ERLANGEN REGIONAL
COMPUTING CENTER

CRAFT: A library for application-level Checkpoint/Restart Automatic Fault Tolerance

Faisal Shahzad

06.04.17

ESSEX
Equipping sparse solvers for Exascale

SPPEXA

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Challenge

- Nowadays, the increasing computational capacity is mainly due to extreme level of hardware parallelism.

- The reliability of the hardware components do not increase with the similar rate.

- At exascale-level, the Mean Time To Failure (MTTF) is expected to reduce to the order of hours or minutes.

- e.g.:
  - ‚Intrepid', BlueGene/P, debuted # 4 on top 500, june 08: MTTF 7.5 days[1]
  - ‚Sequoia', BlueGene/Q, debuted # 3 on top 500, Nov. 13: MTTF 19 hrs[2]

- The absence of fault tolerant environment will put the precious data at risk.

1) Dongarra J (2013) Emerging Heterogeneous Technologies for High Performance Computing. http://www.netlib.org/utk/people/JackDongarra/SLIDES/hcw-0513.pdf.
2) M Snir et al (2014) Addressing failures in exascale computing. *International Journal of High Performance Computing Applications.*

CRAFT Introduction:

1. Checkpoint/restart(CR):
   - Target: A simple and extendable library tool for creating „Application-level checkpoints“ with minimal code modifications.
   - Default checkpointable data types: i) PODs(int, double etc.) ii) POD arrays iii) POD multiarrays
   - Extendable --> to include more data-types.
   - Takes care of checkpoint management issues (e.g. restart checkpoint version).
   - Optimizations:
     i. Supports SCR-library (LLNL) for neighbor level C/R.
     ii. MPI-IO for PFS-level checkpoints.
   - Enables: i) Multi-level checkpoint ii) nested checkpoints.
2. Automatic fault tolerance (AFT):
   - Dynamic process failure recovery in case of processor failures*

* Terms and conditions apply

CR: Toy example

- A toy-code without(left) and with CRAFT application-level CR functions (right).

```
//  WITHOUT CRAFT CHECKPOINTS

int main(int argc, char* argv[])
{
  int n=5, iteration=1;
  double dbl = 0.0;
  int * dataArr   = new int[n];









  for(; iteration <= 100 ; iteration++)
  {
    // Computation-communication loop
    modifyData(&dbl, dataArr);





  }
  return EXIT_SUCCESS;
}
```

```
//  WITH CRAFT CHECKPOINTS
#include <craft.h>
int main(int argc, char* argv[])
{
  int n=5, iteration=1, cpFreq=10;
  double dbl = 0.0;
  int * dataArr   = new int[n];
  // ===== DEFINE CHECKPOINT ===== //
  Checkpoint  myCP("myCP", MPI_COMM_WORLD);
  myCP.add("dbl", &dbl);
  myCP.add("iteration", &iteration);
  myCP.add("dataArr", dataArr, n);
  myCP.commit();
  if( myCP.needRestart() == true ){
    myCP.read();  // RESTART CASE
  }
  for(; iteration <= 100 ; iteration++)
  {
    // Computation-communication loop
    modifyData(&dbl, dataArr);
    if(iteration % cpFreq == 0){
      myCP.update();
      myCP.write();
    }
  }
  return EXIT_SUCCESS;
}
```

Checkpoint: A collection of data objects of checkpointable types.

Checkpointable data-type: A data-type that is recognized by add() function of CRAFT.

CR: CRAFT Interface(I)

- Checkpoint::add(…)
  - Default CRAFT *checkpointable* data types include:

    i. Plain Old Data(int, double, float, etc.)

    cp.add („myfloat“, &myfloat);

    ii. POD arrays:

    cp.add („myint“, &myint, arraySize);

    iii. POD multiarrays:

    cp.add („mydbl“, &mydbl, nRows, nCols, toCpCol);

    toCpCol: Column to checkpoint.ALL(default), CYCLIC, An Integer.

- New data types (User-extension): Arguments depend on the data type.

CR: CRAFT Interface(II)

