Metropolis Hasting Monte-Carlo Markov Chain Simulation

Generated by Doxygen 1.9.2

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

multiway_partitioning< data_t, score_t, sort_func_t >::subset_t
Subset of a partition 72
uniform_configuration_generator< cfg_vector_t >
Initialize a configuration vector using random values 73
uuid< crtp, uuid_type, reserved_ids >
Uuid using crtp 74
wg_leader_exec_env 74

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 probsat_return_cause Namespace Reference

probsat execution results

Enumerations

- enum reason : uint8_t

  probsat execution return code

Functions

- template<typename S >
  void **serialize** (S &s, reason &r)

  serialization of probsat return cause

Variables

- static const std::string **as_string** [reason::NUM_REASONS]

  convert probsat return code into string

5.1.1 Detailed Description

probsat execution results

Chapter 6

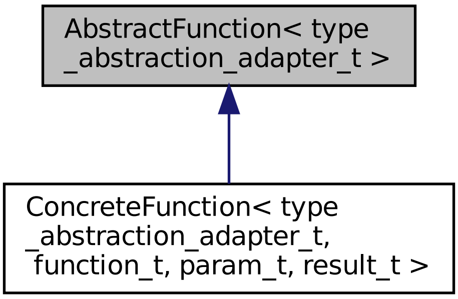
Class Documentation

6.1 AbstractFunction< type_abstraction_adapter_t > Class Template Reference

base class to represent an arbitrary function

`#include <function_abstraction.hpp>`

Inheritance diagram for AbstractFunction< type_abstraction_adapter_t >:


AbstractFunction< type _abstraction_adapter_t >
ConcreteFunction< type _abstraction_adapter_t, function_t, param_t, result_t >

Public Member Functions

- type_abstraction_adapter_t::abstract_type **operator()** (type_abstraction_adapter_t &type_abstraction, const type_abstraction_adapter_t::abstract_type &abstract_param)

  execution operator wrapper around function execution
- virtual type_abstraction_adapter_t::abstract_type **execute** (type_abstraction_adapter_t &type_abstraction, const type_abstraction_adapter_t::abstract_type &abstract_param)

  function execution implementation (virtual)
- virtual ∼**AbstractFunction** ()

  destructor

6.1.1 Detailed Description

template<typename type_abstraction_adapter_t>
requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
class AbstractFunction< type_abstraction_adapter_t >

base class to represent an arbitrary function

Definition at line 226 of file function_abstraction.hpp.

The documentation for this class was generated from the following file:

- distributed_computation/function_abstraction.hpp

6.2 Any_TypeAbstractionAdapter Class Reference

data type abstraction using std::any (for testing)

```
#include <type_abstraction.hpp>
```

Public Types

- using **abstract_type** = std::any

  abstract type

Static Public Member Functions

- template<typename t >
  static abstract_type **serialize** (t &v)

  serialization
- template<typename t >
  static bool **deserialize** (const abstract_type &v, t &result)

  deserialization

Static Public Attributes

- static const bool **is_TypeAbstractionAdapter** = true

  member used for static compile time polymorphism

6.2.1 Detailed Description

data type abstraction using std::any (for testing)

Definition at line 9 of file type_abstraction.hpp.

The documentation for this class was generated from the following file:

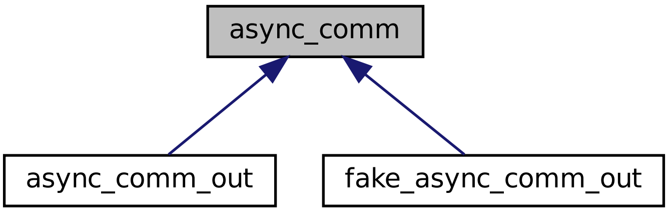
- distributed_computation/type_abstraction.hpp

6.3 async_comm Class Reference

base helper class for async MPI communcation

```
#include <mpi_async_communication.hpp>
```

Inheritance diagram for async_comm:



Public Member Functions

- void **wait_for_one_message** ()

  wait until one open message transfer is completed
- void **wait_for_all_messages** ()

  wait until all open message transfers are completed
- void **wait_for_some_messages** ()

  wait until one ore more open message transfers completed
- void **test_for_one_message** ()

  poll and complete one open message transfer if possible
- void **test_for_messages** ()

  poll to complete all open transfers which finished
- auto **communications_in_queue** () const

  number of open transfers
- bool **communication_is_done** () const

  check if no transfer which has to complete is left
- ∼**async_comm** ()

  destructor

Protected Types

- using **data_t** = std::vector< uint8_t >

  message data type

Protected Attributes

- std::vector< MPI_Request > **outstanding_requests**

  open transfers
- std::vector< data_t > **outstanding_data**

  data of incomplete transfers

6.3.1 Detailed Description

base helper class for async MPI communcation

Definition at line 8 of file mpi_async_communication.hpp.

The documentation for this class was generated from the following files:

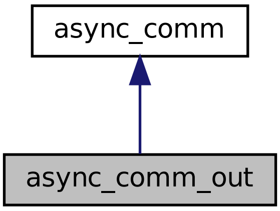
- util/mpi/mpi_async_communication.hpp
- util/mpi/mpi_async_communication.cpp

6.4 async_comm_out Class Reference

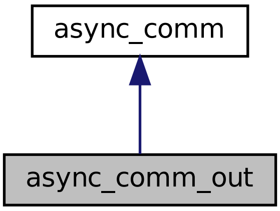
derived helper class providing async message sending

```
#include <mpi_async_communication.hpp>
```

Inheritance diagram for async_comm_out:



Collaboration diagram for async_comm_out:



Public Member Functions

- void **send_message** (const MPI_Comm &comm, int dest, int tag, data_t &&content)

  send a message asynchronous using MPI

Additional Inherited Members

6.4.1 Detailed Description

derived helper class providing async message sending

Definition at line 47 of file mpi_async_communication.hpp.

The documentation for this class was generated from the following files:

- util/mpi/mpi_async_communication.hpp
- util/mpi/mpi_async_communication.cpp

6.5 basic_statistical_metrics< prec_t > Class Template Reference

calculate multiple statistic metrics efficiently at once

```
#include <statistics.hpp>
```

Public Member Functions

- **basic_statistical_metrics** ()

  constructor
- std::size_t **counted** ()

  overall counted numbers, finite and non finite
- template<typename it_t >
  void **operator()** (it_t begin, it_t end)

  Iterator to add numbers in range to the statistic.

Public Attributes

- std::size_t **n**

  number of finite numbers counted
- prec_t **sum_xi**

  sum if all numbers
- prec_t **sum_xi_sq**

  sum if all squared numbers
- prec_t **varianz**

  variance
- prec_t **stddev**

  standard deviation
- prec_t **average**

  arithmetic mean (average)
- prec_t **min**

  minimum
- prec_t **max**

  maximum
- std::size_t **not_finite**

  count of excluded non finite numbers

6.5.1 Detailed Description

template<typename prec_t>
class basic_statistical_metrics< prec_t >

calculate multiple statistic metrics efficiently at once

Calculates minimum, maximum, average, standard deviation, variance, sum and count in O(n). Non finite numbers are counted separately and are excluded in the metrics. Updates are efficiently possible! The math for standard deviation and variance is from: `https://coderodde.wordpress.com/2016/04/12/computing-standard-deviation-in-one-pass/`

Definition at line 17 of file statistics.hpp.

The documentation for this class was generated from the following file:

- util/statistics.hpp

6.6 binary_configuration_vector< use_bitfield > Struct Template Reference

wrapper for a binary configuration vector used in the MCMC simulation

`#include <configuration_vector.hpp>`

Public Types

- using **value_type** = bool

  value type
- using **vector_type** = std::conditional< use_bitfield, bitfield< std::size_t >, std::vector< value_type > >::type

  type of the vector representation (bitfield or std::vector)
- using **size_type** = decltype(vector_type().size())

  size type of the configuration vector

Public Member Functions

- **binary_configuration_vector** ()

  default constructor
- **binary_configuration_vector** (size_type length)

  construct binary config vector of size length
- **binary_configuration_vector** (const binary_configuration_vector &other)

  copy constructor
- binary_configuration_vector & **operator=** (const binary_configuration_vector< use_bitfield > &other)

  assignment operator
- **binary_configuration_vector** (binary_configuration_vector &&other)

  move semantics by using swap
- std::size_t **count_ones** () const

  count number of ones in config vector
- bool **operator==** (const binary_configuration_vector< use_bitfield > &rhs) const

  test for equality of binary config vectors
- bool **operator!=** (const binary_configuration_vector< use_bitfield > &rhs) const

  test for inequality of binary config vectors

Public Attributes

- vector_type **data**

  configuration vector data

Static Public Attributes

- static constexpr value_type **min_value** = false

  value range: minimum
- static constexpr value_type **max_value** = true

  value range: maximum

Friends

- void **swap** (binary_configuration_vector< use_bitfield > &l, binary_configuration_vector< use_bitfield > &r)

  swap function for copy and swap idiom
- std::ostream & **operator**<< (std::ostream &os, const binary_configuration_vector< use_bitfield > &bf)

  stream operator to print binary config vector to std::ostream

6.6.1 Detailed Description

template<bool use_bitfield = false>
struct binary_configuration_vector< use_bitfield >

wrapper for a binary configuration vector used in the MCMC simulation

Definition at line 14 of file configuration_vector.hpp.

The documentation for this struct was generated from the following file:

- metropolis_hastings/configuration/configuration_vector.hpp

6.7 bitfield< length_t > Class Template Reference

bitfield to represent a bitvector compact

```
#include <bitfield.hpp>
```

Public Member Functions

- **bitfield** ()

  default constructor, empty bitvector of size 0
- **bitfield** (length_t n)

  constructor for bitvectors of size n
- **bitfield** (const bitfield &other)

  copy constructor
- ∼**bitfield** ()

  deconstructor (is freeing memory)
- **bitfield** (bitfield &&other)

  move semantics by using swap
- bitfield & **operator=** (bitfield other)

  assignment operator by using swap
- void resize (length_t new_length)

  resize bitvector length
- length_t **get_bytelength** () const

  get amount of bytes required to store current length bits
- uint8_t ∗ **data** ()

  get raw pointer to the array of bits
- bool & operator[ ] (const length_t index)

  access position index by returning a boolean reference
- bool **operator[ ]** (const length_t index) const

  return value at position index as bool
- length_t **size** () const

  get the number of bits which can be stored
- std::ostream & **print** (std::ostream &os, const char ∗separator=" ")

  print the bitvector to std::ostream

Private Member Functions

- bool **get** (const length_t index) const

  get bit at position index as bool
- void **set** (const length_t index, const bool value)

  set bit at position index to value (bool)

Static Private Member Functions

- static length_t **to_bytelength** (const length_t l)

  compute number of bytes required to store l bits

Private Attributes

- length_t **length**

  number of bits
- length_t **last_mod_index**

  used internally for returning a reference (see update function)
- bool **last_mod_value**

  used internally for returning a reference (see update function)
- uint8_t ∗ **array**

  pointer to the actual data

Friends

- void **swap** (bitfield &a, bitfield &b) noexcept(std::is_nothrow_swappable_v< length_t > &&std::is_nothrow_swappable_v< bool > &&std::is_nothrow_swappable_v< uint8_t * >)

  swap implementation for copy & swap idiom

6.7.1 Detailed Description

template<typename length_t>
class bitfield< length_t >

bitfield to represent a bitvector compact

The configuration vector can be stored in a compact bitvector

Definition at line 14 of file bitfield.hpp.

6.7.2 Member Function Documentation

6.7.2.1 operator[]()

```
template<typename length_t >
bool & bitfield< length_t >::operator[] (
            const length_t index )  [inline]
```

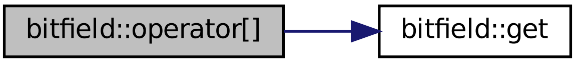
access position index by returning a boolean reference

access position index by returning a boolean reference to a class member. changes are applied later when using other functions. WARNING: may cause trouble when used for multiple indices at once as references cant be invalidated!

