# Parallel Programing with MPI

# Parallel Programming Languages

**Evolution of programming methods:**

- MPI is still the dominant programming technique
- Hybrid OpenMP/MPI approach most effective on supercomputers
- GPU programming develops quickly
  - CUDA
  - OpenACC and OpenMP
- Message Passing directly within the GPU
- New specific parallel programming languages are developed:
  - Co-array Fortran, PGAS, X10, Chapel…
- New runtime systems to handle task-based parallelism:
  - Charm++, HPX, Kokkos

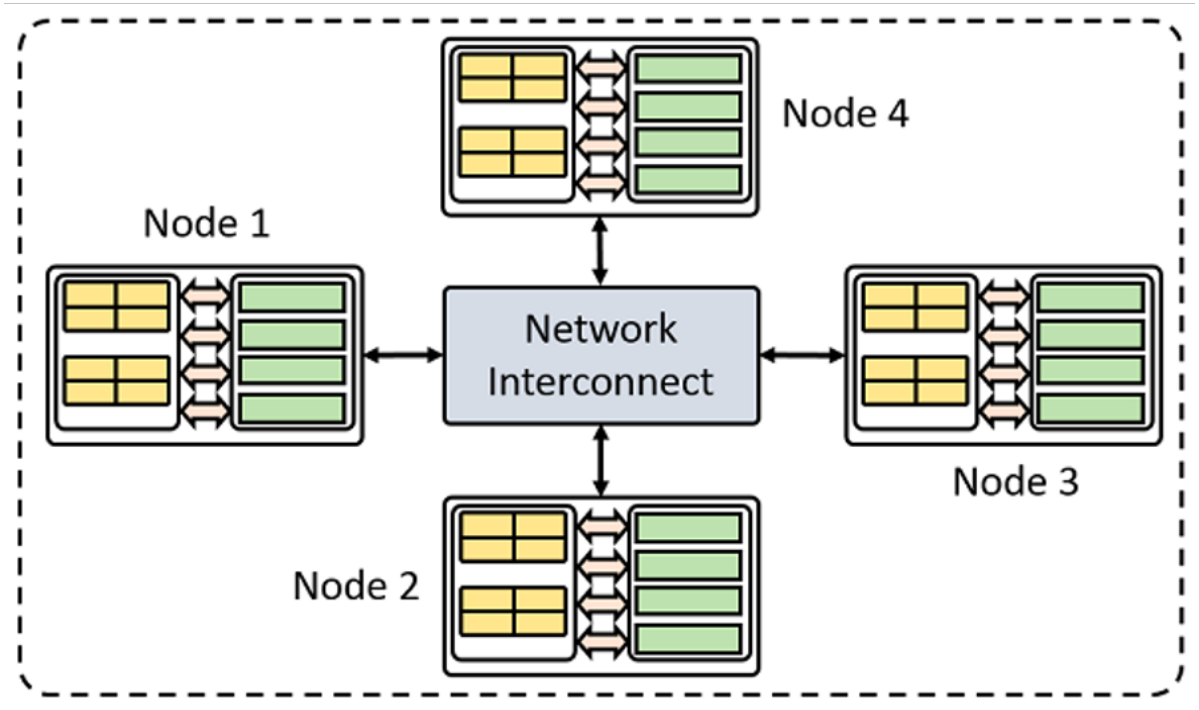
# Distributed memory versus shared memory

**MPI uses a distributed memory paradigm**

- Data are transferred explicitly between nodes through the network

**OpenMP uses a shared memory paradigm**

- Data are shared implicitly within the node through the Random-Access Memory.