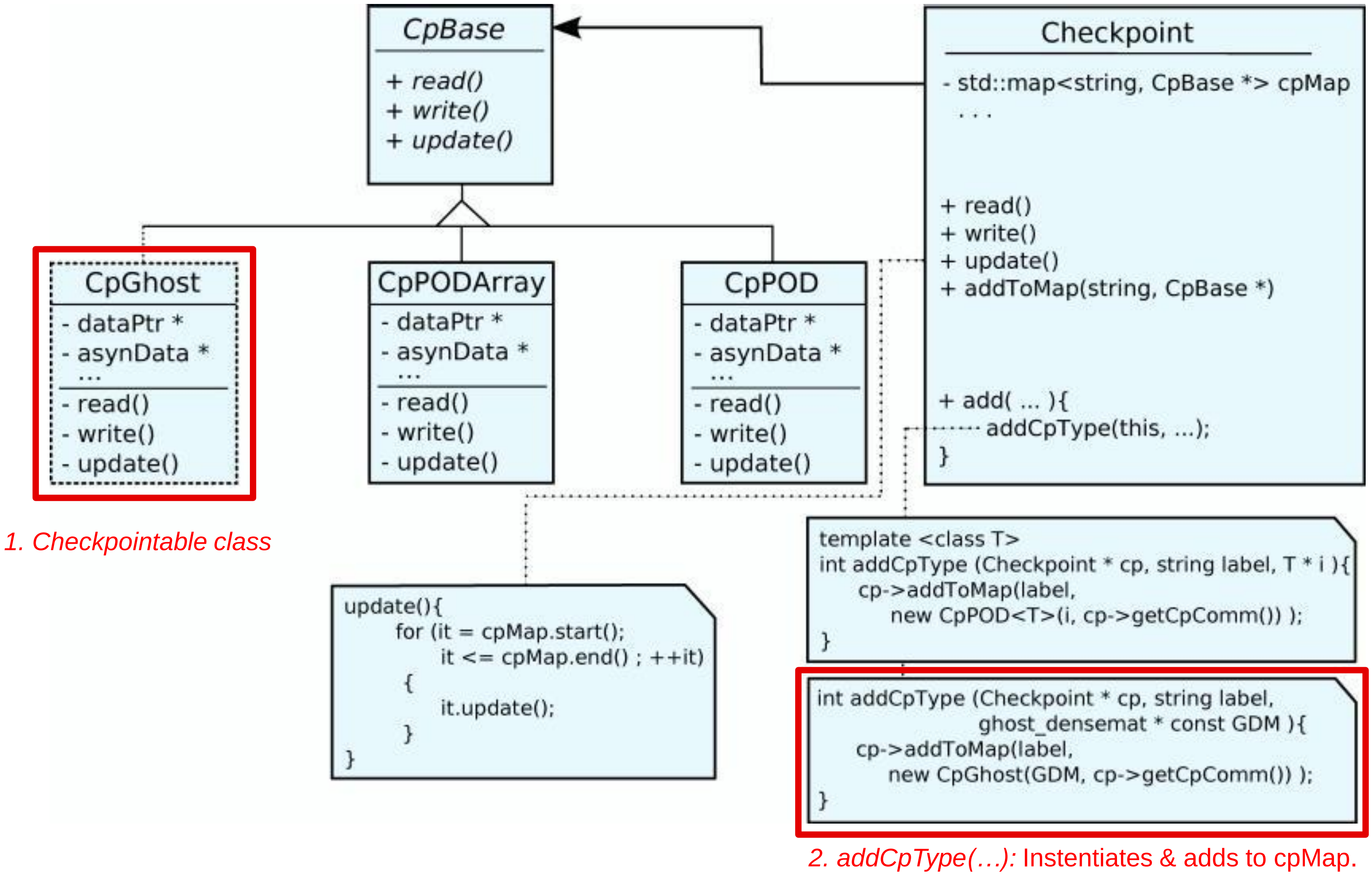
- Checkpoint::update()
  - Update the asynchronous copy of data.

- Checkpoint::write()
  - Iterates through all objects in std::map and updates calles their corresponding write() function, e.g. cpPOD.write() etc.

- Checkpoint::read()

- Checkpoint::needRestart()
  - Checks if there there exist already a copy of the defined checkpoint in the filesystem.

CR: CRAFT design flow



CR: CRAFT extension example(I)

- Extending CRAFT for any arbitrary data-type following 2 steps.

  1. Implementing a "*Checkpointable class*", derived from CpBase, and implementing the read(), write(), and update() functions along with constructor and destructor for the corresponding data type.
  2. Implementing a function addCpType(Checkpoint * cp, ...) in include/addedCpTypes.hpp that instantiates a new object of the above defined class and adds to the 'cpMap'. The structure of this function can be seen in include/addedCpTypes.hpp.

- For example, let us consider the following data type.

```
class rectDomain
{
public:
  rectDomain( const int length, const int width);
  rectDomain( const rectDomain &obj );
  ~rectDomain();
private:
  int length;
  int width;
  double * val;
};
```

CR: CRAFT extension example(II)

- Step 1:

```
#include "cpBase.hpp"

class cpRectDomain: public CpBase
{
public:
  cpRectDomain( rectDomain * dataPtr_, const MPI_Comm cpMpiComm_=MPI_COMM_WORLD){
    dataPtr = dataPtr_;
    *asynData = *dataPtr_;
  }
  ~cpRectDomain(){}
private:
  rectDomain * dataPtr;
  rectDomain * asynData;

  int update(){
    // update asynData from dataPtr
  }

  int write(const std::string * filename){
    // write asynData to the given filename
  }

  int read(const std::string * filename){
    // read asynData to the given filename
  }
};
```

- Step 2:

```
//#include "<CHECKPOINTABLE_CLASS>.hpp"
//int addCpType(Checkpoint * cp, std::string label, <ARGS>){
//  cp->addToMap(label, new <CHECKPOINTABLE_CLASS>( <ARGS>, cp->getCpComm()));
//}

#include "cpTypes/cpRectDom/cpRectDom.hpp"
int addCpType(Checkpoint * cp, std::string label, rectDomain * dataPtr){
  cp->addToMap(label, new cpRectDomain(dataPtr, cp->getCpComm()));
}
```

