
Introduction to High-Performance Computing

Giorgio Amati
Alessandro Ceci

Corso di dottorato in Ingegneria Aeronautica e Spaziale 2025
g.amati@cineca.it/g.amaticode@gmail.com
alessandro.ceci@uniroma1.it

Agenda

- ✓ **HPC: What is it?**
 - ✓ **Hardware: how it works**
 - ✓ **Algorithm vs. Implementation**
 - ✓ **Compiler + Floating point + I/O**
 - ✓ **HW & Parallel Paradigm**
 - **Shared Memory parallelization: openmp et al.**
 - **GPU programming**
 - **Distributed Memory parallelization: MPI**
 - ✓ **Conclusions & Comments**
-

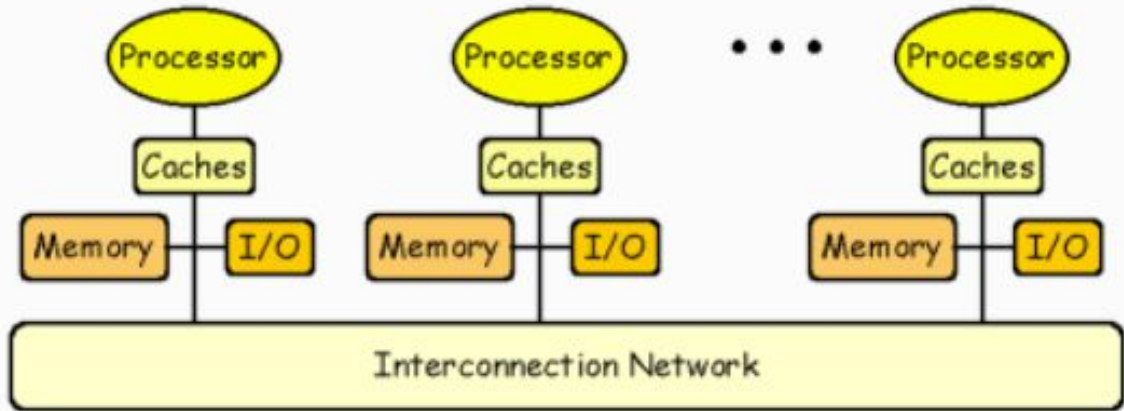
HPC: what it is?

- ✓ These are the main skills for efficient HPC



Distributed Memory Machine

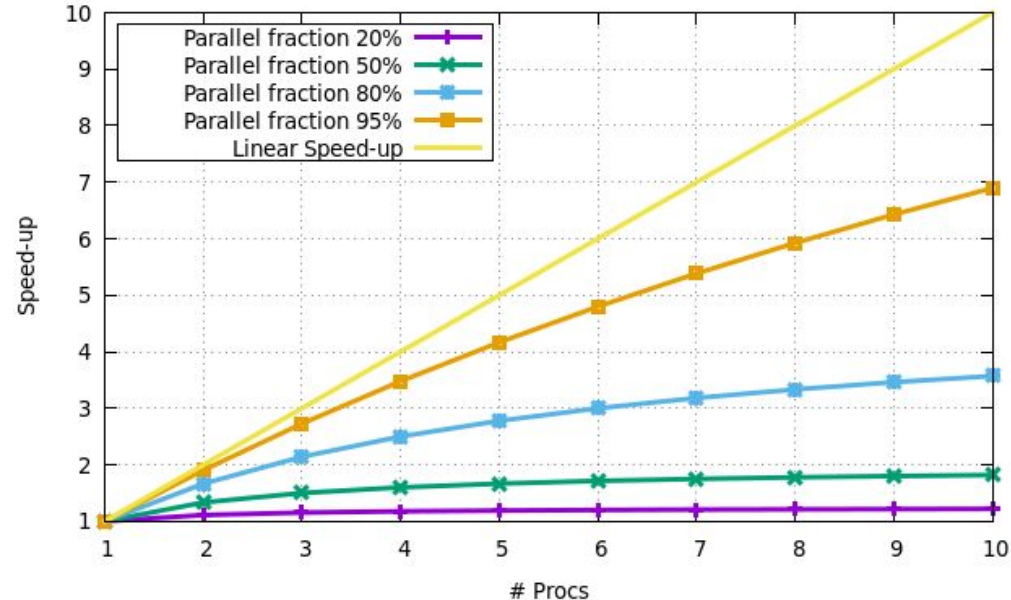
- ✓ Distributed memory refers to a computing system in which each processor (or a node) has its memory. Computational tasks efficiently operate with local data, but when remote data is required, the task must communicate (using explicit messages) with remote processors to transfer the right data.