# MPI: History

**MPI 1:**

- Version 1.0 (June 1994): 40 different organizations develop the MPI standard, with various subroutines defining the first MPI library
- Version 1.1 (June 1995)
- Version 1.2 (1997)
- Version 1.3 (September 2008): final version

**MPI 2:**

- Version 2.0 (July 1997): include new features intentionally left out of MPI 1 such as dynamical process management, one-sided communication, parallel I/O
- Version 2.1 (June 2008)
- Version 2.2 (September 2009)

**MPI 3:**

- Version 3.0 (September 2012): include new features left out of MPI 2 such as collective non-blocking communications, Fortran 2003 bindings, interfacing with external tools

**MPI 4**:

- work in progress (hybrid programming, fault tolerance, million-way parallelism)

# MPI: Implementations

**Open Source libraries :** can be installed on almost any architecture (for example on your laptop)

- MPICH2 http://www.mpich.org
- Open MPI https://www.open-mpi.org

**Vendors:**

- Intel MPI
- Platform MPI (IBM)
- Bullx MPI

# MPI: Tools

**Debuggers and performance analysis tools**:

- Totalview https://www.roguewave.com/products-services/totalview
- DDT https://www.arm.com/products/development-tools/hpc-tools/cross-platform/forge/ddt
- Scalasca http://www.scalasca.org

**Scientific libraries**

- Scalapack http://www.netlib.org/scalapack/
- PETSc http://www.mcs.anl.gov/petsc/
- FFTW http://www.fftw.org

# MPI: General Concepts

**Parallel processing**:

MPI is a library which allows process coordination and scheduling between millions of processors using a message-passing paradigm.

**Message attributes**

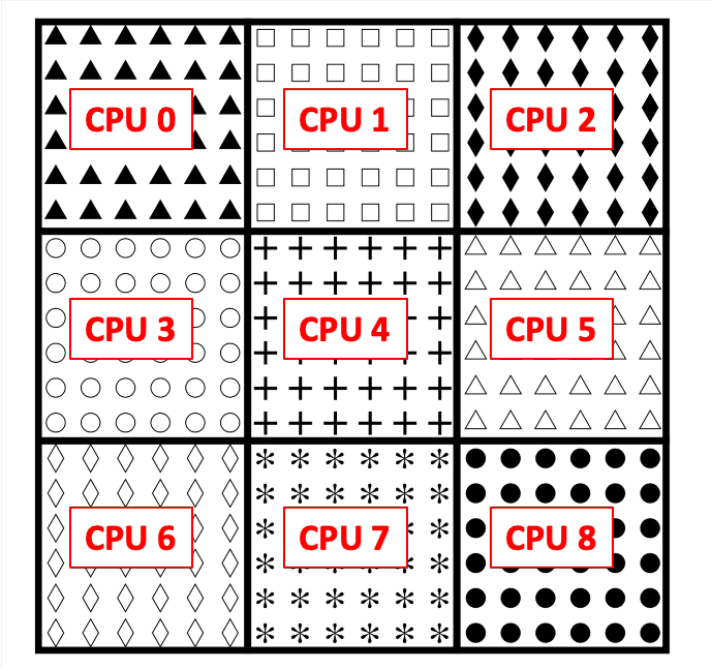
- The message is sent from a source process to a target process: sender address and recipient address
- The message contains a header
  - Identifier of the sending process (sender id)
  - The type of the message data (datatype)
  - The length of the message data (data length)
  - Identifier of the receiving process (receiver id)
- The message contains data

