,1)	5(1,2)

2D Cyclic Block Distribution

🟪 0(0,0)	🟩 1(0,1)	🟦 2(0,2)
🟨 3(1,0)	🟥 4(1,1)	⬜ 5(1,2)



MPI_TYPE_CREATE_DARRAY(…)

Memory Allocation in F90

Since the size of a distributed array in a process usually depends on the total number of processes (determined at running time), it is better to allocate the memory *dynamically*.

FORTRAN 90 also allows so by providing ALLOCATE() statement. We suggest to use language facilities rather than to call MPI routines to allocate memories, then they will be working in both serial and parallel versions.

The purpose of array distribution is

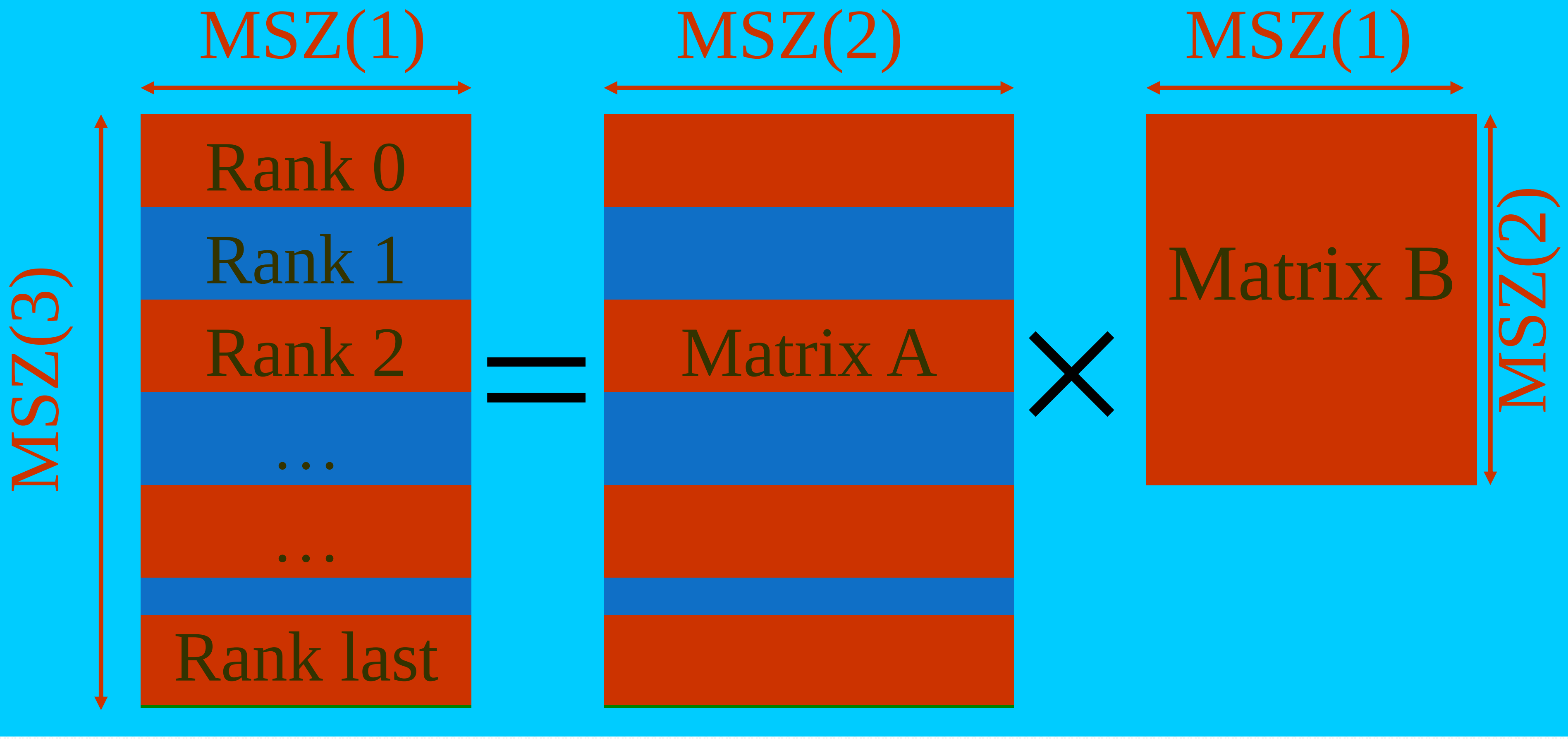
to save memory.

However it also makes some additional (complicated) MPI communications necessary.

Since array distribution is not so straight-forward, it is usually done at a later stage in coding an MPI parallel code.

Examples of distributed arrays

MPI Example 3



MSZ(1)

MSZ(2)

MSZ(1)

MSZ(3)

Rank 0

Rank 1

Rank 2

…

…

Rank last

=

Matrix A

×

Matrix B

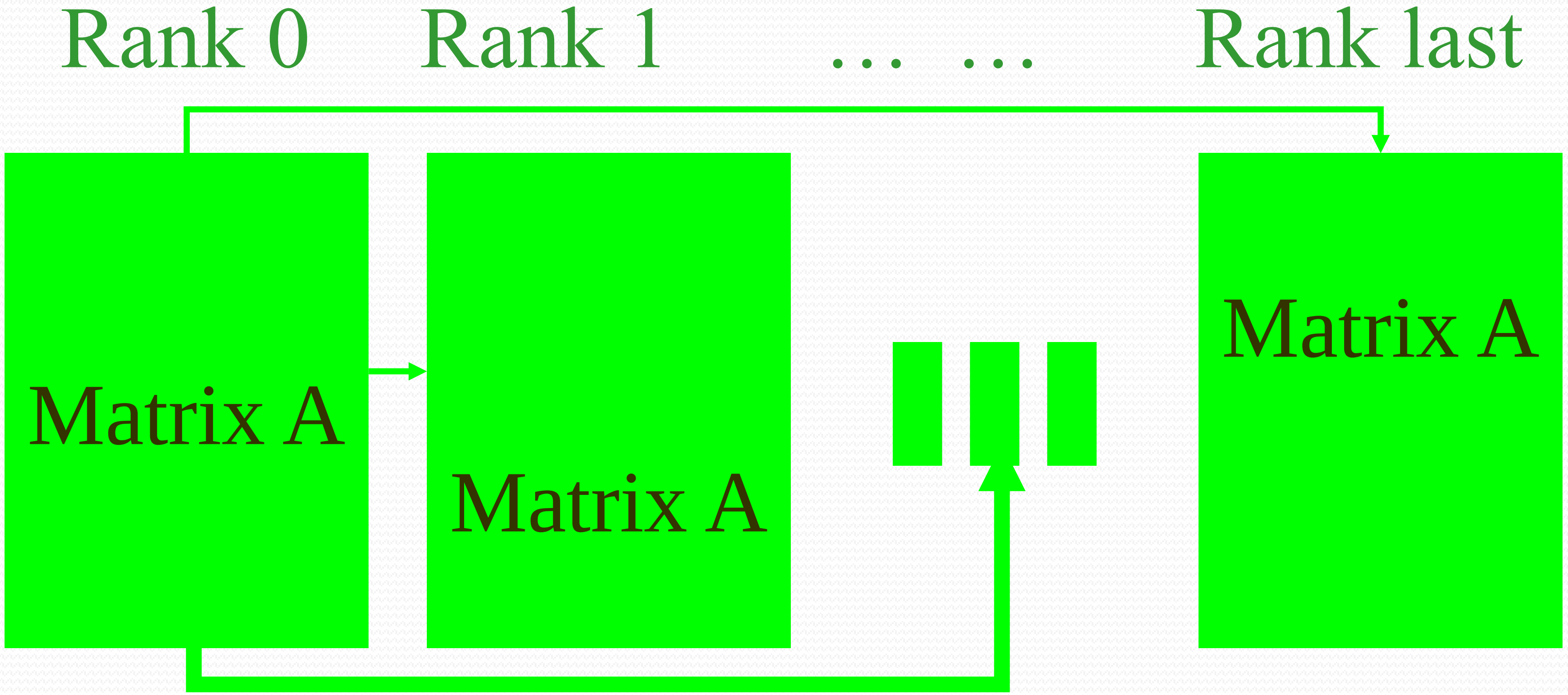
MSZ(2)