CR: CRAFT extension example(III)

- Application usage after CRAFT extention for ‚rectDomain‘ data-type:

```
#include <craft.h>
#include <rectDomain.h>
...
int main(int argc, char* argv[])
{
   ...
   int iteration = 1, n = 100, cpFreq = 10;
   rectDomain myRecDom(3, 4);

   Checkpoint myCP( "myCP", MPI_COMM_WORLD);
   myCP.add("iteration", &iteration);
   myCP.add("myRecDom", &myRecDom);
   myCP.commit();

   if( myCP.needRestart() ) { myCP.read(); }
   for(; iteration <= n; iteration++)
   {
     // Computation-communication loop
     if(iteration % cpFreq == 0)
       { myCP.update(); myCP.write();}
   }
   ...
}
```

CR: Optimizations

- Scalable Checkpoint/Restart (SCR) Library support* (developed by LLNL)
    i. Enables node-level & neighbor node-level CR.
    ii. Less frequent PFS-level checkpoints.
- SCR Limitations
    i. Only one Checkpoint instance can be created. (no multi-level & nested checkpoints).
    ii. Each process must write its own checkpoint file independently.

- MPI-IO can be used for all default-supported data-types for PFS-level checkpoint. (without SCR).

* Terms and conditions apply

CR: Directory structure

- Each Checkpoint object maintains and updates its own directroy.
- The directory structure of all checkpoints is flat, i.e, no nested checkpoints.
- Each checkpoints keeps the value of latest valid checkpoint in ‚metadata.ckpt‘.

```
|
-----CP1
          |
         -----v-2/a.ckpt, b.ckpt
         -----v-3/a.ckpt, b.ckpt
         -----metadata.ckpt
|
-----CP2
         |
         -----v-5/x.ckpt, y.ckpt
         -----v-6/
        -----metadata.ckpt
```

CR: Multi-layered checkpoints

```
Checkpoint CL1("CL1", path, FT_Comm);
Checkpoint CL2("CL2", path, FT_Comm);
[...]   // L1iter, L1data
[...]   // L2iter, L2data

if( CL1.needRestart() ){
  CL1.read();
}
for( L1iter ---> nL1iter  )
{
    // L1 COMPUTATION COMMUNICATION
    if( CL2.needRestart() ){
        CL2.read();
    }
    for( L2iter ---> nL2iter  ) {
        // L2 COMPUTATION COMMUNICATION
        CL2.write();
    }
    // L1 COMPUTATION COMMUNICATION
  CL1.write();
}

Checkpoint CL3(...);
[...]   // L3iter, L3data

if( CL3.needRestart() ){
  CL3.read();
}
for( L3iter ---> nL3iter  ) {
    // COMPUTATION COMMUNICATION
  CL3.write();
}
```

CL2

CL1

CL3

- **„CL1“ and „CL2“ form a nested structure of checkpoints.**
- **All initializations of nested Checkpoints must be done only once.**

CR: hybrid PFS/node-level CRAFT Checkpoints

- **High frequency CPs => node-level via SCR**
- **Low frequency CPs => PFS.**
- **In multi-level checkpoint environment, only one checkpoint can be stored using SCR.**

- **CRAFT compiled with SCR.**
- **SCR can be disabled for any particular Checkpoint object.**
- **Disabled SCR-checkpoints are stored at PFS**

```
Checkpoint CL1("CL1", FT_Comm);
Checkpoint CL2("CL2", FT_Comm);
[...]    // L1iter, L1data
[...]    // L2iter, L2data
CL1.disableSCR();

************************************
if( CL1.needRestart() ){            *
  CL1.read();                       *
}                                   *
for( L1iter ---> nL1iter  )         *
{                                   *
  /* L1 COMPUTATION COMMUNICATION */ *
  ****************************      *
  if( CL2.needRestart() ){     *    *
    CL2.read();                *    *
  }                            *    *
  for( L2iter ---> nL2iter ) { *   CL1
 /* L2 COMPUTATION COMMUNICATION */ CL2 *
    CL2.update();              *    *
    CL2.write();               *    *
  }                            *    *
  ****************************      *
  /* L1 COMPUTATION COMMUNICATION */ *
  CL1.update();                     *
  CL1.write();                      *
}                                   *
************************************
```

Listing 3: A pseudo example of multilevel, nested checkpoints using CRAFT. The usage of SCR is disabled for CL1. Thus only high-freqency, smaller checkpoints of CL2 are stored on node-levels.

Automatic fault tolerant (AFT):

- AFT: Dynamic process(es) recovery in case of process failure(s).

1. A fault tolerant communication (avoids deadlocks in case of failed processes)
   - ULFM-MPI
   - Error handler / MPI call return value
   - Error propagation via MPI_Comm_revoke()

2. Communication recovery
   - Shrinking/Spawning

3. Data recovery
   - Easy option: Checkpoint/Restart
   - Algorithm Based Fault Tolerance

Automatic fault tolerance(AFT):

- An ‚AFT-zone‘ is created between AFT_BEGIN() and AFT_END() region.

