Task-mapping Library using a Space-filling curve

1 Introduction

A task-mapping library is developed using the space-filling curve to improve the communication performance of parallel applications with structured grids in Cartesian coordinate. The library includes public APIs and tools for application developers (see Table 1). Main functionality of the library is to remap tasks to computational resources along the Hilbert curve. The tools for the performance analysis are accompanied as well. The library is implemented in Fortran and some APIs are also developed in C language.

Fig. 1 illustrates the task-mapping workflow using the library APIs and tools. The workflow has two phases: (a) training and (b) execution phase. In the training phase, the PIR (performance improvement) predictor is trained using the execution profile and fragmentation value. The PIR is defined as the ratio of the elapsed application time between with and without the proposed strategy. In the execution phase, the task-remapping is performed only when the predictor expects improvement ($PIR > 1$). Below we describe details of the task-mapping library and then how to install and run it.

Table 1: Task-mapping Library APIs and Tools.

Name	Functionality	Languages
`remap_tasks_Hilbert_2D(3D)`	Function for Hilbert-curve-based task-mapping for 2D(3D)	Fortran and C
`calculate_fragmentation_2D(3D)`	Function to calculate the fragmentation for 2D(3D)	Fortran
`record_compute_nodes`	Tool to record list of `MPI_ID-hostname` pairs	Fortran
`generate_job_log`	Tool to generate job log including execution info. & fragmentation	Fortran

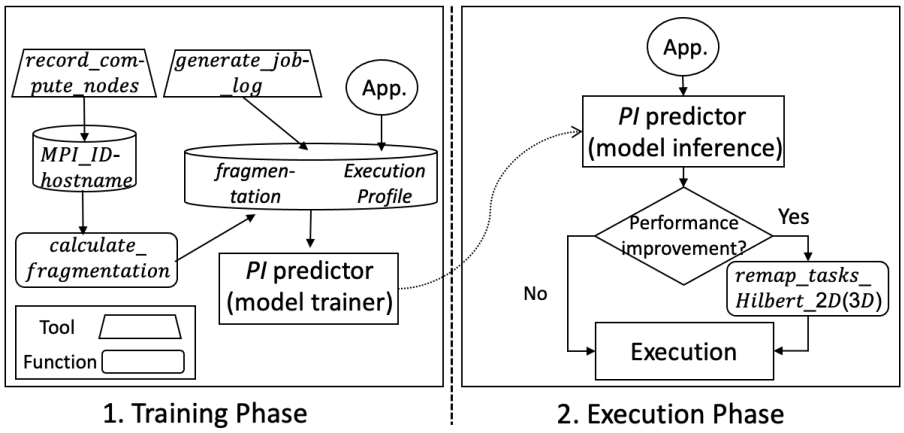

Figure 1: Workflow of performance analysis and execution using the task-mapping library.

2 APIs and Tools

2.1 Functions for Hilbert-curve-based task-mapping

```
/* a task-mapping function in C language */
void remap_tasks_Hilbert_2D(int Nx,
                            int Ny,
                            MPI_Comm *cur_comm,
                            MPI_Comm *new_comm);

! task-mapping functions in Fortran language
subroutine remap_tasks_Hilbert_2D(Nx, Ny, cur_comm, new_comm)
subroutine remap_tasks_Hilbert_3D(Nx, Ny, Nz, cur_comm, new_comm)
```

The `remap_tasks_Hilbert_2D(3D)` functions are provided to application developers for 2D(3D) domain decomposition. The functions could be easily incorporated just after calling `MPI_Init` function. The purpose of the functions is to reassign the tasks (MPI processes) according to the Hilbert-curve-based strategy upon 2D(3D) domain decomposition. They first generate a list of remapped-tasks depending on domain decomposition strategy. For example, as for 2D decomposition, a new order of tasks in the remapping is obtained from the input parameters, i.e, `Nx`, `Ny` and `cur_comm`. The `Nx` and `Ny` specifies the number of processes in each dimension. And the `cur_comm` specifies the current MPI communicator including the list of tasks, e.g. `MPI_COMM_WORLD`. It then creates and returns a new MPI communicator(`new_comm`) from the new order of tasks.