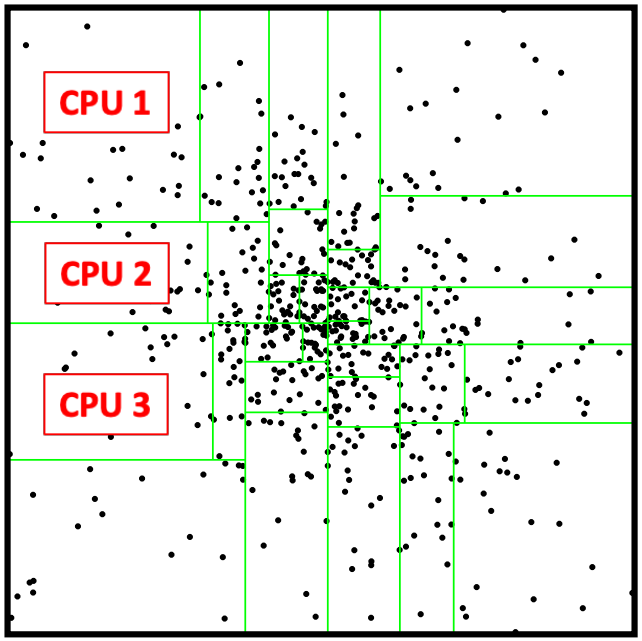
# The MPI Environment

- The messages are managed and interpreted by a runtime system comparable to a telephone provider, email system or postal company.
- Message are sent to a specific address. Receiving processes must be able to classify and interpret incoming messages.
- An MPI application is a group of autonomous processes deployed on different nodes, each one executing its own code and communicating to the other processes via calls to routines in the MPI library.

# MPI: Data Distribution

Data (grid cells or particles) are distributed between nodes and cores using a domain decomposition strategy.



# MPI: Basics

**MPI environment variables:**

`include mpi.h` file (MPI1-Fortran or C/C++)

`use mpi` module (MPI2-Fortran)

**Launching the MPI environment**:

`MPI_INIT()` routine

**Terminating the MPI environment**:

`MPI_FINALIZE()` routine

**C syntax**

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

**Fortran syntax**

```
call MPI_INIT(code)
call MPI_FINALIZE(code)
```

```
1  program who_I_am
2    use mpi
3    implicit none
4    integer :: nb_procs,rank,code
5
6    call MPI_INIT(code)
7
8    call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
9    call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
10
11   print *,'I am the process ',rank,' among ',nb_procs
12
13   call MPI_FINALIZE(code)
14 end program who_am_I
```

```
> mpiexec -n 7 who_am_I

I am process 3 among 7
I am process 0 among 7
I am process 4 among 7
I am process 1 among 7
I am process 5 among 7
I am process 2 among 7
I am process 6 among 7
```

# MPI: Blocking Send and Receive

```
1  program point_to_point
2    use mpi
3    implicit none
4
5    integer, dimension(MPI_STATUS_SIZE) :: status
6    integer, parameter                  :: tag=100
7    integer                             :: rank,value,code
8
9    call MPI_INIT(code)
10
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
12
13   if (rank == 2) then
14      value=1000
15      call MPI_SEND(value,1,MPI_INTEGER,5,tag,MPI_COMM_WORLD,code)
16   elseif (rank == 5) then
17      call MPI_RECV(value,1,MPI_INTEGER,2,tag,MPI_COMM_WORLD,status,code)
18      print *,'I, process 5, I received ',value,' from the process 2'
19   end if
20
21   call MPI_FINALIZE(code)
22
23 end program point_to_point
```

```
> mpiexec -n 7 point_to_point

I, process 5, I received 1000 from the process 2
```

With blocking send and receive, if you are not careful, you can have a **deadlock**. This means the processors will wait for something that will never happen. The classical mistake is to first call a blocking send *towards the next processor*, and then call a blocking receive *from the previous processor*.