- The process failures (of the given communicator) within the AFT-zone are recovered dynamically.

- Communicator Recovery options:
  1. Shrinking
  2. Non-shrinking

```
#include <craft.h>
int main(int argc, char* argv[])
{
   ...
   int myrank;
   MPI_Comm FT_Comm;
   MPI_Comm_dup(MPI_COMM_WORLD, &FT_Comm);
   AFT_BEGIN(FT_Comm, &myrank, argv);
   double data = 0;
   int iteration = 0, cpFreq = 10;
   Checkpoint myCP( "myCP", FT_Comm);
   myCP.add("data", &data);
   myCP.add("iteration", &iteration);
   myCP.commit();

   if( myCP.needRestart() ) {myCP.read();}
   for(; iteration <= n; iteration++)
   {
     // Computation-communication loop
     if(iteration % cpFreq == 0)
       { myCP.update();  myCP.write();}
   }
   ...
   AFT_END();
}
```

AFT-zone

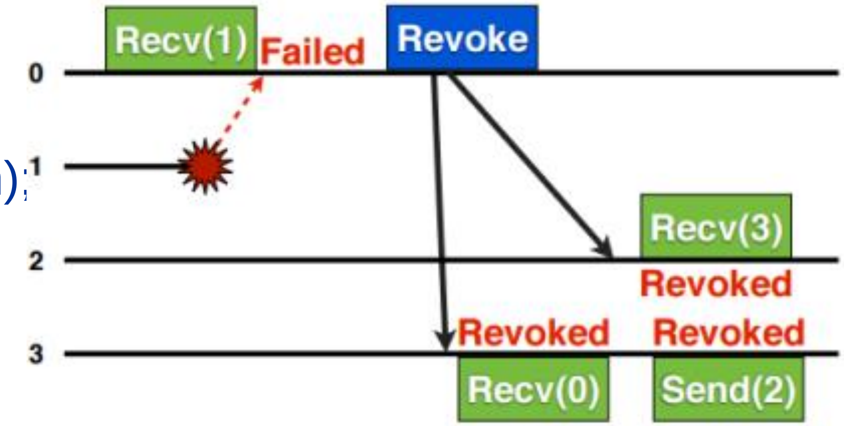
Introduction to ULFM* (I)

- The User Level Failure Mitigation (ULFM) proposal is developed by MPI-Forum‘s Fault Tolerance Working Group.
- Target: To provide a simple, flexible and deadlock-free API that helps users to recover from failed communications due to process-failure.
- This is NOT an application recovery API.
- Once the communication is restored, the data recovery is user‘s responsibility.
- Implemented as an mpi-extension on top of Open MPI implementation
- Early stage implementation: to test correctness, not performance.

* http://fault-tolerance.org/

Introduction to ULFM (II)

- Error handler on working communicator:
  - MPI_ERRORS_RETURN
  - User defined error handler
- **MPIX_Comm_revoke**(MPI_Comm comm);



- **MPIX_Comm_shrink**(MPI_Comm old_comm, MPI_Comm * new_comm);
- **MPIX_Comm_agree**(MPI_Comm comm, int * flag);
- **MPI_Comm_spawn**(… numproc_spawn, spawn_info, &icomm …);
- Merging intercomm to give an interacomm + reordering ranks to give spawned process same rank as dead-rank.
  - MPI_Intercomm_merge()
  - MPI_Group_translate_ranks()

Image courtesy: http://meetings.mpi-forum.org/2014-11-scbof-ft.pdf

AFT: AFT-zone

```
#include <craft.h>
int main(int argc, char* argv[])
{
   ...
   int myrank;
   MPI_Comm FT_Comm;
   MPI_Comm_dup(MPI_COMM_WORLD, &FT_Comm);
   AFT_BEGIN(FT_Comm, &myrank, argv);
   double data = 0;
   int iteration = 0, cpFreq = 10;
   Checkpoint myCP( "myCP", FT_Comm);
   myCP.add("data", &data);
   myCP.add("iteration", &iteration);
   myCP.commit();

   if( myCP.needRestart() ) {myCP.read();}
   for(; iteration <= n; iteration++)
   {
      // Computation-communication loop
      if(iteration % cpFreq == 0)
        { myCP.update();  myCP.write();}
   }
   ...
   AFT_END();
}
```