2.2 Functions to calculate the fragmentation

```
! fragmentation-calculating functions in Fortran language
! at edge level switch for 2D domain decomposition
subroutine calculate_fragmentation_2D_edge_level_x(
                    Nx, Ny, fragmentation, hilbert)
subroutine calculate_fragmentation_2D_edge_level_y(
                    Nx, Ny, fragmentation, hilbert)
! at core level switch for 2D domain decomposition
subroutine calculate_fragmentation_2D_core_level_x(
                    Nx, Ny, fragmentation, hilbert)
subroutine calculate_fragmentation_2D_core_level_y(
                    Nx, Ny, fragmentation, hilbert)
! at edge level switch for 3D domain decomposition
subroutine calculate_fragmentation_3D_edge_level_x(
                    Nx, Ny, Nz, fragmentation, hilbert)
subroutine calculate_fragmentation_3D_edge_level_y(
                    Nx, Ny, Nz, fragmentation, hilbert)
subroutine calculate_fragmentation_3D_edge_level_z(
                    Nx, Ny, Nz, fragmentation, hilbert)
! at core level switch for 3D domain decomposition
subroutine calculate_fragmentation_3D_core_level_x(
                    Nx, Ny, Nz, fragmentation, hilbert)
subroutine calculate_fragmentation_3D_core_level_y(
                    Nx, Ny, Nz, fragmentation, hilbert)
subroutine calculate_fragmentation_3D_core_level_z(
                    Nx, Ny, Nz, fragmentation, hilbert)
```

The `calculate_fragmentation_2D(3D)_edge(core)_level_x(y|z)` functions are provided for applications using 2D(3D) domain decomposition. The `Nx`, `Ny` and `Nz` specifies the number of processes in each dimension. And `hilbert` specifies whether it is calculated with the Hilbert-based method or not. Each function calculates the task fragmentation (`fragmentation`) in the communicator(`x`-axis, `y`-axis, or `z`-axis) at the level (`edge` or `core` level switch) of the system topology. It requires the information of switch levels of hosts. It could use the physical topology file (`readfile_from_node.list`) which is statically generated from the service that manages the interconnection network, e.g., Fabric Manager.

2.3 `record_compute_nodes` Tool

The `record_compute_nodes` tool records a list of compute resources allocated to the given MPI tasks according to the domain decomposition strategy. Specifically, it generates a `stdout` file containing a list of `MPI_ID-hostname` pairs, i.e., MPI task IDs and their host mappings. This list is further leveraged to compute the fragmentation of the currently mapped tasks.

2.4 `generate_job_log` Tool

The `generate_job_log` tool generates log data including execution profile and fragmentation value of jobs to analyze performance. It first reads the list of `MPI_ID-hostname` pairs generated by the `record_computed_nodes` tool. It then calculates the task fragmentation in each communicator at each level of the system topology by invoking the `calculate_fragmentation` function. The tool outputs a csv (comma-separated values) file containing the job log as described in Table 2.

Table 2: An example of job log file (CSV format).

ID	overall time	comm. time	f_{core}-sw, z-$axis$	f_{edge}-sw, z-$axis$	...
803	4.86	2.53	0.19	0.41	...
804	4.7	2.4	0.18	0.38	...
⋮	⋮	⋮	⋮	⋮	⋱

3 Directory structure

Following is directory structure.

- **APIs**
  The library is provided as both C functions and Fortran module, in separate sub-directories. Each sub-directory has its source code and Makefile to build the library. After building of C functions, the sub-directory will have `hilbert.o` and `hilbert.h`. Also, after building of Fortran modules, the sub-directory will have `calculate_fragmentation_2D.mod`, `alculate_fragmentation_3D.mod`, and `hilbert.mod`.

- **Tools**
  This directory contains two core tools for performance analysis and the development of PI predictor; `record_compute_nodes` and `generate_job_log`, in separate sub-directories. Each has its source code and Makefile to build the tool. `generate_job_log` has two Examples (for P3DFFT solver and Poisson solver), in separate sub-directories.

- **Examples**
  This directory has two example programs (P3DFFT solver and Poisson solver) in Fortran, in separate sub-directories. Each has its working directory to contain the example of a batch job script and an input parameter. The batch job script includes the two versions of the workload and execute them one after another. P3DFFT solver has patch files to adapt the Hilbert-based strategy. Poisson solver has its source code to build executable files. The baseline and Hilbert executable is to be built from source files using a "make" command. The details are described at 4.1.

