# CheckMPI-Documentation

Giorgio Amati

January 2024

## Contents

## List of Tables

## List of Figures

## 1 Introduction

This code was developed to exploit MPI communication pattern and to built a *template* to check mpi performance using different HW.
It implements a simple 3D domain decomposition for a periodic domain. It was intended to check different communication patterns and their issues/performance.
In detail we will (try to) explore

- `sendrecv`
- Non-blocking comms
- `mpi datatype`
- handmade packing/unpacking data
- `OpenACC` directives
- `Cuda Aware` MPI
- Communication-Computation overlap
- MPI profiling with `mpiP`

## 2 Code structure

The code is written in `Fortran90` and ask for only unix `make` utility and a Fotratran compiler and a MPI library.
Dynamic allocation is used.

```
.
|- DOC
|- LICENSE
|- README.md
|- RUN
|- SRC
|- TEST
‘- UTIL
```

Where:

- `DOC`: this directory contains documentation.
- `RUN`: in this directory, the exe will be copied.
- `SRC`: this directory contains all the source files.
- `TEST`: in this directory are present some stript for testing the different options.
- `UTIL`: this directory some script and input files are present.

### 2.1 Licence

This code is released under the MIT license.
It can be downloaded from `https://github.com/gamati01/Check_MPI`.

## 3 Test Case

The test case is a *synthetic* test case. The focus is on the efficiency of the communication pattern. It is a:

- 3D domain.
- Periodic boundary conditions.
- 3 different fields to *be propagated* (`field1,field2,field3`).
- The fields are initialized with a complete sinusoidal for each task.
- All the tests are done using single precision unless stated elsewhere.
- The default propagation is
  - from rear to front (direction $x$)
  - from left to right (direction $y$)
  - from bottom to right (direction $z$)
- a *reversed* propagation can be activated

Note that no flops are performed, only "rigid" data movement.
The default propagation is:

```
        do k = n, 1, -1
           do j = m, 1, -1
              do i = l, 1, -1
                 field1(i,j,k) = field1(i-1,j,k)
                 field2(i,j,k) = field2(i,j-1,k)
                 field3(i,j,k) = field3(i,j,k-1)
              end do
           end do
        end do
```

The input file used for the test is the following:

```
&parameters
lx = 600
ly = 600
lz = 600
proc_x = 2
proc_y = 2
proc_z = 2
itfin=15000
icheck=1510    /
```

### 3.1 Validation

The initial condition, a sinusoidal wave, is the same for each task, so the profiles should be *exactly the same* independently for all the tests: this means that the UNIX command `md5sum` should return the same hash.

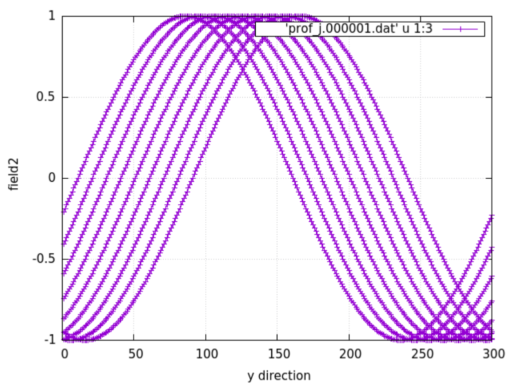

Figure 1: Validation for variable field2.

```
md5sum RUN_STEP*/prof_i*0.dat
3cdb4eef4d04a99e786046dead098237  RUN_STEP0/prof_i.000000.dat
3cdb4eef4d04a99e786046dead098237  RUN_STEP1/prof_i.000000.dat
3cdb4eef4d04a99e786046dead098237  RUN_STEP2/prof_i.000000.dat
3cdb4eef4d04a99e786046dead098237  RUN_STEP3/prof_i.000000.dat
3cdb4eef4d04a99e786046dead098237  RUN_STEP4/prof_i.000000.dat
...
```

In fig. 1 the value of `field2` is reported.