AFT-zone

AFT: AFT-zone

```
#define AFT_BEGIN(CRAFT_comm_working, CRAFT_myrank, CRAFT_argv)
{
    int CRAFT_aftFailed = false;
    do
    {
        try{
        MPI_Comm CRAFT_parent;
        MPI_Errhandler CRAFT_errh;
        MPI_Comm_create_errhandler(&CRAFT_errhandlerRespawn, &CRAFT_errh);
        MPI_Comm_get_parent( &CRAFT_parent );
        if( CRAFT_parent == MPI_COMM_NULL && CRAFT_aftFailed == false){
            MPI_Comm_dup (MPI_COMM_WORLD, &CRAFT_comm_working);
            MPI_Comm_rank (CRAFT_comm_working, CRAFT_myrank);
            CRAFT_getEnvParam();
            CRAFT_removeMachineFiles(&CRAFT_comm_working);
            CRAFT_initRescueNodeList(&CRAFT_comm_working);
            MPI_Comm_set_errhandler(CRAFT_comm_working, CRAFT_errh);
        }
        if ( CRAFT_parent != MPI_COMM_NULL || CRAFT_aftFailed == true)  {
            if( CRAFT_parent != MPI_COMM_NULL){
                CRAFT_comm_working = MPI_COMM_NULL;
                CRAFT_getEnvParam();
                CRAFT_aftFailed = true;
            }
            CRAFT_appNeedsRepair(&CRAFT_comm_working, CRAFT_argv);
            MPI_Comm_set_errhandler(CRAFT_comm_working, CRAFT_errh);
            MPI_Comm_rank(CRAFT_comm_working, CRAFT_myrank);
        }

#define AFT_END()
    CRAFT_aftFailed = false;
        }catch(int CRAFT_exception_val){
         CRAFT_aftFailed = true;
        }
    }
  while(CRAFT_aftFailed == true);
}
```

AFT_BEGIN

- ‚try-catch‘ block in a ‚do-while‘ loop.
- ‚do-while‘ loop runs untill try block is run successfully

First run: duplication of MPI_COMM_WORLD is created and errhandler is assigned to new comm.

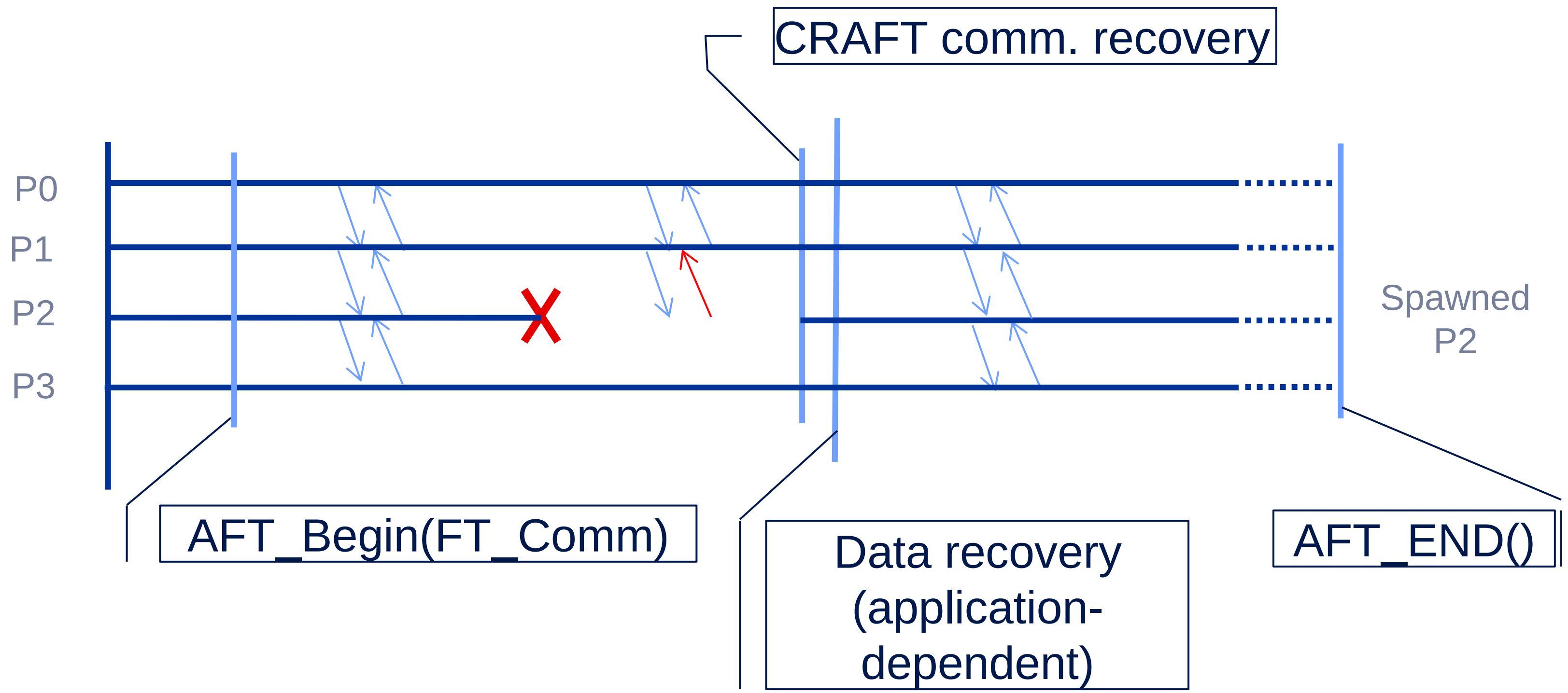
In case of Failure: errhandler revokes the comm_working.