Amdhal's law

- ✓ F = parallel fraction of the code ($F < 1,00$)
- ✓ N = #of processors
- ✓ S = velocity increment (Speed-up)

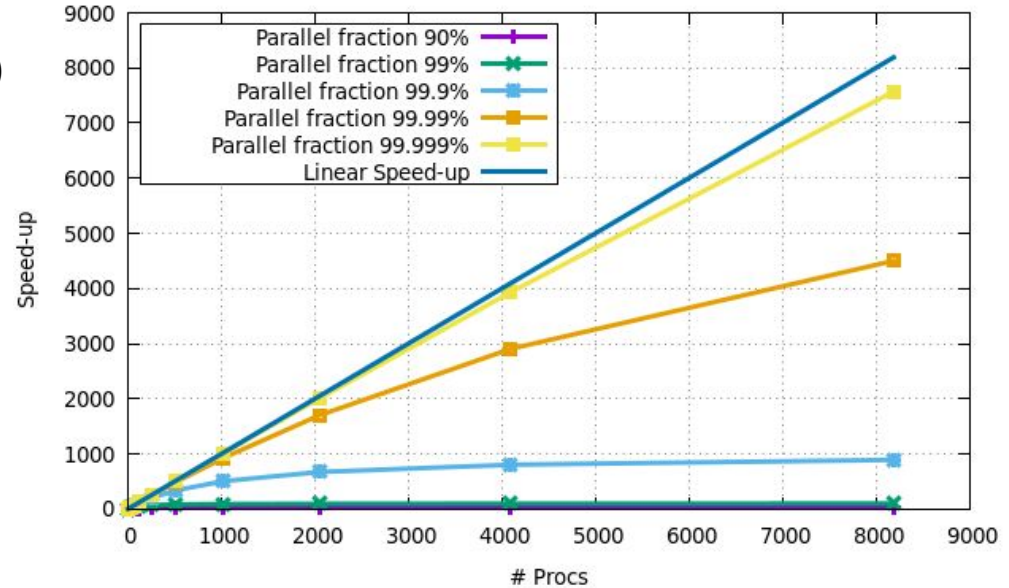
$$S = \frac{1}{(1 - F) + \frac{F}{N}}$$



Amdhal's law

- ✓ F = parallel fraction of the code ($F < 1,00$)
- ✓ N = #of processors
- ✓ S = velocity increment (Speed-up)

$$S = \frac{1}{(1 - F) + \frac{F}{N}}$$



- ✓ Any hint?

Heterogeneous Machine/1

- ✓ Real world systems are “heterogenous”.
- ✓ Up to three different level of parallelization to manage
- ✓ Usually they are a cluster (i.e. distributed memory system) of nodes
 - Each of them is a shared memory system of cores (up to 128 or more)
 - Eventually with GPUs (up to 8)

For example Leonardo has

- ✓ 3456 Nodes, interconnect via an infiniband network at 200Gb/s
 - Each node has 32 Core with 512GB Ram
 - Each node has 4 GPU with 64GB HBM

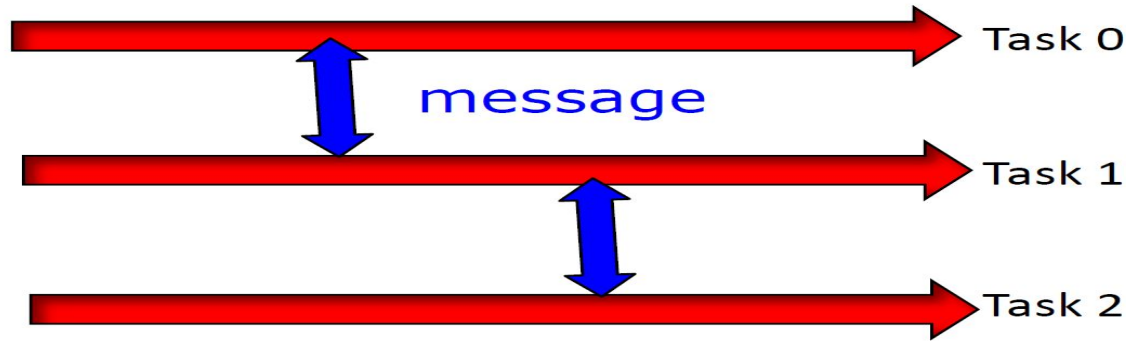
How to handle this “complexity”???

Heterogeneous Machine/2

How to handle this “complexity”???

- ✓ Shared Memory parallelism
 - OpenMP (CPU)
 - OpenACC (GPU)
 - OpenMP Offload (GPU)
 - Cuda/Cudafortran/HIP (GPU)
 - ...
 - ✓ Distributed memory parallelism
 - MPI
 - ...
-

Message Passing Parallelism



Different processes (tasks) that:

- ✓ Run independently one from the other
 - ✓ each task has its own memory
 - ✓ data exchange & synchronization is done via messages
 - ✓ **no risk of “race conditions”**
 - ✓ **risk of “deadlock”**
-

Message Passing

- ✓ Unlike the shared memory model, resources are local;
 - ✓ Each process operates in its own environment (logical address space) and communication occurs via the exchange of messages;
 - ✓ Messages can be instructions, data or synchronisation signals;
 - ✓ The message passing scheme can also be implemented on shared memory architectures but obviously Shared Memory doesn't work on Distributed Memory Systems
-