Definition at line 197 of file bitfield.hpp.

```
        {
            update();

            #if DEBUG_ASSERTIONS
            assert(index < length);
            #endif

            last_mod_index = index;
            last_mod_value = get(index);
            return last_mod_value;
        }
```

Here is the call graph for this function:



6.7.2.2 resize()

```
template<typename length_t >
void bitfield< length_t >::resize (
            length_t new_length )  [inline]
```

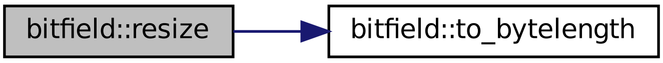
resize bitvector length

resize bitvector length only reallocates memory if new space requirement is larger than the current allocated memory

Definition at line 153 of file bitfield.hpp.

```
{
    update();

    if (to_bytelength(new_length) > to_bytelength(length)) {
        assert(0 < to_bytelength(new_length));

        uint8_t *old_array = array;

        array = new uint8_t[to_bytelength(new_length)];

        for (length_t i = 0; i < to_bytelength(length); i++) {
            array[i] = old_array[i];
        }

        if (old_array) {
            delete[] old_array;
        }
    }

    length = new_length;
    last_mod_index = length;
}
```

Here is the call graph for this function:



The documentation for this class was generated from the following file:

- util/bitfield.hpp

6.8 Bitsery_TypeAbstractionAdapter< buffer_t, input_adapter_t, output_adapter_t > Class Template Reference

data type abstraction using bitsery providing (de-)serialization

```
#include <type_abstraction.hpp>
```

Public Types

- using **abstract_type** = buffer_t

  abstract type

Static Public Member Functions

- template<typename t >
  static abstract_type **serialize** (t &v)

  serialization
- template<typename t >
  static bool **deserialize** (const abstract_type &buffer, t &result)

  deserialization

Static Public Attributes

- static const bool **is_TypeAbstractionAdapter** = true

  member used for static compile time polymorphism

6.8.1 Detailed Description

template<typename buffer_t, typename input_adapter_t, typename output_adapter_t>
class Bitsery_TypeAbstractionAdapter< buffer_t, input_adapter_t, output_adapter_t >

data type abstraction using bitsery providing (de-)serialization

Definition at line 43 of file type_abstraction.hpp.

The documentation for this class was generated from the following file:

- distributed_computation/type_abstraction.hpp

6.9 change< size_type, value_t > Struct Template Reference

struct representing a single change at a config vector

```
#include <change_generator.hpp>
```

Public Member Functions

- auto **operator==** (const change< size_type, value_t > &rhs) const

  equality test
- auto **operator!=** (const change< size_type, value_t > &rhs) const

  inequality test

Public Attributes

- size_type **index**

  index of changed position
- value_t **new_value**

  new value

6.9.1 Detailed Description

template<typename size_type, typename value_t>
struct change< size_type, value_t >

struct representing a single change at a config vector

Definition at line 14 of file change_generator.hpp.

The documentation for this struct was generated from the following file:

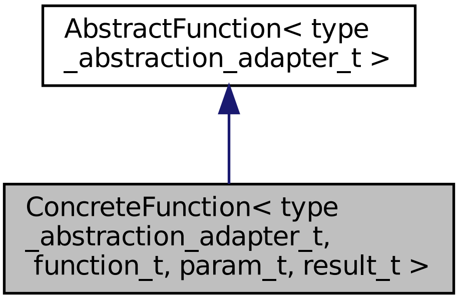
- metropolis_hastings/configuration/change_generator.hpp

6.10 ConcreteFunction< type_abstraction_adapter_t, function_t, param_t, result_t > Class Template Reference

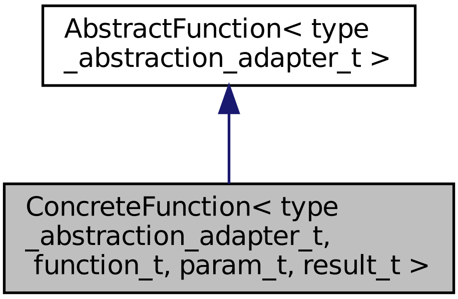
concrete implementation of an arbitrary function

```
#include <function_abstraction.hpp>
```

Inheritance diagram for ConcreteFunction< type_abstraction_adapter_t, function_t, param_t, result_t >:



Collaboration diagram for ConcreteFunction< type_abstraction_adapter_t, function_t, param_t, result_t >:



Public Member Functions

- **ConcreteFunction** (auto function, auto param, auto result)

  constructor
- virtual type_abstraction_adapter_t::abstract_type **execute** (type_abstraction_adapter_t &type_abstraction, const type_abstraction_adapter_t::abstract_type &abstract_param) override

  concrete function execution
- virtual ∼**ConcreteFunction** ()

  destructor of derived class

Private Attributes

- function_t **function**

  function to abstract
- param_t **parameters**

  abstract function parameters
- result_t **result**

  abstract result

6.10.1 Detailed Description

template<typename type_abstraction_adapter_t, typename function_t, typename param_t, typename result_t>
requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
class ConcreteFunction< type_abstraction_adapter_t, function_t, param_t, result_t >

concrete implementation of an arbitrary function

Definition at line 259 of file function_abstraction.hpp.

The documentation for this class was generated from the following file:

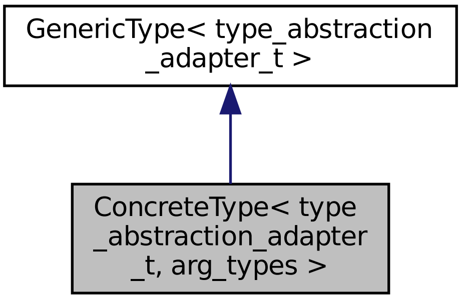
- distributed_computation/function_abstraction.hpp

6.11 ConcreteType< type_abstraction_adapter_t, arg_types > Class Template Reference

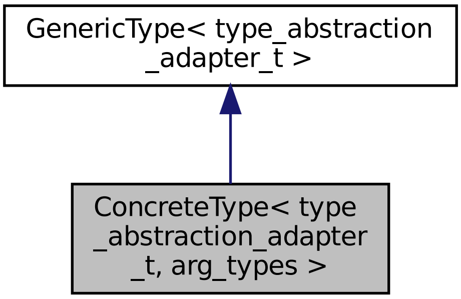
a concrete tuple of arbitrary types

```
#include <rtti.hpp>
```

Inheritance diagram for ConcreteType< type_abstraction_adapter_t, arg_types >:


GenericType< type_abstraction _adapter_t >
ConcreteType< type _abstraction_adapter _t, arg_types >

Collaboration diagram for ConcreteType< type_abstraction_adapter_t, arg_types >:


GenericType< type_abstraction _adapter_t >
ConcreteType< type _abstraction_adapter _t, arg_types >

Public Types

- using **tuple_t** = std::tuple< arg_types ... >

  tuple data type

Public Member Functions

- **ConcreteType** ()

  default constructor
- **ConcreteType** (arg_types... args, bool use_value_constructor=true)

  constructor variant with additional value to resolve ambiguous calls
- **ConcreteType** (std::tuple< arg_types ... > &values)

  normal constructor passing values
- **ConcreteType** (const ConcreteType &other)

copy constructor

- template<typename other_type_abstraction_adapter_t >
  operator ConcreteType< other_type_abstraction_adapter_t, arg_types... > () const

  type conversion to differen type abstraction adapters
- virtual type_abstraction_adapter_t::abstract_type **serialize** (type_abstraction_adapter_t &adapter) const override

  serialization of values using the type abstraction adapter
- virtual bool **deserialize** (type_abstraction_adapter_t &adapter, const type_abstraction_adapter_t::abstract_type &v) override

  deserialization of values using the type abstraction adapter

Public Attributes

- tuple_t **values**

  tuple with values

6.11.1 Detailed Description

template<typename type_abstraction_adapter_t, typename... arg_types>
requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
class ConcreteType< type_abstraction_adapter_t, arg_types >

a concrete tuple of arbitrary types

Definition at line 39 of file rtti.hpp.

The documentation for this class was generated from the following file:

- distributed_computation/rtti.hpp

6.12 constant_configuration_generator< cfg_vector_t > Class Template Reference

initialize a configuration vector using a constant

```
#include <configuration_generator.hpp>
```

Public Member Functions

- **constant_configuration_generator** (value_t constant_value)

  constructor
- void **operator()** (auto &prng, cfg_vector_t &v) const

  apply initialization method to config vector

Private Attributes

- value_t **cvalue**

  constant value used for initialization

6.12.1 Detailed Description

template<typename cfg_vector_t>
class constant_configuration_generator< cfg_vector_t >

initialize a configuration vector using a constant

Definition at line 12 of file configuration_generator.hpp.

The documentation for this class was generated from the following file:

- metropolis_hastings/configuration/configuration_generator.hpp

6.13 multiway_partitioning< data_t, score_t, sort_func_t >::entry_t Struct Reference

single entry associating a score with some data

```
#include <multiway_partitioning.hpp>
```

Public Attributes

- score_t **score**

  score of the entry (used for partitioning, i.e. workload progress)
- data_t **data**

  associated data (i.e. workload identifier)

6.13.1 Detailed Description

template<typename data_t, typename score_t = std::size_t, class sort_func_t = std::greater<score_t>>
struct multiway_partitioning< data_t, score_t, sort_func_t >::entry_t

single entry associating a score with some data

Definition at line 28 of file multiway_partitioning.hpp.

The documentation for this struct was generated from the following file:

- distributed_computation/multiway_partitioning.hpp

6.14 exponential_acceptance_probability< prec_t > Class Template Reference

compute an acceptance probability used by the metropolis hastings algorithm

```
#include <acceptance_probability.hpp>
```

Public Member Functions

- **exponential_acceptance_probability** (prec_t T_)
  constructor
- prec_t **operator()** (prec_t old_value, prec_t new_value) const
  compute acceptance probability given a new and old value

Private Attributes

- prec_t **T**
  exponential falloff (new - old = T (old was better) => P(accept) = 1/e)

6.14.1 Detailed Description

template<typename prec_t>
class exponential_acceptance_probability< prec_t >

compute an acceptance probability used by the metropolis hastings algorithm

Definition at line 8 of file acceptance_probability.hpp.

The documentation for this class was generated from the following file:

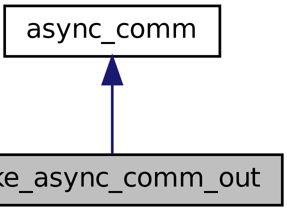
- metropolis_hastings/acceptance_computation/acceptance_probability.hpp

6.15 fake_async_comm_out Class Reference

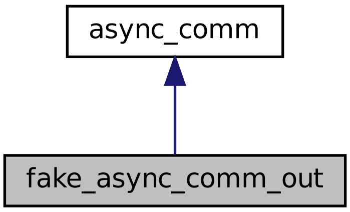
derived helper class for synchronous messaging (debug communication)

```
#include <mpi_async_communication.hpp>
```

Inheritance diagram for fake_async_comm_out:



Collaboration diagram for fake_async_comm_out:



Additional Inherited Members

6.15.1 Detailed Description

derived helper class for synchronous messaging (debug communication)

Definition at line 56 of file mpi_async_communication.hpp.