Spawned + surviving procs.

After app_needs_repair() call, the spawned process is merged into ‚comm_working‘ and it has the same rank as dead processs.

AFT_END

Automatic fault tolerant (AFT): Process recovery



AFT: Communication recovery options

- Shrinking

Pros	Cons
✓ No extra resources (nodes) needed	○ Domain may need redistribution for effective resource utilization. ○ For domain redis., one checkpoint for whole job => no SCR.

- Non-shrinking

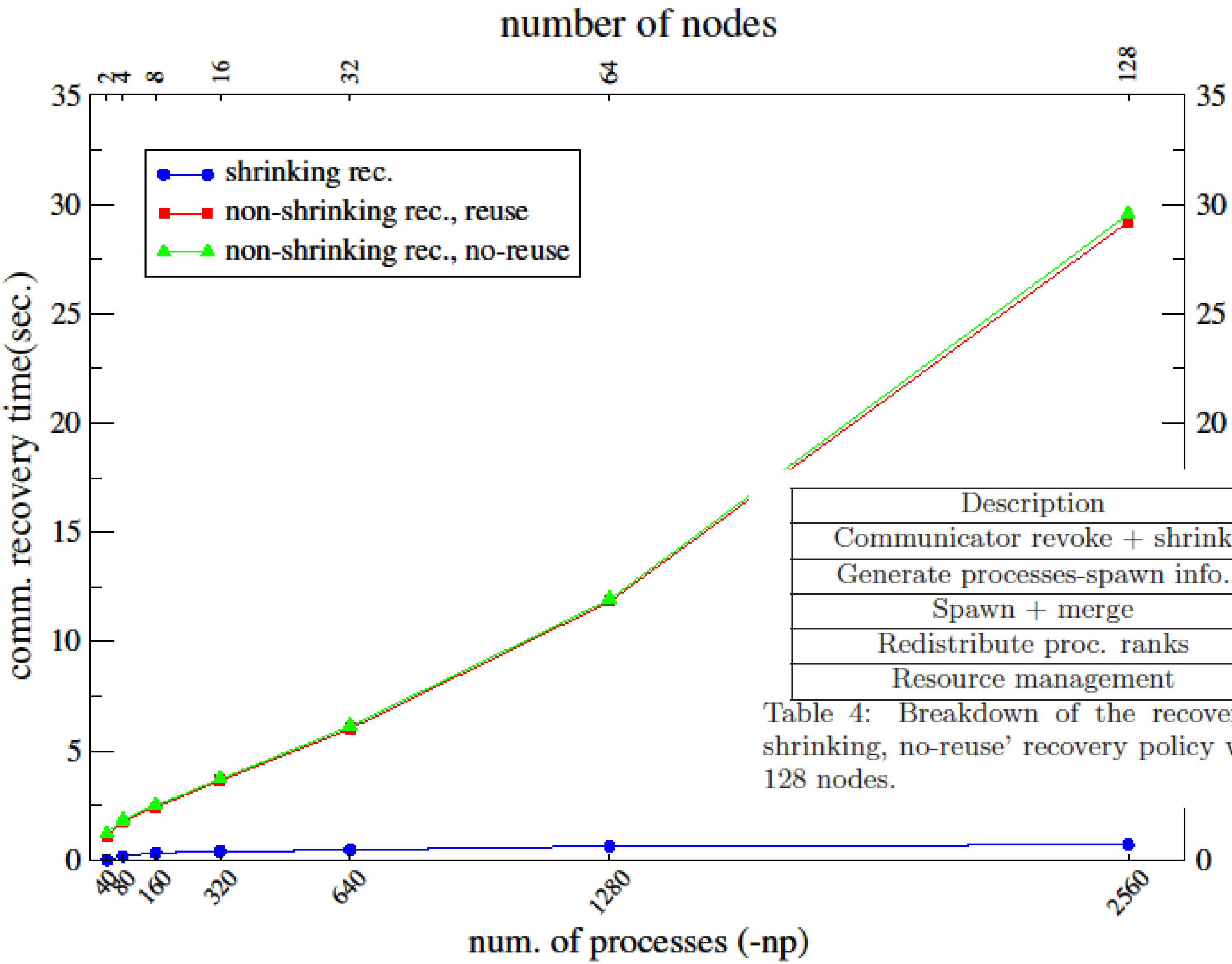
Pros	Cons
✓ If procs. are spawned at same node, no extra resources needed. ✓ No redistribution of domain.	○ If new nodes are used for spawned procs., preallocation of extra nodes is necessary.