# MPI: Non-Blocking Send and Receive

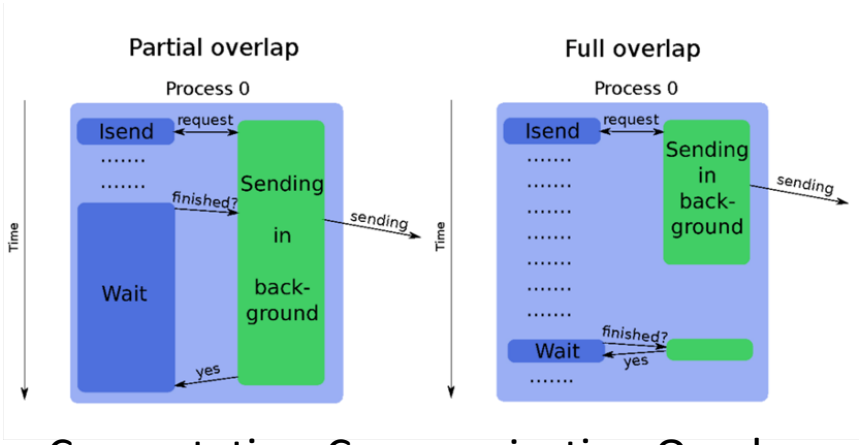
Communication costs can be large. The Infiniband network latency is around microseconds, or equivalent to thousands of processor cycles. One has to add the cost of the message itself (limited by the bandwidth of the Infiniband network around 10 GB/sec). Non-blocking routines can be used to perform calculations in parallel to the communication (additional level of parallelism). No risk of deadlock but risk of memory leak if the communication is not properly terminated.

Non-blocking send: **`MPI_ISEND()`**

Non-blocking receive: **`MPI_IRECV()`**

Wait or test: **`MPI_WAIT(), MPI_TEST()`**

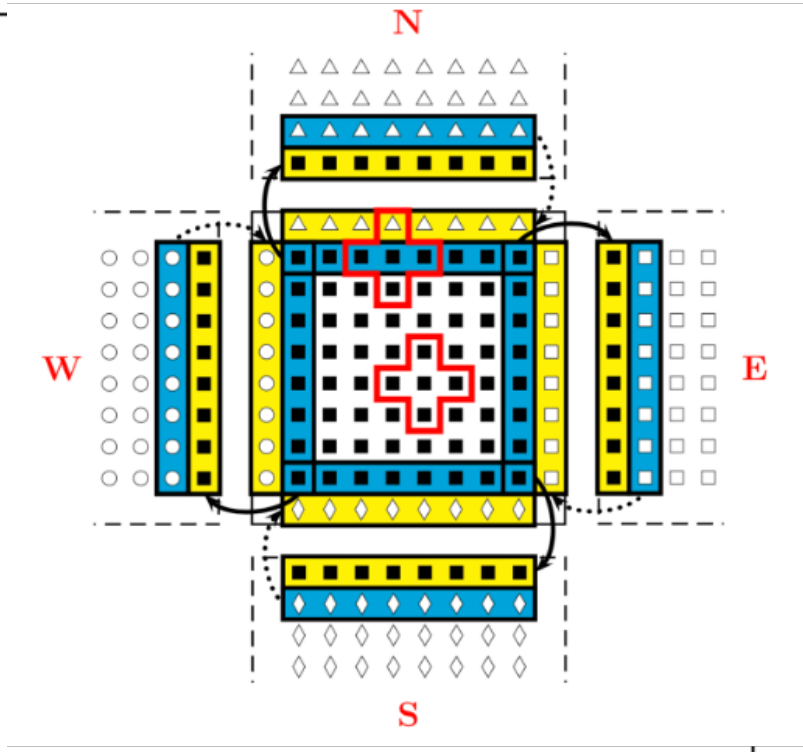
The communication buffers however cannot be used before the proper completion of the send or receive. The user needs to test that the communication is successful, in order to proceed with the data -> Higher algorithmic complexity.