The documentation for this class was generated from the following files:

- util/mpi/mpi_async_communication.hpp
- util/mpi/mpi_async_communication.cpp

6.16 function_typeinfo< type_abstraction_adapter_t > Struct Template Reference

helper struct to get type of function parameters and return value

```
#include <function_abstraction.hpp>
```

Static Public Member Functions

- template<typename return_type >
  static ConcreteType< type_abstraction_adapter_t > **get_concrete_argument_type** (return_type(∗)())

  get function argument type of function pointer without parameters
- template<typename fobject , typename return_type >
  static ConcreteType< type_abstraction_adapter_t > **get_concrete_argument_type** (return_type(fobject::∗)())

  get function argument type of function object without parameters
- template<typename return_type >
  static ConcreteType< type_abstraction_adapter_t > **get_concrete_argument_type** (std::function< return_type()> f)

  get function argument type of std::function without parameters

- template<typename first_arg_type , typename... tuple_tail_types>
  static auto **create_concrete_type** (first_arg_type ∗first_arg, std::tuple< tuple_tail_types... > &tail)

  helper function for recursive argument reduction
- template<typename return_type , typename first_arg_type , typename... arg_types>
  static auto **get_concrete_argument_type** (return_type(∗)(first_arg_type, arg_types...))

  recursive argument reduction to get argument type (fct pointer)
- template<typename fobject , typename return_type , typename first_arg_type , typename... arg_types>
  static auto **get_concrete_argument_type** (return_type(fobject::∗)(first_arg_type, arg_types...))

  recursive argument reduction to get argument type (fct object)
- template<typename return_type , typename first_arg_type , typename... arg_types>
  static auto **get_concrete_argument_type** (std::function< return_type(first_arg_type, arg_types...)> f)

  recursive argument reduction to get argument type (std::function)
- template<typename return_type , typename first_arg_type , typename... arg_types>
  static auto **get_concrete_argument_type** (return_type(∗)(const first_arg_type &, arg_types...))

  recursive argument reduction (const reference) to get argument type (fct pointer)
- template<typename fobject , typename return_type , typename first_arg_type , typename... arg_types>
  static auto **get_concrete_argument_type** (return_type(fobject::∗)(const first_arg_type &, arg_types...))

  recursive argument reduction (const reference) to get argument type (fct object)
- template<typename return_type , typename first_arg_type , typename... arg_types>
  static auto **get_concrete_argument_type** (std::function< return_type(const first_arg_type &, arg_types...)> f)

  recursive argument reduction (const reference) to get argument type (fct object)
- template<typename return_type , typename... arg_types>
  static auto **get_concrete_return_type** (return_type(∗)(arg_types...))

  get function return type for function pointer
- template<typename fobject , typename return_type , typename... arg_types>
  static auto **get_concrete_return_type** (return_type(fobject::∗)(arg_types...))

  get function return type for function object
- template<typename return_type , typename... arg_types>
  static ConcreteType< type_abstraction_adapter_t > **get_concrete_return_type** (std::function< return_type(arg_types...)> f)

  get function return type for std::function

6.16.1 Detailed Description

template<class type_abstraction_adapter_t>
requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
struct function_typeinfo< type_abstraction_adapter_t >

helper struct to get type of function parameters and return value

Definition at line 11 of file function_abstraction.hpp.

The documentation for this struct was generated from the following file:

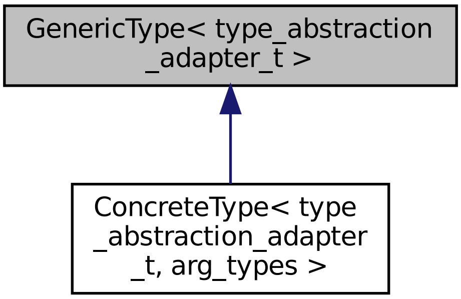
- distributed_computation/function_abstraction.hpp

6.17 GenericType< type_abstraction_adapter_t > Class Template Reference

base class to represent tuples of arbitrary types

```
#include <rtti.hpp>
```

Inheritance diagram for GenericType< type_abstraction_adapter_t >:


GenericType< type_abstraction _adapter_t >
ConcreteType< type _abstraction_adapter _t, arg_types >

Public Member Functions

- virtual type_abstraction_adapter_t::abstract_type **serialize** (type_abstraction_adapter_t &adapter) const

  serialization to type abstraction
- virtual bool **deserialize** (type_abstraction_adapter_t &adapter, const type_abstraction_adapter_t::abstract_type &v)

  deserialization from type abstraction
- virtual ∼**GenericType** ()

  virtual destructor

6.17.1 Detailed Description

template<typename type_abstraction_adapter_t>
requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
class GenericType< type_abstraction_adapter_t >

base class to represent tuples of arbitrary types

Definition at line 12 of file rtti.hpp.

The documentation for this class was generated from the following file:

- distributed_computation/rtti.hpp

6.18 get_mpi< t > Struct Template Reference

the MPI datatype for type T can be queried by get_mpi<T>::type

```
#include <mpi_types.hpp>
```

6.18.1 Detailed Description

template<typename t>
struct get_mpi< t >

the MPI datatype for type T can be queried by get_mpi<T>::type

Definition at line 8 of file mpi_types.hpp.

The documentation for this struct was generated from the following file:

- util/mpi/mpi_types.hpp

6.19 inaccurate_modell< modell_t > Class Template Reference

an enhanced modell with added irregularities for probsat simulation

```
#include <modell.hpp>
```

Classes

- struct prepared_computation

  represents a prepared computation for a given configuration

Public Member Functions

- **inaccurate_modell** (modell_t &modell_, prec_t error_strength_)

  constructor

- void **prepare_computation_with_cfg_vector** (const cfg_vector_t &v, prepared_computation &pcom)

  prepare computation for a given config vector

- template<typename prng_t , typename rpmanager_t >
  auto **operator()** (prng_t &prng, rpmanager_t &rpmanager, prepared_computation &pcom) const

  create a remote function call for one execution

- void **finish_computation** (prepared_computation &pcom)

  finish a given computation

Private Attributes

- modell_t **modell_**

  using the accurate modell internally
- const prec_t **error_strength**

  error strength to apply (errors follow a lognormal distribution with sigma error_strength and mu 0)
- const prec_t **error_offset**

  error offset to correct = exp(mu + sigma∗∗2 /2) - 1

6.19.1 Detailed Description

template<typename modell_t>
requires (modell_t::is_value_computation_implementation)
class inaccurate_modell< modell_t >

an enhanced modell with added irregularities for probsat simulation

Definition at line 36 of file modell.hpp.

The documentation for this class was generated from the following file:

- metropolis_hastings/value_computation/modell.hpp

6.20 lazy_round_robin_scheduler< workload_capacity > Class Template Reference

round robin load balancing of up to workload_capacity computations inside a workgroup

```
#include <lazy_round_robin_scheduler.hpp>
```

Public Member Functions

- bool **is_process_without_work** ()

  check if any worker is without work
- bool **worker_available** ()

  check if any worker has less than workload_capacity jobs
- int **get_id_to_schedule_task_on** ()

  get a worker id (or -1 if none available) for an task
- void **task_finished** (int id)

  mark one task on a worker with the given id as finished
- **lazy_round_robin_scheduler** (int size)

  constructor

Private Attributes

- const int **size**

  number of workers
- std::vector< std::size_t > **num_tasks_scheduled**

  number of tasks on each worker
- std::multimap< std::size_t, int > **ids_for_scheduled**

  lookup table for worker id based upon number of currently scheduled tasks

6.20.1 Detailed Description

template<std::size_t workload_capacity>
class lazy_round_robin_scheduler< workload_capacity >

round robin load balancing of up to workload_capacity computations inside a workgroup

Definition at line 11 of file lazy_round_robin_scheduler.hpp.

The documentation for this class was generated from the following file:

- distributed_computation/lazy_round_robin_scheduler.hpp

6.21 make_generic_serializeable< t > Struct Template Reference

wrapper code for generic serialization

```
#include <basic_serialization.hpp>
```

6.21.1 Detailed Description

template<typename t>
struct make_generic_serializeable< t >

wrapper code for generic serialization

Definition at line 137 of file basic_serialization.hpp.

The documentation for this struct was generated from the following file:

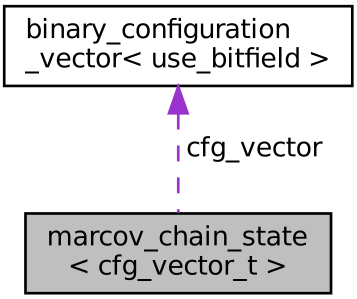
- util/basic_serialization.hpp

6.22 marcov_chain_state< cfg_vector_t > Class Template Reference

represents a single state of a markov chain

```
#include <mcmc.hpp>
```

Collaboration diagram for marcov_chain_state< cfg_vector_t >:


binary_configuration _vector< use_bitfield >
cfg_vector
marcov_chain_state < cfg_vector_t >

Public Types

- using **cfg_vector_type** = cfg_vector_t

    type of the config vector used

Public Member Functions

- **marcov_chain_state** (std::size_t cfg_vector_length=0)

    default constructor
- **marcov_chain_state** (std::size_t cfg_vector_length, auto prn_engine)

    normal constructor to set cfg vector length and random number engine
- **marcov_chain_state** (std::size_t iteration, cfg_vector_t &cfg_vector, prn_engine_t &prn_engine)

    constructor to set all class members
- **marcov_chain_state** (const marcov_chain_state &other)

    copy constructor
- **marcov_chain_state** (marcov_chain_state &&other)

    movement constructor using swap
- marcov_chain_state & **operator=** (const marcov_chain_state< cfg_vector_t > &other)

    assignment operator
- bool **operator==** (const marcov_chain_state< cfg_vector_t > &rhs) const

    equality test
- bool **operator!=** (const marcov_chain_state< cfg_vector_t > &rhs) const

    inequality test
- std::string **get_prn_engine_as_string** () const

    function to convert pseudo random number into string
- void **set_prn_engine_from_string** (auto prn_state)

    function to restore pseudo random number from string

Public Attributes

- std::size_t **iteration**

  current iteration
- cfg_vector_t **cfg_vector**

  configuration vector
- prn_engine_t **prn_engine**

  pseudo random number engine (state)

Friends

- void **swap** (marcov_chain_state< cfg_vector_t > &l, marcov_chain_state< cfg_vector_t > &r)

  swap function
- std::ostream & **operator**<< (std::ostream &os, const marcov_chain_state< cfg_vector_t > &mcs)

  ostream operator to print state

6.22.1 Detailed Description

template<typename cfg_vector_t>
class marcov_chain_state< cfg_vector_t >

represents a single state of a markov chain

Definition at line 15 of file mcmc.hpp.

The documentation for this class was generated from the following file:

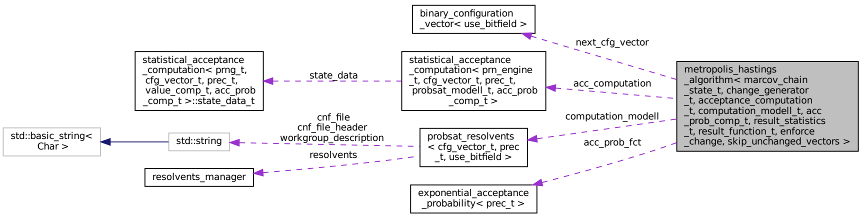
- metropolis_hastings/mcmc.hpp

6.23 metropolis_hastings_algorithm< marcov_chain_state_t, change_generator_t, acceptance_computation_t, computation_modell_t, acc_prob_comp_t, result_statistics_t, result_function_t, enforce_change, skip_unchanged_vectors > Class Template Reference

generic implementation of the metropolis hastings algorithm for one markov chain

`#include <algorithm.hpp>`

Collaboration diagram for metropolis_hastings_algorithm< marcov_chain_state_t, change_generator_t, acceptance_computation_t, computation_modell_t, acc_prob_comp_t, result_statistics_t, result_function_t, enforce_change, skip_unchanged_vectors >:


binary_configuration _vector< use_bitfield >
next_cfg_vector
statistical_acceptance _computation< prng_t, cfg_vector_t, prec_t, value_comp_t, acc_prob _comp_t >::state_data_t
state_data
statistical_acceptance _computation< prn_engine _t, cfg_vector_t, prec_t, probsat_modell_t, acc_prob _comp_t >
acc_computation
metropolis_hastings _algorithm< marcov_chain _state_t, change_generator _t, acceptance_computation _t, computation_modell_t, acc _prob_comp_t, result_statistics _t, result_function_t, enforce _change, skip_unchanged_vectors >
cnf_file
cnf_file_header
workgroup_description
computation_modell
std::basic_string< Char >
std::string
probsat_resolvents < cfg_vector_t, prec _t, use_bitfield >
acc_prob_fct
resolvents
resolvents_manager
exponential_acceptance _probability< prec_t >

