Implementation and performance evaluation of MPI persistent collective in MPC: a case study

Stephane Bouhrour [1], Julien Jaeger [1,2]

[1] Exascale Computing Research, France

[2] CEA, France

Tuesday 22nd September, 2020

Sommaire

Plan

Non-blocking Collective interface

- Two procedures

`MPI_Icoll(coll_args, MPI_Request)`

  - Iniatialize and start the collective
  - May return before the operation is completed/started
  - `MPI_Request` id operation
  - Operation can be done at any time by the runtime

\+ `MPI_Wait/Test_(any, some, all)(MPI_Request)`

  - Return when collective is completed
  - Freeing ressources
  - Invalidate request

Persistent Collective interface

- Four procedures → Allow finer compute cost distribution

`MPI_Coll_init(coll_args, MPI_Request)`
  - Iniatialize and return before operation starts
  - Set request to inactive (can be freed or started)
  - `MPI_Request` id operation

\+ `MPI_Start(all)(MPI_Request)`
  - Trigger the associated operation
  - Set request to active (can't be freed or started)

\+ `MPI_Wait/Test_(any, some, all)(MPI_Request)`
  - Completion
  - Set request to inactive (can be freed or started)

\+ `MPI_Request_free(MPI_Request)`
  - Freeing ressources
  - Invalidate persistent

Non-blocking and persistent interface example (1/4)

Exascale computing research

Non-blocking collective

```
1  MPI_Request *req_ptr
2
3  /* some code */
4  for(...){
5          /* some code */
6          MPI_Icoll(args, req_ptr);
7          /* some code */
8          MPI_Wait(req_ptr, status);
9  }
10 /* some code */
11
```

- Initialization not done before loop

Persistent collective

```
1  MPI_Request *req_ptr
2  MPI_Coll_init(args_coll, req_ptr);
3  /* some code */
4  for(...){
5          /* some code */
6          MPI_Start(req_ptr);
7          /* some code */
8          MPI_Wait(req_ptr, status);
9  }
10 /* some code */
11 MPI_Request_free(req_ptr);
```

- `MPI_Coll_init`

  ➔Initialization

Persistent and non-blocking interface example (2/4)

Exascale computing research

Non-blocking collective

```
1  MPI_Request *req_ptr
2
3  /* some code */
4  for(...){
5          /* some code */
6          MPI_Icoll(args, req_ptr);
7          /* some code */
8          MPI_Wait(req_ptr, status);
9  }
10 /* some code */
11
```

- `MPI_Icoll`

  ➔Initialization + starting

Persistent collective

```
1  MPI_Request *req_ptr
2  MPI\_Coll\_init(args\_coll, req\_ptr);
3  /* some code */
4  for(...){
5          /* some code */
6          MPI_Start(req_ptr);
7          /* some code */
8          MPI_Wait(req_ptr, status);
9  }
10 /* some code */
11 MPI_Request_free(req_ptr);
```

- `MPI_Start`

  ➔Starting

Persistent and non-blocking interface example (3/4)

Exascale computing research

Non-blocking collective

```
1  MPI_Request *req_ptr
2
3  /* some code */
4  for(...){
5          /* some code */
6          MPI_Icoll(args, req_ptr);
7          /* some code */
8          MPI_Wait(req_ptr, status);
9  }
10 /* some code */
11
```

- MPI_Wait
  - ➔ Completion + freeing

Persistent collective

```
1  MPI_Request *req_ptr
2  MPI\_Coll\_init(args\_coll, req\_ptr);
3  /* some code */
4  for(...){
5          /* some code */
6          MPI_Start(req_ptr);
7          /* some code */
8          MPI_Wait(req_ptr, status);
9  }
10 /* some code */
11 MPI_Request_free(req_ptr);
```

- MPI_Wait
  - ➔ Completion

Persistent and non-blocking interface example (4/4)

Exascale computing research

Non-blocking collective

```
1  MPI_Request *req_ptr
2
3  /* some code */
4  for(...){
5          /* some code */
6          MPI_Icoll(args, req_ptr);
7          /* some code */
8          MPI_Wait(req_ptr, status);
9  }
10 /* some code */
11
```

- Freeing already done

Persistent collective

```
1  MPI_Request *req_ptr
2  MPI\_Coll\_init(args\_coll, req\_ptr);
3  /* some code */
4  for(...){
5          /* some code */
6          MPI_Start(req_ptr);
7          /* some code */
8          MPI_Wait(req_ptr, status);
9  }
10 /* some code */
11 MPI_Request_free(req_ptr);
```

- `MPI_Request_free`

  ➔ Freeing

Persistent advantages

What is the advantage of persistent communication?

- In context of repetitive invocation (example: loop)

  ➔ Remove setup overhead

  ➔ Remove freeing overhead

  ➔ Allow Algorithms costly to initialize but more effective

Contribution

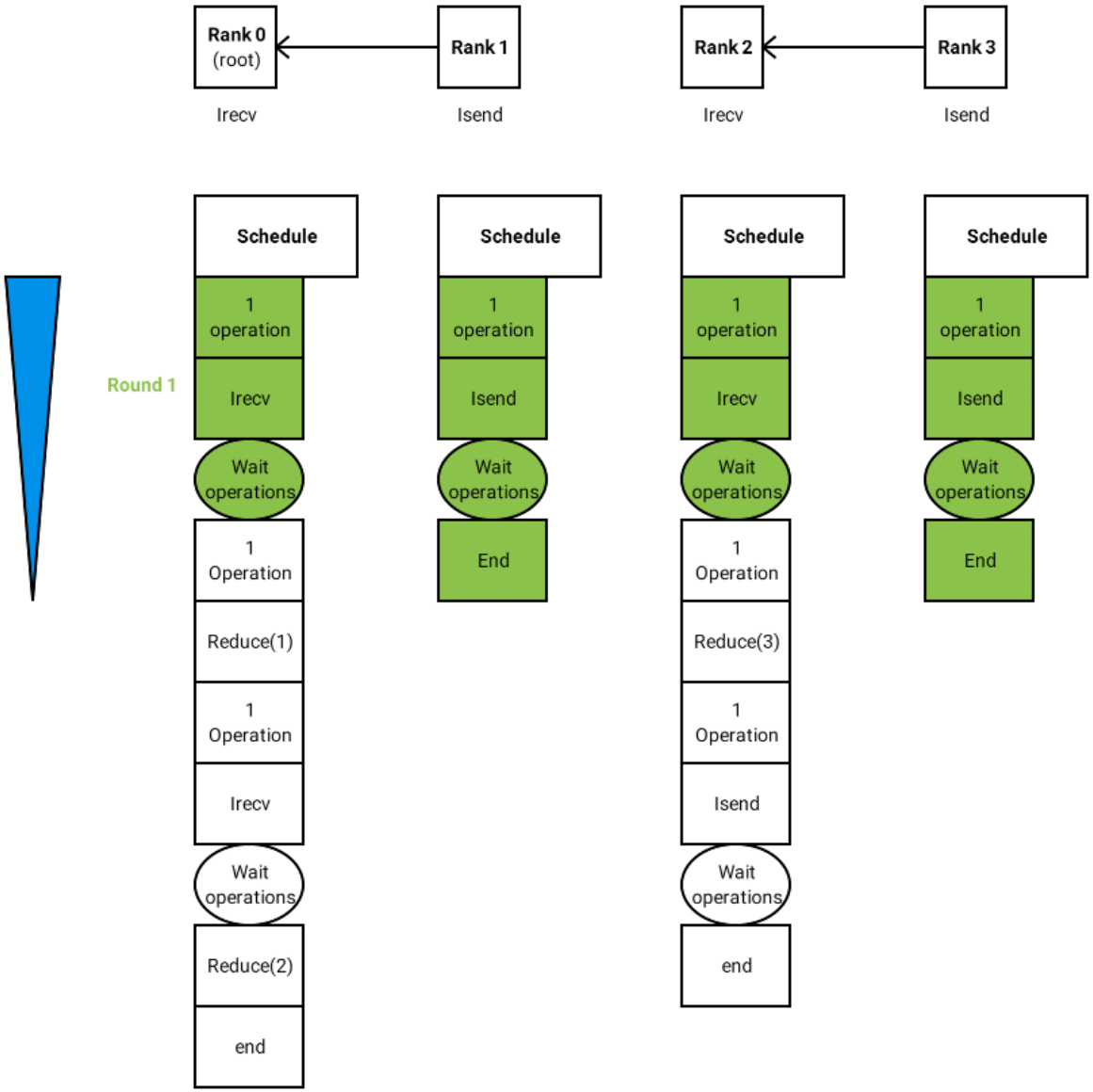
1 - Naïve persistent collective implementation