File f03.f90 c03.c cpp03.cpp

MPI Example 3

$C = A \times B$

Rank 0 reads values for matrix A & B, then broadcast



Rank 0 Rank 1 … … Rank last

Matrix A

Matrix A

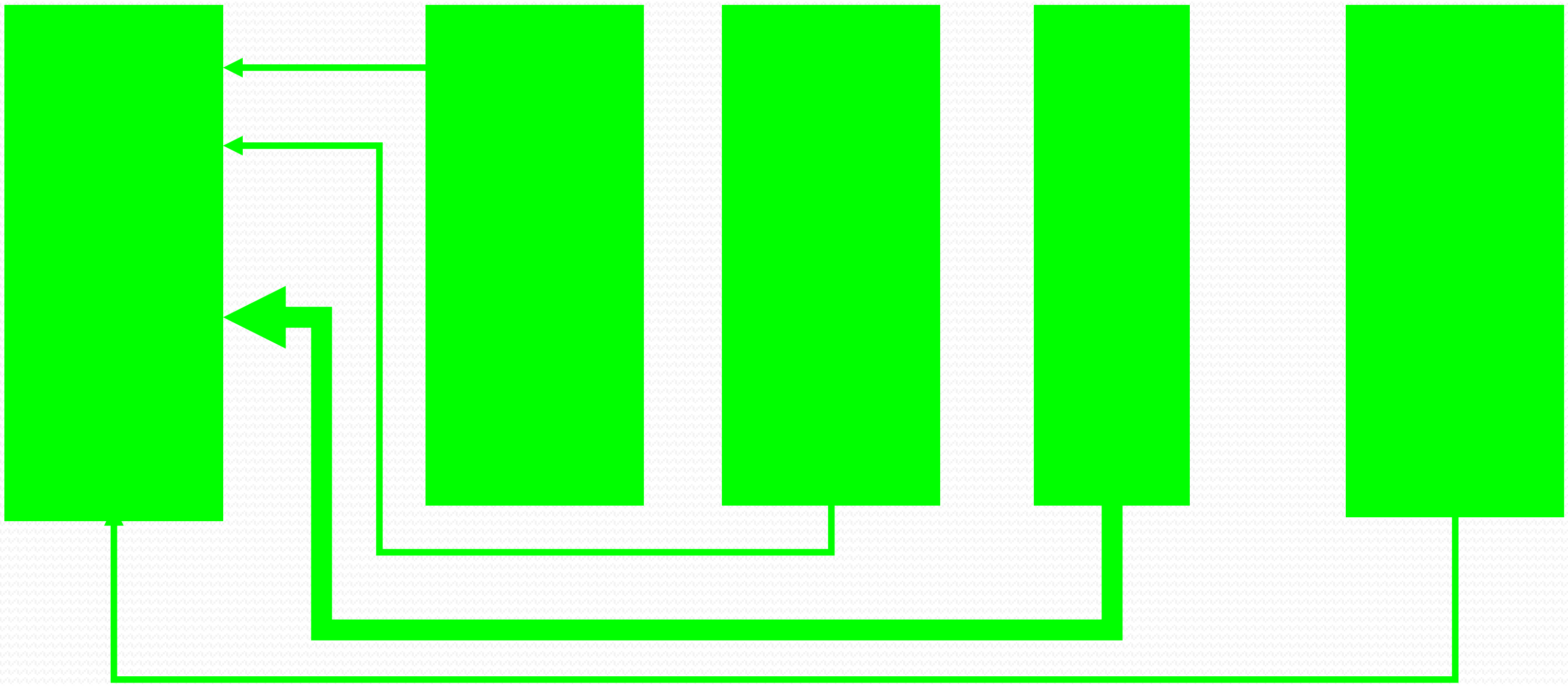
Matrix A

File f03.f90 c03.c cpp03.cpp

MPI Example 3

$$C = A \times B$$

To collect the final results into matrix C of Rank 0 with MPI_REDUCE



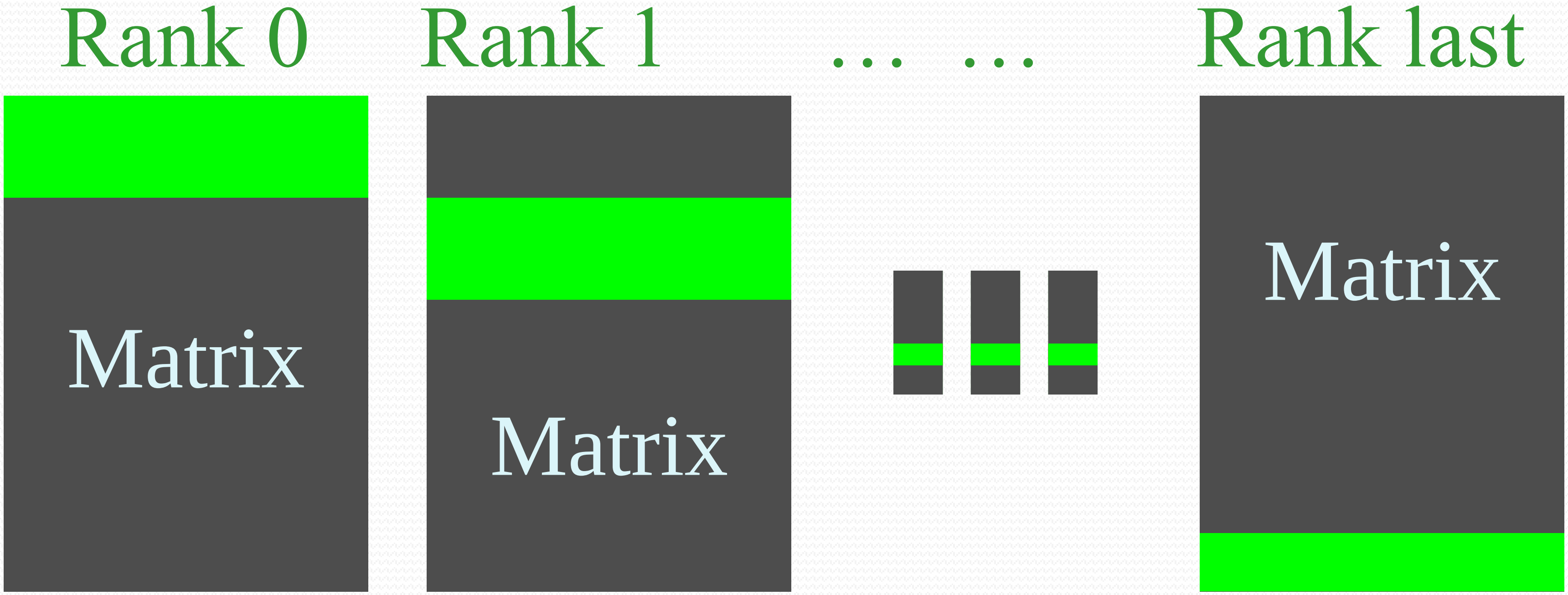
Rank 0 Rank 0 Rank 1 … Rank last

Click for f03.f90 c03.c cpp03.cpp

MPI Example 4

$C = A \times B$

Memory for matrix A and C(P) in Example 4



File f04.f90 c04.c cpp04.cpp

MPI Example 4

Normally, neither MPI_BCAST nor MPI_REDUCE can be used for communications for distributed arrays.