MPI in few slides

- ✓ Message passing standard ([started in 1991](#))
 - MPI 2.2 (2009) has 647 pages....
 - huge number of routine/function et al...
 - point to point communications
 - collective communications
 - ✓ N different and independent workers (**tasks**): each one does its own work
 - ✓ **Programmer has to care of data movement between tasks and data coherence and synchronization between tasks**
 - ✓ Performance deeply depends on the mpi library used
 - ✓ send/receive model (i.e. post office model)
 - Send a message in an envelope with an address of the receiver
 - ✓ MPI 2.0 introduce MPI-IO operations
 - ✓ MPI 3.0 introduce many advanced features (e.g. one-sided communications)
 - But it still not fully supported from different mpi libraries
-

Message Passing

✓ Advantages

- Communications hardware and software are important components of HPC system and often very highly optimised;
- Portable and scalable;
- Long history (many applications already written for it);
- **Can overlap communication with computation**

✓ Drawbacks

- Explicit nature of message-passing is error-prone and discourages frequent communications;
 - Most serial programs need to be completely re-written;
 - High memory overheads.
 - **No fault tolerant**
 - **Deadlock**
-

classical MPI example

✓ Hello world!!!!!!

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

main(int argc, char **argv)
{
    int node;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &node);
    printf("Hello World from Node %d\n",node);
    MPI_Finalize();
}
```

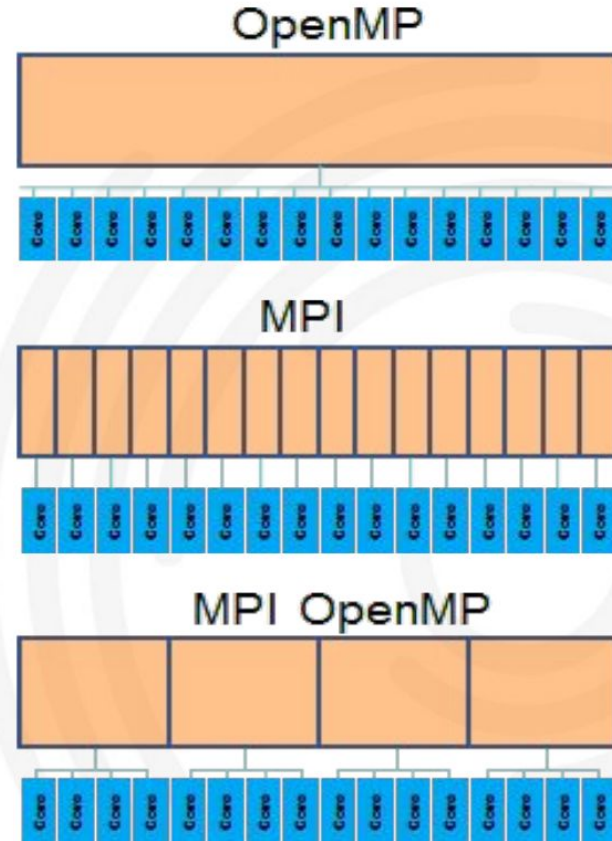
✓ A little more complex in programming.....

- ✓ With MPI+X mixing of different programming languages are presented:
 - MPI+OpenMP
 - MPI+OpenACC
 - MPI+OpenMP-Offload
 - MPI+OpenMP+OpenACC
 - ,,,,

.....

Just a recap

- ✓ for a single node we can have different approaches



post office “model”

- ✓ To send data between task you need
 - an envelope
 - the sender
 - the receiver
 - the “payload”
 - a tag (an identifier)
 - a communicator

```
call mpi_send(temp(1), 18, MPI_INTEGER, taskid, tag,  
MPI_COMM_WORLD, ierr)  
...  
call mpi_recv(temp(1), 18, MPI_INTEGER, taskid, tag,  
MPI_COMM_WORLD, ierr)
```


Communicators/1

- ✓ MPI Communicator link all tasks
 - Default: `MPI_COMM_WORLD`
- ✓ can be re-defined
 - e.g.: select all the odd/even task

```
! set the gpu to the task id
call MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, &
                        MPI_INFO_NULL, localcomm, ierr)
call MPI_Comm_rank(localcomm, mydev, ierr)
call MPI_get_processor_name(hname,len,ierr)
call acc_set_device_num(mydev,acc_device_nvidia)
write(6,*) "INFO: rank",myrank," GPU",mydev, "node ", hname
```

Communicators/2

- ✓ MPI Communicator link all tasks
 - Default: `MPI_COMM_WORLD`
 - derived: `localcomm`