## 4 Code versions

At the moment 9 different versions are available. They can be selected at compile time via pre-processing flag syntax.
The syntax, e.g. to choose `Step6` version, is:

```
make STEP6=1
```

In detail, these are the different options to choose from, for each task and for each timestep:

- `Step0`: $3 \times 2\times$ `sendrecv` calls per direction and using `mpi datatype`.
- `Step1`: $3 times 2\times$ `sendrecv` calls and explicit pack-unpack (only for $x$ direction), development step.
- `Step2`: $2\times$ `sendrecv` calls and explicit pack-unpack (all directions).
- `Step3`: $2\times$ `sendrecv` calls and explicit pack-unpack (all directions) with `kernel` OpenACC directive.
- `Step4`: $2\times$ `sendrecv` calls and explicit pack-unpack (all directions) with `kernel` OpenACC directive and using `CUDAWARE` MPI calls.
- `Step5`: $2\times$ `isend` and `recv` calls only for $x$ direction and explicit pack-unpack with `kernel` OpenACC directive and using `CUDAWARE` MPI calls (development step).
- `Step6`: like `Step5` but with `isend` and `recv` for all the three directions (development step).
- `Step7`: like `Step6` but with `isend` and `irecv` for all the three directions (development step).
- `Step8`: like `Step7` with communication-computation overlap.
- `Step9`: like `Step7` with communication-computation overlap and `asyn` clause.

In tab. 1 is reported for each step which device is supported and if it is a *develop* version or a production one.

Table 1: GPU/CPU support for different versions

| version | CPU | GPU | production ver. |
|---|---|---|---|
| Step0 | yes | no | yes |
| Step1 | yes | no | no |
| Step2 | yes | no | yes |
| Step3 | yes* | yes | yes |
| Step4 | yes* | yes | yes |
| Step5 | yes* | yes | no |
| Step6 | yes* | yes | no |
| Step7 | yes* | yes | no |
| Step8 | yes* | yes | yes |
| Step9 | yes* | yes | yes |

## 4.1 Step0

It is the starting point version: it uses one different single `sendrecv` call for each population to propagate information. It calls `bcond_comm_step0` subroutine.
It means 3 calls to `sendrecv` for each direction $(+/-)$ and each decomposition along $x, y, z$. In total for each timestep and task 6 calls are required.
In the following the calls for positive propagation are shown.

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

It use explicitly `mpi datatype`

```
! yz plane is composed by single points (stride.ne.1)
    call MPI_type_vector((n+2)*(m+2),1,l+2, MYMPIREAL,yzplane,ierr)
    call MPI_type_commit(yzplane,ierr)
!
! xz plane is composed by arrays (the single vector has stride.eq.1)
    call MPI_type_vector(n+2,l+2,(m+2)*(l+2), MYMPIREAL,xzplane,ierr)
    call MPI_type_commit(xzplane,ierr)
!
! xy plane is a contiguous array (stride.eq.1)
    call MPI_type_contiguous((l+2)*(m+2),MYMPIREAL,xyplane,ierr)
    call MPI_type_commit(xyplane,ierr)
```

These are the options activated with the command `make`

```
# default (NVIDIAcompiler)
 CC = mpicc
 FC = mpifort
 FIX = -DPGI
 FOPT = -fast -Mcontiguous -Mnodepchk
 OPT = -fast -Mcontiguous -Mnodepchk
 VER = step0
```

This version is a CPU-only.

## 4.2 Step1

In this version, data for the $x$ direction are explicitly packed/unpacked using a `do` loop. It calls `bcond_comm_step1` subroutine.

```
!
   do k = 0,n+1
      do j = 0,m+1
         bufferXIN(j,k)=field1(l,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
!
```

These are the options activated with the command `make STEP1=1`

```
# default (NVIDIAcompiler)
 CC = mpicc
 FC = mpifort
 FIX = -DPGI
 FOPT = -fast -Mcontiguous -Mnodepchk
 COPT =
 OPT = -fast -Mcontiguous -Mnodepchk
 ifdef STEP1
   FIX=-DSTEP1
   VER=step1
 endif
```

This version is a CPU-only.