Public Member Functions

- **metropolis_hastings_algorithm** (std::size_t id, auto mc_state_, computation_modell_t &computation_modell, acc_prob_comp_t &acc_prob_fct)

  constructor
- marcov_chain_state_t **get_last_state** ()

  get snapshot of last state
- template<typename scheduler_t >
  void **prepare_iteration** (scheduler_t &scheduler)

  prepare an iteration
- bool **still_waiting_for_computation** ()

  check if still waiting or progress can be made
- template<typename scheduler_t >
  bool **continue_iteration** (scheduler_t &scheduler)

  continue an iteration
- void **finish_iteration** (bool last_iteration=false)

  finish an iteration
- void **skip_computation** (bool accept)

  skip one step for fast forwarding
- void **cleanup** ()

  cleanup of unfinished computations to end markov chain
- bool **can_start_next_iteration** () const

  check if iteration was finished and next can be started
- std::size_t **get_iteration** () const

  get current iteration
- void **print_execution_statistic** ()

  print statistics
- template<typename scheduler_t >
  std::size_t **operator()** (scheduler_t &scheduler, bool last_iteration=false)

  automate iteration calls to make progress

Private Member Functions

- void **generate_next_cfg_vector** ()

  generate the next config vector

Private Attributes

- std::size_t **id**

  id of the markov chain (for output)
- marcov_chain_state_t **mc_state**

  current state of the markov chain
- decltype(mc_state.prn_engine) **last_prng**

  pseudo random number generator of last state
- change_generator_t **change_gen**

  change generator for the config vector
- cfg_vector_t **next_cfg_vector**

  next config vector
- acceptance_computation_t **acc_computation**

  acceptance computation

- bool **is_processing**

  internal computation state
- computation_modell_t & **computation_modell**

  computation modell (i.e. probsat)
- acc_prob_comp_t & **acc_prob_fct**

  acceptance function
- result_statistics_t **execution_statistics**

  execution statistics of computation modell

6.23.1 Detailed Description

template<typename marcov_chain_state_t, typename change_generator_t, typename acceptance_computation_t, typename computation_modell_t, typename acc_prob_comp_t, typename result_statistics_t, typename result_function_t, const bool enforce_change, const bool skip_unchanged_vectors>
class metropolis_hastings_algorithm< marcov_chain_state_t, change_generator_t, acceptance_computation_t, computation_↩ modell_t, acc_prob_comp_t, result_statistics_t, result_function_t, enforce_change, skip_unchanged_vectors >

generic implementation of the metropolis hastings algorithm for one markov chain

Definition at line 24 of file algorithm.hpp.

The documentation for this class was generated from the following file:

- metropolis_hastings/algorithm.hpp

6.24 modell< cfg_vector_t, prec_t > Class Template Reference

a modell with exponential falloff as probsat substitution for simulation

```
#include <modell.hpp>
```

Classes

- struct prepared_computation

  represents a prepared computation for a given configuration

Public Member Functions

- **modell** (prec_t s_)

  constructor
- void **prepare_computation_with_cfg_vector** (const cfg_vector_t &v, prepared_computation &pcom)

  prepare computation for a given config vector
- template<typename prng_t , typename rpmanager_t >
  auto **operator()** (prng_t &prng, rpmanager_t &rpmanager, prepared_computation &pcom) const

  create a remote function call for one execution
- void **finish_computation** (prepared_computation &pcom)

  finish a given computation (dummy)

Private Attributes

- const prec_t **s**

  slope factor

6.24.1 Detailed Description

template<typename cfg_vector_t, typename prec_t>
class modell< cfg_vector_t, prec_t >

a modell with exponential falloff as probsat substitution for simulation

Definition at line 112 of file modell.hpp.

The documentation for this class was generated from the following file:

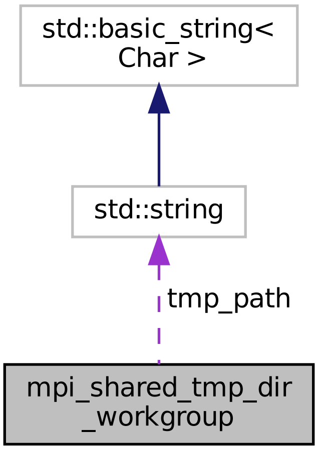
- metropolis_hastings/value_computation/modell.hpp

6.25 mpi_shared_tmp_dir_workgroup Class Reference

creates workgroups based on an available shared tmp directory

```
#include <mpi_shared_tmp_workgroup.hpp>
```

Collaboration diagram for mpi_shared_tmp_dir_workgroup:



Public Member Functions

- bool **is_workgroup_head** () const

  checks if this node is head of a workgroup
- **mpi_shared_tmp_dir_workgroup** (MPI_Comm parent_comm=MPI_COMM_WORLD, std::string app_↩ name="main")

  constructor (parent communicator, application name)
- ∼**mpi_shared_tmp_dir_workgroup** ()

  destructor

Public Attributes

- MPI_Comm **parent_comm**

  parent MPI communicator
- int **parent_comm_id**

  node id in parent comm
- int **parent_comm_size**

  number of nodes in parent comm
- int **parent_comm_workgroup_head_id**

  workgroup head id
- MPI_Comm **workgroup_comm**

  intra workgroup communicator
- int **workgroup_comm_id**

  node if in intra workgroup comm
- int **workgroup_comm_size**

  number of nodes in workgroup
- MPI_Comm **mpi_comm_leaders**

  inter workgroup communicator, workgroup heads only
- int **workgroup_count**

  number of workgroups
- int **workgroup_id**

  id of this workgroup

Private Member Functions

- std::string **build_tmp_dir_path** (std::string app_name)

  create tmp path to partition workgroups
- int **identify_workgroup_head** ()

  helper function to get id of workgroup head in parent communicator

Private Attributes

- std::string **tmp_path**

  tmp path to check

6.25.1 Detailed Description

creates workgroups based on an available shared tmp directory

Creates subworkgroups of a MPI communicator based on an available shared tmp directory. Provides inter and intra workgroup communicators.

Definition at line 12 of file mpi_shared_tmp_workgroup.hpp.

The documentation for this class was generated from the following files:

- util/mpi/mpi_shared_tmp_workgroup.hpp
- util/mpi/mpi_shared_tmp_workgroup.cpp

6.26 multiway_partitioning< data_t, score_t, sort_func_t > Class Template Reference

class to solve the multiway number partitioning problem

```
#include <multiway_partitioning.hpp>
```

Classes

- struct entry_t

    single entry associating a score with some data
- struct partitioning_t

    a simple partition containing multiple subsets
- struct subset_t

    subset of a partition

Public Member Functions

- **multiway_partitioning** (std::size_t num_partitions, auto sort_func=std::greater< score_t >())

    constructor
- void **reset** (std::size_t n_partitions)

    reset object to initial state
- void **add** (auto score, auto data)

    add an element
- void **sort** ()

    sort datasets (initial step after dataset is complete)
- bool **is_finished** ()

    returns true if only one partition is left
- bool iterate ()

    execute one step of the largest differencing method
- void **partitionate** ()

    execute the largest differencing method by iterating until done
- auto **result** ()

    returns the resulting partition

Public Attributes

- std::list< partitioning_t > **dataset**

  list of partitionings for the largest differencing method
- std::size_t **num_partitions**

  number of target partitions
- sort_func_t **sort_func**

  function to compare scores

Friends

- std::ostream & **operator**<< (std::ostream &os, const multiway_partitioning &mp)

  print all partitionings for debugging purpose

6.26.1 Detailed Description

template<typename data_t, typename score_t = std::size_t, class sort_func_t = std::greater<score_t>>
class multiway_partitioning< data_t, score_t, sort_func_t >

class to solve the multiway number partitioning problem

Multi-way partitioning using the largest differencing method. This is an efficient approximation with statistically better results than a greedy algorithm. Implementation based upon: https://en.wikipedia.org/wiki/Largest_differencing_method https://en.wikipedia.org/wiki/Multiway_number_partitioning

Definition at line 23 of file multiway_partitioning.hpp.

6.26.2 Member Function Documentation

6.26.2.1 iterate()

```
template<typename data_t , typename score_t = std::size_t, class sort_func_t = std::greater<score_t>>
bool multiway_partitioning< data_t, score_t, sort_func_t >::iterate ( )  [inline]
```

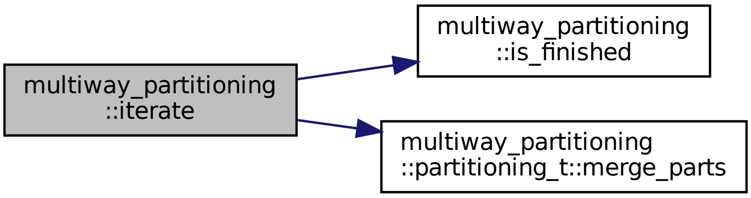
execute one step of the largest differencing method

each iteration fuses two partitions in the dataset until one partition is left.

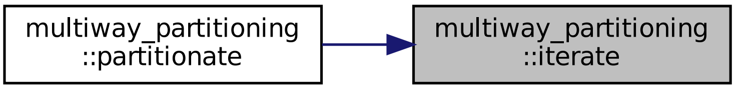
Definition at line 289 of file multiway_partitioning.hpp.

```
        {
            if (1 < dataset.size()) {
                partitioning_t s1 = dataset.front();
                dataset.pop_front();
                partitioning_t s2 = dataset.front();
                dataset.pop_front();

                auto cmp_entry_f =
                    [this](const entry_t &l, const entry_t &r)
                {
                    return sort_func(l.score, r.score);
                };

                s1.merge_parts(s2, num_partitions, cmp_entry_f);
                std::list<partitioning_t> nel = { s1 };

                auto cmp_f =
                [this](const partitioning_t &l, const partitioning_t &r)
                {
                    return sort_func(
                        l.difference(num_partitions),
                        r.difference(num_partitions));
                };
                dataset.merge(nel, cmp_f);
            }

            return is_finished();
        }
```

Here is the call graph for this function:


multiway_partitioning ::iterate
multiway_partitioning ::is_finished
multiway_partitioning ::partitioning_t::merge_parts

Here is the caller graph for this function:


multiway_partitioning ::partitionate
multiway_partitioning ::iterate

The documentation for this class was generated from the following file:

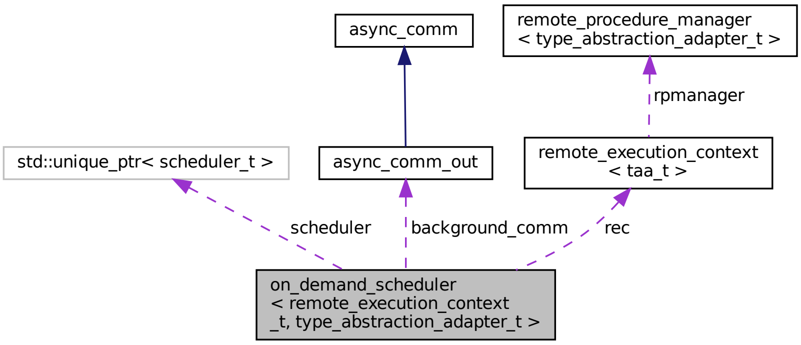
- distributed_computation/multiway_partitioning.hpp

6.27 on_demand_scheduler< remote_execution_context_t, type_abstraction_adapter_t > Class Template Reference

scheduler for load balancing of probsat computations inside a workgroup

```
#include <on_demand_scheduler.hpp>
```

Collaboration diagram for on_demand_scheduler< remote_execution_context_t, type_abstraction_adapter_t >:


async_comm
remote_procedure_manager < type_abstraction_adapter_t >
rpmanager
std::unique_ptr< scheduler_t >
async_comm_out
remote_execution_context < taa_t >
scheduler
background_comm
rec
on_demand_scheduler < remote_execution_context _t, type_abstraction_adapter_t >