- Based on non-blocking
- For every MPI runtime implementing non-blocking

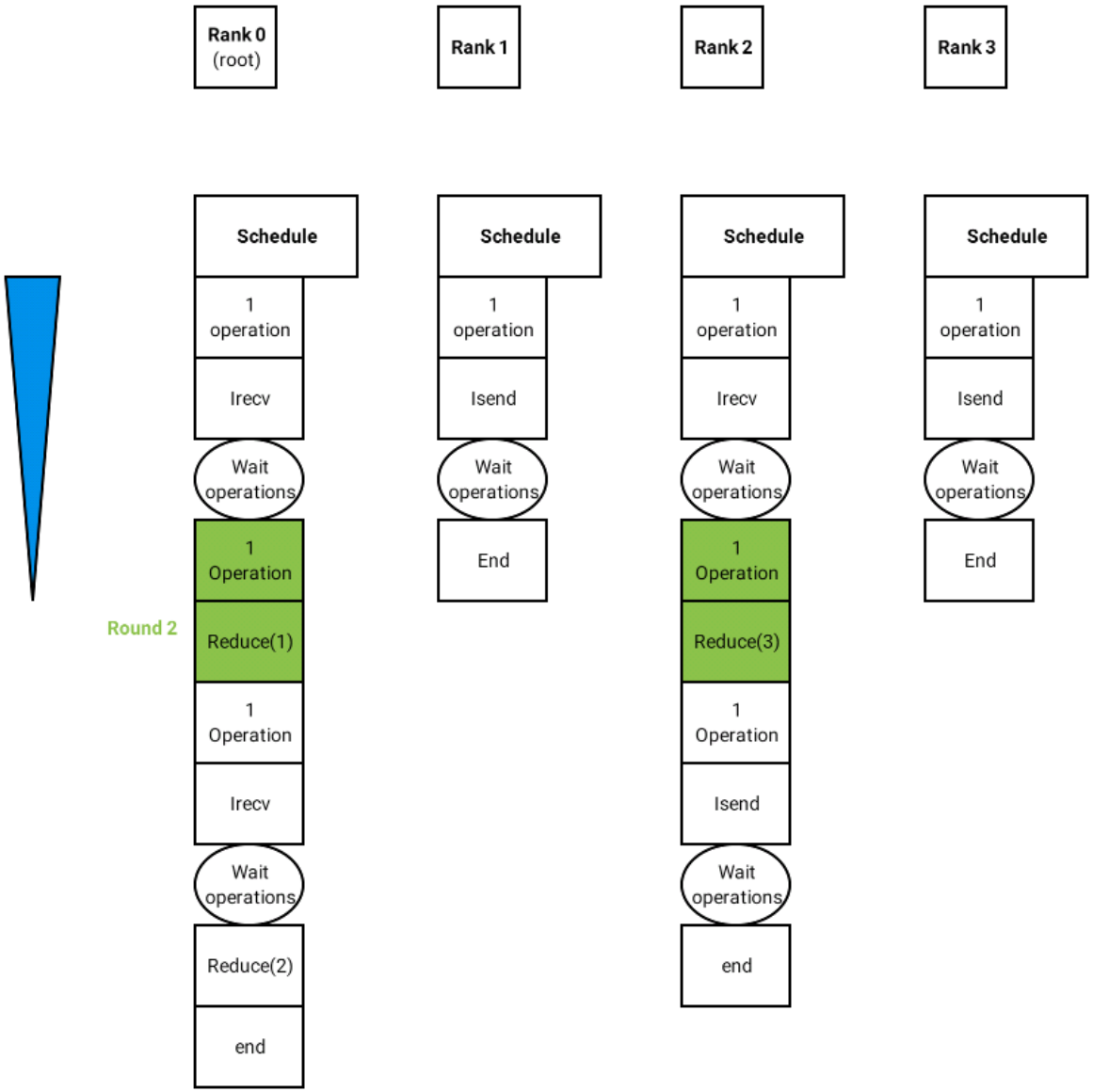
2 - Optimizations based on caching

- Internal scheduling of the collective (similar to libPNBC, 2017)
- Intermediate communications ➔ persistent
- Evaluation of naïve and optimized implementation

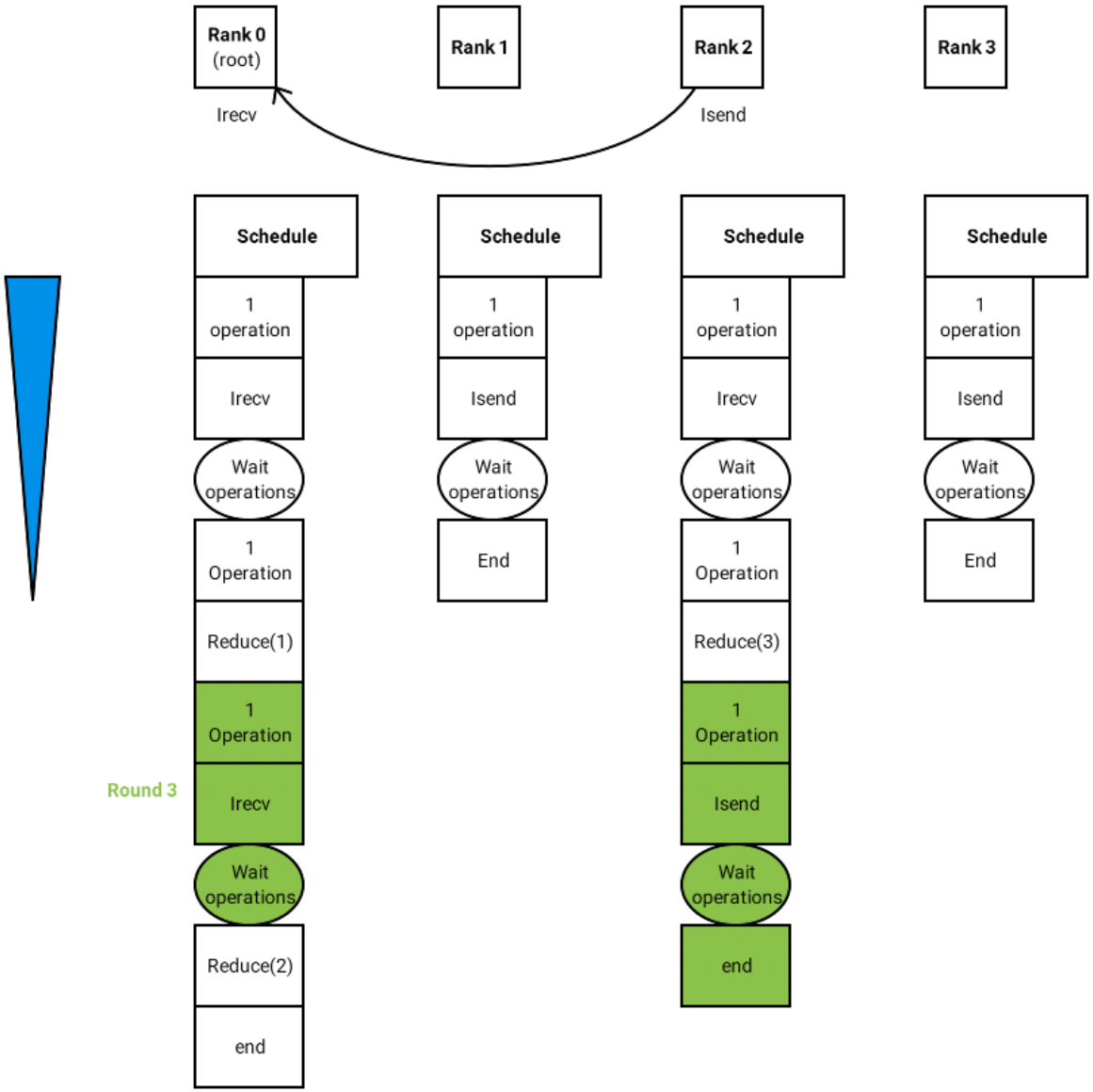
LibNBC mechanism : example binomial reduce (1/4)



