Parallel Data Interface

how to decouple applications from I/O concerns

Leonardo Bautista Gomez[2],
Julien Bigot[1], Mohamed Gaalich[1],
Kai Keller[2], Guillaume Latu[3],
Corentin Roussel[1]

[1] Maison de la Simulation
[2] Barcelona Supercomputing Center
[3] CEA/IRFM

Per3s – DDN Meudon – 2017 Jan. 30th

I/O's for HPC

- We want it easy to use
- We want it fast
- We want a portable library
- We want large language support
- We want parallelization independent file format
- We want a portable file format
- We want...

Kind of a “*Pick 2 out of 3*” situation (except with more than 3)

Example: I/Os in Gysela

- Gysela: 5D Semi-Lagrangian Gyrokinetic code
  - Large scale plasma simulation (16k+ core, ~month wallclock time)
  - Large memory footprint (*e.g.* on Curie, 100+ TB)
- 2 I/O kinds: Diagnostics & checkpoints
- Checkpoint
  - Intermediate checkpoints (Fault tolerance)
    - Performance, fast temporary storage, overlap w. computation
  - Final checkpoints (Segmentation)
    - File portability, domain decomposition change, permanent
- … Except for small debug cases (ease of use 1st)

Example: I/Os in Gysela

- Gysela: 5D Semi-Lagrangian Gyrokinetic code
  - Large scale plasma simulation (16k+ core, ~month wallclock time)
  - Large memory footprint (*e.g.* on Curie, 100+ TB)
- 2 I/O kinds: Diagnostics & checkpoints
- **Checkpoint**
  - Intermediate checkpoints (Fault tolerance)
    - Performance, fast temporary storage, overlap w. computation
  - Final checkpoints (Segmentation)
    - File portability, domain decomposition change, permanent
- … Except for small debug cases (ease of use 1st)

Context : I/O libraries

Many libraries for I/O's in HPC

- POSIX I/O
- ANSI C lib
- MPI I/O
- HDF5 (seq / par)
- NetCDF (seq/par)
- FTI
- SCR
- XIOS
- SIONlib
- DDN Infinite Memory Engine
- Damaris
- ADIOS
- ...

I/O libraries: make a choice

- Stream *vs.* Object API
- Imperative *vs.* declarative API
- General purpose *vs.* Domain specific
- Dedicated HW use (none, NVRAM, SSDs, I/O nodes, …)
- File format: ASCII *vs.* HDF5 *vs.* …
- Single file *vs.* one file / process
- ...

PDI: The requirements

1) Start a new code
   - Use a simple API for I/O's
   - Don't depend on anything fancy
2) When the code grows
   - Have efficient parallel I/O's
   - Different requirement in different places
     - Checkpoint/Restart *vs.* code outputs
3) When the hardware changes
   - Optimal I/O strategy changes
   - Is our chosen library available there?

Minimize changes in the code

Introducing PDI

© XKCD
https://xkcd.com/927/

Introducing PDI

SITUATION:
THERE ARE 14 COMPETING STANDARDS.

14?! RIDICULOUS!
WE NEED TO DEVELOP
ONE UNIVERSAL STANDARD

SOON:
SITUATION:
THERE ARE 15 COMPETING STANDARDS.

No!

PDI is an Interface...
just an interface!

© XKCD
https://xkcd.com/927/

The I/O stack

User code	User code	…	User code	
MPI				
API-adapt	API-adapt		API-adapt	
API	API		API	
Frontend	Frontend		Frontend	
Parallel strategy				
Backend	Backend	…	Backend	… Backend
Parallel File-System				

The I/O stack

- Complex-type
- App life-cycle
- Serialization
- …

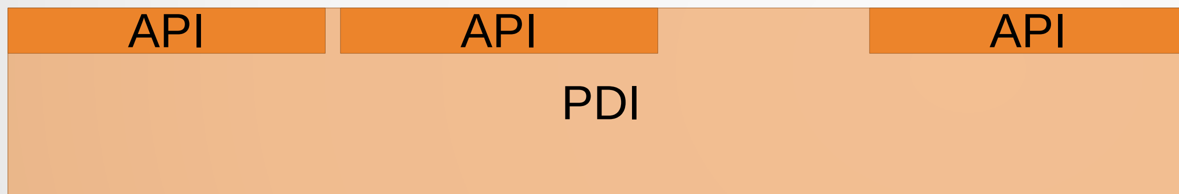
- File open & close
- Data read & write
  - Stream/Object
- Distribution spec.
- …

- Burst buffering
- Compression
- Data processing
  - workflow
- ...