Public Types

- using **rpc_message_t** = remote_execution_context_t::rpc_message_t

  task type: remote procedure call message
- using **buffer_t** = type_abstraction_adapter_t::abstract_type

  communication buffer type

Public Member Functions

- **on_demand_scheduler** (MPI_Comm env_comm, int mpi_tag, remote_execution_context_t &rec)

  constructor
- void **schedule** (auto task, int target=0)

  schedule a task or add it to the waiting queue
- std::size_t **num_tasks_waiting** () const

  number of waiting tasks in queue
- std::size_t **num_tasks_pending** () const

  number of already scheduled but pending tasks
- bool **is_work_outstanding** () const

  check if any work is left or everything completed
- size_t **do_work** (bool wait_for_work=false)

  do available work and optionally wait for work if nothing todo

Public Attributes

- MPI_Comm **env_comm**

  communicator this scheduler manages
- const int **env_comm_id**

  id of node in communicator
- const int **env_comm_size**

  number of nodes in communicator
- const int **mpi_tag**

  tag used for scheduler communication
- remote_execution_context_t & **rec**

  remote execution context

Private Types

- using **scheduler_t** = lazy_round_robin_scheduler< workload_capacity >

  scheduler base type
- using **uptr_scheduler_t** = std::unique_ptr< scheduler_t >

  scheduler type, unique ptr only on head node used

Private Member Functions

- void **process_msg** (MPI_Status &status)

  process a result message
- rpc_message_t **get_msg** (MPI_Status &status)

  recieve a pending message
- rpc_message_t **wait_for_msg** ()

  wait until a message arrives
- void **execute** (auto msg)

  execute a message using the remote execution context
- void **feed_hungry_workers** ()

  only schedule queued tasks to workers in idle state

Private Attributes

- uptr_scheduler_t **scheduler**

  lazy round robin scheduler
- std::deque< rpc_message_t > **tasks**

  queue of waiting tasks (available to process or schedule)
- type_abstraction_adapter_t **taa**

  serialization of abstract types
- std::size_t **num_waiting_for_reply**

  number of pending (already scheduled) tasks
- async_comm_out **background_comm**

  asynchronous background communication

Static Private Attributes

- static constexpr const std::size_t **workload_capacity** = 2

  least workload capacity to keep workers busy

6.27.1 Detailed Description

template<typename remote_execution_context_t, class type_abstraction_adapter_t = remote_execution_context_t::type_abstraction_adapter_t>
requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
class on_demand_scheduler< remote_execution_context_t, type_abstraction_adapter_t >

scheduler for load balancing of probsat computations inside a workgroup

uses internally the lazy round robin scheduler with up to two jobs per worker

Definition at line 27 of file on_demand_scheduler.hpp.

The documentation for this class was generated from the following file:

- distributed_computation/on_demand_scheduler.hpp

6.28 multiway_partitioning< data_t, score_t, sort_func_t >::partitioning_t Struct Reference

a simple partition containing multiple subsets

```
#include <multiway_partitioning.hpp>
```

Public Member Functions

- auto difference (std::size_t num_partitions) const

  difference between minimum and maximum sum of scores in any subset contained
- **partitioning_t** (std::size_t num_partitions)

  construct an empty partition
- **partitioning_t** (std::size_t num_partitions, auto first_subset)

  construct a partition with one subset
- void **merge_parts** (partitioning_t &other, std::size_t num_partitions, auto cmp_function)

  merge two partitions
- void **sort** (auto sort_func)

  sort subsets in this partition by the sum of scores of their elements
- void **final_sort** (auto sort_func)

  modified sort to consider set size and sum of scores in each subset

Public Attributes

- std::vector< subset_t > **subsets**

  subsets of this partition
- score_t **min_sum**

  minimal sum of entry scores in any subset contained
- score_t **max_sum**

  maximum sum of entry scores in any subset contained

Friends

- std::ostream & **operator**<< (std::ostream &os, const partitioning_t &p)

  print a partition to std::ostream

6.28.1 Detailed Description

template<typename data_t, typename score_t = std::size_t, class sort_func_t = std::greater<score_t>>
struct multiway_partitioning< data_t, score_t, sort_func_t >::partitioning_t

a simple partition containing multiple subsets

Definition at line 84 of file multiway_partitioning.hpp.

6.28.2 Member Function Documentation

6.28.2.1 difference()

```
template<typename data_t , typename score_t = std::size_t, class sort_func_t = std::greater<score_t>>
auto multiway_partitioning< data_t, score_t, sort_func_t >::partitioning_t::difference (
            std::size_t num_partitions ) const  [inline]
```

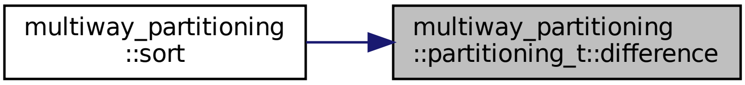
difference between minimum and maximum sum of scores in any subset contained

if partition size is less than the target number of partitions its required by the algorithm to assume subsets with size zero are contained

Definition at line 101 of file multiway_partitioning.hpp.

```
        {
            if (subsets.size() < num_partitions) {
                return max_sum - std::min(min_sum, (std::size_t) 0);
            }
            return max_sum - min_sum;
        }
```

Here is the caller graph for this function:


multiway_partitioning ::sort
multiway_partitioning ::partitioning_t::difference

The documentation for this struct was generated from the following file:

- distributed_computation/multiway_partitioning.hpp

6.29 inaccurate_modell< modell_t >::prepared_computation Struct Reference

represents a prepared computation for a given configuration

`#include <modell.hpp>`

Public Attributes

- modell_t::prepared_computation **pcom**

  embedding the accurate modell
- std::lognormal_distribution< prec_t > **error_distr**

  error distribution

6.29.1 Detailed Description

template<typename modell_t>
struct inaccurate_modell< modell_t >::prepared_computation

represents a prepared computation for a given configuration

Definition at line 46 of file modell.hpp.

The documentation for this struct was generated from the following file:

- metropolis_hastings/value_computation/modell.hpp

6.30 modell< cfg_vector_t, prec_t >::prepared_computation Struct Reference

represents a prepared computation for a given configuration

```
#include <modell.hpp>
```

Public Attributes

- prec_t **f**

  value to return

6.30.1 Detailed Description

template<typename cfg_vector_t, typename prec_t>
struct modell< cfg_vector_t, prec_t >::prepared_computation

represents a prepared computation for a given configuration

Definition at line 120 of file modell.hpp.

The documentation for this struct was generated from the following file:

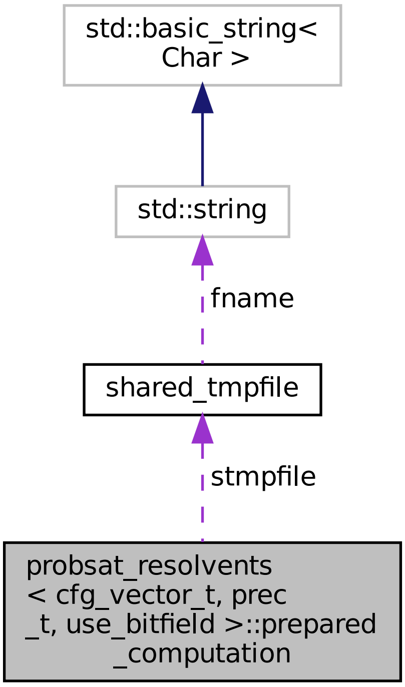
- metropolis_hastings/value_computation/modell.hpp

6.31 probsat_resolvents< cfg_vector_t, prec_t, use_bitfield >::prepared_computation Struct Reference

represents a prepared probsat computation for a given configuration

```
#include <probsat-with-resolvents.hpp>
```

Collaboration diagram for probsat_resolvents< cfg_vector_t, prec_t, use_bitfield >::prepared_computation:


std::basic_string< Char >
std::string
fname
shared_tmpfile
stmpfile
probsat_resolvents < cfg_vector_t, prec _t, use_bitfield >::prepared _computation

Public Attributes

- shared_tmpfile **stmpfile**

  name of the temporary file
- seed_distr_t **seed_distr**

  seed distribution

6.31.1 Detailed Description

template<typename cfg_vector_t, typename prec_t, bool use_bitfield>
struct probsat_resolvents< cfg_vector_t, prec_t, use_bitfield >::prepared_computation

represents a prepared probsat computation for a given configuration

Definition at line 52 of file probsat-with-resolvents.hpp.

The documentation for this struct was generated from the following file:

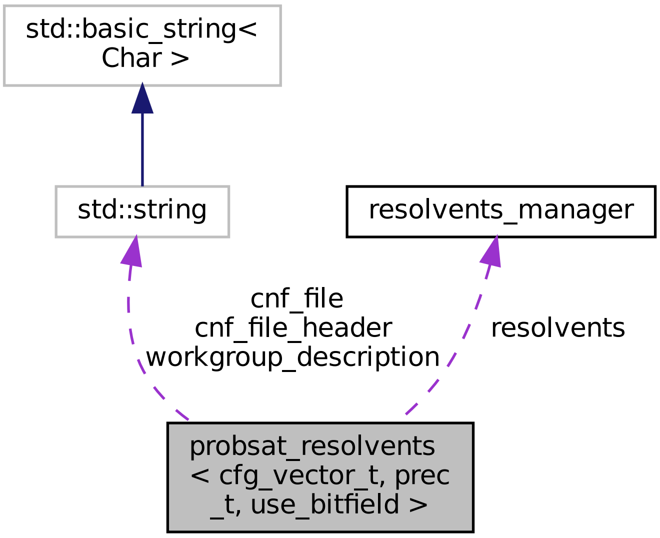
- metropolis_hastings/value_computation/probsat-with-resolvents.hpp

6.32 probsat_resolvents< cfg_vector_t, prec_t, use_bitfield > Class Template Reference

wrapper to create and solve probsat instances with resolvents

```
#include <probsat-with-resolvents.hpp>
```

Collaboration diagram for probsat_resolvents< cfg_vector_t, prec_t, use_bitfield >:



Classes

- struct prepared_computation

  represents a prepared probsat computation for a given configuration

Public Member Functions

- **probsat_resolvents** (std::string cnf_filename, std::string resolvents_fn)

  constructor loading cnf formula and resolvents
- **probsat_resolvents** (const probsat_resolvents &other)=delete

  disable copy constructor
- void **prepare_computation_with_cfg_vector** (const cfg_vector_t &v, prepared_computation &pcom)

  prepare probsat computations for a given config vector
- template<typename prng_t , typename rpmanager_t >
  auto **operator()** (prng_t &prng, rpmanager_t &rpmanager, prepared_computation &pcom) const

  create a remote function call for one execution

Public Attributes

- std::string **workgroup_description**

  string to adapt workgroup description for parallel execution
- resolvents_manager **resolvents**

  resolvents manager

Private Types

- using **seed_distr_t** = std::uniform_int_distribution$<$ probsat_seed_t $>$

  seed distribution type

Private Member Functions

- FILE $*$ **create_cnf_file_from_cfg_vector** (const cfg_vector_t &v, FILE $*$fptr) const

  add resolvents to the cnf formula based on a given config vector

Private Attributes

- std::size_t **cnf_num_vars**

  number of variables
- std::size_t **cnf_num_clauses**

  number of clauses
- std::string **cnf_file**

  cnf formula
- std::string **cnf_file_header**

  header of the cnf formula

6.32.1 Detailed Description

template$<$typename cfg_vector_t, typename prec_t, bool use_bitfield$>$
requires (std::is_same$<$binary_configuration_vector$<$use_bitfield$>$, cfg_vector_t$>$::value)
class probsat_resolvents$<$ cfg_vector_t, prec_t, use_bitfield $>$

wrapper to create and solve probsat instances with resolvents

Definition at line 23 of file probsat-with-resolvents.hpp.