## 4.3 Step2

In this version, the data for the three directions is explicitly packed/unpacked using a `do` loop and a single `sendrecv` is used for all the 3 fields to be propagated It calls `bcond_comm_step2` subroutine.

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

These are the options activated with the command `make STEP2=1`

```
# default (NVIDIAcompiler)
 CC = mpicc
 FC = mpifort
 FIX = -DPGI
 FOPT = -fast -Mcontiguous -Mnodepchk
 COPT =
 OPT = -fast -Mcontiguous -Mnodepchk
 ifdef STEP2
   FIX=-DSTEP2
   VER=step2
 endif
```

This version is a CPU-only.

## 4.4 Step3

In this version `OpenACC` directives are used for

- loop inside `do_somethingGPU` subroutine
- pack and unpack fields to propagate

It calls `bcond_comm_step3` subroutine. For the loop inside `do_somethingGPU` subroutine, an explicit copy has to be done to exploit GPU parallelism, it is activated by `FAST` pre-processing flags.

```
ifdef FAST
!$acc kernels
        do k = 0, n+1
           do j = 0, m+1
              do i = 0, l+1
                 temp1(i,j,k) = field1(i,j,k)
                 temp2(i,j,k) = field2(i,j,k)
                 temp3(i,j,k) = field3(i,j,k)
              end do
           end do
        end do
!$acc end kernels
!
!$acc kernels
        do k = 1, n
           do j = 1, m
              do i = 1, l
                 field1(i,j,k) = temp1(i-1,j,k)
                 field2(i,j,k) = temp2(i,j-1,k)
                 field3(i,j,k) = temp3(i,j,k-1)
              end do
           end do
        end do
!$acc end kernels
#else
....
```

For the pack/unpack section these are the `kernels` directives were used

```
!$acc kernels
     do k = 0,n+1
        do j = 0,m+1
           bufferXIN(j,k,1)=field1(l,j,k)
           bufferXIN(j,k,2)=field2(l,j,k)
           bufferXIN(j,k,3)=field3(l,j,k)
        enddo
     enddo
!$acc end kernels
!
   call mpi_sendrecv(bufferXIN(0,0,1),msgsizeX,MYMPIREAL,front(2),    &
                     tag,bufferXOUT(0,0,1),msgsizeX,MYMPIREAL,rear(2),&
                     tag,lbecomm,status,ierr)
!
```

```
!$acc kernels
     do k = 0,n+1
        do j = 0,m+1
           field1(0,j,k) = bufferXOUT(j,k,1)
           field2(0,j,k) = bufferXOUT(j,k,2)
           field3(0,j,k) = bufferXOUT(j,k,3)
        enddo
     enddo
!$acc end kernels
```

These are the options activated with the command `make STEP3=1`

```
# default (NVIDIAcompiler)
 CC = mpicc
 FC = mpifort
 FIX = -DPGI
 FOPT = -fast -Mcontiguous -Mnodepchk
 COPT =
 OPT = -fast -Mcontiguous -Mnodepchk
 ifdef STEP3
   FIX=-DSTEP3 -acc -Minfo=acc -DOPENACC -DFAST
   VER=step3
 endif
```

By default only the CPU version is compiled. To activate the GPU version the correct compile line is:

```
make STEP3=1 GPUENABLE=1
```

## 4.5 Step4

In this version `OpenACC` directives are used to exploit `Cuda-aware` MPI comms. It calls `bcond_comm_step4` subroutine.

```
...
!$acc host_data use_device(bufferZIN,bufferZOUT)
 call mpi_sendrecv(bufferZIN(0,0,1),msgsizeZ,MYMPIREAL,up(2),    &
                tag,bufferZOUT(0,0,1),msgsizeZ,MYMPIREAL,down(2),&
                tag,lbecomm,status,ierr)
!$acc end host_data
....
```