4 Installation and Execution

4.1 Two workloads to exploit the Hilbert-based strategy

Two examples (P3DFFT solver and Poisson solver) are given to compare the performance between the baseline and the Hilbert-based strategy. Channel code is excluded due to due to copyright issues. Each example has its source code to build executable files. The baseline and Hilbert executable is to be built from source files using a `make` command.

4.1.1 P3DFFT solver

P3DFFT solver is a popular software package to implement fast Fourier transforms (FFTs) in three dimensions in a highly efficient and scalable way. It exploits a 2D domain decomposition strategy. It needs to build two versions (baseline and Hilbert executable) of P3DFFT solver and execute them one after another. Patch files are given to build Hilbert version. Following is how to install and test the comparison between the baseline and the Hilbert version.

Prerequisites

- **Platform and compiler**: this was tested on a conventional Linux cluster system(x86-64) and Intel parallel compiler. But, it might be functional on other Linux cluster systems(e.g. Power systems) and compilers(e.g. GNU compilers).
- **Task-mapping library**: export the path of the root directory to the `ROOT` variable.

  ```
  $ export ROOT=‘pwd‘
  ```

- **FFT library**: FFT library such as FFTW, FFTE, or MKL is required to install P3DFFT. It should be installed in both single and double precision. Following is an example of installation of FFTW library.
  - Download and extract: obtain a latest stable release at the following URL: http://www.fftw.org/download.html, and untar it

    ```
    $ tar xvf fftw-3.3.10.tar.gz
    ```

  - Installation: move into the directory and install in both single and double precision as follows.

    ```
    $ cd fftw-3.3.10
    $ ./confugre CC=icc --enable-shared
      --prefix=/scratch/abc/external/fftw-3.3.10
    $ make ; make install
    $ make clean
    $ ./confugre CC=icc --enable-shared --enable-float
    ```

```
  --prefix=/scratch/abc/external/fftw-3.3.10
$ make ; make install
```

P3DFFT(baseline) Installation

- **Download and extract**: obtain a latest stable release at following URL: `https://github.com/sdsc/p3dfft.3`, and unzip it.

```
$ unzip p3dfft.3-master.zip
```

- **Build**: move into the download directory (`~/p3dfft.3-master`) and build as follows:

```
$ cd ~/p3dfft.3-master
$ ./configure --enable-fftw
 --with-fftw=/scratch/abc/external/fftw-3.3.10 CC=mpiicc
 FC=mpiifort CXX=mpiicpc CXXFLAGS=-std=c++11
$ make
```

- **Installation**: move into the working directory (`$ROOT/Examples/p3dfft/working`) and copy an executable file(`test3D_r2c_cpp`) to the directory

```
$ cd $ROOT/Examples/p3dfft/working
$ cp ~/p3dfft.3-master/sample/C++/test3D_r2c_cpp .
```

P3DFFT(Hilbert) Installation

- **Copy patch files**: copy patch files to the build directory of P3DFFT (`~/p3dfft.3-master/build`).

```
$ cp $ROOT/Examples/p3dfft/init.C ~/p3dfft.3-master/build/
$ cp $ROOT/Examples/p3dfft/hilbert.* ~/p3dfft.3-master/build/
```

- **Configuration**: modify a `Makefile` file as follows:

```
$ vi ~/p3dfft.3-master/build/Makefile
CXXFLAGS = -std=c++11
==>
CXXFLAGS = -std=c++11 -DHILBERT

init.o: $(top_builddir)/include/p3dfft.h
==>
hilbert.o:  hilbert.c
    $(CC) -c hilbert.c
init.o: $(top_builddir)/include/p3dfft.h hilbert.o

am_libp3dfft_3_a_OBJECTS = init.$(OBJEXT) exec.$(OBJEXT)
```

```
templ.$(OBJEXT) wrap.$(OBJEXT) deriv.$(OBJEXT) fwrap.$(OBJEXT)
fp3dfft++mod.$(OBJEXT)
==>
am_libp3dfft_3_a_OBJECTS = hilbert.$(OBJEXT) init.$(OBJEXT)
exec.$(OBJEXT) templ.$(OBJEXT) wrap.$(OBJEXT) deriv.$(OBJEXT)
fwrap.$(OBJEXT) fp3dfft++mod.$(OBJEXT)

am_libp3dfft_3_a_OBJECTS = init.$(OBJEXT) exec.$(OBJEXT)
templ.$(OBJEXT) wrap.$(OBJEXT) deriv.$(OBJEXT) fwrap.$(OBJEXT)
fp3dfft++mod.$(OBJEXT)
==>
am_libp3dfft_3_a_OBJECTS = hilbert.$(OBJEXT) init.$(OBJEXT)
exec.$(OBJEXT) templ.$(OBJEXT) wrap.$(OBJEXT) deriv.$(OBJEXT)
fwrap.$(OBJEXT) fp3dfft++mod.$(OBJEXT)
```

Note that `am_libp3dfft_3_a_OBJECTS` variable would be defined twice.