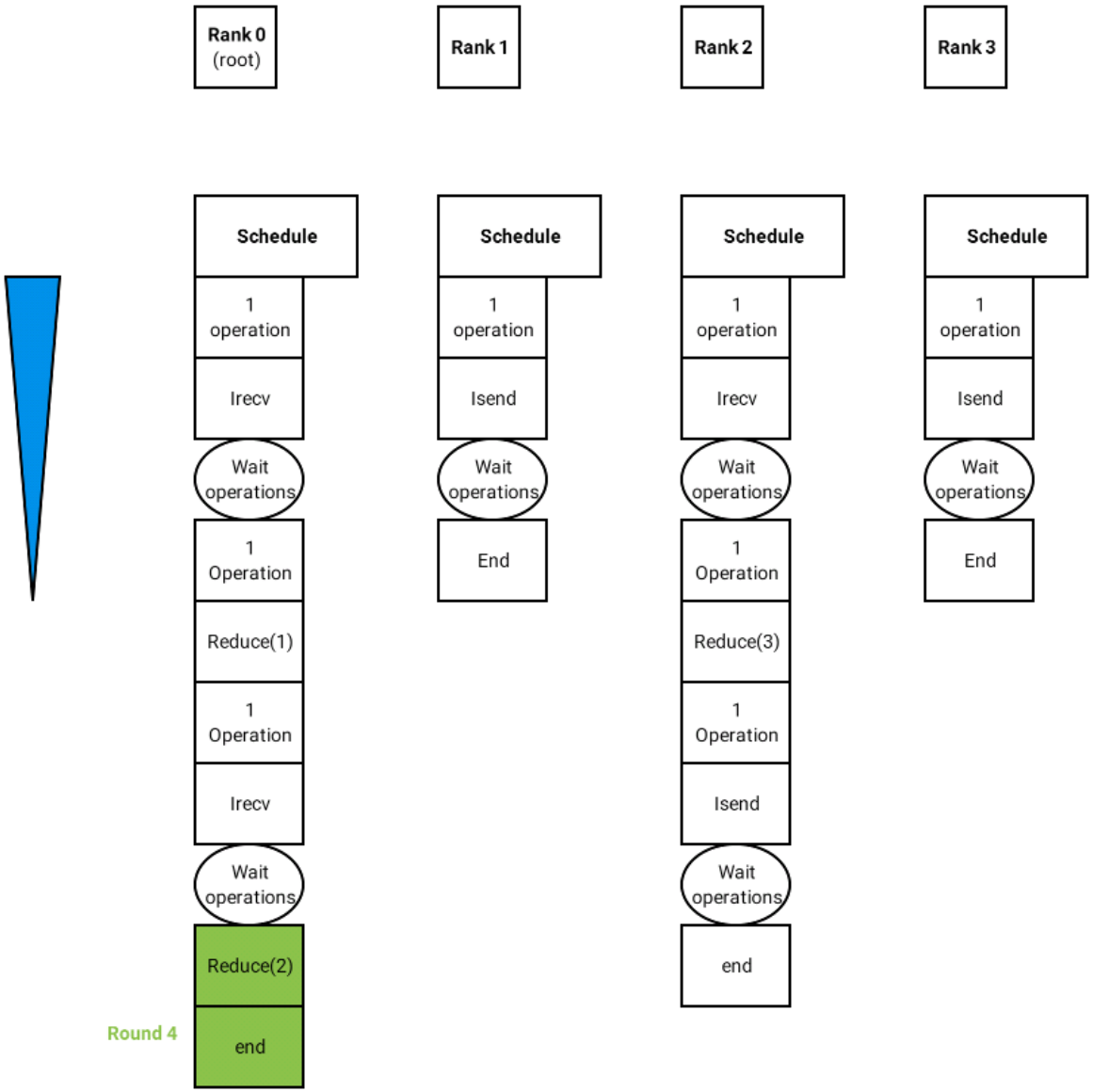
LibNBC mechanism : example binomial reduce (2/4)



LibNBC mechanism : example binomial reduce (3/4)



LibNBC mechanism : example binomial reduce (4/4)