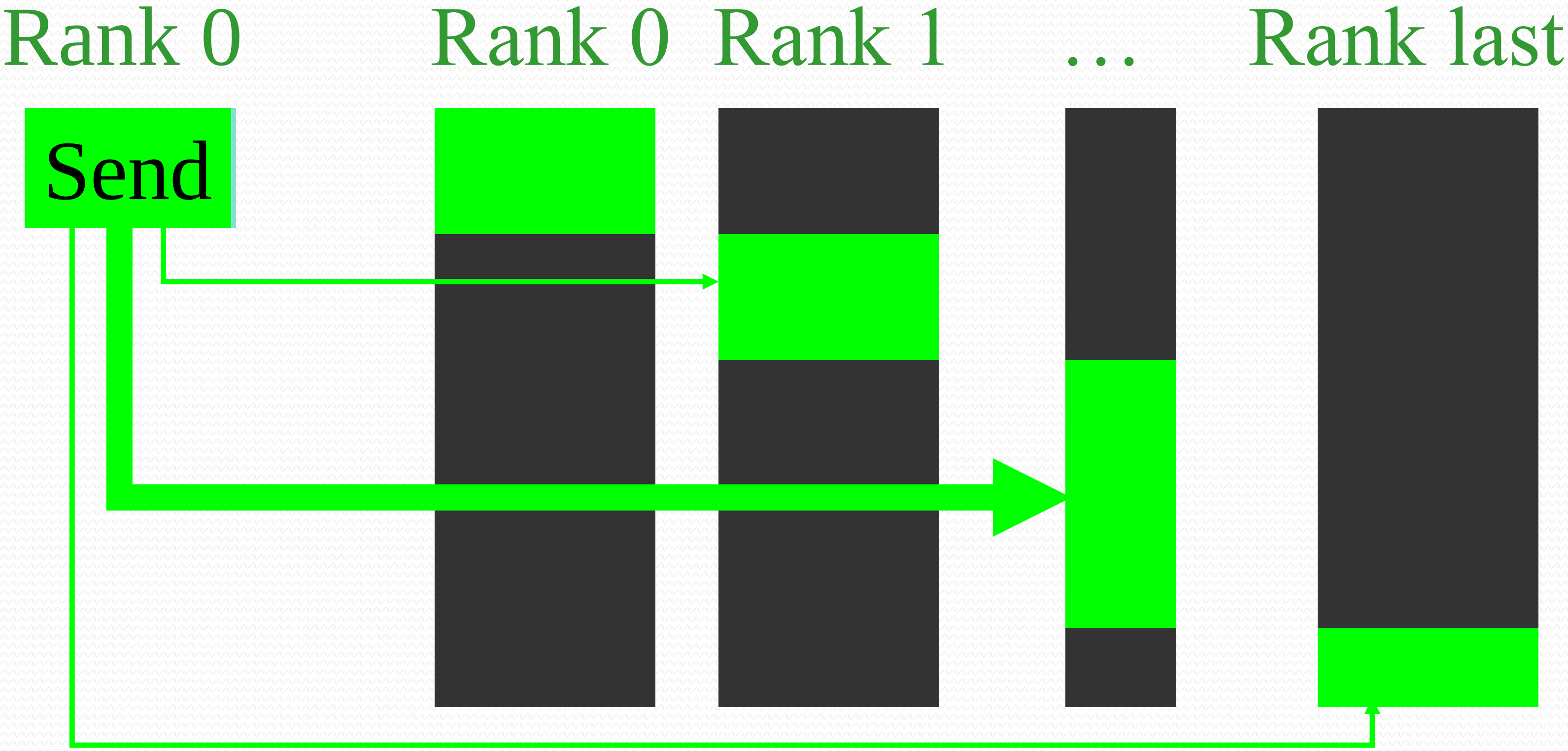
Instead, Point to Point communications will work.