- Redistribution
- Dedicated nodes
- Staged I/O's
- ...

- File format
- Buffering
- ...

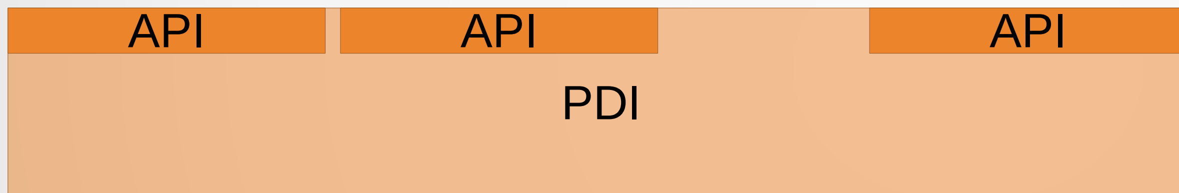
The I/O stack: PDI



- File open & close
- Data read & write
  - Stream/Object
- Distribution spec.
- …

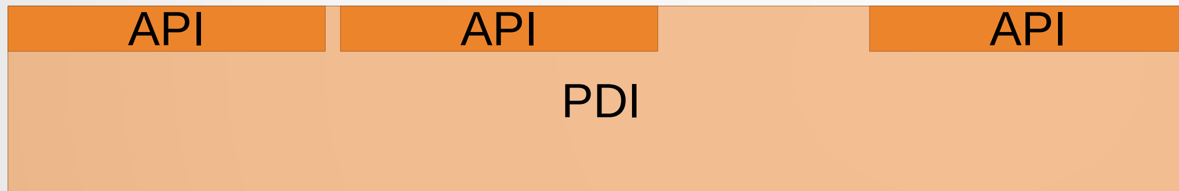
The I/O stack: PDI

- Standardize a declarative API
- Describe actions in a config file
- Rely on existing libraries through plugins



The I/O stack: PDI

- Standardize a declarative API
- Describe actions in a config file
- Rely on existing libraries through plugins




```yaml
metadata:
  width: int
  height: int
  pwidth: int
  pheight: int
  nb_iter: int
  iter: int
  coord: { size: 2, type: int }
data:
  main_field:
    type: double
    sizes: [$width, $height]
    subsizes: [$width-2, $height-2]
    starts: [1, 1]
plugins:
  hdf5_per_process:
    outputs:
      main_field: # ref to a variable
        select: $iter%16=0
        file: output_${coord[0]}x${coord[1]}.h5
        var: iter_${iter}/main_field
```

The I/O stack: PDI

- Standardize a declarative API
- Describe actions in a config file
- Rely on existing libraries through plugins


```
metadata:
  width: int
  height: int
  pwidth: int
  pheight: int
  nb_iter: int
  iter: int
```

PDI: our choices

- Stream *vs.* Object API
- Imperative *vs.* declarative API
- General purpose *vs.* Domain specific
- Dedicated HW use (none, NVRAM, SSDs, I/O nodes, …)
- File format: ASCII *vs.* HDF5 *vs.* …
- Single file *vs.* one file / process
- ...

PDI: our choices

- Stream *vs.* **Object API**
- Imperative *vs.* **declarative API**
- **General purpose** *vs.* Domain specific
- Dedicated HW use (none, NV… …AM, SSDs, I/O nodes, …)
- File format: AS… …-5 *vs.* …
- Singl… …e file / process
- ...