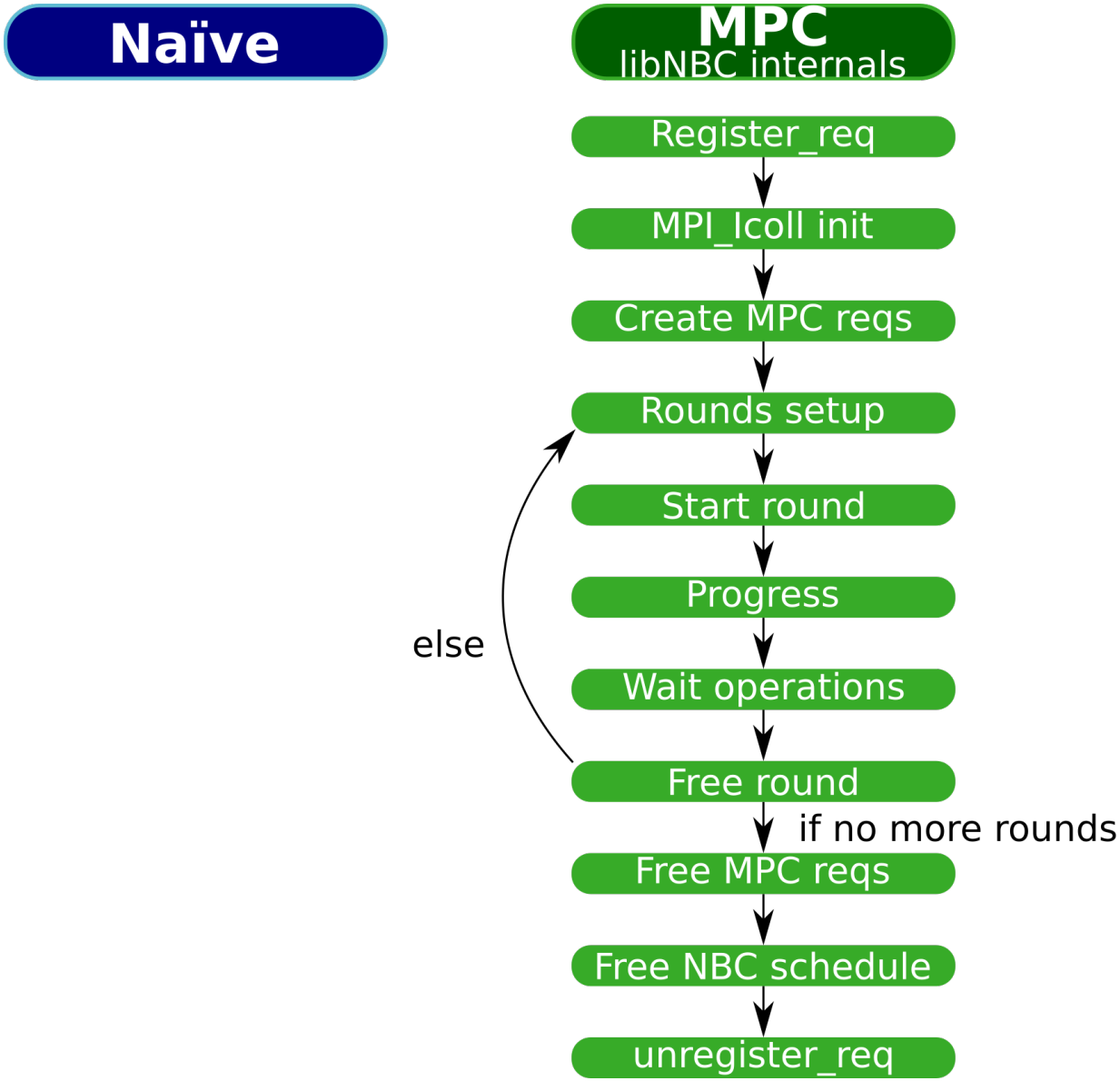


Integration libNBC in MPC

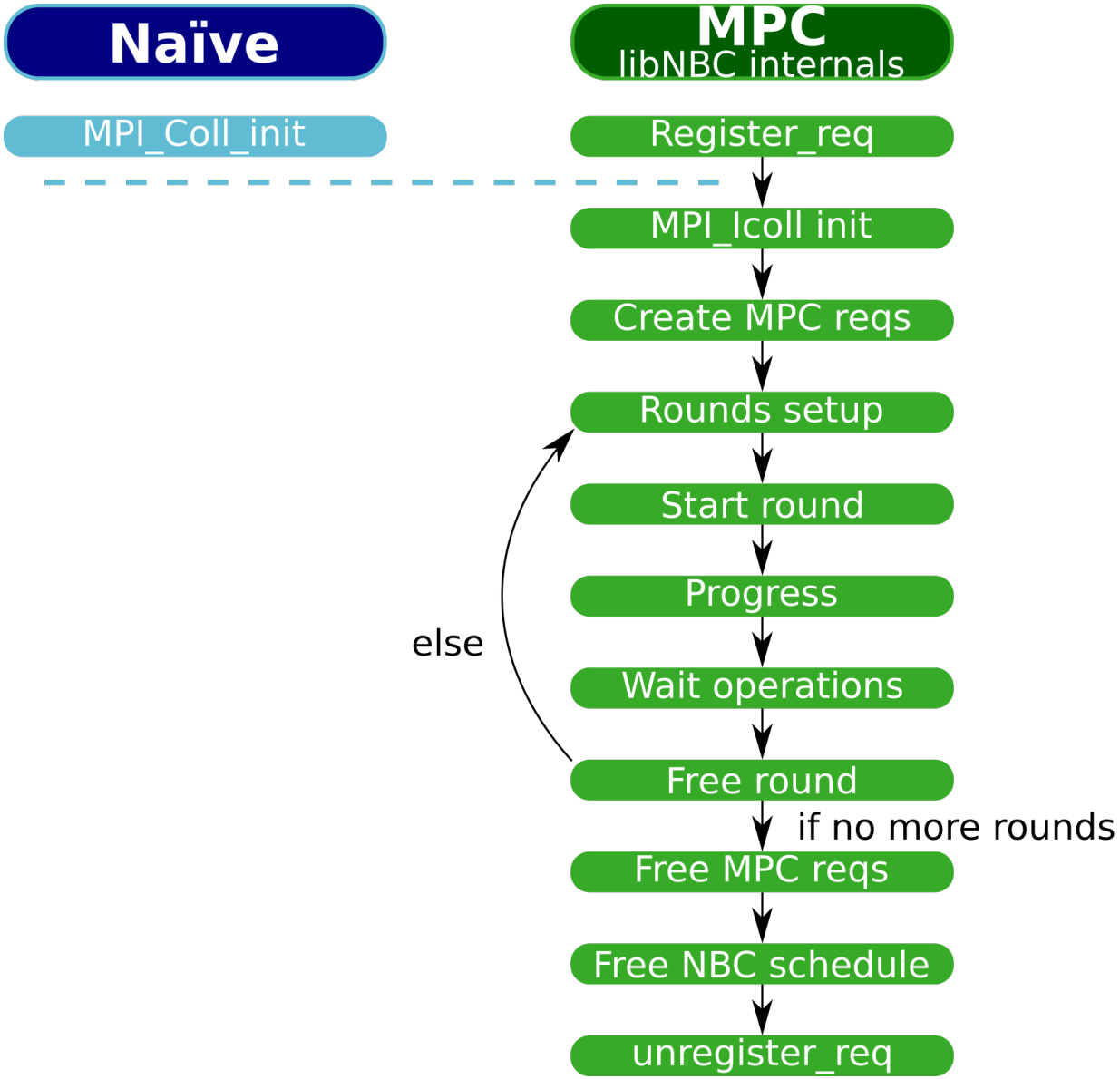
LibNBC optimizations in MPC :

- Schedule was build using realloc for each operation
  ➔ Compute size statically to allocate memory once
- MPI calls
  ➔ MPC internal calls
- Duplicate communicator
  ➔ Internal structures and message tags
- Progress thread was pthread
  ➔ user-level MPC internal thread