```
...
INFO: rank      340  GPU      0 node lrdn1115.leonar
INFO: rank      390  GPU      2 node lrdn1233.leonar
INFO: rank      391  GPU      3 node lrdn1233.leonar
INFO: rank      388  GPU      0 node lrdn1233.leonar
INFO: rank      389  GPU      1 node lrdn1233.leonar
INFO: rank      342  GPU      2 node lrdn1115.leonar
INFO: rank      120  GPU      0 node lrdn0494.leonar
INFO: rank      122  GPU      2 node lrdn0494.leonar
INFO: rank      127  GPU      3 node lrdn0506.leonar
...
```

Topology

- ✓ You can create a cartesian grid (3D) of task and “easy” understand yours neighbours

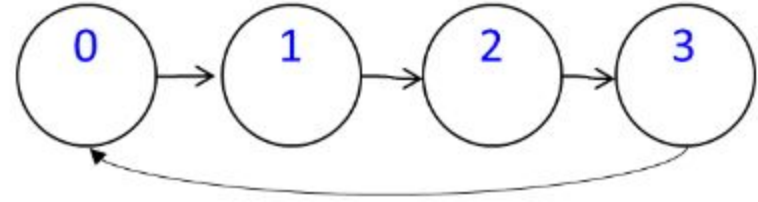
```
! building virtual topology
call MPI_cart_create(mpi_comm_world, mpid, prgrid, &
                    periodic,rreorder,lbecomm,ierr)

call MPI_comm_rank(lbecomm, myrank, ierr)
call MPI_cart_coords(lbecomm, myrank, mpid, mpicoords, ierr)

call MPI_cart_shift(lbecomm, 0, 1, rear(2), front(2), ierr)
call MPI_cart_shift(lbecomm, 1, 1, left(2), right(2), ierr)
call MPI_cart_shift(lbecomm, 2, 1, down(2), up(2), ierr)
```

sendrecv

- ✓ Classical mpi function (send+recv)
- ✓ e.g. receive from left and send to right....

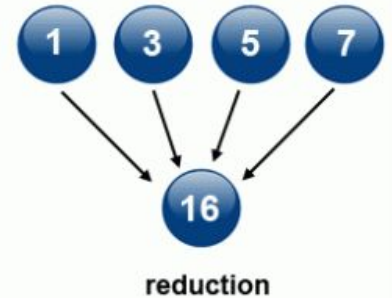
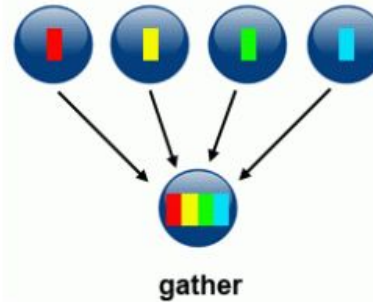
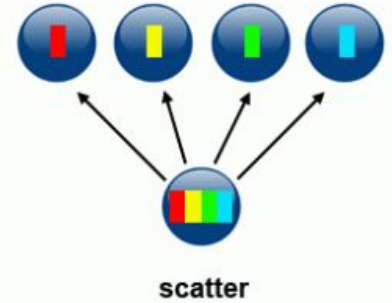
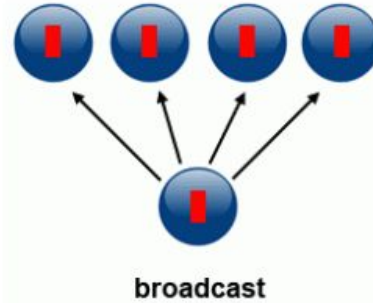


```
int MPI_Sendrecv(const void* buffer_send,
                 int count_send,
                 MPI_Datatype datatype_send,
                 int recipient,
                 int tag_send,
                 void* buffer_recv,
                 int count_recv,
                 MPI_Datatype datatype_recv,
                 int sender,
                 int tag_recv,
                 MPI_Comm communicator,
                 MPI_Status* status);
```

Collective

Communication involving more than 2 tasks..

- ✓ broadcast
- ✓ scatter
- ✓ gather
- ✓ reduction



An example...

- ✓ One call to `sendrecv`, one for each field to “propagate”
 - 3x3x2 calls every timestep
- ✓ using `mpi-datatype`

```
tag = 04
call mpi_sendrecv(field1(0,0,n), 1, xyplane, up(2), tag,      &
                  field1(0,0,0), 1, xyplane, down(2), tag,    &
                  lbcomm, status,ierr)

tag = 06
call mpi_sendrecv(field2(0,0,n), 1, xyplane, up(2), tag,      &
                  field2(0,0,0), 1, xyplane, down(2), tag,    &
                  lbcomm, status,ierr)

tag = 07
call mpi_sendrecv(field3(0,0,n), 1, xyplane, up(2), tag,      &
                  field3(0,0,0), 1, xyplane, down(2), tag,    &
                  lbcomm, status,ierr)
```

Step0 (Default)

- ✓ 1024³, 64 tasks
- ✓ 1'000 timestep
- ✓ Intel Intel(R) Xeon(R) Platinum 8480+ (Sapphire Rapids)