Plugins

PDI: The API

```
/** Initializes PDI
 * \param[in] conf the configuration
 * \param[in,out] world the main MPI communicator
 * \return an error status
 */
PDI_status_t PDI_init(PC_tree_t conf, MPI_Comm* world);

/** Finalizes PDI
 * \return an error status
 */
PDI_status_t PDI_finalize();
```

- Initialization
  - Takes a yaml config
  - Might steal ranks from the provided MPI comm.
- Finalization releases all resources

PDI: an example

```
int main( int argc, char *argv[] )
{
    MPI_Init(&argc, &argv); PC_tree_t conf = PC_parse_path(argv[1]);
    int width, height, max_iterations;
    read_conf(conf, &width, &height, &max_iterations);
    MPI_Comm main_comm=MPI_COMM_WORLD;
    int rank; MPI_Comm_rank(main_comm, &rank);

    double *cur = malloc(sizeof(double)*width*height);
    double *next = malloc(sizeof(double)*width*height);

        init(cur, width, height, car_coord[0], car_coord[1]);


    int ii; for(ii=0; ii<max_iterations ; ++ii) {



        iter(cur, next, width, height);
        exchange(cart_com, next, width, height);
        double *tmp = cur; cur = next; next = tmp;
    }



                    PC_tree_destroy(&conf);
    free(cur); free(next);
    MPI_Finalize();
}
```

PDI: an example

```c
int main( int argc, char *argv[] )
{
    MPI_Init(&argc, &argv); PC_tree_t conf = PC_parse_path(argv[1]);
    int width, height, max_iterations;
    read_conf(conf, &width, &height, &max_iterations);
    MPI_Comm main_comm=MPI_COMM_WORLD; PDI_init(PC_get(conf, ".pdi"), &main_comm);
    int rank; MPI_Comm_rank(main_comm, &rank);

    double *cur = malloc(sizeof(double)*width*height);
    double *next = malloc(sizeof(double)*width*height);

        init(cur, width, height, car_coord[0], car_coord[1]);


    int ii; for(ii=0; ii<max_iterations ; ++ii) {



        iter(cur, next, width, height);
        exchange(cart_com, next, width, height);
        double *tmp = cur; cur = next; next = tmp;
    }



    PDI_finalize(); PC_tree_destroy(&conf);
    free(cur); free(next);
    MPI_Finalize();
}
```

PDI: The API

```
/** Shortly exposes some data to PDI.
 * \param[in] name the data name
 * \param[in] data the exposed data
 */
PDI_status_t PDI_expose(const char *name, const void *data);

/** Imports some data from PDI.
 * \param[in] name the data name
 * \param[out] data the data to initialize
 */
PDI_status_t PDI_import(const char *name, void *data);
```

- Expose data to PDI (out, ≈write)
  - **Might or might not** use it (write it, use it for computation, send it over MPI, etc.)
- Import data from PDI
  - **Might or might not** fill the content

PDI: an example

```
int main( int argc, char *argv[] )
{
    MPI_Init(&argc, &argv); PC_tree_t conf = PC_parse_path(argv[1]);
    int width, height, max_iterations;
    read_conf(conf, &width, &height, &max_iterations);
    MPI_Comm main_comm=MPI_COMM_WORLD; PDI_init(PC_get(conf, ".pdi"), &main_comm);
    int rank; MPI_Comm_rank(main_comm, &rank);

    double *cur = malloc(sizeof(double)*width*height);
    double *next = malloc(sizeof(double)*width*height);

        init(cur, width, height, car_coord[0], car_coord[1]);


    int ii; for(ii=0; ii<max_iterations ; ++ii) {



        iter(cur, next, width, height);
        exchange(cart_com, next, width, height);
        double *tmp = cur; cur = next; next = tmp;
    }



    PDI_finalize(); PC_tree_destroy(&conf);
    free(cur); free(next);
    MPI_Finalize();
}
```