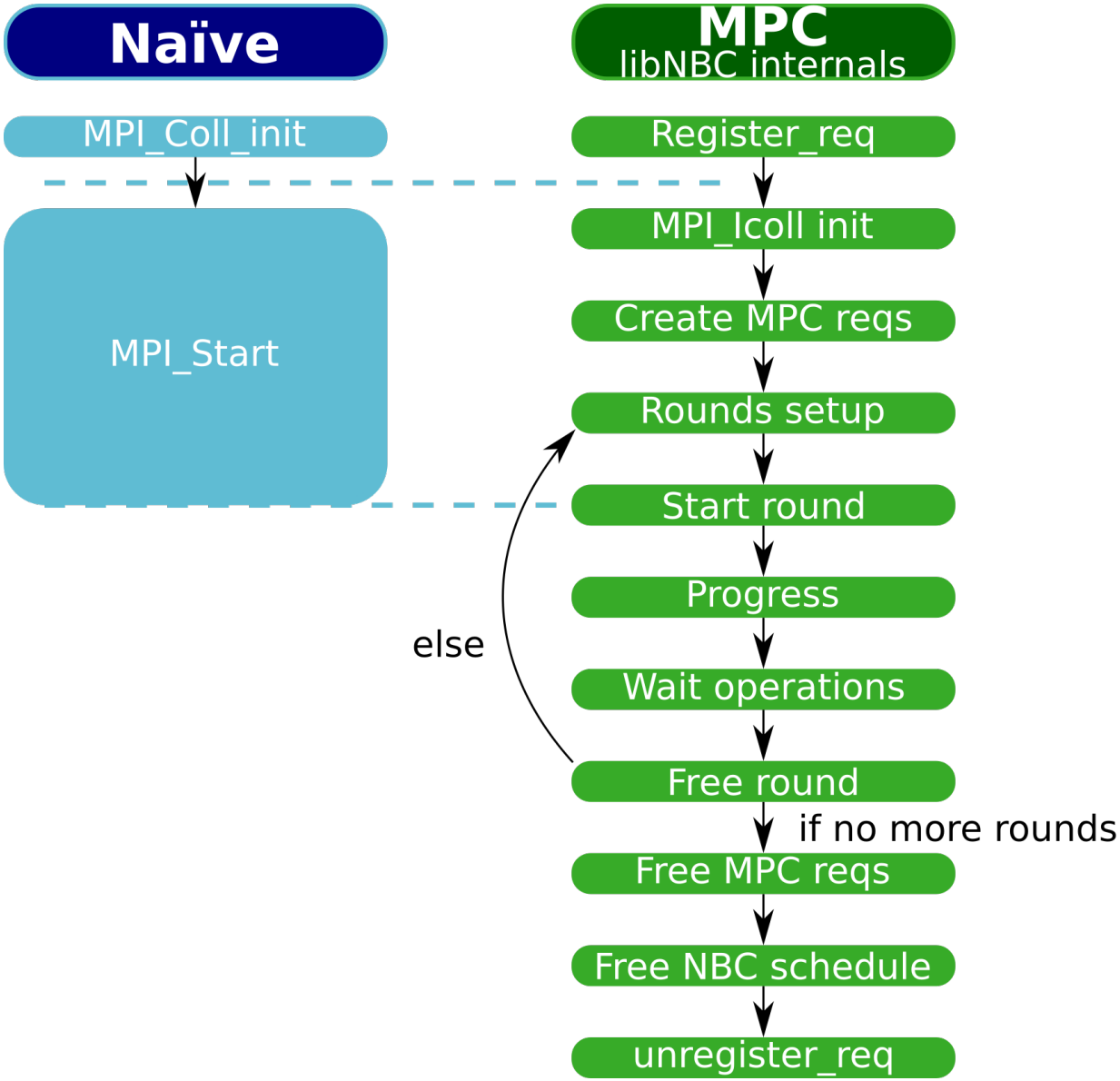
Naïve implementation



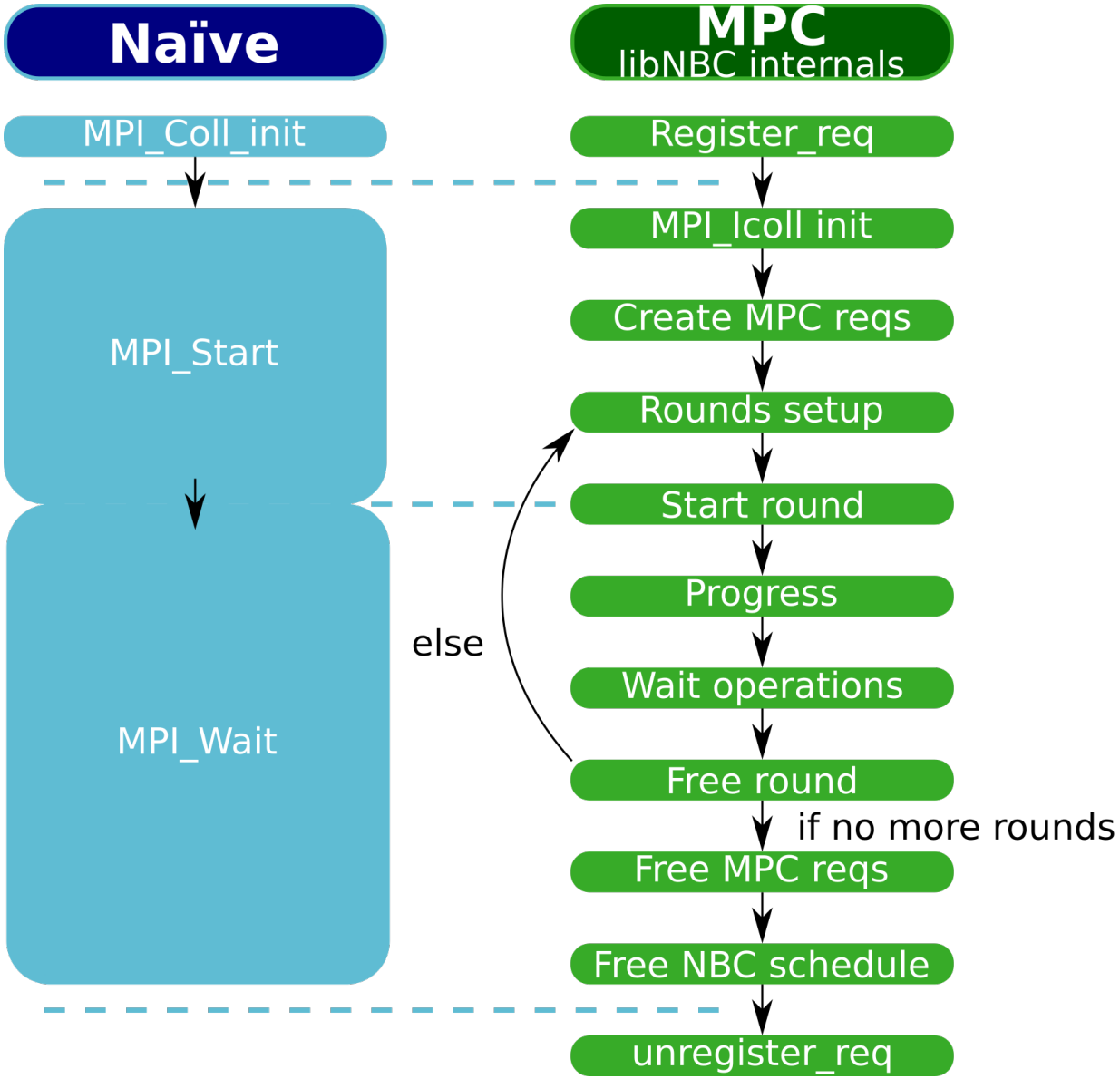
Naïve implementation



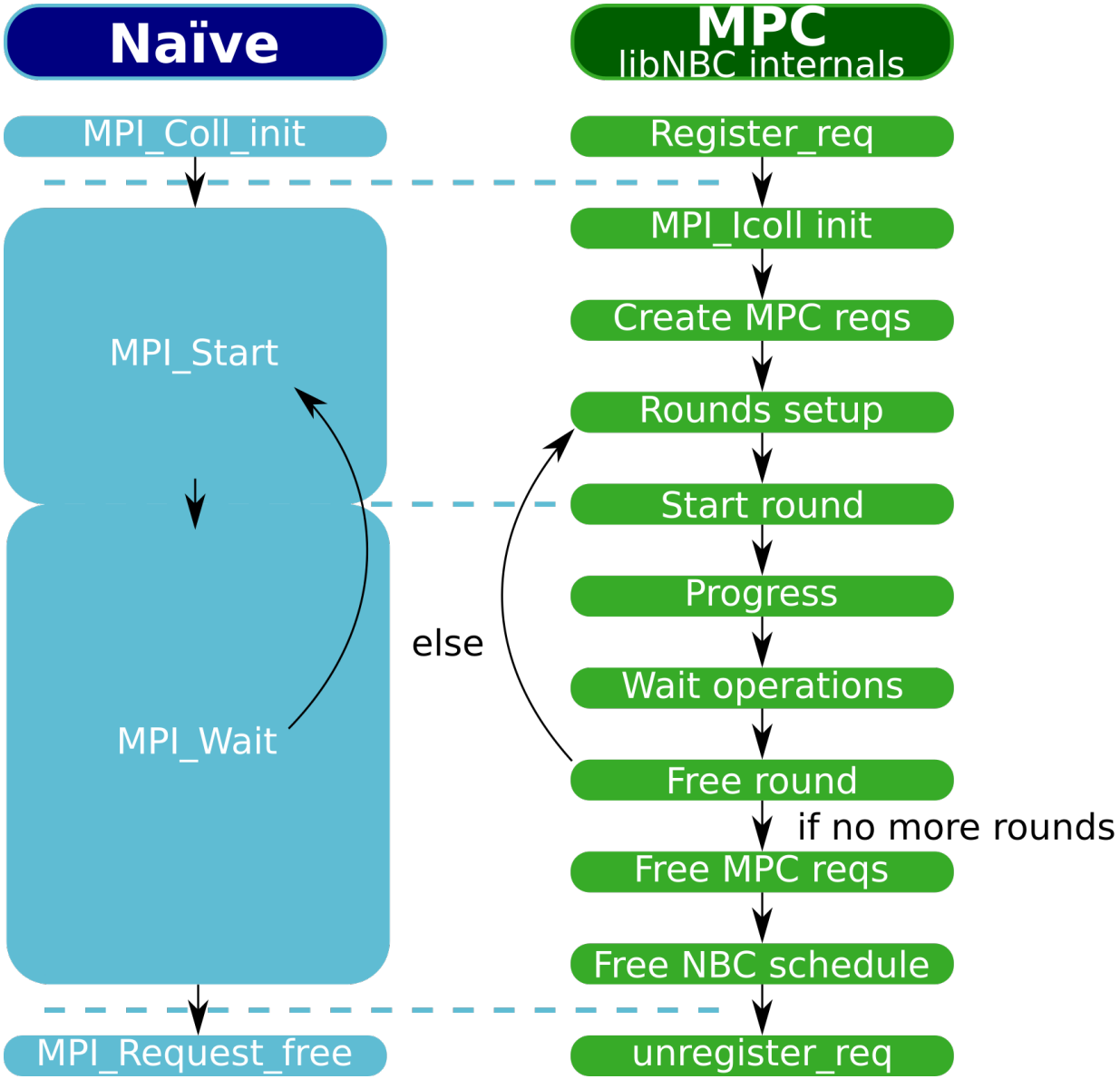
Naïve implementation



Naïve implementation



Naïve implementation



Naïve implementation

Exascale computing research

What about Naïve implementation?