```
# Time for section
# loop    time    0.750499E+02    0.750508E+02
# comp.    time    0.560540E+02    0.559492E+02
# diag.    time    0.000000E+00    0.000000E+00
# MPI      time    0.189785E+02    0.190547E+02
#-----
# Ratio
# Ratio Coll    0.747    0.745
# Ratio Dg.     0.000    0.000
# Ratio MPI     0.253    0.254
# Check         0.000    0.001
#-----
# Z-MPI time, BW (MB/s) --> 0.828125    306.6222
# Y-MPI time, BW (MB/s) --> 2.097656    121.0501
# X-MPI time, BW (MB/s) --> 15.84766    16.02265
```

Step1

- ✓ One call to `sendrecv`, one for each field to “propagate”
 - 3x3x2 calls every timestep
 - cache “unfriendly”
- ✓ no mpi-datatype, explicit packing/unpacking

```
      do k = 0,n+1
        do j = 0,m+1
          bufferXIN(j,k)=field1(1,j,k)
        enddo
      enddo
!
      call mpi_sendrecv(bufferXIN(0,0),msgsizeX,MYMPIREAL,front(2),tag,&
                        bufferXOUT(0,0),msgsizeX,MYMPIREAL,rear(2),tag,&
                        lbecomm,status,ierr)
!
      do k = 0,n+1
        do j = 0,m+1
          field1(0,j,k) = bufferXOUT(j,k)
        enddo
      enddo
```

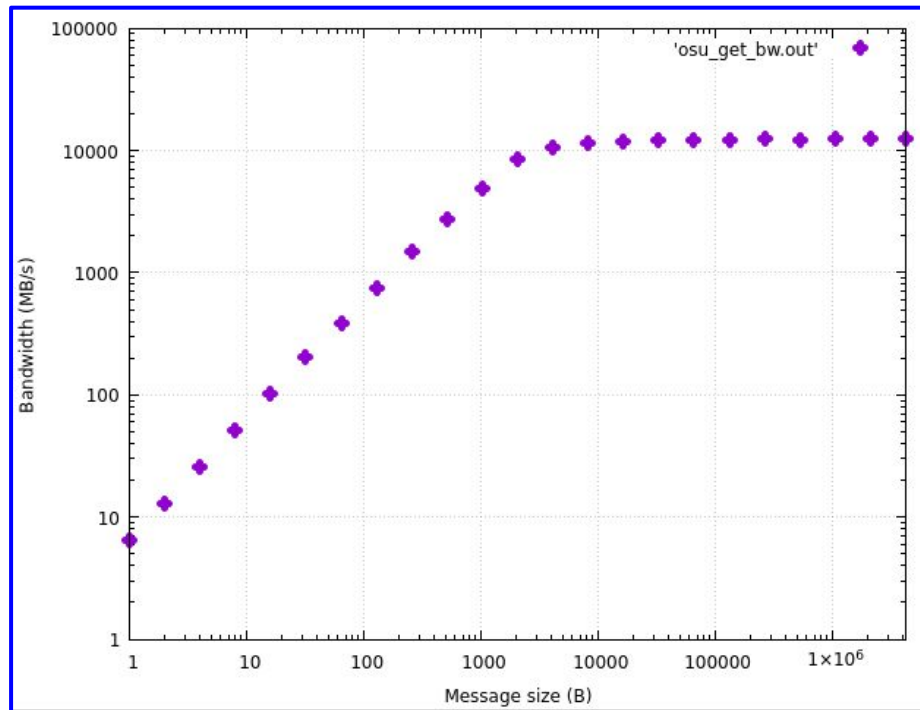

Step1

- ✓ 1024^3, 64 tasks
- ✓ 1'000 timestep
- ✓ Intel Intel(R) Xeon(R) Platinum 8480+ (Sapphire Rapids)

```
# Time for section
# loop      time    0.727114E+02    0.727109E+02
# comp.     time    0.558754E+02    0.558281E+02
# diag.     time    0.000000E+00    0.000000E+00
# MPI       time    0.168161E+02    0.167930E+02
#-----
# Ratio
# Ratio Coll  0.768    0.768
# Ratio Dg.   0.000    0.000
# Ratio MPI   0.231    0.231
# Check       0.000    0.001
#-----
# Z-MPI time, BW (MB/s) --> 0.648437    391.5898
# Y-MPI time, BW (MB/s) --> 2.125000    119.4925
# X-MPI time, BW (MB/s) --> 13.79297    18.40949
```

MPI bandwidth

- ✓ **MPI latency:** is the time it takes to send a 0-size message (overhead)
- ✓ One single big MPI message is better than many little MPI messages



Step2

- ✓ One call to **sendrecv**, one for all three fields to “propagate”
 - 3x1x2 calls every time step
- ✓ no mpi-datatype, explicit packing/unpacking