File f04.f90 c04.c cpp04.cpp

MPI Example 4

$C = A \times B$

For assigning values to matrix A



Rank 0

Send

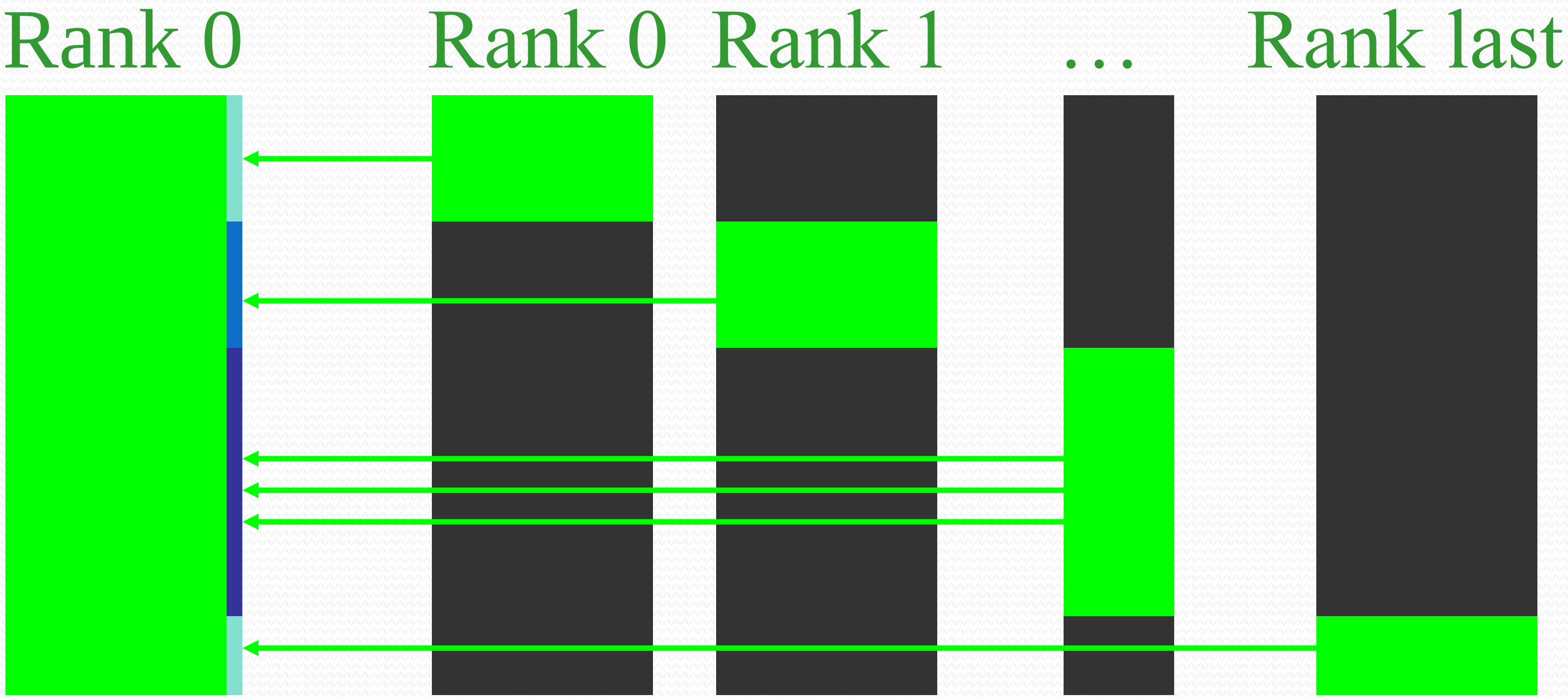
Rank 0 Rank 1 … Rank last

File f04.f90 c04.c cpp04.cpp

MPI Example 4

$$C = A \times B$$

To collect data for matrix C

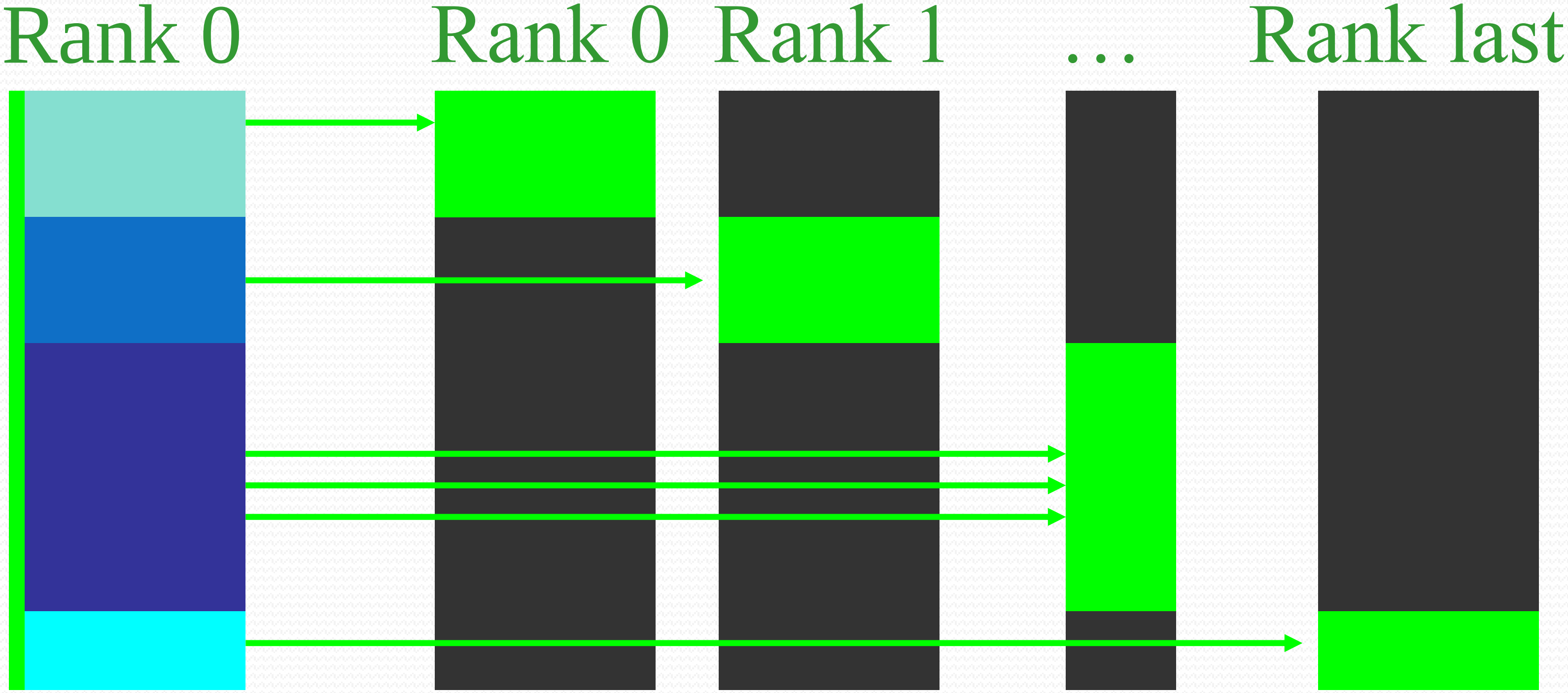


Click for f04.f90 **c04.c** **cpp04.cpp**

$$C = A \times B$$

MPI Example 5

For assigning values to matrix A



Rank 0 Rank 0 Rank 1 … Rank last

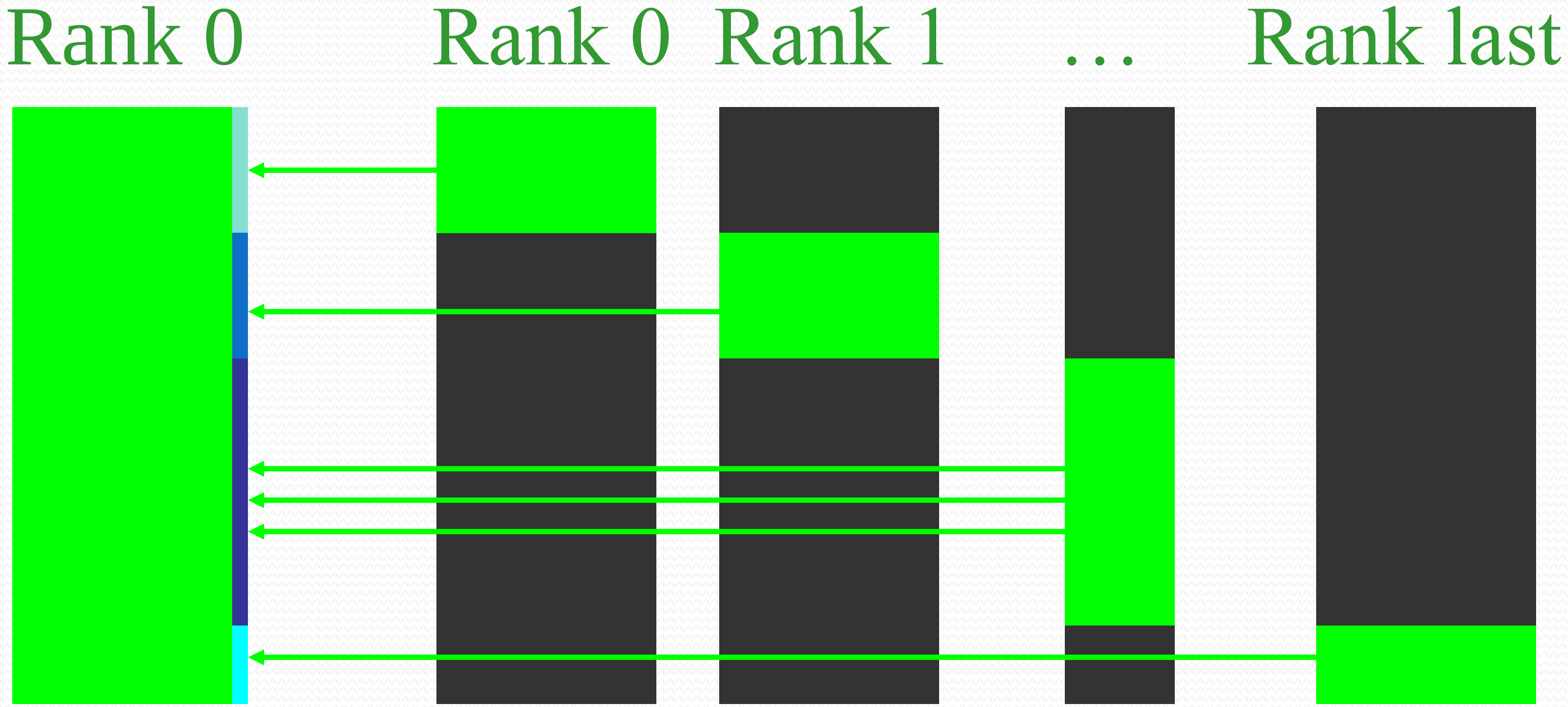
CALL MPI_SCATTERV

File f05.f90 c05.c cpp05.cpp

MPI Example 5

$C = A \times B$

To collect data for matrix C



CALL MPI_GATHERV

Click for f05.f90 c05.c cpp05.cpp

Comparison among

- Example : 3, 4, and 5
- Calculation job : same, same, same
- Parallelization: same, same, same
- Memory for matrixes A and C:
  duplicated, distributed, distributed
  , , full memory in one process
- Communication :
  broadcast & reduce, P-to-P, scatter & gather
- Programming :
  concise, tedious, compromised
- Suggestion : earlier try, later try, later try

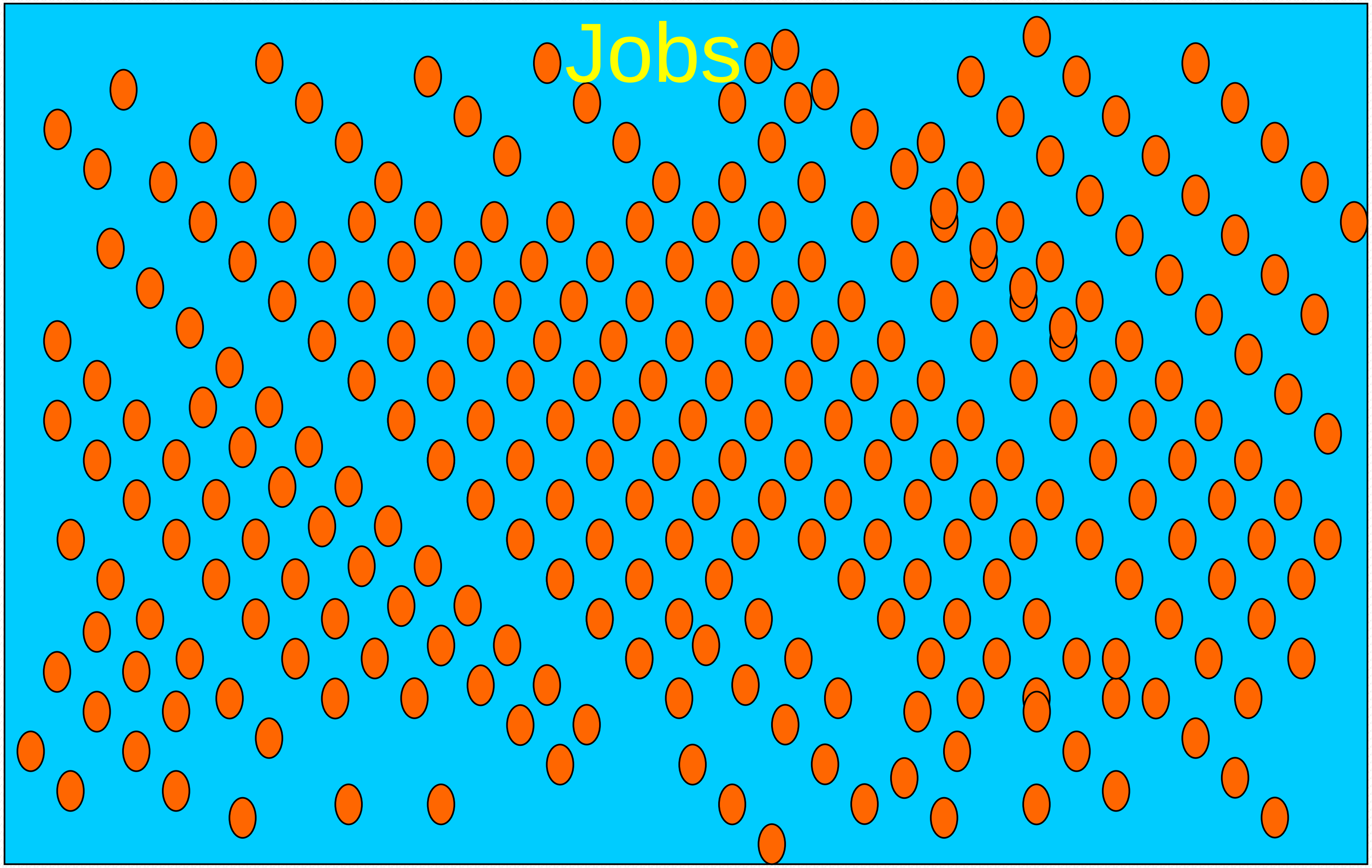
Outlines

- Introduction
- MPI basics
    - Programming environments
    - MPI predefined data types
    - Communications
    - User defined data types
    - Runtime environments
    - Some remarks
- Array distribution
- Sub-task distribution
- CAC bonus libraries
- References