CRAFT Parameters:

Parameter Name	Description
CRAFT_CP_PATH	The path to checkpoint. Default: The application directory.
CRAFT_ENABLE	Enables/disables CRAFT checkpointing. Values: 1(default),0
CRAFT_USE_SCR	Controls the usage of SCR. If CRAFT is compiled with SCR, its usage can still be disabled by this variable. Values: 1(default), 0
CRAFT_READ_CP_ ON_RESTART	Controls whether the restarted-run should resume by reading checkpoints or not. Values: 1(default), 0
CRAFT_COMM_ RECOVERY_POLICY	Controls the method of communicator recovery. Values: NON-SHRINKING(default), SHRINKING
CRAFT_COMM_ SPAWN_POLICY	Determines the node-locality of spawned processes in case of Non-shrinking recovery. Values: NO-REUSE(default), REUSE

Table 1: The CRAFT parameters description. Note: 1=enable, 0=disable

CRAFT Benchmarks: comm. recovery scaling

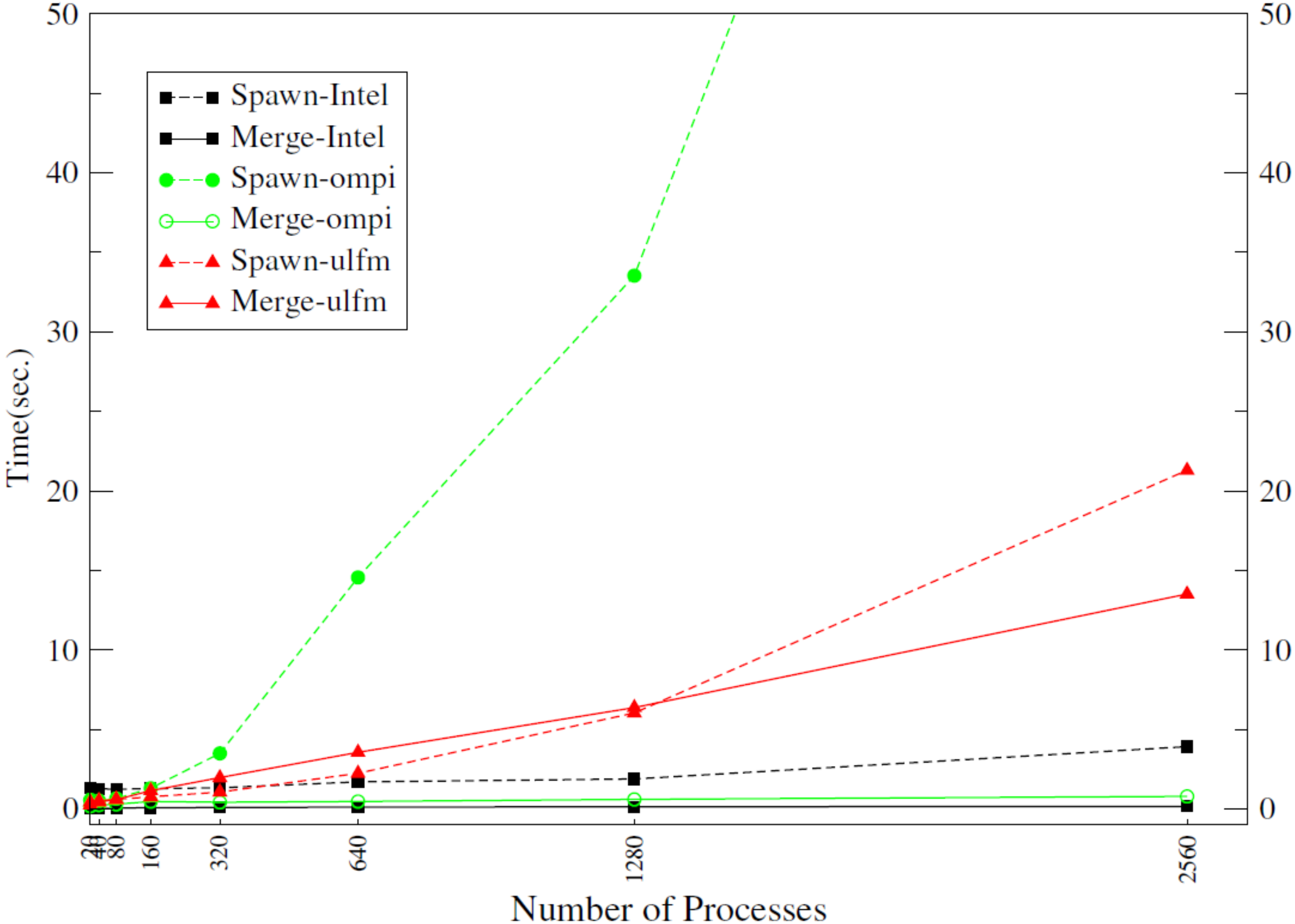


Description	Time(sec.)
Communicator revoke + shrink	0.34
Generate processes-spawn info.	0.23
Spawn + merge	26.10
Redistribute proc. ranks	1.39
Resource management	0.68