These are the options activated with the command `make STEP4=1`

```
# default (NVIDIAcompiler)
 CC = mpicc
 FC = mpifort
 FIX = -DPGI
 FOPT = -fast -Mcontiguous -Mnodepchk
 OPT = -fast -Mcontiguous -Mnodepchk
 ifdef STEP4
   FIX=-DSTEP4 -acc -Minfo=acc -DOPENACC -DFAST
   VER=step4
 endif
```

By default, only the CPU version is compiled. To activate the GPU version the compile line is:

```
make STEP4=1 GPUENABLE=1
```

## 4.6 Step5

This is a developmentslurm-2480891.out step.
It calls `bcond_comm_step5` subroutine. It is like `Step4` but using `isend` and `recv` subroutines only for the $x$ direction.

```
! First pack data.....
!$acc kernels
      do k = 0,n+1
         do j = 0,m+1
! x+ direction
            bufferXINP(j,k,1)=field1(l,j,k)
            bufferXINP(j,k,2)=field2(l,j,k)
            bufferXINP(j,k,3)=field3(l,j,k)
!
! x- direction
            bufferXINM(j,k,1)=field1(1,j,k)
            bufferXINM(j,k,2)=field2(1,j,k)
            bufferXINM(j,k,3)=field3(1,j,k)
         enddo
      enddo
$acc end kernels
!
! Second send pack data.....
      tag = 11
!$acc host_data use_device(bufferXINP)
      call mpi_isend(bufferXINP(0,0,1),msgsizeX,MYMPIREAL, &
                     front(2),tag,lbecomm,reqs_front(1),ierr)
!$acc end host_data
      tag = 10
!$acc host_data use_device(bufferXINM)
      call mpi_isend(bufferXINM(0,0,1),msgsizex,MYMPIREAL, &
                     rear(2),tag,lbecomm,reqs_rear(1),ierr)
!$acc end host_data
!
      tag = 11
!Third receive data
!$acc host_data use_device(bufferXOUTP)
      call mpi_recv(bufferXOUTP(0,0,1),msgsizeX,MYMPIREAL, &
                    rear(2),tag,lbecomm,status_front, ierr)
!$acc end host_data
      tag = 10
!$acc host_data use_device(bufferXOUTM)
      call mpi_recv(bufferXOUTM(0,0,1),msgsizeX,MYMPIREAL, &
                    front(2),tag,lbecomm,status_rear,ierr)
!$acc end host_data
!
! Fourth unpack data
!$acc kernels
      do k = 0,n+1
         do j = 0,m+1
! x+ direction
            field1(0,j,k) = bufferXOUTP(j,k,1)
```

```
              field2(0,j,k) = bufferXOUTP(j,k,2)
              field3(0,j,k) = bufferXOUTP(j,k,3)
!
! x- direction
              field1(l+1,j,k) = bufferXOUTM(j,k,1)
              field2(l+1,j,k) = bufferXOUTM(j,k,2)
              field3(l+1,j,k) = bufferXOUTM(j,k,3)
           enddo
        enddo
!$acc end kernels
!
! Fifth wait...
        call mpi_wait(reqs_rear(1), status_rear, ierr)
        call mpi_wait(reqs_front(1), status_front, ierr)
!
```

These are the options activated with the command `make STEP5=1`

```
# default (NVIDIAcompiler)
 CC = mpicc
 FC = mpifort
 FIX = -DPGI
 FOPT = -fast -Mcontiguous -Mnodepchk
 OPT = -fast -Mcontiguous -Mnodepchk
 ifdef STEP5
   FIX=-DSTEP5 -acc -Minfo=acc -DOPENACC -DFAST
   VER=step5
 endif
```

By default, only the CPU version is compiled. To activate the GPU version the compile line is:

```
make STEP5=1 GPUENABLE=1
```

## 4.7 Step6

This is a development step. It calls `bcond_comm_step6` subroutine.
It is like `Step6` but with `isend` and `recv` subroutines for all the three directions.
These are the options activated with the command `make STEP6=1`

```
# default (NVIDIAcompiler)
 CC = mpicc
 FC = mpifort
 FIX = -DPGI
 FOPT = -fast -Mcontiguous -Mnodepchk
 OPT = -fast -Mcontiguous -Mnodepchk
 ifdef STEP6
   FIX=-DSTEP6 -acc -Minfo=acc -DOPENACC -DFAST
   VER=step6
 endif
```