Computation-Communication Overlap

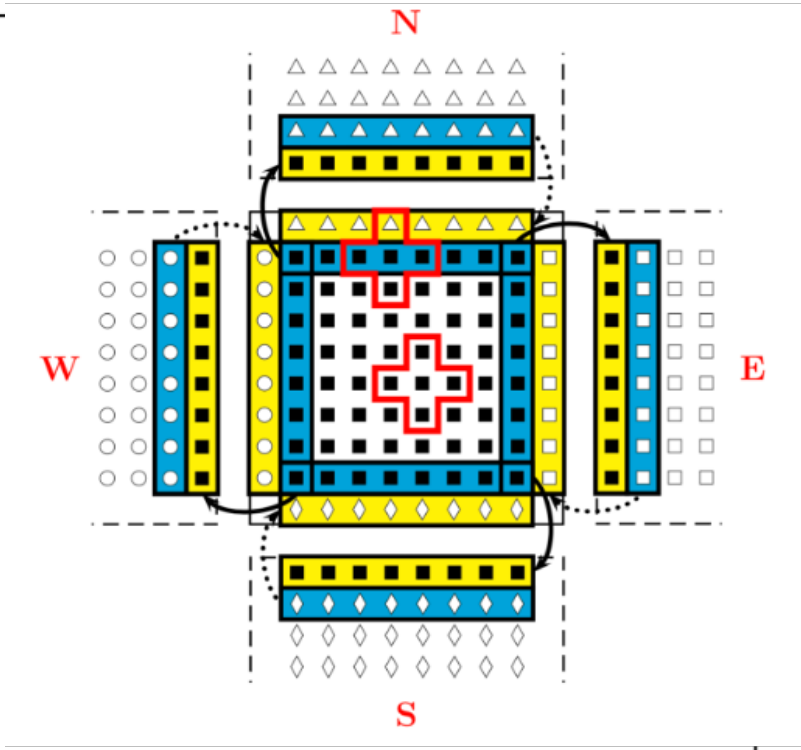
# MPI: Non-Blocking Send and Receive

```
1   SUBROUTINE start_communication(u)
2     ! Send to the North and receive from the South
3     CALL MPI_IRECV( u(,), 1, rowtype, neighbor(S), &
4          tag, comm2d, request(1), code)
5     CALL MPI_ISEND( u(,), 1, rowtype, neighbor(N), &
6          tag, comm2d, request(2), code)
7
8     ! Send to the South and receive from the North
9     CALL MPI_IRECV( u(,), 1,rowtype, neighbor(N), &
10         tag, comm2d, request(3), code)
11    CALL MPI_ISEND(  u(,), 1,rowtype,neighbor(S), &
12         tag, comm2d, request(4),code)
13
14    ! Send to the West and receive from the East
15    CALL MPI_IRECV( u(,), 1, columntype, neighbor(E), &
16         tag, comm2d, request(5), code)
17    CALL MPI_ISEND( u(,), 1, columntype, neighbor(W), &
18         tag, comm2d, request(6),code)
19
20    ! Send to the East and receive from the West
21    CALL MPI_IRECV( u(,), 1, columntype, neighbor(W), &
22         tag, comm2d, request(7),code)
23    CALL MPI_ISEND( u(,), 1, columntype, neighbor(E), &
24         tag, comm2d, request(8),code)
25  END SUBROUTINE start_communication
26  SUBROUTINE end_communication(u)
27    CALL MPI_WAITALL(2*NB_NEIGHBORS,request,tab_status,code)
28  END SUBROUTINE end_communication
```



# MPI: Non-Blocking Send and Receive

```
1   DO WHILE ((.NOT. convergence) .AND. (it < it_max))
2     it = it +1
3     u(sx:ex,sy:ey) = u_new(sx:ex,sy:ey)
4
5     ! Exchange value on the interfaces
6     CALL start_communication( u )
7
8     ! Compute u
9     CALL calcul( u, u_new, sx+1, ex-1, sy+1, ey-1)
10
11    CALL end_communication( u )
12
13    ! North
14    CALL calcul( u, u_new, sx, sx, sy, ey)
15    ! South
16    CALL calcul( u, u_new, ex, ex, sy, ey)
17    ! West
18    CALL calcul( u, u_new, sx, ex, sy, sy)
19    ! East
20    CALL calcul( u, u_new, sx, ex, ey, ey)
21
22    ! Compute global error
23    diffnorm =  global_error (u, u_new)
24
25    convergence = ( diffnorm < eps )
26
27  END DO
```



# MPI: Collective Communication

**General concepts**

- Collective communications are making a series of point-to-point calls hidden to the user within one single subroutine
- A collective communication always involves all processes within the communicator.
- A collective communication is blocking. It is finished when all the necessary point-to-point communications are completed.
- No need to add a barrier.
- No need to specify tags.