The documentation for this class was generated from the following file:

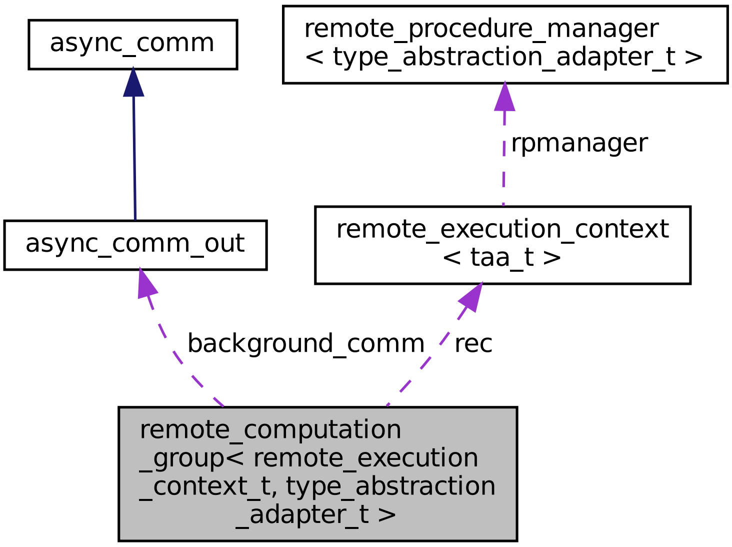
- metropolis_hastings/value_computation/probsat-with-resolvents.hpp

6.33 remote_computation_group< remote_execution_context_t, type_abstraction_adapter_t > Class Template Reference

manager class to execute functions on remote nodes using a remote execution context

```
#include <remote_computation_group.hpp>
```

Collaboration diagram for remote_computation_group< remote_execution_context_t, type_abstraction_adapter_t >:


async_comm
remote_procedure_manager < type_abstraction_adapter_t >
rpmanager
async_comm_out
remote_execution_context < taa_t >
background_comm
rec
remote_computation_group< remote_execution_context_t, type_abstraction_adapter_t >

Public Types

- using **rpc_message_t** = remote_execution_context_t::rpc_message_t

  task type: remote procedure call message
- using **buffer_t** = type_abstraction_adapter_t::abstract_type

  communication buffer type

Public Member Functions

- **remote_computation_group** (MPI_Comm env_comm, int mpi_tag, remote_execution_context_t &rec)

  constructor
- void **schedule** (auto task, int target_id)

  schedule a task on a node
- std::size_t **num_tasks_waiting** () const

  number tasks waiting to be executed
- std::size_t **num_tasks_pending** () const

  number of pending tasks on remote nodes
- bool **is_work_outstanding** () const

  check if any work is left or everything completed
- void **do_work** (bool wait_for_work=false)

  communicate, do available work and optionally wait for work

Public Attributes

- MPI_Comm **env_comm**

    communicator this scheduler manages
- const int **env_comm_id**

    id of node in communicator
- const int **env_comm_size**

    number of nodes in communicator
- const int **mpi_tag**

    tag used for scheduler communication
- remote_execution_context_t & **rec**

    remote execution context

Private Member Functions

- std::pair< int, rpc_message_t > **get_msg** (MPI_Status &status)

    recieve a pending message
- std::pair< int, rpc_message_t > **wait_for_msg** ()

    wait until a message arrives
- void **execute** (const std::pair< int, rpc_message_t > &task)

    execute a message using the remote execution context

Private Attributes

- std::deque< std::pair< int, rpc_message_t > > **tasks**

    number of tasks waiting to be executed
- type_abstraction_adapter_t **taa**

    serialization of abstract types
- std::size_t **num_waiting_for_reply**

    number of pending (scheduled) tasks
- async_comm_out **background_comm**

    asynchronous background communication

6.33.1 Detailed Description

template<typename remote_execution_context_t, class type_abstraction_adapter_t = remote_execution_context_t::type_↩
abstraction_adapter_t>
requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
class remote_computation_group< remote_execution_context_t, type_abstraction_adapter_t >

manager class to execute functions on remote nodes using a remote execution context

Definition at line 20 of file remote_computation_group.hpp.

The documentation for this class was generated from the following file:

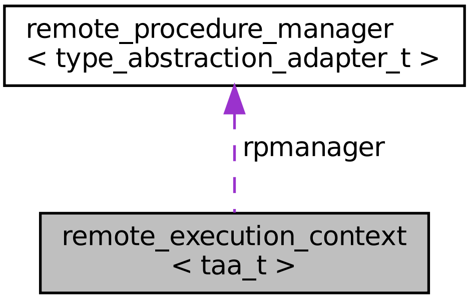
- distributed_computation/remote_computation_group.hpp

6.34 remote_execution_context< taa_t > Class Template Reference

context for remote function execution including on_result functions

```
#include <remote_execution_context.hpp>
```

Collaboration diagram for remote_execution_context< taa_t >:



Public Types

- using **type_abstraction_adapter_t** = taa_t

  type of data type abstraction adapter
- using **data_t** = type_abstraction_adapter_t::abstract_type

  abstract data type
- using **rpc_message_t** = rpc_message< data_t >

  remote procedure call message type
- using **remote_procedure_manager_t** = remote_procedure_manager< type_abstraction_adapter_t >

  remote procedure call manager type

Public Member Functions

- **remote_execution_context** (remote_procedure_manager_t &rpmanager)

  constructor
- template<typename function_t >
  rpc_message_t **on_result** (auto msg, function_t function)

  adding an on_result function to a remote procedure call message
- auto **execute_rpc** (rpc_message_t &msg)

  execute a function by passing a remote prodedure call message

Public Attributes

- remote_procedure_manager_t & **rpmanager**

  remote procedure call manager

Private Types

- using **uuid_t** = uuid< remote_execution_context< type_abstraction_adapter_t > >

  uuid data type for function call tracking

6.34.1 Detailed Description

template<typename taa_t>
requires (taa_t::is_TypeAbstractionAdapter)
class remote_execution_context< taa_t >

context for remote function execution including on_result functions

Definition at line 9 of file remote_execution_context.hpp.

The documentation for this class was generated from the following file:

- distributed_computation/remote_execution_context.hpp

6.35 remote_procedure_manager< type_abstraction_adapter_t > Class Template Reference

class to manager remote procedure calls

```
#include <rpc.hpp>
```

Public Types

- using **rpc_message_t** = rpc_message< typename type_abstraction_adapter_t::abstract_type >

  remote procedure call message type

Public Member Functions

- template<typename function_t >
  void **add_function** (std::string name, function_t function)

  make function available for remote procedure calls
- template<typename... param_types>
  auto **prepare_call** (std::string name, param_types... params)

  create a remote procedure call message for a function call
- auto **execute_rpc** (const rpc_message_t &msg)

  call function by passing a remote procedure call message

Public Attributes

- std::map< std::string, std::unique_ptr< abstract_function_t > > **function_map**

  functions available for call

Private Types

- using **abstract_function_t** = AbstractFunction< type_abstraction_adapter_t >

  function abstraction

Private Attributes

- type_abstraction_adapter_t **type_abstraction**

  used for parameter abstraction

6.35.1 Detailed Description

template<typename type_abstraction_adapter_t>
requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
class remote_procedure_manager< type_abstraction_adapter_t >

class to manager remote procedure calls

Definition at line 146 of file rpc.hpp.

The documentation for this class was generated from the following file:

- distributed_computation/rpc.hpp

6.36 rescheduling_manager Class Reference

class is used to reschedule workload between workgroups

```
#include <rescheduling.hpp>
```

Public Member Functions

- **rescheduling_manager** (int num_workgroups, std::size_t num_mcs, bool use_mc_limit, bool rescheduling_required=false)

  constructor
- void **start_rescheduling** ()

  start a new scheduling by reseting internal state
- bool **is_rescheduling_in_progress** () const

  checks rescheduling state
- void **add_progress_for_id** (std::size_t id, std::size_t progress, int wg_id)

  add progress of a markov chain on a workgroup after start of rescheduling
- void **remove_mc** (std::size_t id)

  remove a markov chain from the scheduling if finished
- bool **information_complete** () const

  check if all progress information for rescheduling is available
- bool **is_rescheduling_unnecessary** (std::size_t reschedule_requires_absolute_difference) const

  check if rescheduling is appropriate
- void **cancel_rescheduling** (auto do_rescheduling)

  cancel a rescheduling (i.e. if unnecessary)
- void **reschedule** (auto do_rescheduling)

  perform a rescheduling

Private Attributes

- bool **rescheduling_required**

  rescheduling state
- int **num_workgroups**

  number of workgroups
- std::size_t **number_of_active_mcs**

  number of markov chains which are to be scheduled
- multiway_partitioning< std::pair< std::size_t, int >, std::size_t, std::function< bool(std::size_t, std::size_t)> > **mwpart**

  use multiway partitioning (largest differencing method) for rescheduling
- std::vector< std::size_t > **workload**

  current workload on each workgroup

6.36.1 Detailed Description

class is used to reschedule workload between workgroups

Definition at line 14 of file rescheduling.hpp.

The documentation for this class was generated from the following file:

- distributed_computation/rescheduling.hpp

6.37 resolvents_manager Class Reference

class to manage resolvents

```
#include <resolvents_manager.hpp>
```

Public Member Functions

- **resolvents_manager** ()=default

  constructor
- bool **load** (std::string filename)

  load file with resolvents
- std::size_t **get_num_resolvents** () const

  get number of resolvents
- std::string **get_resolvent** (std::size_t i) const

  access resolvent at index i

Private Attributes

- std::vector< std::string > **resolvent**

  list of resolvents

6.37.1 Detailed Description

class to manage resolvents

Definition at line 8 of file resolvents_manager.hpp.

The documentation for this class was generated from the following files:

- metropolis_hastings/value_computation/utility/resolvents_manager.hpp
- metropolis_hastings/value_computation/utility/resolvents_manager.cpp

6.38 result_statistics< prec_t > Struct Template Reference

class to gather statistics about probsat executions per markov chain

```
#include <probsat-execution.hpp>
```

Public Member Functions

- prec_t **operator()** (std::pair< uint64_t, probsat_return_cause::reason > result)

  add a result to the statistic

Public Attributes

- std::size_t **probsat_executions** = 0

  number of executions
- std::size_t **reasons** [probsat_return_cause::NUM_REASONS] = {0}

  count return reasons
- double **total_flips_executed** = 0

  number of total flips executed

Friends

- std::ostream & **operator**<< (std::ostream &os, const result_statistics &rs)

  print statistics

6.38.1 Detailed Description

template<typename prec_t = prec_t>
struct result_statistics< prec_t >

class to gather statistics about probsat executions per markov chain

Definition at line 67 of file probsat-execution.hpp.

The documentation for this struct was generated from the following file:

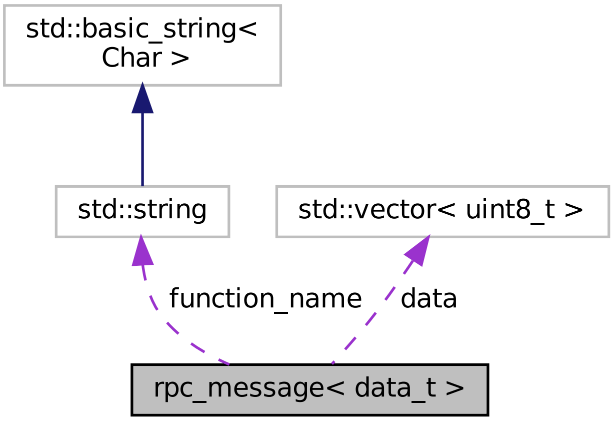
- metropolis_hastings/value_computation/utility/probsat-execution.hpp

6.39 rpc_message< data_t > Class Template Reference

message to code a (remote) procedure call

```
#include <rpc.hpp>
```

Collaboration diagram for rpc_message< data_t >:



Public Types

- using **data_type** = data_t

  message type

Public Member Functions

- **rpc_message** ()

  constructor
- **rpc_message** (const rpc_message &other)

  copy constructor
- rpc_message & **operator=** (const rpc_message &other)

  assignment operator
- **rpc_message** (rpc_message &&other)

  move semantics by using swap
- uint8_t **compute_crc** ()

  compute and set checksum
- bool **check_crc** () const

  test for correct checksum

Public Attributes

- std::size_t **id**

  message id
- std::string **function_name**

  function name
- bool **answer_required**

  wheter an answer is awaited
- data_t **data**

  function parameter data (abstract representation)
- uint8_t **crc**

  checksum (optional, used for debugging and error detection)

Private Member Functions

- uint8_t **compute_crc_** () const

  actual checksum computation (helper function)

Friends

- void **swap** (rpc_message &l, rpc_message &r)

  swap function

6.39.1 Detailed Description

template<typename data_t = std::vector<uint8_t>>
class rpc_message< data_t >

message to code a (remote) procedure call

Definition at line 26 of file rpc.hpp.