By default, only the CPU version is compiled. To activate the GPU version the compile line is:

```
make STEP6=1 GPUENABLE=1
```

## 4.8 Step7

This is a development step. It calls `bcond_comm_step7` subroutine.
It is like `Step6` but with `isend` and `irecv` for all three directions and restructured to allow computation and communication overlap.
It works correctly only if there is more than one task for each direction (i.e. total tasks $\geq 8$). The structure is the following

```
!-----------------------------------------------------------------
! First pack data.....
     call time(tcountZ0)
!$acc kernels
     do j = 0,m+1
        do i = 0,l+1
...
!-----------------------------------------------------------------
! Second receive data
     tag = 34
!$acc host_data use_device(bufferZOUTP)
     call mpi_irecv(bufferZOUTP(0,0,1), msgsizez, MYMPIREAL, &
                    down(2),tag,lbecomm,reqs_up(1),ierr)
!$acc end host_data
...
!-----------------------------------------------------------------
! Third send data.....
     tag = 34
!$acc host_data use_device(bufferZINP)
     call mpi_isend(bufferZINP(0,0,1), msgsizez, MYMPIREAL, &
                    up(2),tag,lbecomm,reqs_up(2),ierr)
!$acc end host_data
...
!-----------------------------------------------------------------
! forth  wait...
     call MPI_Waitall(2,reqs_up   ,MPI_STATUSES_IGNORE, ierr)
...
!-----------------------------------------------------------------
!fifth unpack data
     call time(tcountZ0)
!$acc kernels
     do j = 0,m+1
        do i = 0,l+1
! z+ direction
           temp1(i,j,0)=bufferZOUTP(i,j,1)
           temp2(i,j,0)=bufferZOUTP(i,j,2)
...
```

These are the options activated with the command `make STEP7=1`

```
# default (NVIDIAcompiler)
 CC = mpicc
 FC = mpifort
 FIX = -DPGI
 FOPT = -fast -Mcontiguous -Mnodepchk
 OPT = -fast -Mcontiguous -Mnodepchk
```

```
  ifdef STEP7
    FIX=-DSTEP7 -acc -Minfo=acc -DOPENACC -DFAST
    VER=step7
  endif
```

By default, only the CPU version is compiled. To activate the GPU version the compile line is:

```
make STEP7=1 GPUENABLE=1
```

## 4.9 Step8

In this version, the operations of `do_something` subroutine are split to allow the overlap. It calls `bcond_comm_step8` and `do_somethingGPU_overlap` subroutine.
It works correctly only if there is more than one task for each direction (e.b. total tasks $\geq 8$).
The overlap is done putting the copy of temporary files between the section where data are sent and the wait section:

```
!----------------------------------------------------------------
! Third send data.....
     tag = 34
!$acc host_data use_device(bufferZINP)
     call mpi_isend(bufferZINP(0,0,1), msgsizez, MYMPIREAL, up(2), &
                    tag,lbecomm,reqs_up(2),ierr)
!$acc end host_data
...
!----------------------------------------------------------------
! overlap region
!$acc kernels
        do k = 1, n
           do j = 1, m
              do i = 1, l
                 temp1(i,j,k) = field1(i,j,k)
                 temp2(i,j,k) = field2(i,j,k)
                 temp3(i,j,k) = field3(i,j,k)
              end do
           end do
        end do
!$acc end kernels
!----------------------------------------------------------------
! forth  wait...
     call MPI_Waitall(2,reqs_up   ,MPI_STATUSES_IGNORE, ierr)
...
```

These are the options activated with the command `make STEP8=1`

```
# default (NVIDIAcompiler)
 CC = mpicc
 FC = mpifort
 FIX = -DPGI
 FOPT = -fast -Mcontiguous -Mnodepchk
 OPT = -fast -Mcontiguous -Mnodepchk
 ifdef STEP8
   FIX=-DSTEP8 -acc -Minfo=acc -DOPENACC -DFAST
   OBJ1= bcond.comm.step8.o do_somethingGPU_overlap.o
   VER=step8
 endif
```