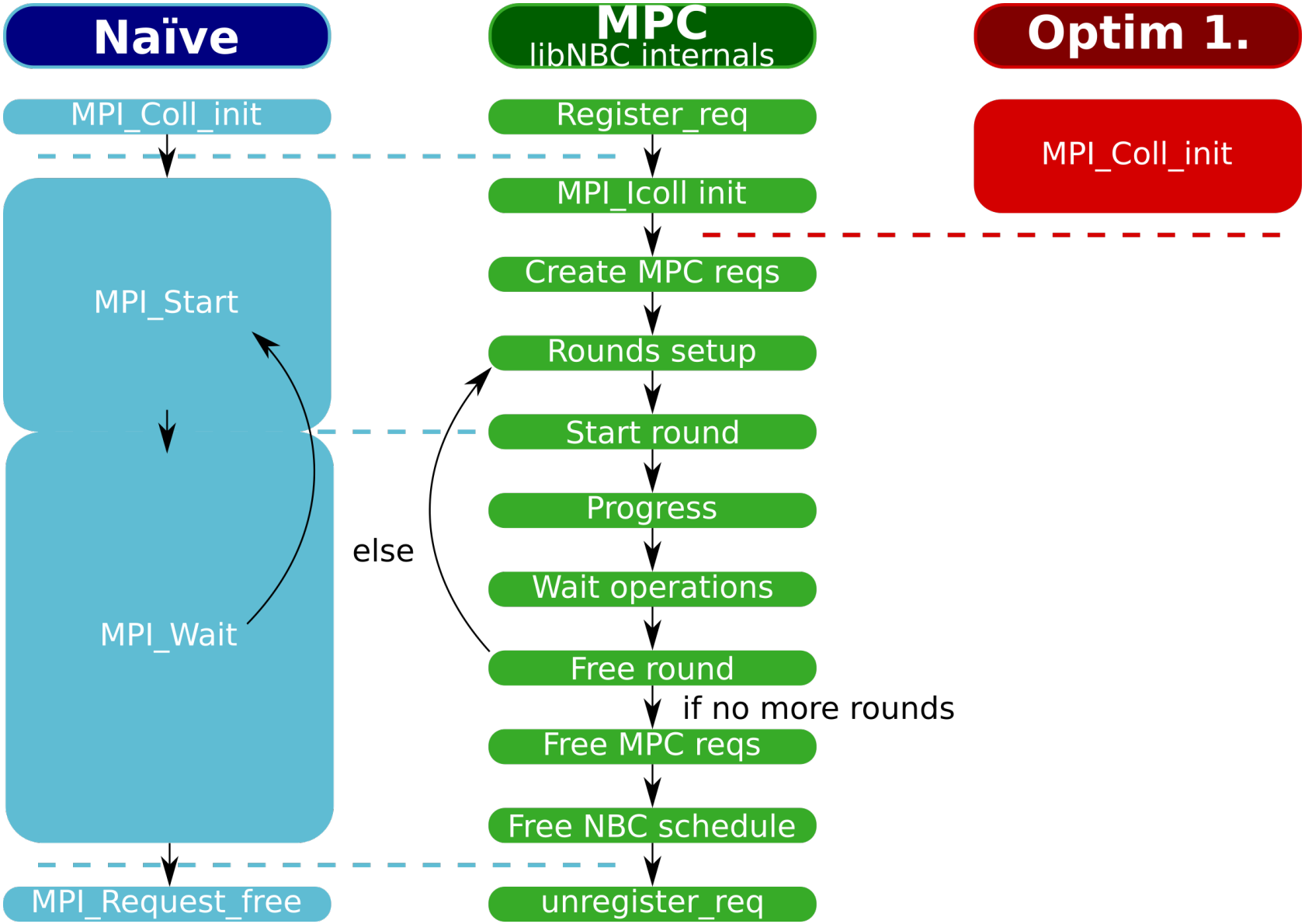
- Based on non-blocking collective
- In accordance with MPI-4 standard
- No benefits via persistent mechanism