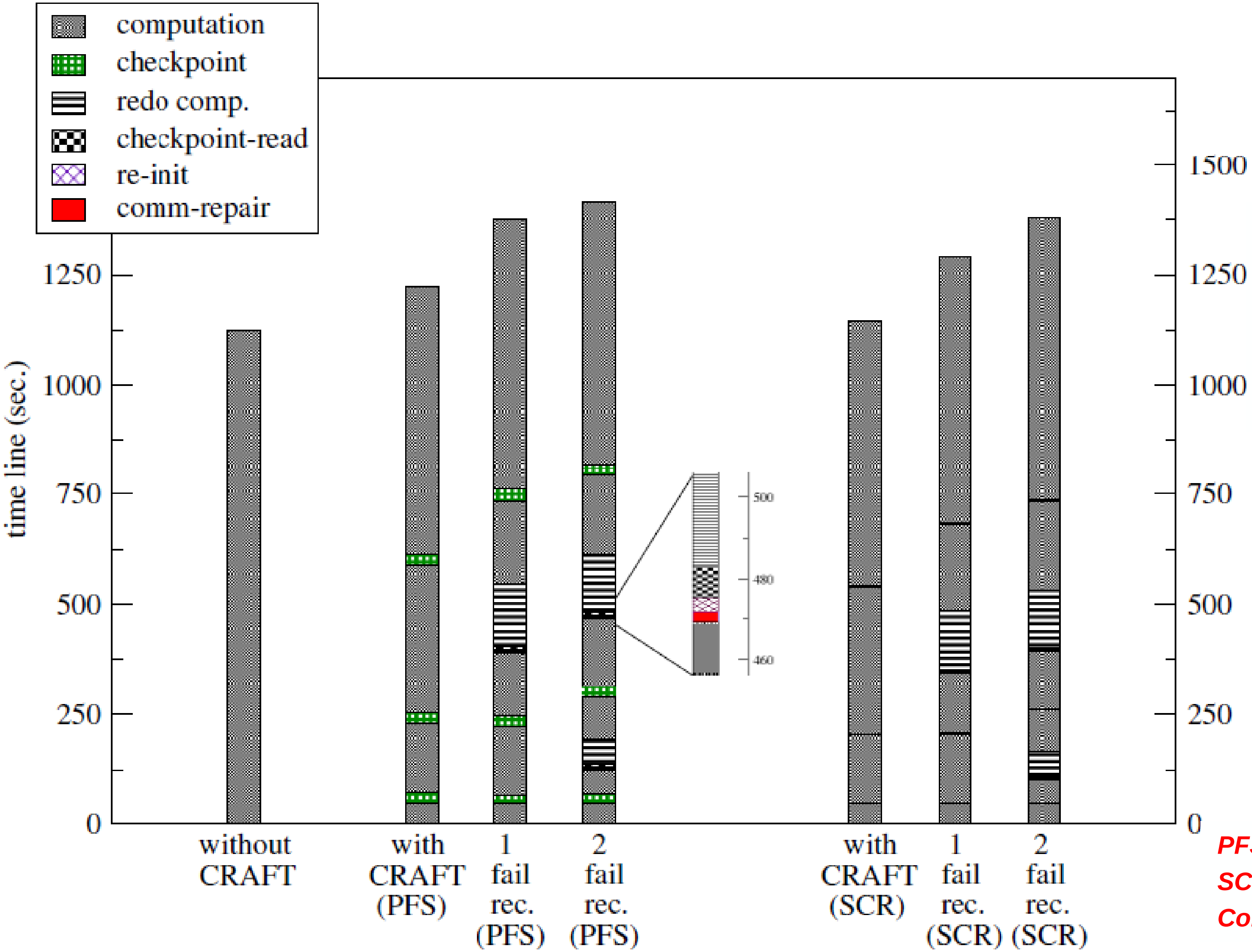
Table 4: Breakdown of the recovery process with 'non-shrinking, no-reuse' recovery policy with 2560 processes on 128 nodes.

CRAFT Benchmarks: spawn + merge comparison

- A scaling comparison of spawn and merge routines for IntelMpi-v5.1 vs. OMPI-v1.10.3 vs. ULFM-1.1 implementations.



CRAFT Benchmarks: Lanczos, 128 Emmy nodes(-np 256)



- *Num. Nodes = 128*
- *np = 256 (npernode=2)*
- *Recovery=Non-shrinking*

PFS CP Overhead= ~6.26%
SCR CP Overhead= ~0.41%
Comm. Rec. overhead=~2.7sec.

*AFT Limitations:

- AFT needs ULFM-MPI.
- Failures are only detected in the next MPI call of the corresponding communicator.
- One-sided & I/O MPI calls are not fault-tolerant.
- Batch system: Torque (SLURM-support in near future)
- AFT with SCR enabled checkpoints: Modified SCR_Init(MPI_COMM)
- Untested: Real physical failure of node. e.g. cable plug-out test.

Summary & outlook:

- CRAFT‘s CR:
  - An easier way to add Application-level CR with little modifications in the application.
  - Extendable interface to add any arbitrary data-type.

- CRAFT‘s AFT:
  - Enables dynamic process recovery in case of failed process(es) by defining ‚AFT-zone‘.
  - Shrinking, non-shrinking recoveries.

- Future work:
  1. Asynchronous checkpoint writing support via tasking.
  2. SLURM support for AFT.
  3. Multiple node failures/recoveries.
  4. Partial node-failure → kill all processes forcefully.

CRAFT checkout @: https://bitbucket.org/essex/craft

Thank you!

Questions!