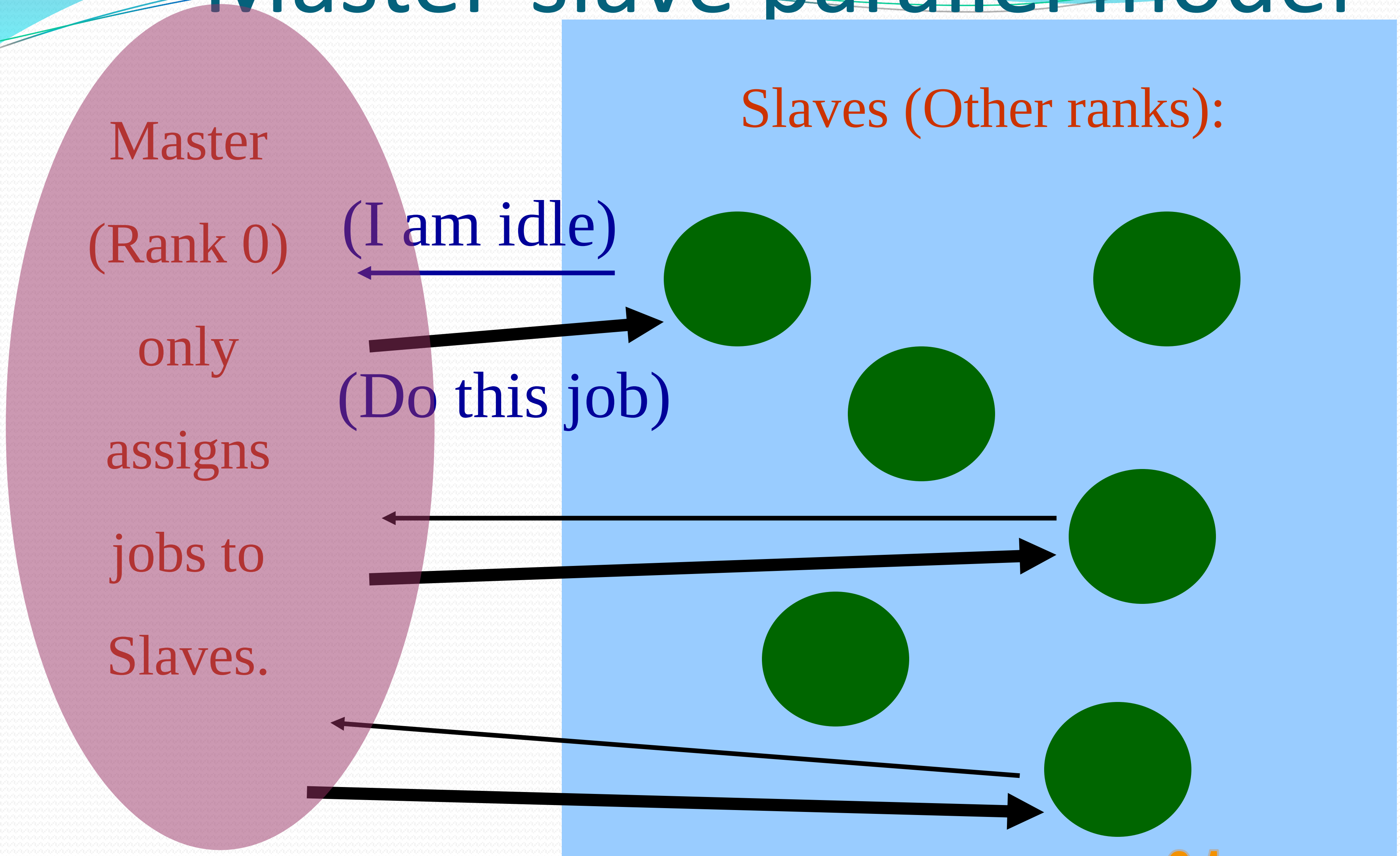
Generic Model

- The big calculation task is consisted of some independent smaller ones, which needs approximately the same CPU time.
- Then the smaller subtasks are distributed to the processes in the order of ranks and as evenly as possible.
- Widely used, as in previous examples.

Master-slave parallel model



Master-slave parallel model



Master (Rank 0) only assigns jobs to Slaves.

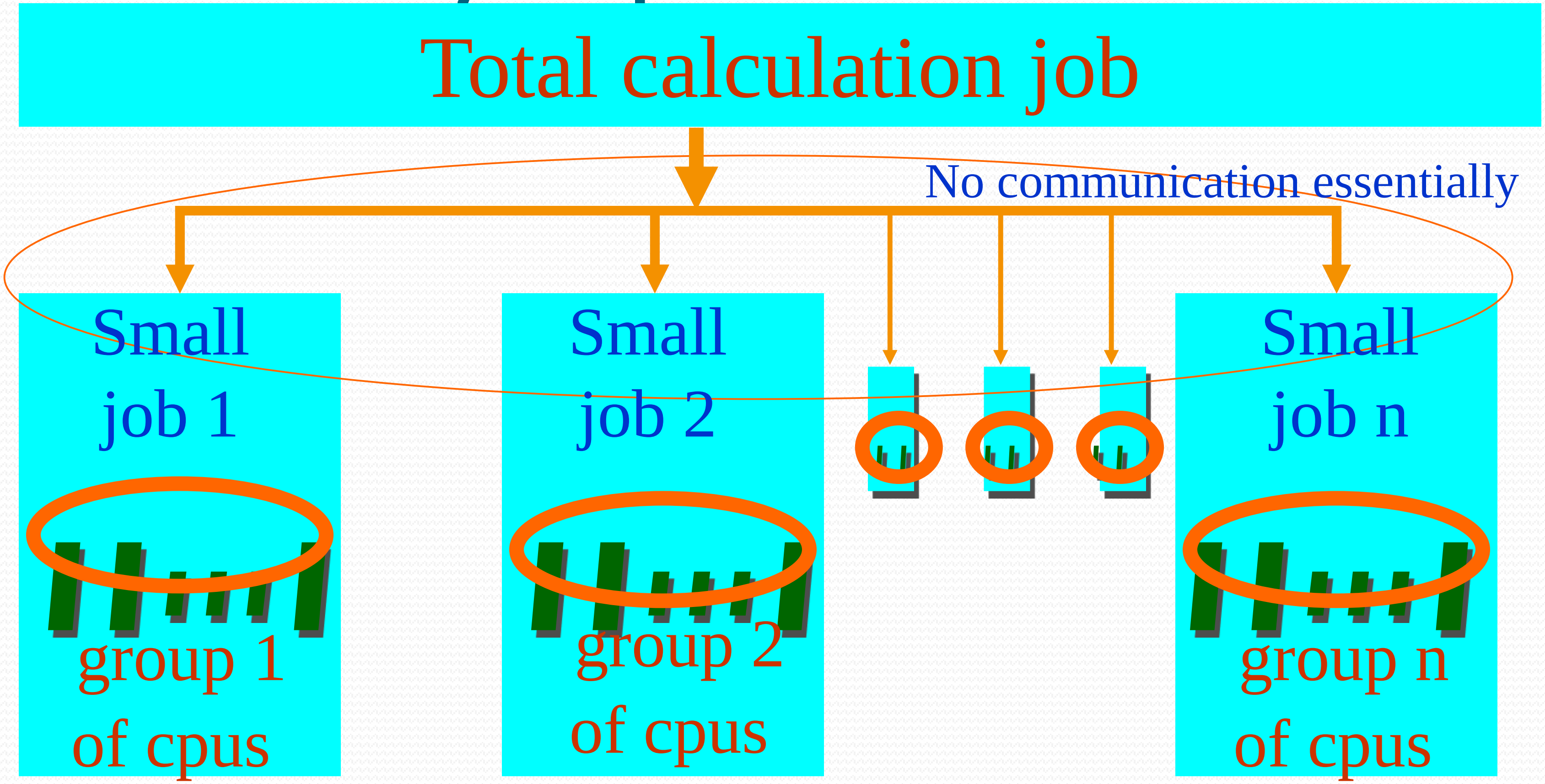
Slaves (Other ranks):