➔ Avoidable non-blocking costs at each starting/completion

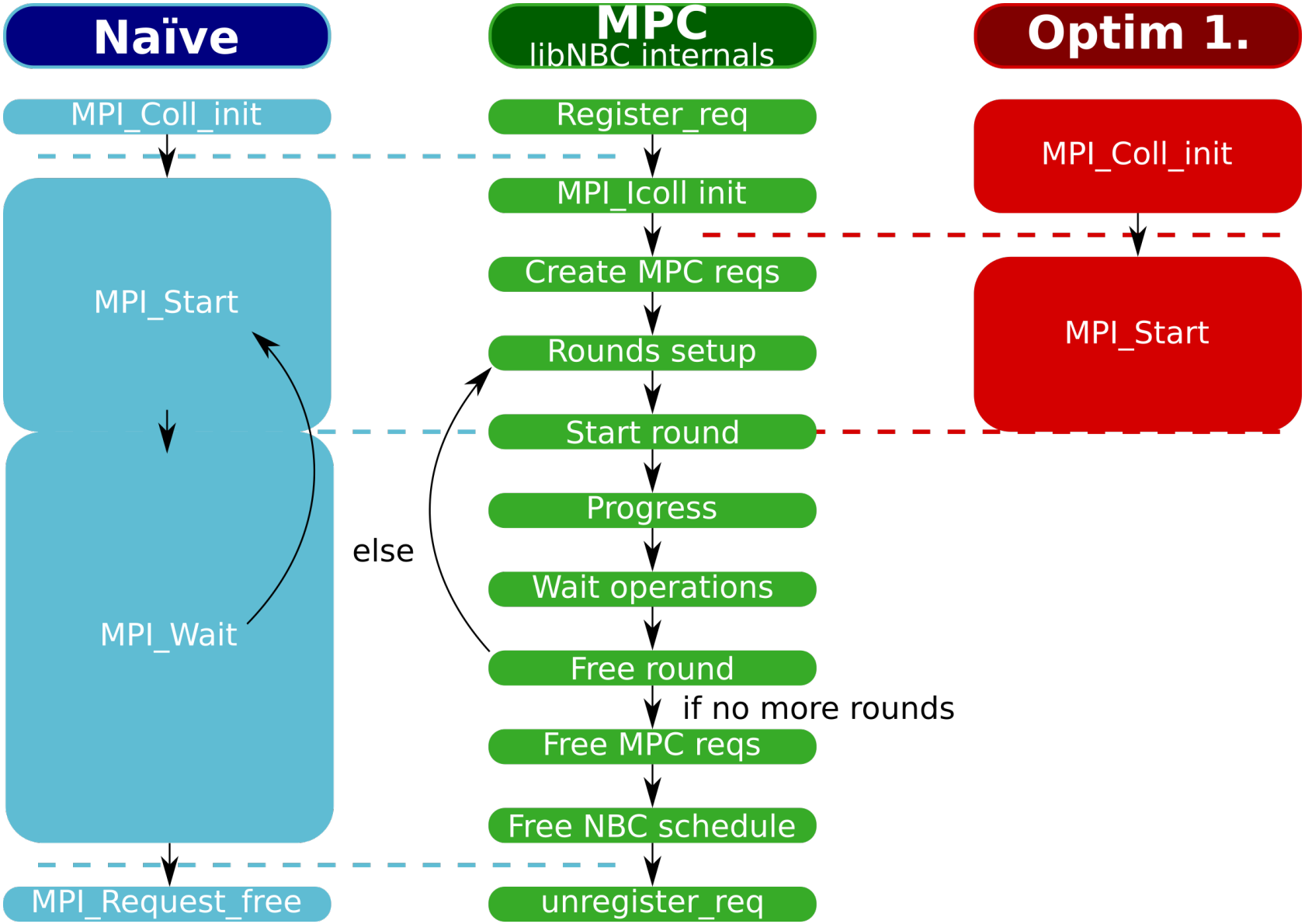
➔ Initialization

➔ Freeing

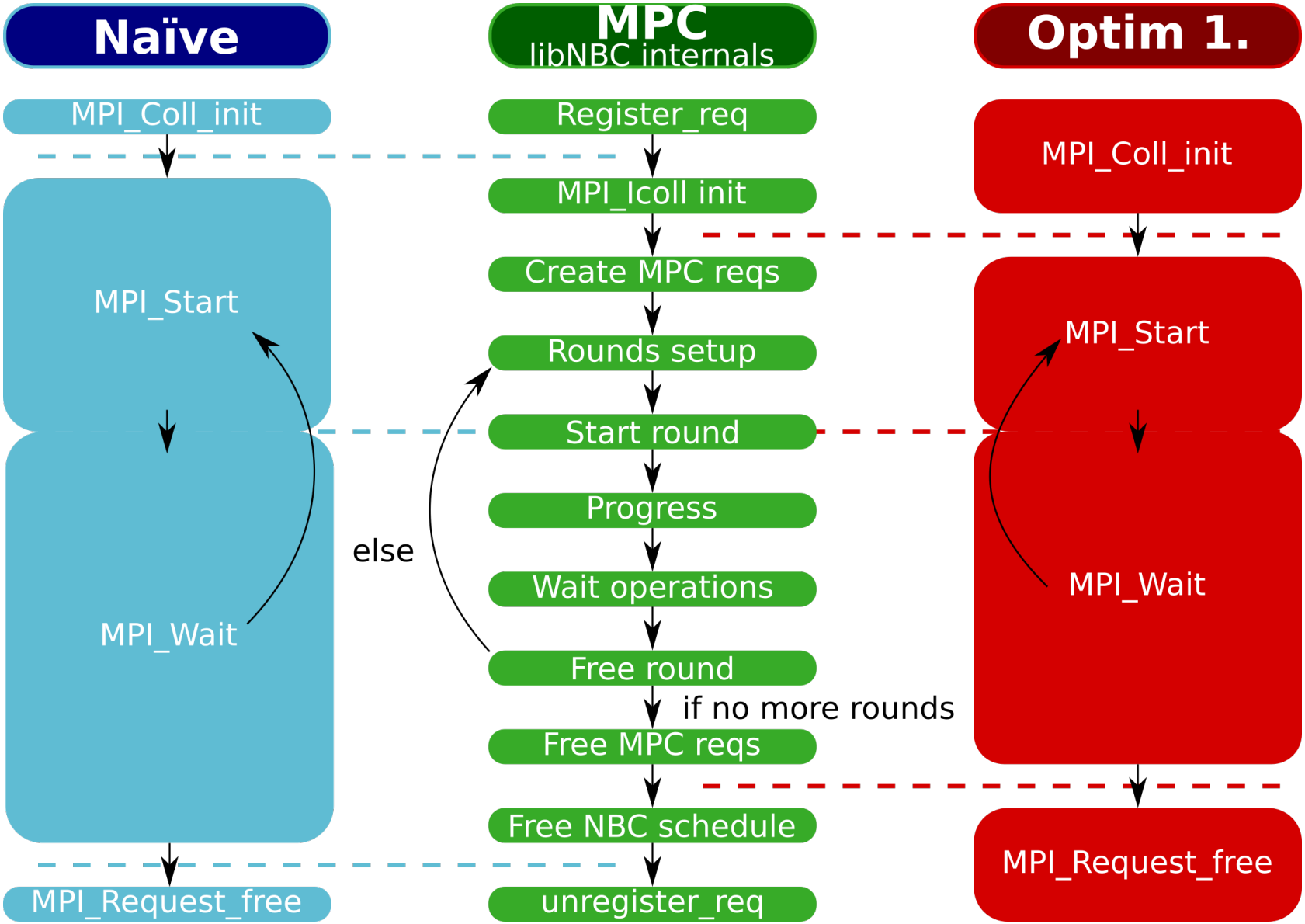
Caching schedule optimization



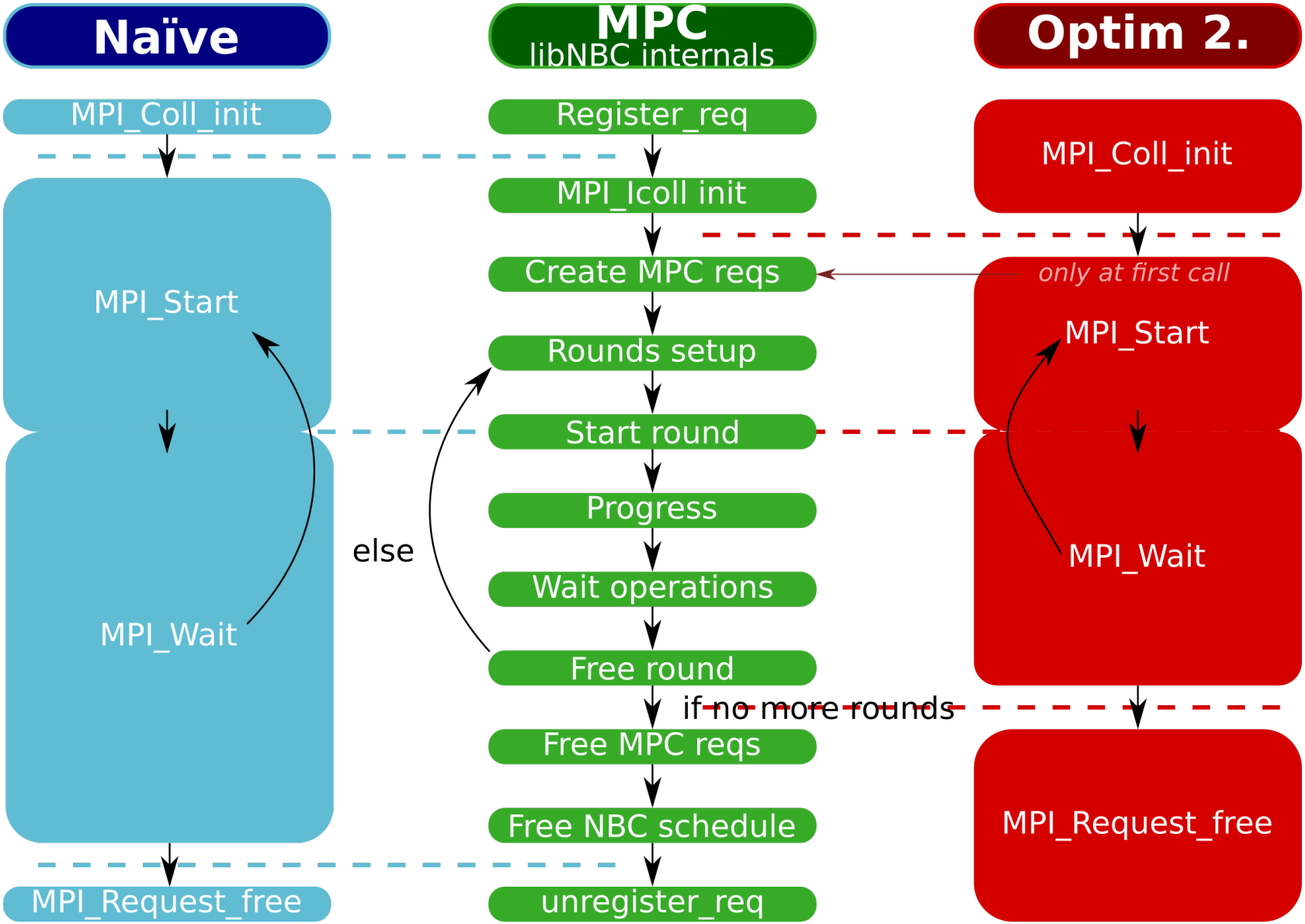
Caching schedule optimization



Caching schedule optimization



Caching internal requests optimization (1/2)



Caching internal request optimization (2/2)

Some insight:

- Why request tagged in first call and not in initialization step?
  - ➔ Avoid case where persistent isn't used but initialized
  - ➔ Waste request structure ressources
- Optimization possible thanks to changes of libNBC in MPC
  - ➔ Static schedule allocation
  - ➔ Arithmetic pointer to parse schedule

Plan

Benchmarks

- Loop, 500 iterations
- Result : slower MPI rank

```
1  MPI_Request req_ptr;
2  double wwtime = 0;
3  MPI_Barrier();
4  START_TIMER(wwtime)
5
6  for(i=0; i<500; i++){
7          MPI_Icoll(args_coll, &req_ptr);
8          MPI_Wait(&req_ptr, status);
9  }
10
11 END_TIMER(wwtime)
12 double recup_max = 0;
13 MPI_Reduce(&wwtime,
   &recup_max, 1, MPI_DOUBLE,
14     MPI_MAX,0, MPI_COMM_WORLD);
```