- **Build**: build as follows:

```
$ cd ~/p3dfft.3-master
$ make clean ; make
```

If it fails to build due to an `init.C` file, please manually edit the `init.C` file as described in Appendix A and make.

- **Installation**: move into the working directory and copy an executable file(`test3D_r2c_cpp`) to the directory.

```
$ cd $ROOT/Examples/p3dfft/working
$ cp ~/p3dfft.3-master/sample/C++/test3D_r2c_cpp test3D_r2c_cpp.hilbert
```

Run two versions of P3DFFT

- **Write a job script**: `run.sh` is an example job script for the PBS batch scheduler. Because the code is implemented on MPI library, it must be run with an `mpirun` program.

```
$ cat run.sh
#!/bin/bash
#PBS -N test
#PBS -V
#PBS -q normal
#PBS -A inhouse
#PBS -l select=4:ncpus=64:mpiprocs=64:ompthreads=1
#PBS -l walltime=00:30:00

cd $PBS_O_WORKDIR
```

```
TOTAL_CPUS=$(wc -l $PBS_NODEFILE | awk '{print $1}')

OUTFILE=out.${TOTAL_CPUS}.${PBS_JOBID}
ERRFILE=err.${TOTAL_CPUS}.${PBS_JOBID}
OUTFILE2=out.hilbert.${TOTAL_CPUS}.${PBS_JOBID}
ERRFILE2=err.hilbert.${TOTAL_CPUS}.${PBS_JOBID}
date
mpirun ./test3D_r2c_cpp.hilbert  >> ${OUTFILE2} 2>> ${ERRFILE2}
date
mpirun ./test3D_r2c_cpp  >> ${OUTFILE} 2>> ${ERRFILE}
date
```

- **Edit input parameters**: the number of grid points are specified in `stdin` file. As for evaluations, to find proper number of grid points according to the number of processes, please refer to Table 3.

  ```
  $ cat stdin
  64 64 64 2 20
  ```

- **Run**: submit the job script(`run.sh`) to run two versions of P3DFFT via the PBS job scheduler. Note that the total number of CPU cores must be $2^n \times 2^n$, n = 1, 2, ..., because P3DFFT is by using 2D domain decomposition. If out-of-memory errors occur, the grid points should be decreased.

  ```
  $ qsub ./run.sh
  ```

Table 3: An example of input parameters according to the number of processes for 2D decomposition.

Number of processes=(npx×npy)	Number of grid points
256=(16×16)	64×64 × 64
1024=(32×32)	128×128 × 128
4096=(64×64)	256×256 × 256
16384=(128×128)	512×512 × 512
65536=(256×256)	1024×1024 × 1024

4.1.2 Poisson solver

The Poisson solver is a parallel solver of the Poisson equation with Multigrid method. Unlike previous workload, this code adapts a 3D domain decomposition strategy. We need to build two versions (baseline and Hilbert executable) of Poisson solver and execute them one after another. Following is how to install

and test the comparison between the baseline and the Hilbert version.

Prerequisites

- **Platform and compiler**: this was tested on a conventional Linux cluster system(x86-64) and Intel parallel compiler. But, it might be functional on other Linux systems(e.g. Power systems) and compilers(e.g. GNU compilers).
- **Task-mapping library**: export the path of the root directory to the `ROOT` variable.

```
$ export ROOT=‘pwd‘
```

Installation

- **Move to the directory**: move into the directory of Poisson solver.

```
$ cd $ROOT/Examples/poisson/
```