The documentation for this class was generated from the following file:

- distributed_computation/rpc.hpp

6.40 seed_type< size, base_type > Class Template Reference

generic seed sequence for random generators

```
#include <seed.hpp>
```

Public Types

- using **base_t** = base_type

  seed sequence number type

Public Member Functions

- auto **get_seed_seq** ()

  provides access to the seed sequence
- **seed_type** (bool generate_random_seed=false)

  (default) constructor with unitialized or random seed
- **seed_type** (std::string seed_str)

  construct seed from string
- base_type **short_rep** ()

  short representation
- bool **operator==** (const seed_type< size, base_type > &rhs) const

  compare operator for seeds
- bool **operator!=** (const seed_type< size, base_type > &rhs) const

  compare operator for seeds

Public Attributes

- std::array< base_type, size > **seed_data**

  seed sequence data
- template<typename rdev_t >

  __pad0__: seed_data() { init(rng_dev

  construct using a random number generator and a seed_type_generator

Private Member Functions

- template<typename rdev_t >

  void **init** (rdev_t &rng_dev, seed_type_generator< base_type > ∗sgen_ptr=nullptr)

  initialize this seed from a random device using a distribution

Friends

- std::ostream & **operator<<** (std::ostream &os, const seed_type &st)

  std::ostream output operator for seeds
- std::istream & **operator>>** (std::istream &is, seed_type &st)

  std::istream input operator for seeds

6.40.1 Detailed Description

template<std::size_t size, typename base_type>
class seed_type< size, base_type >

generic seed sequence for random generators

Definition at line 32 of file seed.hpp.

The documentation for this class was generated from the following file:

- util/seed.hpp

6.41 seed_type_generator< base_type > Struct Template Reference

wrapper to initialize multiple seeds from one distribution

```
#include <seed.hpp>
```

Public Member Functions

- **seed_type_generator** ()
  constructor

Public Attributes

- std::uniform_int_distribution< base_type > **dist**
  seed distribution

6.41.1 Detailed Description

template<typename base_type>
struct seed_type_generator< base_type >

wrapper to initialize multiple seeds from one distribution

Definition at line 15 of file seed.hpp.

The documentation for this struct was generated from the following file:

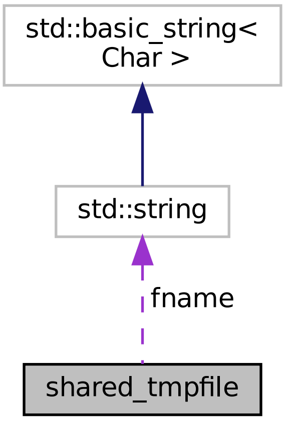
- util/seed.hpp

6.42 shared_tmpfile Class Reference

temporary file implementation

```
#include <shared_tmpfile.hpp>
```

Collaboration diagram for shared_tmpfile:



Public Member Functions

- **shared_tmpfile** ()

  dummy default constructor, does not provide any file access
- shared_tmpfile (std::string purpose)

  create a temporary file for purpose (string)
- **shared_tmpfile** (const shared_tmpfile &other)=delete

  disable copy constructor
- **shared_tmpfile** (shared_tmpfile &&other) noexcept

  move constructor by using swap
- shared_tmpfile & **operator=** (shared_tmpfile other)

  assignment operator by using swap
- void **remove** ()

  remove the tmp file manually
- ∼**shared_tmpfile** ()

  deconstructor, removes the tmp file if required

Public Attributes

- std::string **fname**

  name of tmp file
- FILE ∗ **fptr**

  FILE ∗ pointer.

Private Attributes

- std::size_t **id**

  tmpfile id

Friends

- void **swap** (shared_tmpfile &a, shared_tmpfile &b)

  swap implementation for copy & swap idiom

6.42.1 Detailed Description

temporary file implementation

provides temporary files identified by an uuid and purpose string which can therefore be reused.

Definition at line 13 of file shared_tmpfile.hpp.

6.42.2 Constructor & Destructor Documentation

6.42.2.1 shared_tmpfile()

```
shared_tmpfile::shared_tmpfile (
              std::string purpose )
```

create a temporary file for purpose (string)

the temporary file opened will contain an uuid and the purpose string and therefore can be reused by the application

Definition at line 11 of file shared_tmpfile.cpp.

```
00011                                                  :
00012     id(uuid<shared_tmpfile>::get()+1),
00013     fname(),
00014     fptr(nullptr)
00015 {
00016     std::string tmp_path = std::filesystem::temp_directory_path();
00017     fname = tmp_path + "/tmpfile-" + purpose + "_" + std::to_string(id);
00018 
00019     // std::cout « "created shared_tmpfile: " « fname « std::endl;
00020     fptr = fopen(fname.c_str(), "wbx+");
00021     if (nullptr == fptr) {
00022         std::string errmsg = "fopen error (shared_tmpfile "
00023             + std::to_string(id) + "):";
00024         perror(errmsg.c_str());
00025 
00026         if (false) {
00027             fptr = fopen(fname.c_str(), "wb+");
00028             assert(fptr);
00029             std::cerr
00030                 « "note: recovered by reopening with wb+" « std::endl;
00031         } else {
00032             assert(false);
00033         }
00034     }
00035 }
```

The documentation for this class was generated from the following files:

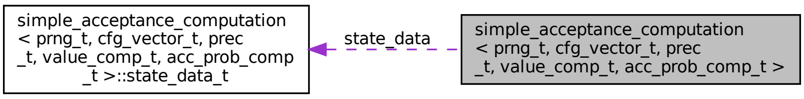
- util/shared_tmpfile.hpp
- util/shared_tmpfile.cpp

6.43 simple_acceptance_computation< prng_t, cfg_vector_t, prec_t, value_comp_t, acc_prob_comp_t > Class Template Reference

decides over acceptance of a new state in a markov chain using a exact value

```
#include <simple_acceptance_computation.hpp>
```

Collaboration diagram for simple_acceptance_computation< prng_t, cfg_vector_t, prec_t, value_comp_t, acc_prob_comp_t >:


simple_acceptance_computation
< prng_t, cfg_vector_t, prec
_t, value_comp_t, acc_prob_comp
_t >::state_data_t
state_data
simple_acceptance_computation
< prng_t, cfg_vector_t, prec
_t, value_comp_t, acc_prob_comp_t >

Classes

- struct state_data_t

    compute and represent a state of a markov chain

Public Member Functions

- **simple_acceptance_computation** ()

    constructor
- void **skip_computation** (prng_t &main_prng)

    skip one step for fast state recovery of random number generator
- template<typename scheduler_t , typename result_projection_t = std::identity, typename result_function_t = std::function<void(prec↩ _t)>>

    void **start_computation** (prng_t &main_prng, const cfg_vector_t &v, scheduler_t &scheduler, value_comp↩ _t &value_comp, result_projection_t &result_projection={})

    schedule computation of new state value
- bool **still_waiting_for_computation** ()

    check whether this computation is still outstanding
- template<typename scheduler_t >

    bool **continue_computation** (scheduler_t &scheduler, value_comp_t &value_comp, acc_prob_comp_t &acc_prob_fct)

    call this function to continue the computation until it returns true
- bool **finish_computation** (value_comp_t &value_comp, acc_prob_comp_t &acc_prob_fct)

    finish the computation to know wheter to accept the new state
- void **cleanup** (value_comp_t &value_comp)

    finish any outstanding computations
- prec_t **get_current_value** ()

    get the value of the current state
- prec_t **get_last_computed_value** ()

    get the value of the last computed state
- prec_t **get_previous_computed_value** ()

    get the value of the previous state

Private Member Functions

- auto **derive_prng** (prng_t &main_prng)

    derive a pseudo random number generator for each step

Private Attributes

- std::uniform_real_distribution< prec_t > **prob_distribution**

    probability distribution to get cutoff values for acceptance
- state_data_t **state_data** [2]

    markov chain state information of current and next state
- bool **is_first_cfg_vector**

    application state to accept the first state anyway
- bool **accept**

    last result
- uint8_t **new_index**

    index into state_data of the next/new state
- bool **computation_scheduled_state**

    application state whether a value computation is scheduled
- std::size_t **remaining_computations**

    number of remaining computations

6.43.1 Detailed Description

template<typename prng_t, typename cfg_vector_t, typename prec_t, typename value_comp_t, typename acc_prob_comp_t = exponential_acceptance_probability<prec_t>>
requires (value_comp_t::is_value_computation_implementation && acc_prob_comp_t::is_acceptance_probability_implementation)
class simple_acceptance_computation< prng_t, cfg_vector_t, prec_t, value_comp_t, acc_prob_comp_t >

decides over acceptance of a new state in a markov chain using a exact value

Definition at line 26 of file simple_acceptance_computation.hpp.

The documentation for this class was generated from the following file:

- metropolis_hastings/acceptance_computation/simple_acceptance_computation.hpp

6.44 simple_change_generator< cfg_vector_t, enforce_change, value_t, value_distr_type, value_distr_minv, value_distr_maxv > Class Template Reference

class to generate changes in config vectors

```
#include <change_generator.hpp>
```

Public Member Functions

- **simple_change_generator** (const cfg_vector_t &v)

  constructor
- auto **operator()** (auto &prng, const cfg_vector_t &v)

  create change for a given config vector

Private Attributes

- std::uniform_int_distribution< index_t > **index_distr**

  index distribution
- value_distr_type **value_distr**

  value distribution

6.44.1 Detailed Description

template<typename cfg_vector_t, bool enforce_change, typename value_t = std::conditional< std::is_same<bool, typename cfg_vector_t::value_type>::value, short, typename cfg_vector_t::value_type >::type, typename value_distr_type = std::uniform_int_distribution<value_t>, decltype(value_distr_type().min()) value_distr_minv = cfg_vector_t::min_value, decltype(value_distr_type().max()) value_distr_maxv = enforce_change ? cfg_vector_t::max_value - 1 : cfg_vector_t::max_value>
class simple_change_generator< cfg_vector_t, enforce_change, value_t, value_distr_type, value_distr_minv, value_distr_maxv >

class to generate changes in config vectors

Definition at line 54 of file change_generator.hpp.

The documentation for this class was generated from the following file:

- metropolis_hastings/configuration/change_generator.hpp

6.45 simple_acceptance_computation< prng_t, cfg_vector_t, prec_t, value_comp_t, acc_prob_comp_t >::state_data_t Struct Reference

compute and represent a state of a markov chain

Public Attributes

- value_comp_t::prepared_computation **pcom**

  data to compute each value
- prng_t **prng**

  pseudo random number generator used in this step
- prec_t **value**

  computed value of this markov chain state

6.45.1 Detailed Description

template<typename prng_t, typename cfg_vector_t, typename prec_t, typename value_comp_t, typename acc_prob_comp_t = exponential_acceptance_probability<prec_t>>
struct simple_acceptance_computation< prng_t, cfg_vector_t, prec_t, value_comp_t, acc_prob_comp_t >::state_data_t

compute and represent a state of a markov chain

Definition at line 33 of file simple_acceptance_computation.hpp.

The documentation for this struct was generated from the following file:

- metropolis_hastings/acceptance_computation/simple_acceptance_computation.hpp

6.46 statistical_acceptance_computation< prng_t, cfg_vector_t, prec_t, value_comp_t, acc_prob_comp_t >::state_data_t Struct Reference

compute a state of a markov chain

Public Attributes

- value_comp_t::prepared_computation **pcom**

  data to compute each value
- prng_t **prng**

  pseudo random number generator used in this step
- std::vector< prec_t > **values**

  number of computed values
- std::size_t **iteration**

  how often more values have been requested
- basic_statistical_metrics< prec_t > **statistics**

  statistical properties of computed values

6.46.1 Detailed Description

template<typename prng_t, typename cfg_vector_t, typename prec_t, typename value_comp_t, typename acc_prob_comp_t = exponential_acceptance_probability<prec_t>>
struct statistical_acceptance_computation< prng_t, cfg_vector_t, prec_t, value_comp_t, acc_prob_comp_t >::state_data_t

compute a state of a markov chain

Definition at line 111 of file statistical_acceptance_computation.hpp.