By default, only the CPU version is compiled. To activate the GPU version the compile line is:

```
make STEP8=1 GPUENABLE=1
```

## 4.10 Step9

This version is the same of `Step8` with the use of `async` clause for `kernels` directives. It calls `bcond_comm_step8`.
It works correctly only if there is more than one task for each direction (e.b. total tasks $\geq 8$).

```
! overlap region
....
!$acc kernels async
        do k = 0,n+1
           do i = 0,l+1
! y+ direction
              temp1(i,0,k)=bufferYOUTP(i,k,1)
              temp2(i,0,k)=bufferYOUTP(i,k,2)
              temp3(i,0,k)=bufferYOUTP(i,k,3)
!
! y- direction
              temp1(i,m+1,k)=bufferYOUTM(i,k,1)
              temp2(i,m+1,k)=bufferYOUTM(i,k,2)
              temp3(i,m+1,k)=bufferYOUTM(i,k,3)
           enddo
        enddo
!$acc end kernels
...
!acc wait
....
```

This implementation shows his efficiency for more than 64 tasks.
These are the options activated with the command `make STEP9=1`

```
# default (NVIDIAcompiler)
 CC = mpicc
 FC = mpifort
 FIX = -DPGI
 FOPT = -fast -Mcontiguous -Mnodepchk
 OPT = -fast -Mcontiguous -Mnodepchk
 ifdef STEP9
   FIX=-DSTEP9 -acc -Minfo=acc -DOPENACC -DFAST
   OBJ1= bcond.comm.step9.o do_somethingGPU_overlap.o
   VER=step9
 endif
```

To activate the GPU version the compile line is:

```
make STEP9=1 GPUENABLE=1
```

Table 2: Figures for different versions (using single DGX node, with 8 tasks

| `version` | tot | `%COLL` | `%MPI` | `x` | `y` | `z` | type |
|---|---|---|---|---|---|---|---|
| `Step0` | 661” | 53% | 47% | 268” | 15.4” | 10.6” | CPU |
| `Step1` | 484” | 72% | 28% | 101” | 15.4” | 11.0” | CPU |
| `Step2` | XXX” | 72% | 28% | 74” | 19.1” | 13.9” | CPU |
| `Step7` | 680” | 77% | 23% | 26.3” | 23.8” | 23.8” | CPU |
| `Step8` | 1650” | 77% | 23% | 26.3” | 23.8” | 23.8” | CPU |
| `Step3` | 97.5” | 16% | 82% | 26.3” | 23.8” | 23.8” | GPU |
| `Step4` | 27.7” | 59% | 35% | 3.52” | 1.74” | 3.62” | GPU |
| `Step5` | 26.5” | 61% | 32% | 2.51” | 2.04” | 3.25” | GPU |
| `Step6` | 25.4” | 64% | 29% | 2.46” | 1.34” | 3.02” | GPU |
| `Step7` | 24.6” | -% | -% | -” | -” | -” | GPU |
| `Step8` | 24.0” | -% | -% | -” | -” | -” | GPU |
| `Step9` | 21.5” | -% | -% | -” | -” | -” | GPU |

## 5 Performance Figures

To obtain the performance figures these HW has been used:

- DGX
  - CPU: AMD
  - GPU: Nvidia A100@40GB
- Leonardo
  - CPU: Intel Xeon Platinum 8358 CPU @ 2.60GHz, 32 core
  - GPU: Nvidia A100@64GB
  - Compiler: nvhpc–23.1
  - MPI: openmpi/4.1.4–nvhpc–23.1-cuda-11.8

All the figures reported in the tables are obtained from file `task.000000.log`.
In tab 2 are reported some performance figures for a 8-task run on DGX.
Each task uses a $300^3$ gridpoint, $100'000$ iterations were performed, and the decomposition was a $2\times2\times2$. Each task uses a $300^3$ gridpoint, $100'000$ iterations were performed, and the decomposition was a $2 \times 2 \times 2$.
In tab. 3 performance for the different CPU versions, for a 16 tasks test case.