- **Configuration**: make sure that the compiler is properly configured at `Makefile` and `Makefile.hilbert`.

```
$ grep -ir "FC" Makefile*
grep -ir "FC" Makefile*
Makefile:FC     = mpiifort
Makefile: $(FC) -o $(EXE) $(FFLAGS) $(OBJS) $(LIBS)
Makefile: $(FC) -c $(FFLAGS_HILBERT) mpi_topology.f90
Makefile: $(FC) -o $(EXE_HILBERT) $(HILBERT_OBJS) $(LIBS)
Makefile: $(FC) -c $(FFLAGS) $?
Makefile.hilbert:FC     = mpiifort
Makefile.hilbert: $(FC) -o $(EXE) $(FFLAGS) $(OBJS) $(LIBS)
Makefile.hilbert: $(FC) -c $(FFLAGS_HILBERT) mpi_topology.f90
Makefile.hilbert: $(FC) -o $(EXE_HILBERT) $(HILBERT_OBJS) $(LIBS)
Makefile.hilbert: $(FC) -c $(FFLAGS) $?
```

- **Build**: build two versions(baseline and Hilbert) as follows:

```
$ make
$ make clean
$ make -f Makefile.hilbert
```

- **Installation**: move into the working directory (`$ROOT/Examples/poisson/working`) and copy executable files(`prun.ex` and `prun_hilbert.ex`) to the working directory.

```
$ cd $ROOT/Examples/poisson/working
$ cp ../prun.ex* .
```

Run two versions of Poisson

- **Write a job script**: `run.sh` is an example job script for the PBS batch scheduler. Because the code is implemented on the MPI library, it must be run with an `mpirun` program.

```
$ cat run.sh
#!/bin/bash
#PBS -N test
#PBS -V
#PBS -q normal
#PBS -A inhouse
#PBS -l select=8:ncpus=64:mpiprocs=64:ompthreads=1
#PBS -l walltime=00:30:00

cd $PBS_O_WORKDIR

TOTAL_CPUS=$(wc -l $PBS_NODEFILE | awk '{print $1}')

OUTFILE=out.${TOTAL_CPUS}.${PBS_JOBID}
ERRFILE=err.${TOTAL_CPUS}.${PBS_JOBID}
OUTFILE2=out.hilbert.${TOTAL_CPUS}.${PBS_JOBID}
ERRFILE2=err.hilbert.${TOTAL_CPUS}.${PBS_JOBID}
date
time mpirun ./prun_hilbert.ex  >> ${OUTFILE2} 2>> ${ERRFILE2}
date
time mpirun ./prun.ex  >> ${OUTFILE} 2>> ${ERRFILE}
date
```

- **Edit input parameters**: the number of processes and grid points in each dimension can be specified in `procs` and `meshes` of `PARA_INPUT.inp` file, respectively. As for evaluations, to find proper input parameters according to the number of processes, please refer to Table 4.

```
$ cat PARA_INPUT.inp
&meshes
nx = 1024
ny = 1024
nz = 1024
/
...
&procs
npx = 8
npy = 8
npz = 8
/
...
```

- **Run**: submit the job script(`run.sh`) to run two versions of P3DFFT via the PBS job scheduler. Note that the total number of CPU cores must be $2^n \times 2^n \times 2^n$, n = 1, 2, ..., because Poisson is by using 3D domain decomposition. If out-of-memory errors occur, the grid points should be decreased.

```
$ qsub ./run.sh
```

Table 4: An example of input parameters according to the number of processes for 3D decomposition.

Number of processes=(npx×npy×npz)	Number of grid points
64=(4×4 × 4)	512×512 × 512
256=(8×8 × 8)	1024×1024 × 1024
4096=(16×16 × 16)	2048×2048 × 2048
32768=(32×32 × 32)	4096×4096 × 4096

4.2 Tools for performance analysis

4.2.1 `record_compute_nodes` Tool

Following is how to install and test `record_compute_nodes` tool.

Prerequisites

- **Platform and compiler**: this was tested on a conventional Linux cluster system(x86-64) and Intel parallel compiler. But, it might be functional on other Linux systems(e.g. Power systems) and compilers(e.g. GNU compilers).
- **Task-mapping library**: export the path of the root directory of it to the `ROOT` variable.

```
$ export ROOT=‘pwd‘
```

Installation

- **Move to the directory**: move into the `$ROOT/Tools/record_compute_nodes` directory.

```
$ cd $ROOT/Tools/record_compute_nodes
```

- **Build**: an executable file(`record_compute_nodes`) is to be built from source files using a `make` command as follows.

```
$ make
```