(I am idle)

(Do this job)

Click for f21.f90 **c21.c** **cpp21.cpp**

Two-layer parallel model



Total calculation job

No communication essentially

Small job 1

group 1 of cpus

Small job 2

group 2 of cpus

Small job n

group n of cpus

Click for f22.f90 c22.c cpp22.cpp

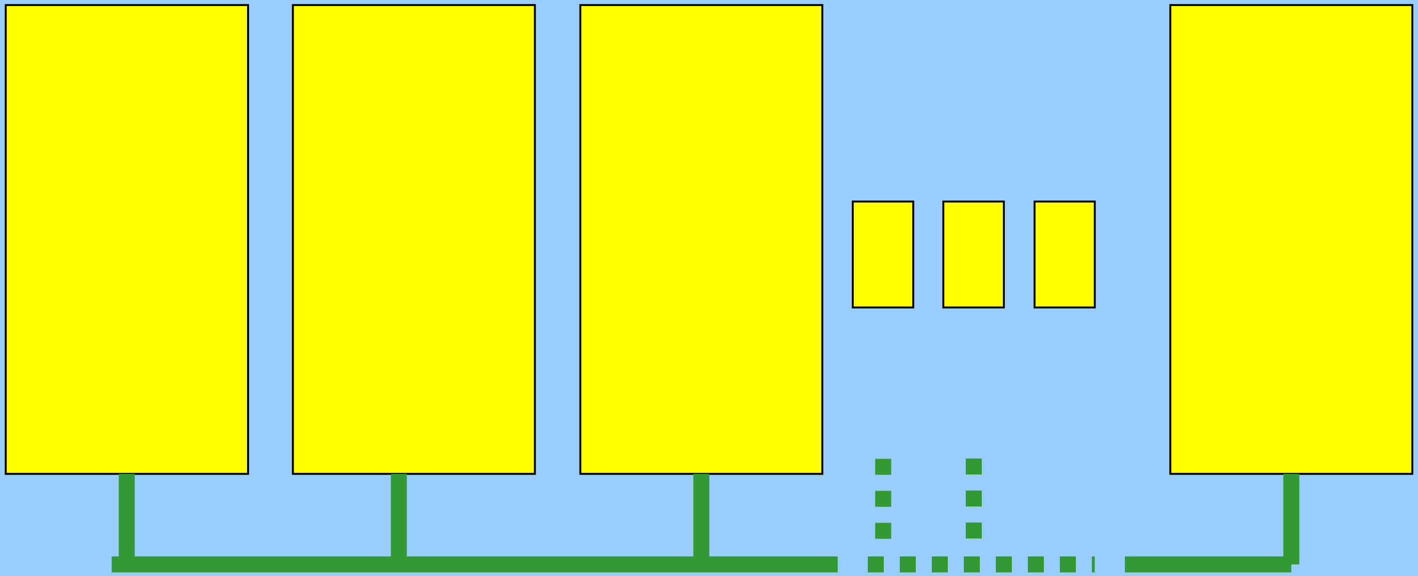
Outlines

- Introduction
- MPI basics
  - Programming environments
  - MPI predefined data types
  - Communications
  - User defined data types
  - Runtime environments
  - Some remarks
- Array distribution
- Sub-task distribution
- CAC bonus libraries
- References