```
!
  msgsizeY = (l+2)*(n+2)*3
!
  do k = 0,n+1
    do i = 0,l+1
      bufferYIN(i,k,1)=field1(i,m,k)
      bufferYIN(i,k,2)=field2(i,m,k)
      bufferYIN(i,k,3)=field3(i,m,k)
    enddo
  enddo
!
  call mpi_sendrecv(bufferYIN(0,0,1),msgsizeY,MYMPIREAL,right(2),tag,&
                    bufferYOUT(0,0,1),msgsizeY,MYMPIREAL,left(2),tag,&
                    lbecomm,status,ierr)
!
```

Performance figure (CPU)

- ✓ 1024³,
- ✓ 64 tasks
- ✓ 1'000 timestep
- ✓ Intel Sapphire Rapids

Version	HW	Time	MPI	Comp
Step0	CPU	75''	25%	74%
Step1	CPU	72''	23%	77%
Step2	CPU	72''	22%	78%

Step3

- ✓ One call to **sendrecv**, one for all three fields to “propagate”
 - 3x1x2 calls every time step
- ✓ no mpi-datatype, explicit packing/unpacking, ported to GPU

```
!$acc kernels
  do k = 0,n+1
    do i = 0,l+1
      bufferYIN(i,k,1)=field1(i,m,k)
      bufferYIN(i,k,2)=field2(i,m,k)
      bufferYIN(i,k,3)=field3(i,m,k)
    enddo
  enddo
!$acc end kernels
!
  call mpi_sendrecv(bufferYIN(0,0,1),msgsizeY,MYMPIREAL,right(2),tag,&
                    bufferYOUT(0,0,1),msgsizeY,MYMPIREAL,left(2),tag,&
                    lbcomm,status,ierr)
!
—
```

Step3

- ✓ 1024³, 8 tasks
- ✓ 10'000 timestep
- ✓ Nvidia A100@64GB (Leonardo)

```
# Time for section
# loop    time    0.119129E+03    0.119129E+03
# comp.   time    0.479769E+02    0.480273E+02
# diag.   time    0.000000E+00    0.000000E+00
# MPI     time    0.654978E+02    0.654102E+02
#-----
# Ratio
# Ratio Coll    0.403    0.403
# Ratio Dg.     0.000    0.000
# Ratio MPI     0.550    0.549
# Check        0.047    0.048
#-----
# Z-MPI time, BW (MB/s) --> 19.58984    514.4644
# Y-MPI time, BW (MB/s) --> 19.83594    508.0817
# X-MPI time, BW (MB/s) --> 20.90625    482.0701
```

Step4

- ✓ One call to `sendrecv`, one for all three fields to “propagate”
 - 3x1x2 calls every time step
- ✓ no mpi-datatype, explicit packing/unpacking, ported to GPU
- ✓ Cuda-aware MPI

```
!$acc kernels
  do k = 0,n+1
    do i = 0,l+1
      bufferYIN(i,k,1)=field1(i,m,k)
      bufferYIN(i,k,2)=field2(i,m,k)
      bufferYIN(i,k,3)=field3(i,m,k)
    enddo
  enddo
!$acc end kernels
!
!$acc host_data use_device(bufferYIN,bufferYOUT)
  call mpi_sendrecv(bufferYIN(0,0,1),msgsizeY,MYMPIREAL,right(2),tag,&
                    bufferYOUT(0,0,1),msgsizeY,MYMPIREAL,left(2),tag,&
                    lbecomm,status,ierr)
!$acc end host_data
```

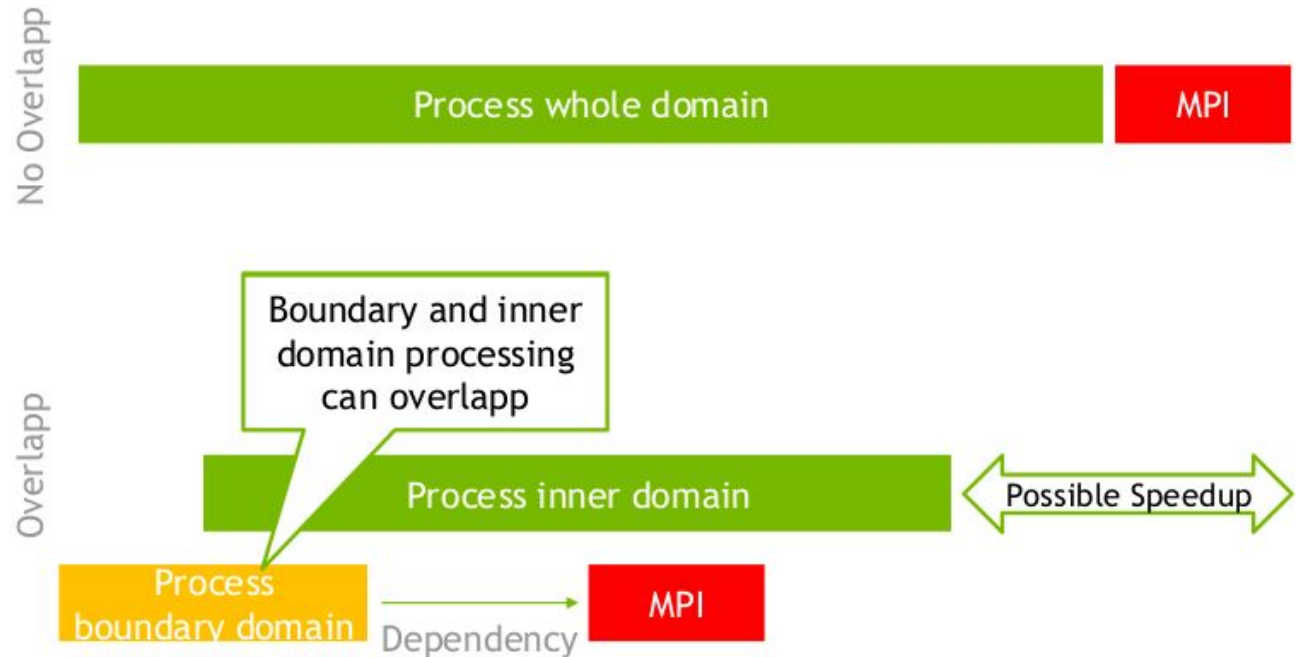
Step4

- ✓ 1024^3 , 8 tasks
- ✓ 10'000 timestep
- ✓ Nvidia A100@64GB (Leonardo)