# MPI: Collective Communication

**Types of collectives**

- Global synchronization (wait for all processors to arrive): **`MPI_BARRIER()`**
- Collective transfer of fixed size data:
  - Send data from one process to all other: **`MPI_BCAST()`**
  - Split data from one process into all other: **`MPI_SCATTER()`**
  - Collect data from all processes into one: **`MPI_GATHER()`**
  - Same but collect into all: **`MPI_ALLGATHER()`, `MPI_ALLTOALL()`**
- Collective transfer with variable size data **`MPI_BCASTV()`, `MPI_SCATTERV()`, `MPI_GATHERV()`, `MPI_ALLGATHERV()`, `MPI_ALLTOALLV()`**
- Collective transfer plus additional operation on the data (MAX, MIN, +, *…) **`MPI_REDUCE()`, `MPI_ALLREDUCE()`**

# MPI Scatter

```
1  program scatter
2    use mpi
3    implicit none
4
5    integer, parameter                :: nb_values=8
6    integer                           :: nb_procs,rank,block_length,i,code
7    real, allocatable, dimension(:) :: values,data
8
9    call MPI_INIT(code)
10   call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
11   call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
12   block_length=nb_values/nb_procs
13   allocate(data(block_length))
14
15   if (rank == 2) then
16      allocate(values(nb_values))
17      values(:)=(/(1000.+i,i=1,nb_values)/)
18      print *,'I, process ',rank,'send my values  array : ',&
19              values(1:nb_values)
20   end if
21
22   call MPI_SCATTER(values,block_length,MPI_REAL,data,block_length, &
23                    MPI_REAL,2,MPI_COMM_WORLD,code)
24   print *,'I, process ',rank,', received ', data(1:block_length), &
25           ' of process 2'
26   call MPI_FINALIZE(code)
27
28 end program scatter
```

```
> mpiexec -n 4 scatter
I, process 2 send my values array :
1001.  1002.  1003.  1004.  1005.  1006.  1007.  1008.

I, process 0, received 1001.  1002. of process 2
I, process 1, received 1003.  1004. of process 2
I, process 3, received 1007.  1008. of process 2
I, process 2, received 1005.  1006. of process 2
```

# MPI All Reduce

```
1  program allreduce
2
3    use mpi
4    implicit none
5
6    integer :: nb_procs,rank,value,product,code
7
8    call MPI_INIT(code)
9    call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_procs,code)
10   call MPI_COMM_RANK(MPI_COMM_WORLD,rank,code)
11
12   if (rank == 0) then
13      value=10
14   else
15      value=rank
16   endif
17
18   call MPI_ALLREDUCE(value,product,1,MPI_INTEGER,MPI_PROD,MPI_COMM_WORLD,code)
19
20   print *,'I,process ',rank,', received the value of the global product ', product
21
22   call MPI_FINALIZE(code)
23
24 end program allreduce
```

```
> mpiexec -n 7 allreduce

I, process 6, received the value of the global product 7200
I, process 2, received the value of the global product 7200
I, process 0, received the value of the global product 7200
I, process 4, received the value of the global product 7200
I, process 5, received the value of the global product 7200
I, process 3, received the value of the global product 7200
I, process 1, received the value of the global product 7200
```