PDI: an example

```
int main( int argc, char *argv[] )
{
    MPI_Init(&argc, &argv); PC_tree_t conf = PC_parse_path(argv[1]);
    int width, height, max_iterations;
    read_conf(conf, &width, &height, &max_iterations);
    MPI_Comm main_comm=MPI_COMM_WORLD; PDI_init(PC_get(conf, ".pdi"), &main_comm);
    int rank; MPI_Comm_rank(main_comm, &rank);
    PDI_expose("rank", &rank); PDI_expose("width", &width); PDI_expose("height", &height);
    double *cur = malloc(sizeof(double)*width*height);
    double *next = malloc(sizeof(double)*width*height);
    if ( PDI_import("main_field", cur) ) {
        init(cur, width, height, car_coord[0], car_coord[1]);
    }

    int ii; for(ii=0; ii<max_iterations ; ++ii) {

        PDI_expose("iter", &ii); PDI_expose("main_field", cur);

        iter(cur, next, width, height);
        exchange(cart_com, next, width, height);
        double *tmp = cur; cur = next; next = tmp;
    }

    PDI_expose("iter", &ii); PDI_expose("main_field", cur);

    PDI_finalize(); PC_tree_destroy(&conf);
    free(cur); free(next);
    MPI_Finalize();
}
```

PDI: The API

```
/** Triggers a PDI "event"
 * \param[in] event the event name
 */
PDI_status_t PDI_event(const char *event);

/** Begin a transaction. All the ::PDI_expose will be exposed together.
 * \param[in] name the name of the transaction
 */
PDI_status_t PDI_transaction_begin( const char *name );

/** Ends the previously opened transaction.
 */
PDI_status_t PDI_transaction_end();
```

- Events notifies PDI of interesting code location
- Transactions
  - Group multiple PDI_expose
  - Triggers an event when all values are exposed

PDI: an example

```
int main( int argc, char *argv[] )
{
    MPI_Init(&argc, &argv); PC_tree_t conf = PC_parse_path(argv[1]);
    int width, height, max_iterations;
    read_conf(conf, &width, &height, &max_iterations);
    MPI_Comm main_comm=MPI_COMM_WORLD; PDI_init(PC_get(conf, ".pdi"), &main_comm);
    int rank; MPI_Comm_rank(main_comm, &rank);
    PDI_expose("rank", &rank); PDI_expose("width", &width); PDI_expose("height", &height);
    double *cur = malloc(sizeof(double)*width*height);
    double *next = malloc(sizeof(double)*width*height);
    if ( PDI_import("main_field", cur) ) {
        init(cur, width, height, car_coord[0], car_coord[1]);
    }

    int ii; for(ii=0; ii<max_iterations ; ++ii) {

        PDI_expose("iter", &ii); PDI_expose("main_field", cur);

        iter(cur, next, width, height);
        exchange(cart_com, next, width, height);
        double *tmp = cur; cur = next; next = tmp;
    }

    PDI_expose("iter", &ii); PDI_expose("main_field", cur);

    PDI_finalize(); PC_tree_destroy(&conf);
    free(cur); free(next);
    MPI_Finalize();
}
```

PDI: an example

```
int main( int argc, char *argv[] )
{
    MPI_Init(&argc, &argv); PC_tree_t conf = PC_parse_path(argv[1]);
    int width, height, max_iterations;
    read_conf(conf, &width, &height, &max_iterations);
    MPI_Comm main_comm=MPI_COMM_WORLD; PDI_init(PC_get(conf, ".pdi"), &main_comm);
    int rank; MPI_Comm_rank(main_comm, &rank);
    PDI_expose("rank", &rank); PDI_expose("width", &width); PDI_expose("height", &height);
    double *cur = malloc(sizeof(double)*width*height);
    double *next = malloc(sizeof(double)*width*height);
    if ( PDI_import("main_field", cur) ) {
        init(cur, width, height, car_coord[0], car_coord[1]);
    }
    PDI_event("loop_start");
    int ii; for(ii=0; ii<max_iterations ; ++ii) {
        PDI_transaction_begin("loop_iter");
        PDI_expose("iter", &ii); PDI_expose("main_field", cur);
        PDI_transaction_end();
        iter(cur, next, width, height);
        exchange(cart_com, next, width, height);
        double *tmp = cur; cur = next; next = tmp;
    }
    PDI_transaction_begin("loop_end");
    PDI_expose("iter", &ii); PDI_expose("main_field", cur);
    PDI_transaction_end();
    PDI_finalize(); PC_tree_destroy(&conf);
    free(cur); free(next);
    MPI_Finalize();
}
```