```
# Time for section
# loop    time    0.730247E+02    0.730234E+02
# comp.   time    0.480949E+02    0.476836E+02
# diag.   time    0.000000E+00    0.000000E+00
# MPI     time    0.193703E+02    0.197539E+02
#-----
# Ratio
# Ratio Coll    0.659    0.653
# Ratio Dg.     0.000    0.000
# Ratio MPI     0.265    0.271
# Check        0.076    0.076
#-----
# Z-MPI time, BW (MB/s) --> 1.667969    6042.246
# Y-MPI time, BW (MB/s) --> 1.875000    5375.081
# X-MPI time, BW (MB/s) --> 15.82422    636.8894
```


Overlapping computation and communication...

- ✓ Communication and computation are independent systems
- ✓ Can be “overlapped”




Overlapping everywhere....

- ✓ Recap: Also with GPU....


Overlapping data transfer with CPU computation:

```
istat = cudaMemcpyAsync(a_d, a_h, N, 0) }  
call kernel<<grid, block>>>(a_d) }  
call cpuFunction(b) ← overlapped
```



Overlapping data transfer and GPU computation:

```
integer (kind=cuda_stream_kind) :: stream1, stream2  
...  
istat = cudaStreamCreate(stream1)  
istat = cudaStreamCreate(stream2)  
istat = cudaMemcpyAsync(a_d, a_h, N, stream1) ← overlapped  
call kernel<<grid, block, 0, stream2>>>(b_d) ← overlapped
```



Step5

- ✓ Using non blocking comms
 - `isend/recv`
 - 3x1x2 calls every time step
- ✓ overlapping some computations via a matrix (border/bulk points)
- ✓ asynchronous
- ✓ More simple to manage borders

```
! First pack data ...
! Second send data ...
! Third receive data ...
! do something (bulk only)
! Fourth unpack data ...
! Fifth wait ...
! do something (border only)
```

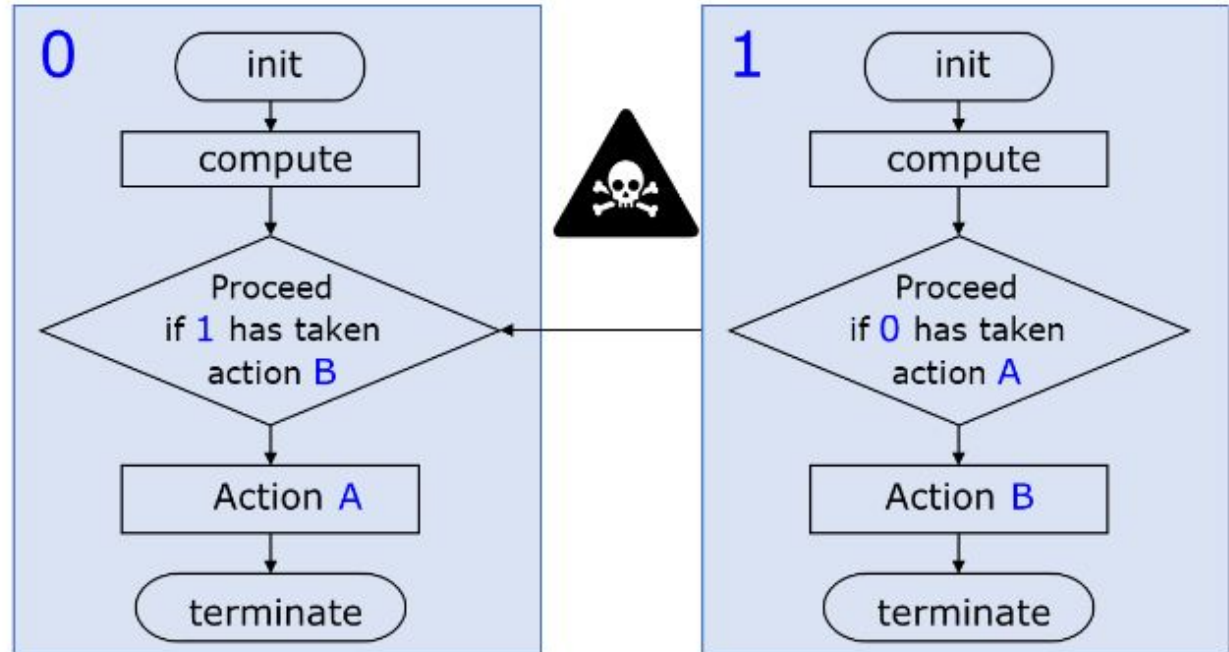
Step5