- Non-blocking

```
1  MPI_Request req_ptr;
2  double wwtime = 0;
3  MPI_Barrier();
4  START_TIMER(wwtime)
5  MPI_Coll_init(args_coll, &req_ptr);
6  for(i=0; i<500; i++){
7          MPI_Start(&req_ptr);
8          MPI_Wait(&req_ptr, status);
9  }
10 MPI_Request_free(&req_ptr);
11 END_TIMER(wwtime)
12 double recup_max = 0;
13 MPI_Reduce(&wwtime,
   &recup_max, 1, MPI_DOUBLE,
14     MPI_MAX,0, MPI_COMM_WORLD);
```

- Persistent

Benchmarks

Exascale computing research

Architectures

- Intel Sandy Bridge

  ➔ Two 8-core sockets (16 cores per node)

Benchmarks variations

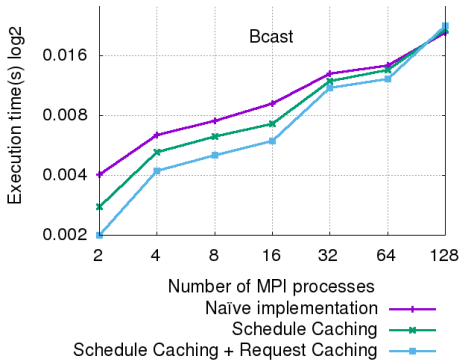
- Number of MPI processes, communicate one integer

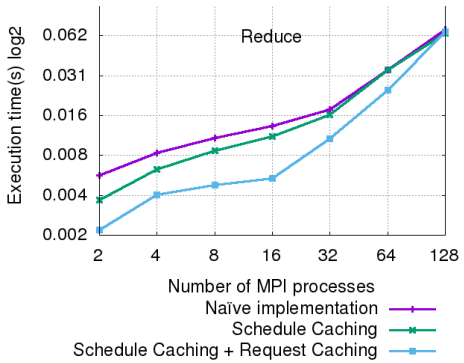
  ➔ Until 8 nodes

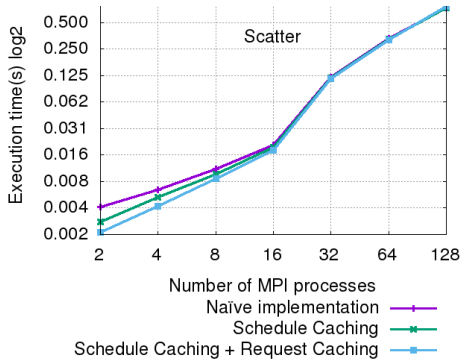
- Size of collective buffer argument on one full node

  ➔ One MPI process per core

One-to-all and All-to-one collectives







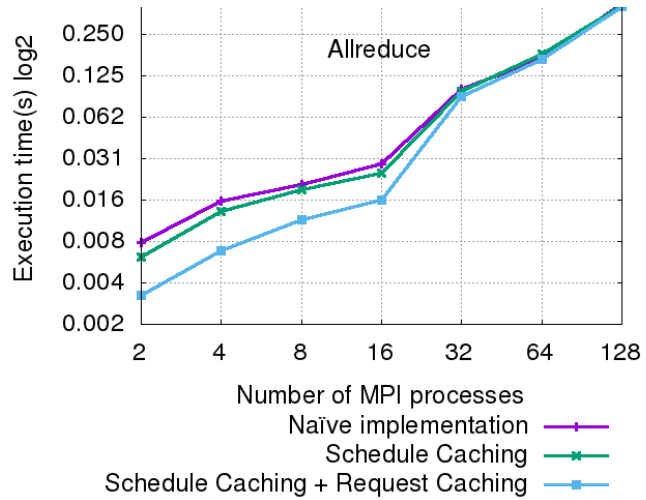
bcast, reduce

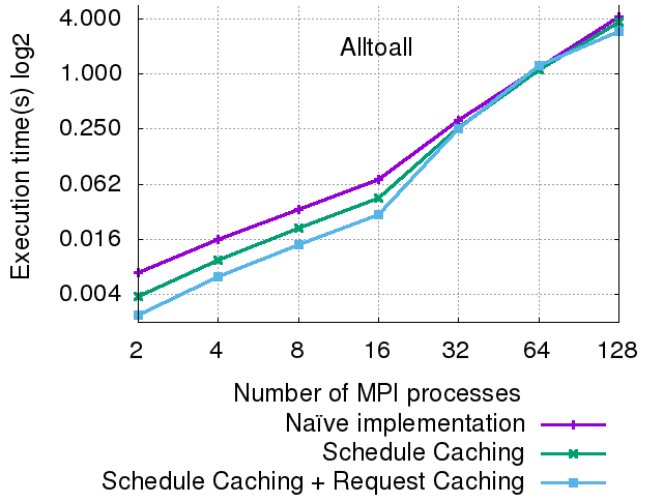
- Binomial tree algorithm
- Several rounds
- Initialization cost
  ➔ + Optimizations impact
- Reduce up to x3 speedup

scatter

- Flat tree algorithm
- Low initialization cost
- One round
- Communication weight
  ➔ - Optimizations impact

All-to-all collectives





Allreduce

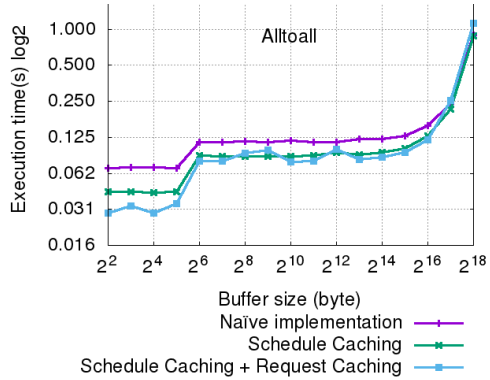
- reduce + bcast

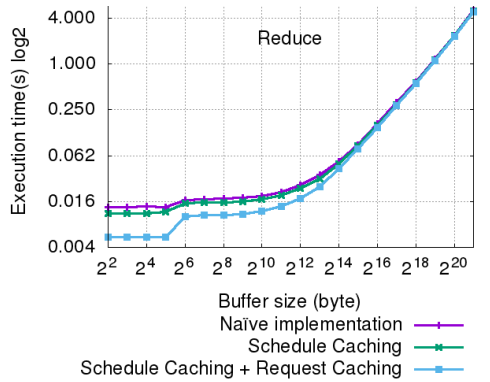
➔ + Optimizations impact

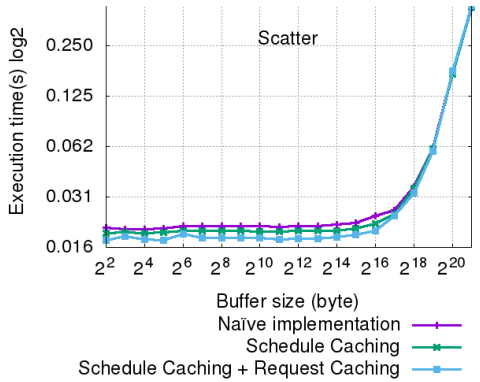
Alltoall

- Parallel communications
- Communication increases

➔ +/- Optimizations impact







- Speedup until very large size 2^{16} bytes
- Impact of optimization decreases with increasing message size
  ➔ Communication time more important
  ➔ Less impact of optimizations
- Alltoall speedup drop from x2 to x1.5
  ➔Due to fast increasing communication time

Conclusion

Naïve persistent based on non-blocking in MPC

- Same performances as non-blocking

Two optimizations

- Schedule caching (similar to libPNBC)
- Intermediate persistent point-to-point
  - ➔ first implementaion, evaluation
  - ➔ Needed deep integration of libNBC in MPC runtime
  - ➔ Optimization impacting the most

x3 speedup on reduce

- ➔ With basic alorithms
- ➔ gains thanks to avoiding building and freeing

Demonstrate potentiel speedup gain

- In libPNBC
- On persistent collective in general

Future works

- Algorithms benefiting from heavier initialization
- Using underlying hardware and network
- Poster : Towards an optimal allreduce communication in message-passing systems

Thank you for your persistent attention.