# OpenMP versus MPI

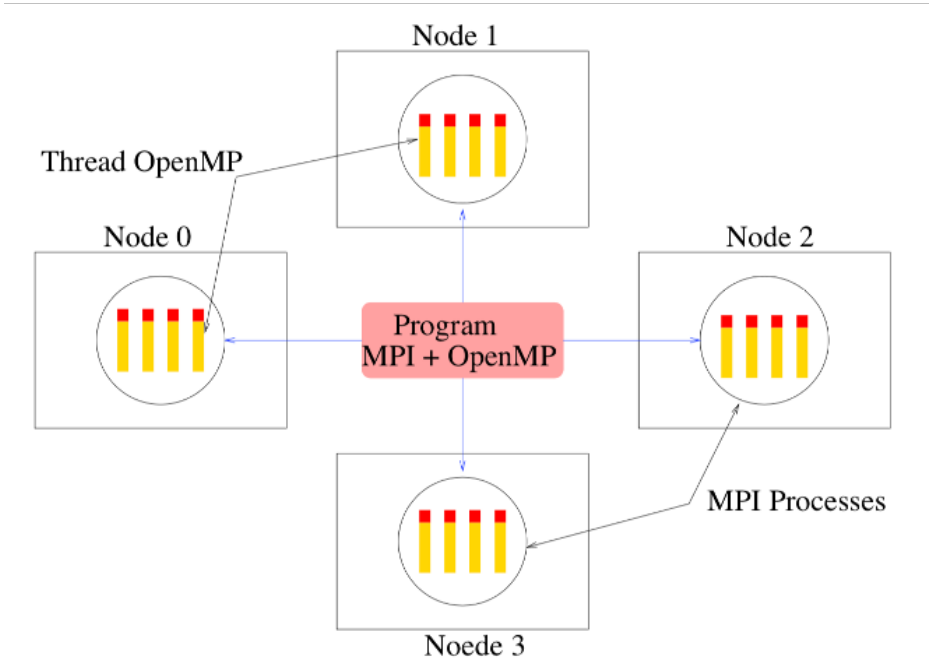
MPI is a multi-process model, for which communication between processes is explicit and under the responsibility of the programmer.

OpenMP is a multi-thread model, within a single process. Communication between threads is implicit . The management of communication is under the responsibility of the compiler (and the operating system).