Run

- **Write a job script**: `run.sh` is an example job script for PBS batch scheduler. Because `record_compute_nodes` tool is implemented on MPI library, it must be run with an `mpirun` program. The first argument on `record_compute_nodes` tool is the domain decomposition strategy, i.e. 2 for 2D domain decomposition and 3 for 3D domain decomposition.

```
$ cat run.sh
#!/bin/bash
#PBS -N test
#PBS -V
#PBS -q normal
#PBS -A inhouse
#PBS -l select=8:ncpus=64:mpiprocs=64:ompthreads=1
#PBS -l walltime=0:10:00

cd $PBS_O_WORKDIR
TOTAL_CPUS=$(wc -l $PBS_NODEFILE | awk '{print $1}')

OUTFILE=out.host.${TOTAL_CPUS}.${PBS_JOBID}
ERRFILE=err.host.${TOTAL_CPUS}.${PBS_JOBID}
# "record_compute_nodes 2" for 2D domain decomposition
# "record_compute_nodes 3" for 3D domain decomposition
mpirun ./record_compute_nodes 2 >> ${OUTFILE} 2>> ${ERRFILE}
```

- **Run**: submit the job script(`run.sh`) via a PBS job scheduler. Note that the total number of CPU cores must be $2^n \times 2^n \times 2^n$, n = 1, 2, ... for 2D domain decomposition and $2^n \times 2^n \times 2^n$ for 3D domain decomposition.

```
$ qsub ./run.sh
```

4.2.2 `generate_job_log` Tool

Two examples (P3DFFT solver and Poisson solver) are given to run the `generate_job_log` tool. Channel code is excluded due to due to copyright issues. Each example has its source code to build a tool. An executable file(`generate_job_log`) is to be built from source files using a `make` command.
Prerequisites

- **Platform and compiler**: this was tested on a conventional Linux system(x86-64) and Intel compiler. But, it might be functional on other Linux systems(e.g. Power systems) and compilers(e.g. GNU compilers).
- **Task-mapping library**: export the path of the root directory to the `ROOT` variable.

```
$ export ROOT=‘pwd‘
```

Installation and run for P3DFFT solver

- **Move to the directory**: move into the `$ROOT/Tools/p3dfft/generate_job_log` directory.

```
$ cd $ROOT/Tools/p3dfft/generate_job_log
```

- **Build**: an executable file(`generate_job_log`) is to be built from source files using a `make` command as follows.

```
$ make
```

- **Run**: run as follows.

```
$ ./generate_job_log
 number, cores, size, normal_overall, hilbert_overall, ...
   8430966,       4096,        2,      0.02788,    0.01686,   ...
   8430967,       4096,        2,      0.04358,    0.02558,  ...
...
```

Installation and run for Poisson solver

- **Move to the directory**: move into the `$ROOT/Tools/poisson/generate_job_log` directory.

```
$ cd $ROOT/Tools/poisson/generate_job_log
```

- **Build**: an executable file(`generate_job_log`) is to be built from source files using a `make` command as follows.

```
$ make
```

- **Run**: run as follows.

```
$ ./generate_job_log
 number, cores, size, normal_overall, hilbert_overall, ...
   8047991,       4096,        2,      0.37070,    0.27355,   ...
   8047993,       4096,        2,      0.30651,    0.31976,  ...
...
```

A Appendix

Modification of the "`init.C`" file

- include the `hilbert.h` header file in the `init.C` file as follows.

```
#include "hilbert.h"
```

- add the following statements just after calling `MPI_Comm_size` in the constructor(`ProcGrid::ProcGrid`) of `ProcGrid` class in the `init.C` file.

```
    MPI_Comm_size(mpi_comm_glob,&numtasks);
    // add
    MPI_Comm new_comm;
#ifdef HILBERT
    if (procdims[0]==1) {
        remap_tasks_Hilbert_2D(procdims[1], procdims[2],
            &mpi_comm_glob, &new_comm);
    } else {
        fprintf(stderr, "Not supported number %d for HILBERT\n",
            procdims[0]);
    }
#else
    new_comm = mpi_comm_glob;
#endif
```

- edit the statement calling `MPI_Cart_create` function in the constructor(`ProcGrid::ProcGrid`) of `ProcGrid` class in the `init.C` file as follows.

```
MPI_Cart_create(mpi_comm_glob,3,ProcDims,periodic,reorder,
    &mpi_comm_cart);
==>
MPI_Cart_create(new_comm,3,ProcDims,periodic,reorder,
    &mpi_comm_cart);
```