In tab. 4 a scale-up using version `STEP4`, `STEP8` and `STEP9` are reported with respect to different tasks. Each task asks for a $300^3$ grid, $100'000$ iterations were performed, and a cubic decomposition was used.

Table 3: Figures for different versions (using single Leonardo node, with 16 tasks, CPU version. The MPI ratio computed via mpip doesn't include the pack and unpack time as the code default timing.

| version | tot | %COLL | %MPI | %MPI (mpiP) |
|---|---|---|---|---|
| Step0 | 1014” | 75% | 25% | 25.7% |
| Step1 | 914” | 82% | 18% | 5.7% |
| Step2 | 931” | 82% | 18% | 8.3% |
| Step3 | 144” | 30% | 69% | 22.7% |
| Step4 | 58” | 72% | 22% | 7.8% |

Table 4: Figures for scale-up (using Leonardo, with 4 tasks per node)

| proc_x | proc_y | proc_x | STEP4 | STEP7 | STEP8 | STEP9 |
|---|---|---|---|---|---|---|
| 2 | 2 | 2 | 185” | 166” | 160” | 143” |
| 3 | 3 | 3 | 267” | 230” | 230” | 196” |
| 4 | 4 | 4 | 290” | 360” | 280” | 218” |
| 5 | 5 | 5 | 404” | 467” | 285” | 224” |
| 6 | 6 | 6 | 568” | 500” | 291” | 233” |
| 7 | 7 | 7 | 621” | 653” | 268” | 232” |
| 8 | 8 | 8 | 683” | 660” | 285” | 241” |
| 10 | 10 | 10 | 809” | 743” | 288” | 285” |

## 5.1 Comments

## 6 To DO LIST

- `MPIP`
- Testy section

## A Leonardo performance issues

In tab. 5 performance issues using Loenarod are present. For each job, using 1000 GPUs, with a $10 \times 10 \times$ task decomposition, 7 runs using different sizes, from $2000^3$ to $5000^3$ were performed, and always the same executable was used. MPI timings are obtained using both internal timings and `mpiP` library.
These is the input file used

```
&parameters
lx = XXXX
ly = XXXX
lz = XXXX
proc_x = 10
proc_y = 10
proc_z = 10
itfin=100000
icheck=10010   /
```

Total timing are obtained from file `task.000000.log`.
In detail. for 12 different job submission, we get

- 1/12 of the jobs (8.3%) was killed because too slow, order $10x$ slower the expected (note=a)
- 1/12 of the jobs (8.3%) didn't start at all (note=b)
- 7/12 of the jobs (58%) presented a failure performing at least 1 size problem (note=a,c,d,e,f,g,h)
- 10/12 of the jobs (83%) presented at least one run significantly slower then expected
- only 4 jobs performed all the 7 problem sizes correctly, regardless elapsed time.
- only 1 job (2490032) presented expected time. In tab. 6 the loss of performance is shown for the different problem sizes.

The following errors were found for the different problem sizes.

- note a: error for `size=2500`:

  ```
  [lrdn0598:212346:0:212346] ib_mlx5_log.c:177  Transport retry count exceeded on mlx5_0:1/IB (synd 0x15 vend 0x81 hw_synd 0/0)
  [lrdn0598:212346:0:212346] ib_mlx5_log.c:177  DCI QP 0x7ad2 wqe[38318]: RDMA_READ s-- [rqpn 0x19c13 rlid 8163] [rva 0x152f93e00000 rkey 0x5a7b] [va 0x152717eba200 le
  ...
  ```

- note c: error for `size=2500`:

  ```
  mpiP: WARNING: BFD format matching failed[lrdn0514:888046:0:888046] Caught signal 11 (Segmentation fault: address not mapped to object at address 0x90)
  ==== backtrace (tid: 888046) ====
   0 0x0000000000012ce0 __funlockfile()  :0
   1 0x00000000000795c5 mpiP_find_src_loc()  /leonardo/home/userinternal/gamati01/LOCAL_SW/mpiP/pc_lookup.c
  ...
  ```