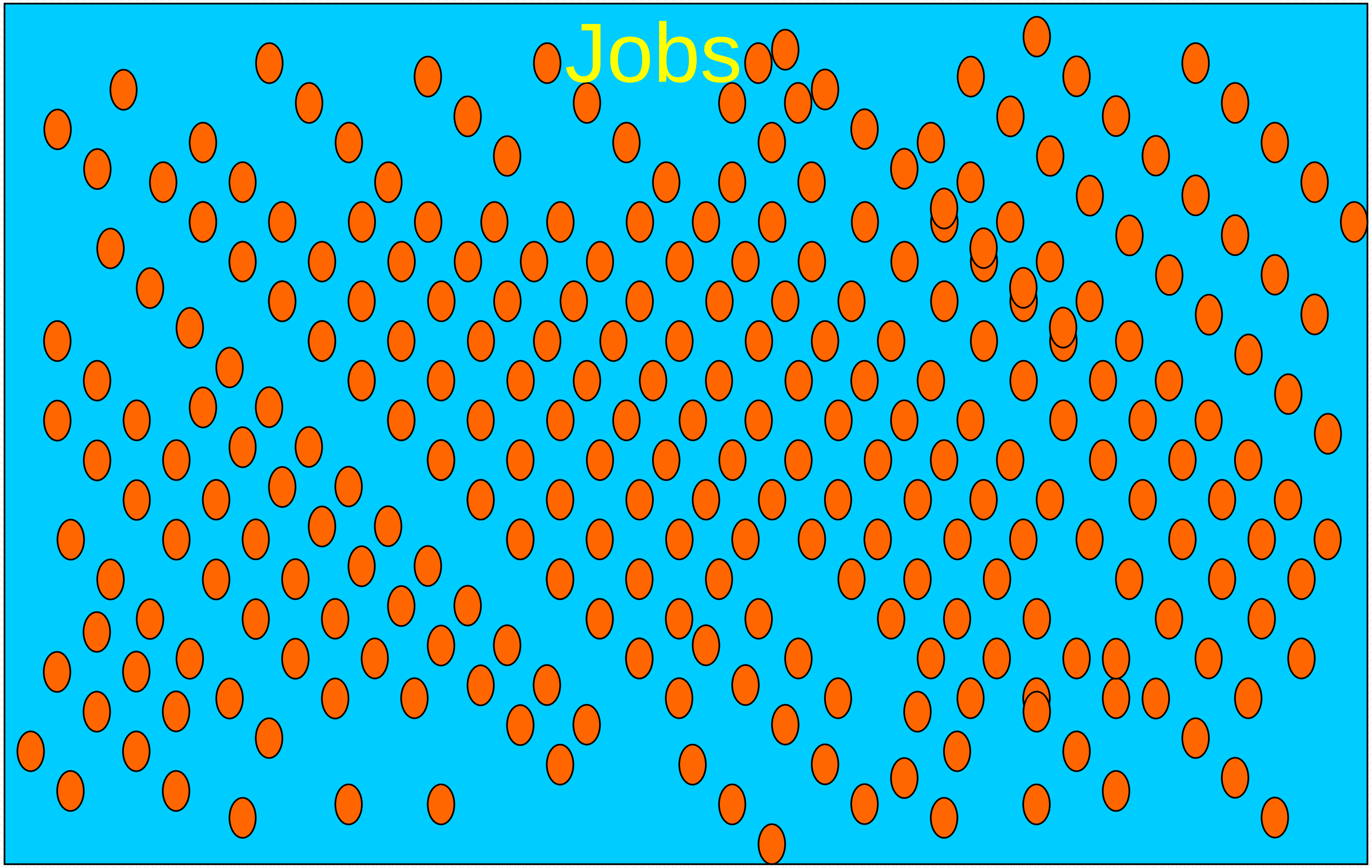
Double-layer Master-Slave Model

Double-layer master-slave model

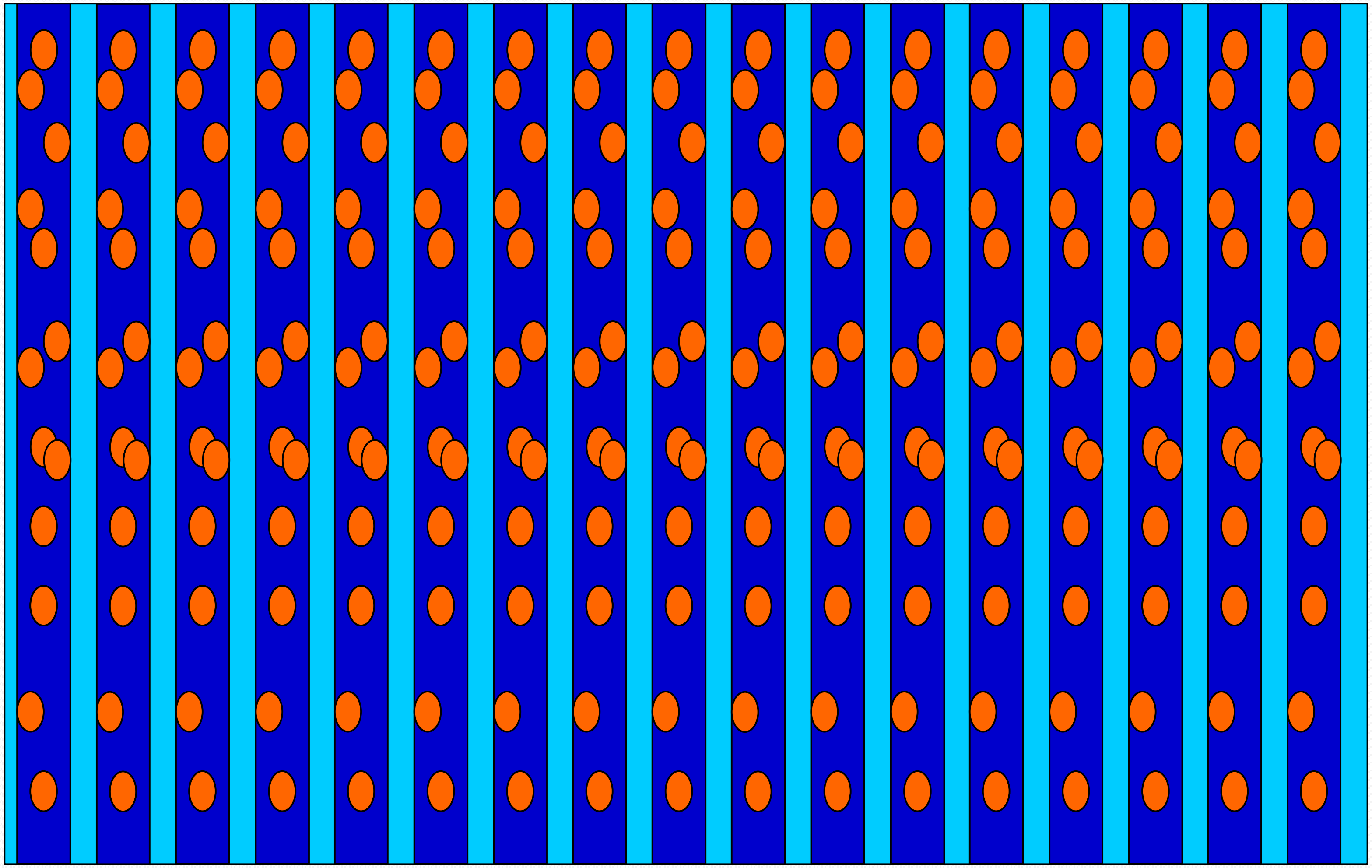
A big cluster of independent nodes
memory distributed between nodes