PDI: an example

```yaml
pdi:
  metadata: #  small values reference-able w. $syntax
    rank: int # rank of the process
    width: int # per proc. width including ghost
    height: int # per proc. height including ghost
    iter: int # curent iteration id
```

PDI: an example

```
int main( int argc, char *argv[] )
{
    MPI_Init(&argc, &argv); PC_tree_t conf = PC_parse_path(argv[1]);
    int width, height, max_iterations;
    read_conf(conf, &width, &height, &max_iterations);
    MPI_Comm main_comm=MPI_COMM_WORLD; PDI_init(PC_get(conf, ".pdi"), &main_comm);
    int rank; MPI_Comm_rank(main_comm, &rank);
    PDI_expose("rank", &rank); PDI_expose("width", &width); PDI_expose("height", &height);
    double *cur = malloc(sizeof(double)*width*height);
    double *next = malloc(sizeof(double)*width*height);
    if ( PDI_import("main_field", cur) ) {
        init(cur, width, height, car_coord[0], car_coord[1]);
    }
    PDI_event("loop_start");
    int ii; for(ii=0; ii<max_iterations ; ++ii) {
        PDI_transaction_begin("loop_iter");
        PDI_expose("iter", &ii); PDI_expose("main_field", cur);
        PDI_transaction_end();
        iter(cur, next, width, height);
        exchange(cart_com, next, width, height);
        double *tmp = cur; cur = next; next = tmp;
    }
    PDI_transaction_begin("loop_end");
    PDI_expose("iter", &ii); PDI_expose("main_field", cur);
    PDI_transaction_end();
    PDI_finalize(); PC_tree_destroy(&conf);
    free(cur); free(next);
    MPI_Finalize();
}
```

PDI: an example

```yaml
pdi:
  metadata: #  small values reference-able w. $syntax
    rank: int # rank of the process
    width: int # per proc. width including ghost
    height: int # per proc. height including ghost
    iter: int # curent iteration id
  data: # large values that should not be copied
    main_field: # type similar to MPI_Type
      type:     double
      sizes:    [$width, $height]
      subsizes: [$width-2, $height-2]
      starts:   [1, 1]
```

PDI: an example

```
int main( int argc, char *argv[] )
{
    MPI_Init(&argc, &argv); PC_tree_t conf = PC_parse_path(argv[1]);
    int width, height, max_iterations;
    read_conf(conf, &width, &height, &max_iterations);
    MPI_Comm main_comm=MPI_COMM_WORLD; PDI_init(PC_get(conf, ".pdi"), &main_comm);
    int rank; MPI_Comm_rank(main_comm, &rank);
    PDI_expose("rank", &rank); PDI_expose("width", &width); PDI_expose("height", &height);
    double *cur = malloc(sizeof(double)*width*height);
    double *next = malloc(sizeof(double)*width*height);
    if ( PDI_import("main_field", cur) ) {
        init(cur, width, height, car_coord[0], car_coord[1]);
    }
    PDI_event("loop_start");
    int ii; for(ii=0; ii<max_iterations ; ++ii) {
        PDI_transaction_begin("loop_iter");
        PDI_expose("iter", &ii); PDI_expose("main_field", cur);
        PDI_transaction_end();
        iter(cur, next, width, height);
        exchange(cart_com, next, width, height);
        double *tmp = cur; cur = next; next = tmp;
    }
    PDI_transaction_begin("loop_end");
    PDI_expose("iter", &ii); PDI_expose("main_field", cur);
    PDI_transaction_end();
    PDI_finalize(); PC_tree_destroy(&conf);
    free(cur); free(next);
    MPI_Finalize();
}
```