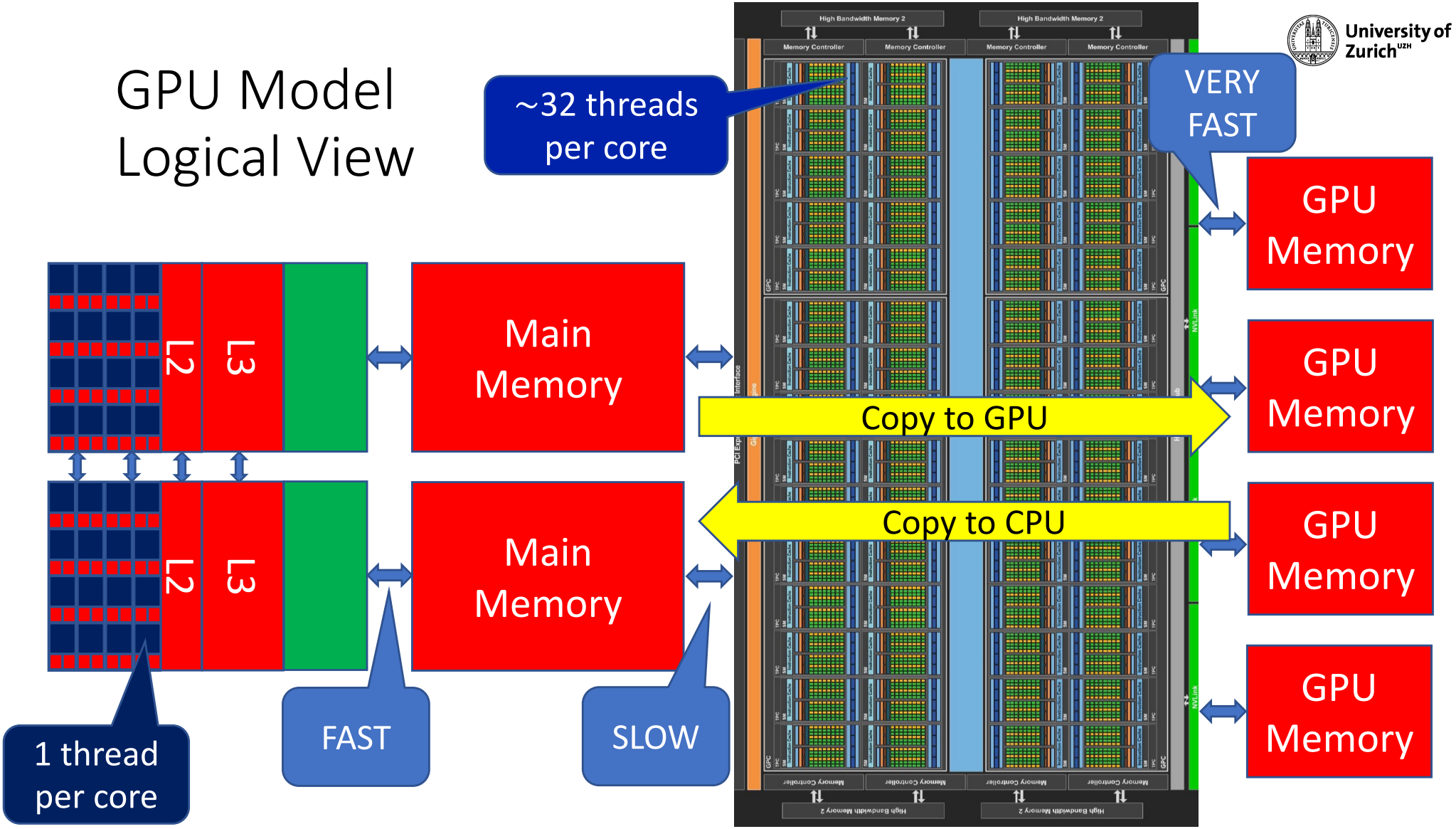
MPI is used on distributed memory architectures (clusters with Infiniband network).

OpenMP is used on shared-memory, multi-core architectures.

On a cluster of many large shared-memory nodes, the hybrid approach (OpenMP within nodes and MPI across nodes) can be optimal



# Graphical Processing Units (GPU)



# Graphical Processing Units (GPU)

**GPU programming with CUDA:**
NVIDIA GPUs are usually programmed using the vendor proprietary language CUDA.
It is interfaced with C and C++ to create GPU kernels.
Warning: CUDA requires a complete rewrite of the computing intensive sections of the code. Lack of portability. Extracting maximum performance is difficult.

**GPU programming with OpenACC:**
Similar to OpenMP directives to load data on and retrieve data from the GPU and translate C and Fortran code into GPU kernels.
Better portability but usually requires also a rewrite of the routines.
Available on the PGI compiler suite.

**GPU programming with OpenMP:**
The newest OpenMP version (4.5) allows for offloading of parallel sections to the GPU using the target directive.
Implemented already in GNU, Intel and PGI.

**GPU programming with new generation of compilers and frameworks:**
The goal is to hide the dirty details to the programmer.
Legion is an example of compiler. Kokkos is an example of framework.

# Conclusion

**Parallel computing methods:**

- MPI is still the dominant programming technique
- Hybrid OpenMP/MPI approach most effective on supercomputers
- GPU programming develops quickly
  - CUDA
  - OpenACC and OpenMP
- Message Passing directly within the GPU
- New specific parallel programming languages are developed:
  - Co-array Fortran, PGAS, X10, Chapel…
- New runtime systems to handle task-based parallelism:
  - Charm++, HPX, Kokkos