- note c: error for `size=4500`:

```
mpiP: WARNING: BFD format matching failed[lrdn0514:888600:0:888600] Caught signal 11 (Segmentation fault: address not mapped to object at address 0x90)
==== backtrace (tid: 888600) ====
 0 0x0000000000012ce0 __funlockfile()  :0
 1 0x00000000000795c5 mpiP_find_src_loc()  /leonardo/home/userinternal/gamati01/LOCAL_SW/mpiP/pc_lookup.c:477
...
```

It's worth noting that a little test case (64 GPUs, from $800^3$ to $2000^3$) presents no issue for 7 distinct submissions.
As can be seen in tab.7 the overhead is completely due to MPI.

Table 5: Total loop time for different problem size

| jobid | $2000^3$ | $2500^3$ | $3000^3$ | $3500^3$ | $4000^3$ | $4500^3$ | $5000^3$ | note |
|---|---|---|---|---|---|---|---|---|
| 2489511 | 1209” | n.a. | 1498” | 1638” | - | - | - | a |
| 2489714 | 329” | 185” | 827” | 668” | 503” | 657” | 843” | - |
| 2490032 | 169” | 200” | 274” | 402” | 500” | 676” | 848” | - |
| 2490463 | n.a. | - | - | - | - | - | - | b |
| 2490633 | 195” | n.a. | 332” | 437” | 557” | n.a. | 904” | c |
| 2491872 | n.a. | 332” | 407” | 520” | 646” | 805” | 999” | d |
| 2491873 | 250” | n.a. | 415” | 511” | n.a | 794” | 986” | e |
| 2492037 | 257” | 301” | 395” | 512” | 648” | 791” | 981” | - |
| 2492038 | 300” | 301” | 387” | 592” | 621” | 787” | 963” | - |
| 2492039 | 241” | 302” | 455” | n.a. | 614” | 778” | 966” | f |
| 2493346 | 436” | 586” | 706” | 729” | 934” | n.a | 1375” | g |
| 2493347 | 350” | n.a. | 598” | 762” | 895” | 1181” | 1462” | h |

Table 6: Max,min and ratio time for different problem size

| jobid | $2000^3$ | $2500^3$ | $3000^3$ | $3500^3$ | $4000^3$ | $4500^3$ | $5000^3$ |
|---|---|---|---|---|---|---|---|
| Max | 1209” | 586” | 1498” | 1638” | 934 | 1181” | 1462” |
| Min | 159” | 185” | 274” | 402” | 500” | 657” | 843” |
| Ratio | 7.6x | 3.2x | 5.5x | 4.1” | 1.9” | 1.8x | 1.7x |

Table 7: MPI% for different problem size, using mpiP

| jobid | $2000^3$ | $2500^3$ | $3000^3$ | $3500^3$ | $4000^3$ | $4500^3$ | $5000^3$ |
|---|---|---|---|---|---|---|---|
| 2489511 | 97.6% | n.a. | 94.9% | 92.4% | - | - | - |
| 2489714 | 90.6% | 72.6% | 90.8% | 81.3% | 66.1% | 63.5% | 61.3% |
| 2490032 | 81.4% | 74.6% | 71.4% | 69.0% | 65.8% | 64.5% | 61.6% |
| 2490463 | n.a. | - | - | - | - | - | - |
| 2490633 | 84.2% | n.a. | 76.5% | 71.6% | 69.5% | n.a. | 64.1% |
| 2491872 | n.a. | 84.8% | 80.9% | 76.1% | 73.8% | 70.4% | 67.5% |
| 2491873 | 87.9% | n.a. | 81.3& | 75.7% | n.a | 69.9% | 67.1% |
| 2493347 | 91.3% | n.a. | 87.0% | 83.7% | 81.1% | 1181” | 79.8% |