PDI: an example

```yaml
pdi:
  metadata: #  small values reference-able w. $syntax
    rank: int # rank of the process
    width: int # per proc. width including ghost
    height: int # per proc. height including ghost
    iter: int # curent iteration id
  data: # large values that should not be copied
    main_field: # type similar to MPI_Type
      type:     double
      sizes:    [$width, $height]
      subsizes: [$width-2, $height-2]
      starts:   [1, 1]
  plugins:
    hdf5: # load the HDF5 plugin
      outputs: # write the following
        main_field:
          select: $iter % 16 = 0
          file: output_${rank}.h5
          var: iter_${iter}/main_field
```

PDI: under the hood

- Replacing I/O libs function calls
- Only 2 concepts
  - “events” : (think Qt signal/slots)
  - Zero copy buffer sharing
    - name, pointer + access rights (R / W / RW)
- 2 APIs
  - “low” level for plugins (events notification, access buffers)
  - Higher-level API for the application

PDI: the plugin API

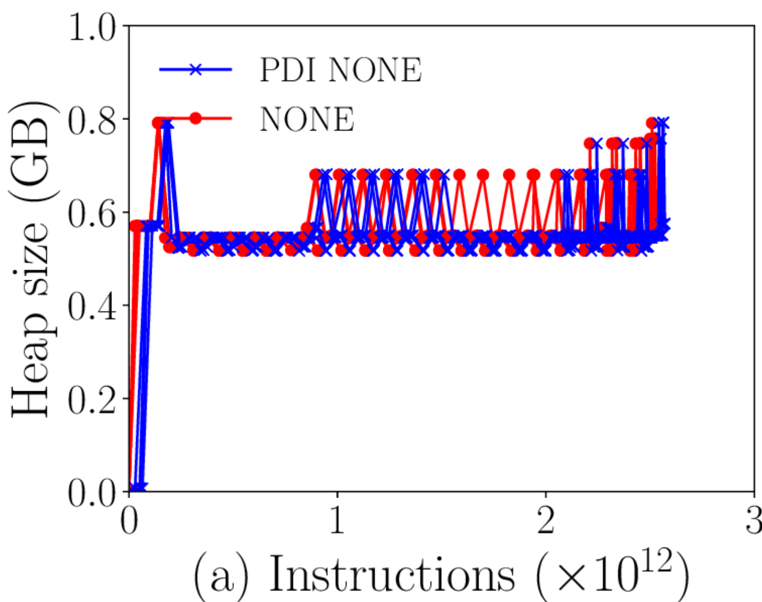
```
/** initialization notification */
PDI_status_t PDI_<plugin>_init(PC_tree_t conf,
                               MPI_Comm *world);

/** Finalization notification*/
PDI_status_t PDI_<plugin>_finalize();

/** Notification of a named event */
PDI_status_t PDI_<plugin>_event(const char *event);

/** Notification of a data availability event */
PDI_status_t PDI_<plugin>_data_start(PDI_variable_t *data);

/** Notification of a data unavailability event */
PDI_status_t PDI_<plugin>_data_end(PDI_variable_t *data);

/** Declare the plugin to PDI */
PDI_PLUGIN(<plugin>)
```

Gysela checkpointng w. PDI

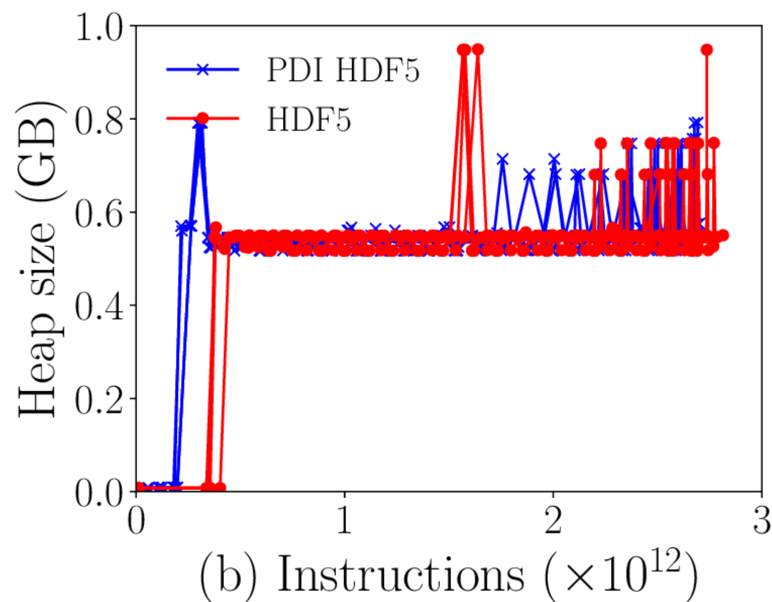
- 4 versions of the production code Gysela
  - No checkpoint
  - HDF5 checkpoints
  - FTI fault-tolerance
  - PDI (none / HDF5 / FTI / HDF5+FTI)
- Implementation cost