```
    tag = 10
!$acc host_data use_device(bufferXINM)
    call mpi_isend(bufferXINM(0,0,1),msgsizeX,MYMPIREAL,rear(2),tag, &
                    lbecomm, reqs_rear(1), ierr)
!$acc end host_data
!
< do something else > ! not using bufferXINM
!
    tag = 11
!
!$acc host_data use_device(bufferXOUTP)
    call mpi_recv(bufferXOUTP(0,0,1),msgsizeX,MYMPIREAL,front(2),tag,&
                    lbecomm, status_front, ierr)
!$acc end host_data
...
! Fourth wait...
    call mpi_wait(reqs_front(1), status_front, ierr)
```

Deadlock

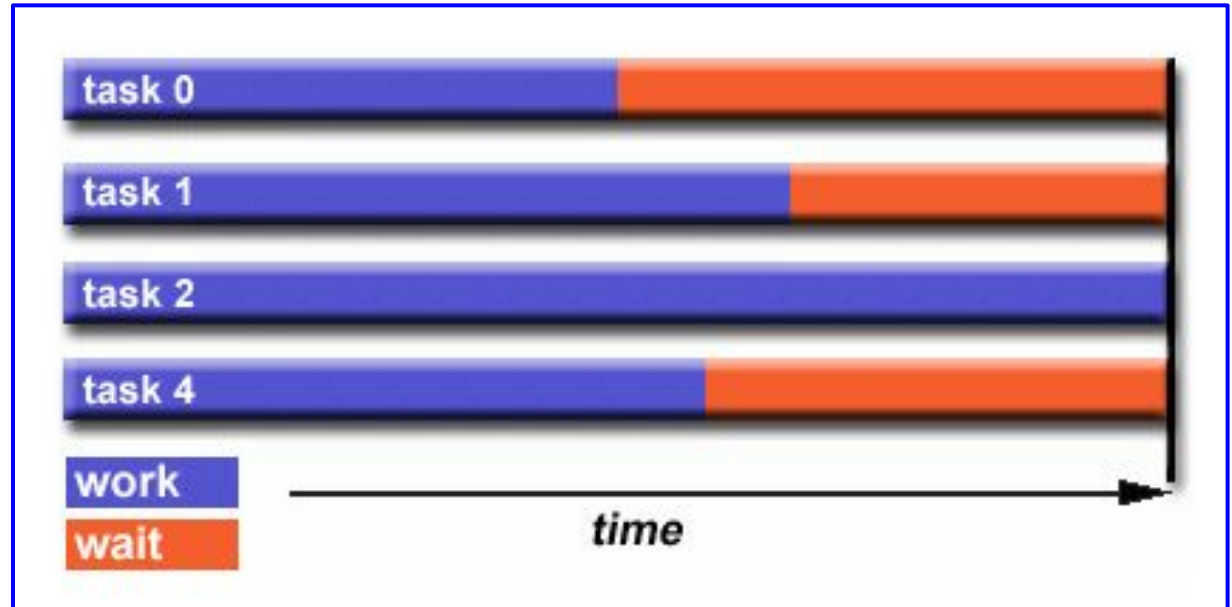
Classical error: one or more tasks are waiting for something that will never happen.

- ✓ Red/Black
- ✓ Even/odd



Load Balancing

- ✓ Keep similar “load” between tasks
- ✓ use domain decomposition programs (e.g. scotch)



some comments

- ✓ Minimize communications
 - ✓ Pack communications
 - ✓ Overlap as possible communication and computation
 - ✓ Always get info about MPI timing
 - ✓ Track mpi overhead (e.g. [MPIp](#))
 - ✓ ...
-

Different Level of optimization

✓ **Single core optimization**

- Vectorization
- Data access
- Serial Optimized libraries
- Compiler optimization

✓ **Single-node optimizations**

- Intra-node optimization
- Data access
- Shared memory Parallelization/Distributed memory Parallelization
- Offloading
- Shared Memory Optimized Libraries

✓ **Multi-node optimizations**

- Distributed memory optimization (i.e. load balancing)
- Parallel Optimized libraries

✓ **I/O issues**

You cannot skip one single level of optimization!!

Topics NOT covered here

- ✓ MPI-IO
 - ✓ One sided communication
 - ✓ MPI datatype
 - ✓ fault tolerance
 - ✓ [PGAS](#)
-