CheckMPI-Documentation

Giorgio Amati

December 2023

Contents

List of Tables

List of Figures

1 Introduction

This code was developed to exploit MPI comms.
It implements a simple 3D domain decomposition for a periodic box. It was intended to check different communication patterns and their issues.
In details we will explore

- `sendrecv`
- `mpi datatype`
- handmade packing/unpaking data
- `OpenACC` directives
- `Cuda Aware` mpi
- Non-blocking comms
- Communication-Computation overlap

2 Code structure

3 Test Case

The test case is a *sinthetic* test case. It is a:

- 3D flow
- with periodic boundary condition
- 3 fields to propagate, initialized with a complete sinusoidal for each task
- All the tests are done using single precision.
- The propagation is
  - from left to right
  - from rear to front
  - from bottom to right

Note that no flops are performed, only "rigid" data movement.

```
        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, it the same for each task, so the profile are the *the same* independently for all the tests: this means that the UNIX `md5sum` command should return the same hash.

```
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
```

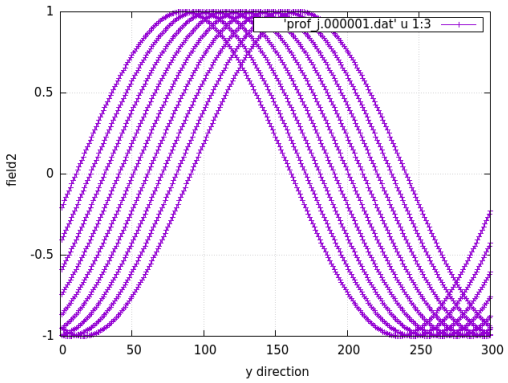

Figure 1: Validation for variable field2.

```
3cdb4eef4d04a99e786046dead098237  RUN_STEP4/prof_i.000000.dat
3cdb4eef4d04a99e786046dead098237  RUN_STEP5/prof_i.000000.dat
```

Table 1: GPU/CPU support for different versions

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

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:

- `Step0`: 3×2 `sendrecv` per task/timestep and `mpi datatype`
- `Step1`: 3×2 `sendrecv` and explicit pack-unpack (only for x direction)
- `Step2`: 2 `sendrecv` and explicit pack-unpack (only all directions)
- `Step3`: 2 `sendrecv` and explicit pack-unpack (only all directions) with OpenACC
- `Step4`: 2 `sendrecv` and explicit pack-unpack (only all directions) with OpenACC and CUDAWARE MPI call
- `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`: communication-computation overlap.
- `Step9`: communication-computation overlap with `async` clause.

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.

```
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 explicity `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, the data for the x direction is 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 use for

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

It calls `bcond_comm_step3` subroutine. For the loop inside `do_somethingGPU` subroutine a copy has to be done to exploit GPU parallelism, activated by `FAST` preprocessing 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 directive 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
```

To activate the GPU version the compile line is:

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

4.5 Step4

In this version `OpenACC` directives are use for 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
 COPT =
 OPT = -fast -Mcontiguous -Mnodepchk
 ifdef STEP4
   FIX=-DSTEP4 -acc -Minfo=acc -DOPENACC -DFAST
   VER=step4
 endif
```

To activate the GPU version the compile line is:

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

4.6 Step5

This is a development step.
It is like `Step4` but using `isend` and `recv` subroutines only for the x direction.

To activate the GPU version the compile line is:

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

4.7 Step6

This is a development step.
It is like `Step6` but with `isend` and `recv` subroutines for all the three directions.

To activate the GPU version the compile line is:

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

4.8 Step7

This is a development step.
It is like `Step7` 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 (e.b. total tasks ≥ 8).

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` is splitted to allow the overlap. It works correctly only if there is more than one task for each direction (e.b. total tasks ≥ 8).

To activate the GPU version the compile line is:

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

4.10 Step9

In this version, the operations of `do_something` is splitted to allow the overlap. It works correctly only if there is more than one task for each direction (e.b. total tasks ≥ 8). Respect `step8` the `async` clause is used for the kernel loops. This implementation shows his efficiency for more than 64 tasks.

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	488”	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

Table 3: Figures for scale-up (using Leonardo, with 4 tasks per node

proc_x	proc_y	proc_x	STEP4	STEP8	STEP9
2	2	2	185”	159”	143”
3	3	3	267”	230”	196”
4	4	4	290”	280”	218”
5	5	5	404”	285”	224”
6	6	6	568”	291”	233”
7	7	7	621”	268”	232”
8	8	8	683”	285”	241”

5 Figures

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. About $100'000$ iterations were performed.
In tab. 3 a scale up using version `STEP4` and `STEP8` are reported with respect different number of tasks. Each task uses a 300^3 gridpoint. About $100'000$ iterations were performed

5.1 Compilers options

5.2 TEST Case Options

5.3 Licence

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

6 Repository Structure

The code is developed to be self-consistent. No external libraries are needed to compile/run the code. The directory structure is the following

- `DOC`: In this directory some documentation
- `RUN`: In this directory the executable file will be created (i.e., `bgk2d.*.x`) after the compilation step.
- `SRC`: In this directory all the source files, with the `Makefile`, are present.
- `UTIL`: In this directory some utility scripts and some input files are present.
- `TEST`: In this directory the four test cases, with some scripts, used for the code validation are present, each in a different directory.
- `BENCH`: In this directory the three benchmark scripts (for single core, single compute node, and GPU) are present, each in a different directory.
- `CI`: In this directory some script to perform a *fast* check for compilation and execution of all possible configurations.

7 How to compile

These are the steps to compile the code:

1. Go in the `SRC` directory.
2. Compile the desired configuration: e.g.,
   `make serial DOUBLE=1 FUSED=1 LDC=1 GNU=1`
   for a lid-driven cavity test in double precision, with the fused implementation and GNU compiler.
3. If the compilation is successful, the exe file, i.e., `bgk2d.serial.x`, will be copied in the `../RUN/` directory.
4. Go in the `RUN` directory.
5. Copy from the `UTIL` directory a `bgk.input` file.
   For the single core run: `../UTIL/bgk.core.input bgk.input`.
6. Run the code: `./bgk2d.serial.x`.

The code uses dynamic allocation, so, once fixed, the test case does not need to be recompiled if you need to vary the size of the simulation box.

7.1 Further Pre-processing Flags

7.2 Input/Output

The code will produce the following file, together with some standard output:

8 Compiler options

8.1 Compiler options for CPU version

- GNU: `-Ofast`
- INTEL: `-O3 -xCORE-AVX512 -mtune=skylake-avx512 -assume contiguous_pointer`
- NVIDIA: `-O2 -Mnodepchk -Mcontiguous`
- AMD: `-O3`
- CRAY: `-O3`

8.2 Compiler options for GPU version

- GNU: `-fopenmp -ftree-parallelize-loops=4`
- INTEL: `-qopenmp`
- NVIDIA:`-stdpar=multicore`
- AMD: `-h thread_do_concurrent`