	Native API	PDI
HDF5	24 func. 535 LOC.	3 func. 207 LOC. 172 l. config.
FTI	7 func. 216 LOC. 29 config	+2 func.+37 LOC. + 7 l. config.
HDF5 + FTI		+0 func. +3 LOC. + 3 l. config.

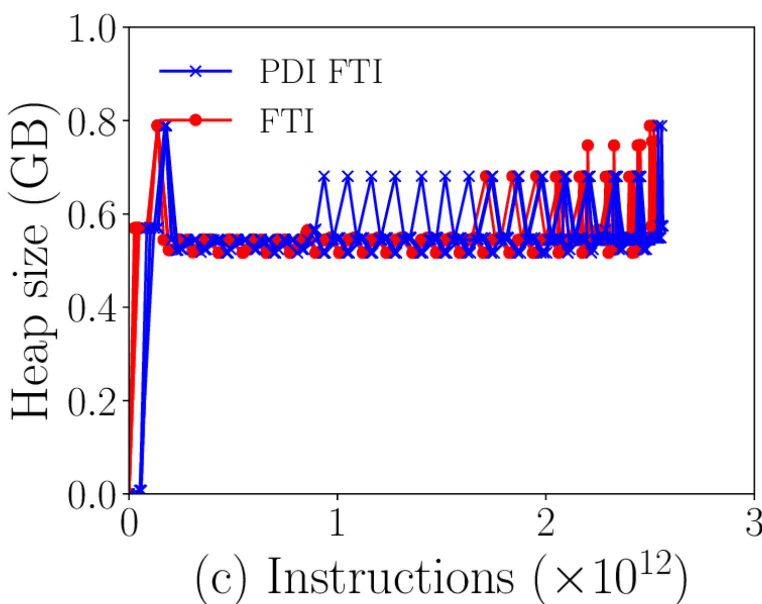
PDI: Memory overhead



(a) Instructions ($\times 10^{12}$)



(b) Instructions ($\times 10^{12}$)



(c) Instructions ($\times 10^{12}$)

Memory usage during a Gysela execution with and without PDI on 4 MareNostrum nodes

PDI: Overhead 1/2

FTI
HDF5
NONE
PDI FTI
PDI FTI+HDF5
PDI HDF5
PDI NONE

Normalized simulation time (s/MB)

0.20
0.18
0.16
0.14
0.12
0.10

10^2 10^3 10^4

(a) Problem size/nodes (MB)

Execution time by MB of checkpointed data on 4 MareNostrum Nodes with and without PDI

PDI: Overhead 2/2

Gysela Wallclock time in weak scaling on Curie with and without PDI

Checkpointed data ~2.1GB/node

Conclusion

PDI enables to decouple I/O's & code

https://gitlab.maisondelasimulation.fr/jbigot/pdi

- A declarative API
  - In the code: data pointers
  - In yaml: type descriptors (quite general, builds on MPI types)
  - Only “optional” calls (without PDI calls, code remains valid)
  - Minimal (read no) overhead
- Actual I/O's
  - Plugin specific
  - In yaml
  - But not only I/O's
    - Think fault tolerance, debug, visualization, post-processing, coupling, …