Double-layer Master-Slave Model

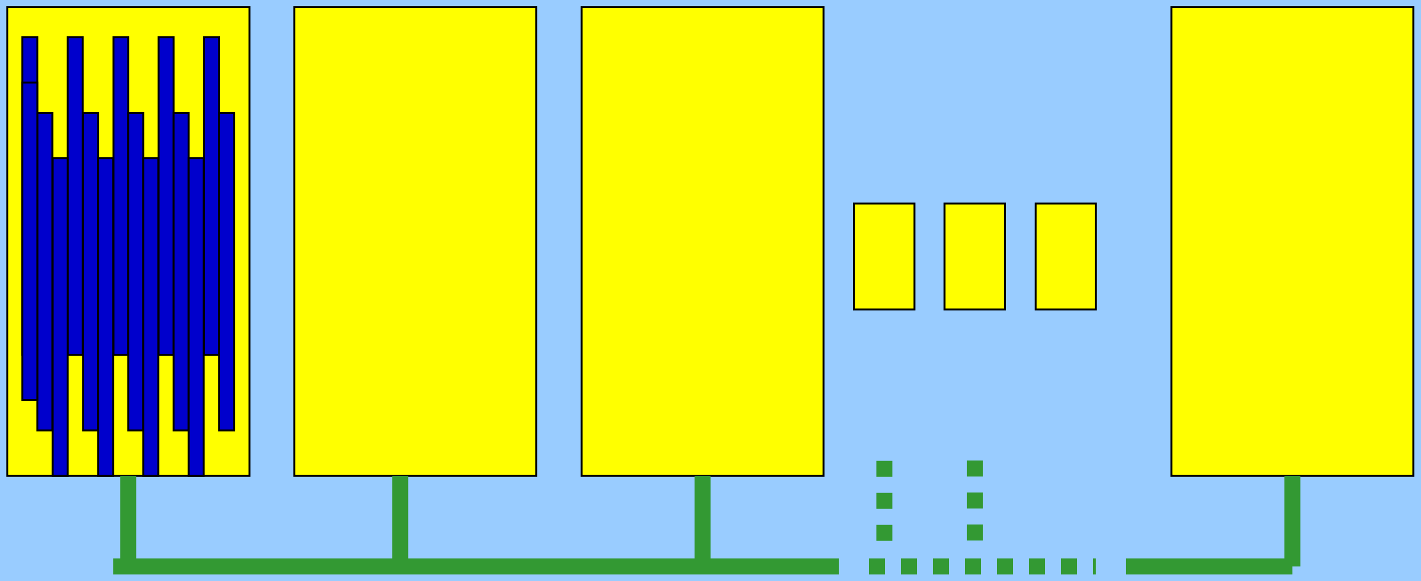


Double-layer Master-Slave Model



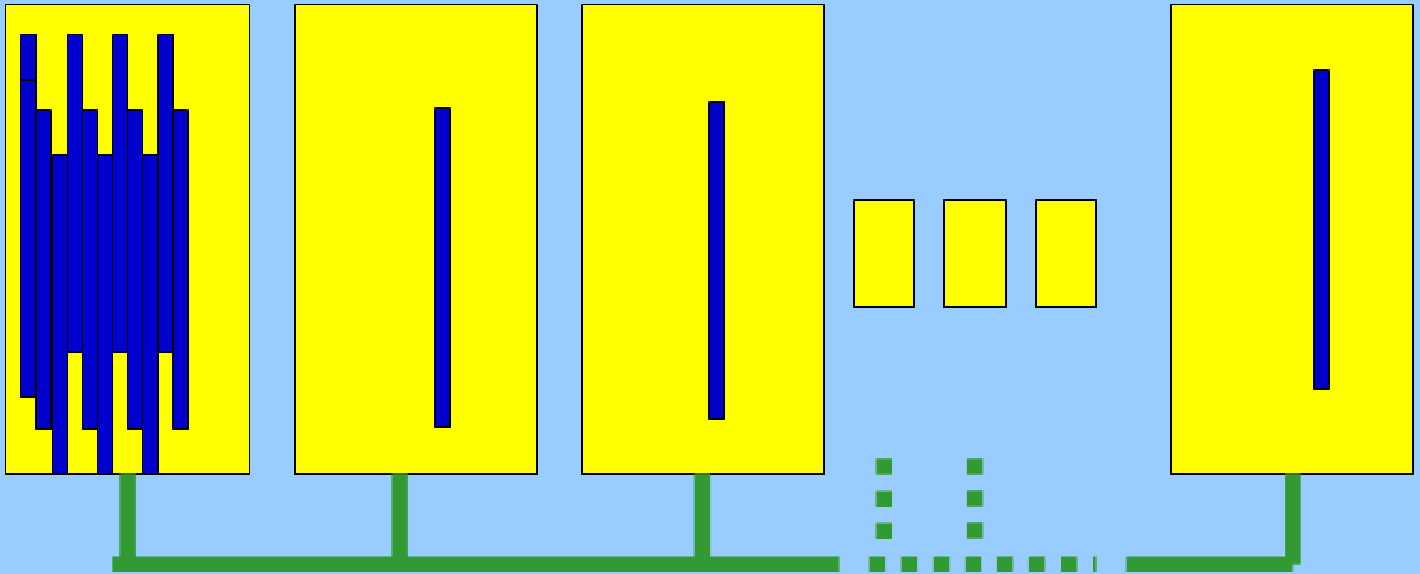
Double-layer Master-Slave Model

Job groups sent to nodes via MPI master-slave model

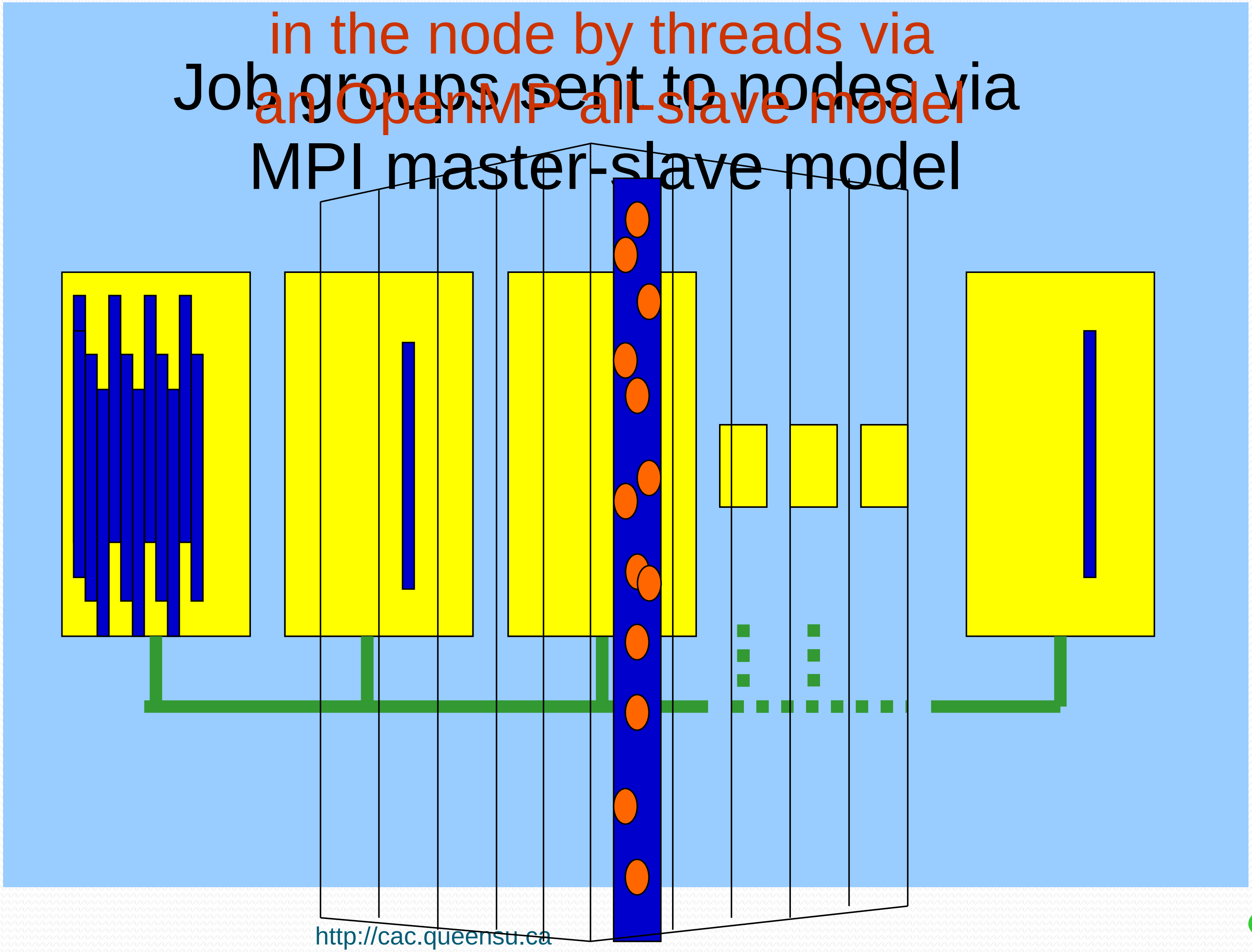


Double-layer Master-Slave Model

Job groups sent to nodes via MPI master-slave model



Double-layer Master-Slave Model



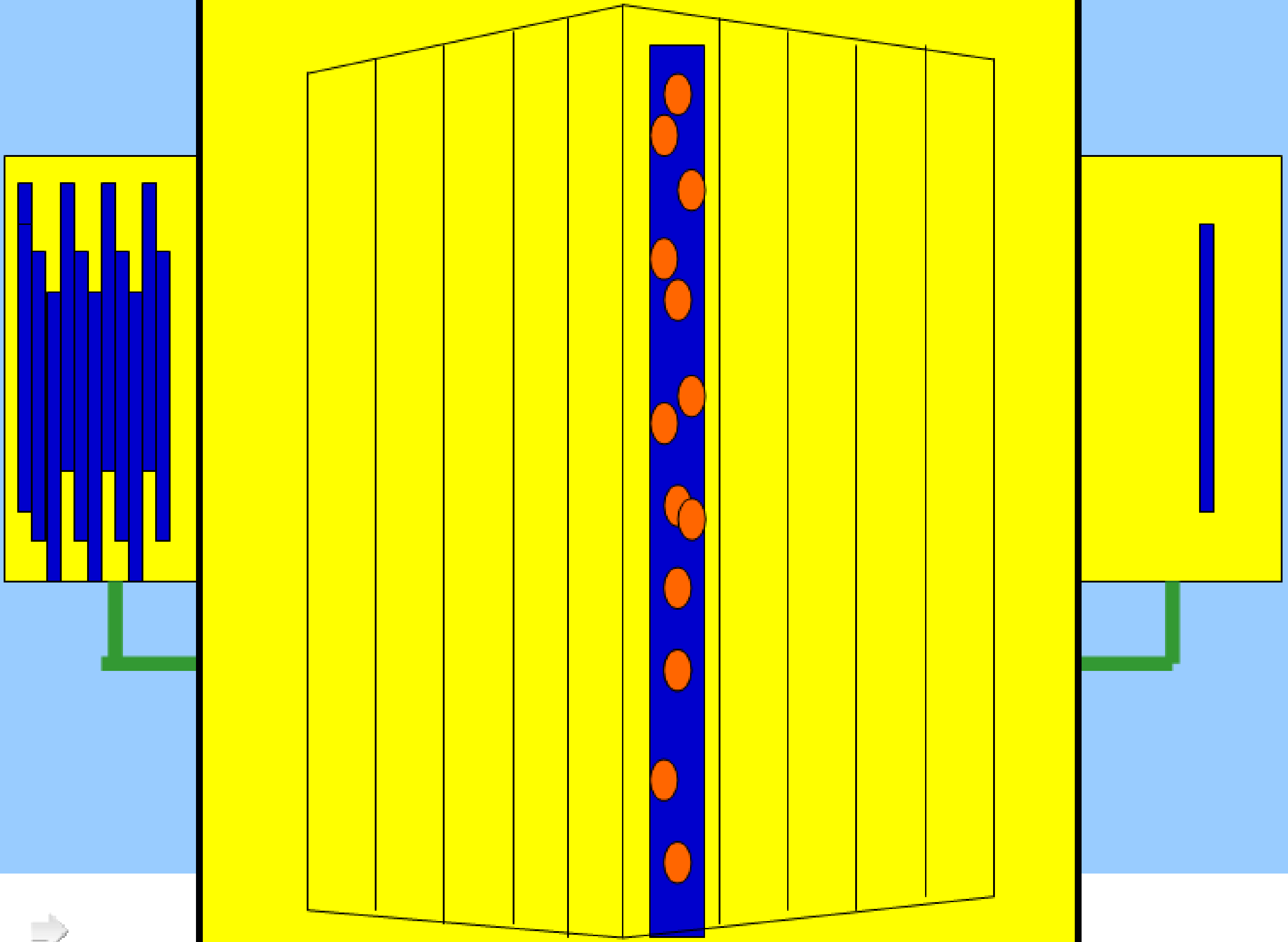
Jobs in a group executed in the node by threads via an OpenMP all-slave model

Job groups sent to nodes via MPI master-slave model

Doubl Model

Jo a

Jobs in a group executed in the node by threads via an OpenMP all-slave model



Double-layer Master-Slave Model

CAC supplies the DMSM library with source code for free.

Topics untouched

- Intercommunicators
- Data packing/unpacking
- Process topologies
- Dynamical process creation and management
- One-sided communications
- Parallel I/O
- Typical Parallelized Libraries with MPI
- Still many other functions in touched topics

References

- http://www.mpi-forum.org

- MPI – The Complete Reference

  Volume 1, The MPI Core

  Marc Snir, et al.

  Volume 2, The MPI Extensions

  William Gropp, et al.

Thank you very much for your attention!

Have a nice day!