The documentation for this struct was generated from the following file:

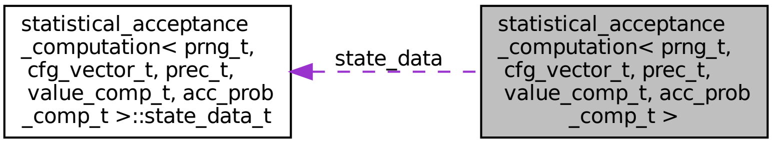
- metropolis_hastings/acceptance_computation/statistical_acceptance_computation.hpp

6.47 statistical_acceptance_computation< prng_t, cfg_vector_t, prec_t, value_comp_t, acc_prob_comp_t > Class Template Reference

decides over acceptance of a new markov chain state based on a coincidence intervall

```
#include <statistical_acceptance_computation.hpp>
```

Collaboration diagram for statistical_acceptance_computation< prng_t, cfg_vector_t, prec_t, value_comp_t, acc_prob_comp_t >:


statistical_acceptance _computation< prng_t, cfg_vector_t, prec_t, value_comp_t, acc_prob _comp_t >::state_data_t
state_data
statistical_acceptance _computation< prng_t, cfg_vector_t, prec_t, value_comp_t, acc_prob _comp_t >

Classes

- struct state_data_t

    compute a state of a markov chain

Public Member Functions

- **statistical_acceptance_computation** (double coincidence_interval_factor=cfg_sac_confidence_↩ interval_scaling_factor, double testniveau=cfg_sac_testniveau, std::size_t values_per_iteration=cfg_↩ sac_num_additional_values_per_iteration, std::size_t max_iterations=cfg_sac_max_iterations, std::size_t values_required=cfg_sac_num_initial_values)

  constructor
- void **skip_computation** (prng_t &main_prng)

  skip one step for fast state recovery of random number generator
- template<typename scheduler_t , typename result_projection_t = std::identity, typename result_function_t = std::function<void(prec↩ _t)>>
  void **start_computation** (prng_t &main_prng, const cfg_vector_t &v, scheduler_t &scheduler, value_comp↩ _t &value_comp, result_projection_t &result_projection={})

  setup computation of new state values
- bool **still_waiting_for_computation** ()

  check if all scheduled value computations finished
- template<typename scheduler_t >
  bool **continue_computation** (scheduler_t &scheduler, value_comp_t &value_comp, acc_prob_comp_t &acc_prob_fct)

  compute values until decision is clear (recall until it returns true)
- bool finish_computation (value_comp_t &value_comp, acc_prob_comp_t &acc_prob_fct)

  get result of wheter to accept the new state
- void **cleanup** (value_comp_t &value_comp)

  cleanup any unfinished computation
- prec_t **get_current_value** ()

  get the value of the current state
- prec_t **get_last_computed_value** ()

  get the value of the last computed state
- prec_t **get_previous_computed_value** ()

  get the value of the previous state

Private Member Functions

- auto **derive_prng** (prng_t &main_prng)

  derive a pseudo random number generator for each step

Private Attributes

- std::uniform_real_distribution< prec_t > **prob_distribution**

  probability distribution to get cutoff values for acceptance
- prec_t **nextz**

  next acceptance probability cutoff value
- prec_t **z**

  acceptance probability cutoff value
- state_data_t **state_data** [2]

  markov chain state information of current and next state
- bool **is_first_cfg_vector**

  application state to accept the first state anyway
- bool **accept**

  last result
- uint8_t **new_index**

index into state_data of the next/new state

- bool **computation_scheduled_state**

  application state whether a value computation is scheduled
- std::size_t **remaining_computations**

  number of remaining computations
- double **coincidence_interval_factor**

  sigma factor for confidence
- double **testniveau**

  probability cutoff off small coincidence intervalls to use average instead
- std::size_t **values_per_iteration**

  number of values to add in each iteration
- std::size_t **max_iterations**

  maximum number of iterations until average is used
- std::size_t **values_required**

  minimum number of initial values required for decision
- std::function< void(void ∗scheduler_ptr, std::size_t i, uint8_t index) > **schedule_computation**

  internally used function to schedule computations

6.47.1 Detailed Description

template<typename prng_t, typename cfg_vector_t, typename prec_t, typename value_comp_t, typename acc_prob_comp_t = exponential_acceptance_probability<prec_t>>
requires (value_comp_t::is_value_computation_implementation && acc_prob_comp_t::is_acceptance_probability_implementation)
class statistical_acceptance_computation< prng_t, cfg_vector_t, prec_t, value_comp_t, acc_prob_comp_t >

decides over acceptance of a new markov chain state based on a coincidence intervall

Definition at line 104 of file statistical_acceptance_computation.hpp.

6.47.2 Member Function Documentation

6.47.2.1 finish_computation()

```
template<typename prng_t , typename cfg_vector_t , typename prec_t , typename value_comp_t ,
typename acc_prob_comp_t = exponential_acceptance_probability<prec_t>>
bool statistical_acceptance_computation< prng_t, cfg_vector_t, prec_t, value_comp_t, acc_prob_comp_t
>::finish_computation (
            value_comp_t & value_comp,
            acc_prob_comp_t & acc_prob_fct )  [inline]
```

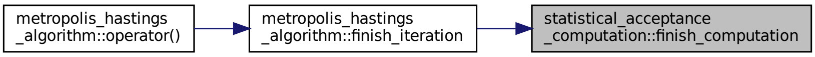
get result of wheter to accept the new state

won't cleanup computation of accepted state, as it may be required to continue with it in the next iteration

Definition at line 468 of file statistical_acceptance_computation.hpp.

```
        {
            assert(0 == remaining_computations);
            assert(computation_scheduled_state);
            computation_scheduled_state = false;

            ignore(&acc_prob_fct);

            uint8_t old_index = 1 - new_index;
            if (is_first_cfg_vector) {
                is_first_cfg_vector = false;
                new_index = old_index;
                return true;
            } else {
                if (accept) {
                    // std::cout « "accept "
                        // « state_data[new_index].statistics.average
                        // « std::endl;
                    value_comp.finish_computation(
                        state_data[old_index].pcom);
                    new_index = old_index;
                } else {
                    // std::cout « "decline "
                        // « state_data[new_index].statistics.average
                        // « std::endl;
                    // std::cout « "keep value "
                        // « state_data[old_index].statistics.average
                        // « std::endl;
                    value_comp.finish_computation(
                        state_data[new_index].pcom);
                }
                return accept;
            }
        }
```

Here is the caller graph for this function:



The documentation for this class was generated from the following file:

- metropolis_hastings/acceptance_computation/statistical_acceptance_computation.hpp

6.48 multiway_partitioning< data_t, score_t, sort_func_t >::subset_t Struct Reference

subset of a partition

`#include <multiway_partitioning.hpp>`

Public Member Functions

- **subset_t** ()

  default constructor
- **subset_t** (entry_t first_entry)

  construct subset of size one by passing one entry
- void **merge_subsets** (subset_t &other, auto cmp_function)

  merge two subsets

Public Attributes

- score_t **sum**

  sum of scores of all elements
- std::list< entry_t > **elements**

  list of elements in this subset

Friends

- std::ostream & **operator<<** (std::ostream &os, const subset_t &s)

  std::ostream operator to print a subset

6.48.1 Detailed Description

template<typename data_t, typename score_t = std::size_t, class sort_func_t = std::greater<score_t>>
struct multiway_partitioning< data_t, score_t, sort_func_t >::subset_t

subset of a partition

Definition at line 38 of file multiway_partitioning.hpp.

The documentation for this struct was generated from the following file:

- distributed_computation/multiway_partitioning.hpp

6.49 uniform_configuration_generator< cfg_vector_t > Class Template Reference

initialize a configuration vector using random values

```
#include <configuration_generator.hpp>
```

Public Member Functions

- **uniform_configuration_generator** ()

  constructor
- template<typename rng_t >
  void **operator()** (rng_t &prng, cfg_vector_t &v) const

  apply initialization method to config vector

6.49.1 Detailed Description

template<typename cfg_vector_t>
class uniform_configuration_generator< cfg_vector_t >

initialize a configuration vector using random values

Definition at line 51 of file configuration_generator.hpp.

The documentation for this class was generated from the following file:

- metropolis_hastings/configuration/configuration_generator.hpp

6.50 uuid< crtp, uuid_type, reserved_ids > Class Template Reference

uuid using crtp

`#include <uuid.hpp>`

Static Public Member Functions

- static uuid_type **get** ()

  get an unused uuid
- static void **free** (const uuid_type id)

  return an uuid for reuse

Static Private Attributes

- static uuid_type **new_uuid** = 0

  next new uuid
- static std::vector< uuid_type > **unused**

  list of unused/free ids smaller new_uuid

6.50.1 Detailed Description

template<class crtp, typename uuid_type = std::size_t, uuid_type reserved_ids = 1>
class uuid< crtp, uuid_type, reserved_ids >

uuid using crtp

A Universally Unique Identifier Implementation using the curiously recurring template pattern. UUID Type can be specified as template parameter. Also some id's can be reserved.

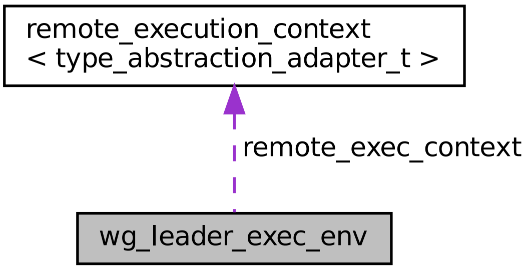
Definition at line 15 of file uuid.hpp.

The documentation for this class was generated from the following file:

- util/uuid.hpp

6.51 wg_leader_exec_env Struct Reference

Collaboration diagram for wg_leader_exec_env:



6.51.1 Detailed Description

Definition at line 102 of file main.cpp.

The documentation for this struct was generated from the following file:

- main.cpp

Chapter 7

File Documentation

7.1 config.cpp

```

#include "config.hpp"

// regarding the statistical acceptance computation
double cfg_sac_confidence_interval_scaling_factor = 1.0;

double cfg_sac_testniveau = 0.01;

std::size_t cfg_sac_num_initial_values = 2400;

std::size_t cfg_sac_num_additional_values_per_iteration = 240;

std::size_t cfg_sac_max_iterations = 12;
```

7.2 config.hpp

```
#ifndef CONFIG_HPP
#define CONFIG_HPP

#include <cstddef>
#include <string>
#include <random>
#include <chrono>

using prec_t = double;
static_assert(std::numeric_limits<prec_t>::has_quiet_NaN);

// typedef std::mt19937 prn_engine_t;
// using seed_t = seed_type<prn_engine_t::state_size, uint32_t>;

typedef std::mt19937 prn_engine_t;
// using seed_t = seed_type<prn_engine_t::state_size, uint32_t>;
// using seed_type_generator_t = seed_type_generator<typename seed_t::base_t>;


// regarding probsat
const std::string probsat_cmd
    = "./probSAT/probSAT --fct 0 --eps 0.9 --cb 2.06 --runs 1";

constexpr uint64_t probsat_max_flips = 20’000’000;

constexpr const std::chrono::minutes probsat_max_exec_time{1};

constexpr const bool interpret_timeout_as_max_flips_reached = false;

using probsat_seed_t = int32_t;

// regarding the statistical acceptance computation
extern double cfg_sac_confidence_interval_scaling_factor;

extern double cfg_sac_testniveau;

extern std::size_t cfg_sac_num_initial_values;

extern std::size_t cfg_sac_num_additional_values_per_iteration;

extern std::size_t cfg_sac_max_iterations;

#endif

```

7.3 function_abstraction.hpp

```
#ifndef FUNCTION_ABSTRACTION_HPP
#define FUNCTION_ABSTRACTION_HPP

#include <functional>

#include "rtti.hpp"

template <class type_abstraction_adapter_t>
    requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
struct function_typeinfo
{
    // functions without arguments

    template <typename return_type>
    static ConcreteType<type_abstraction_adapter_t>
        get_concrete_argument_type(return_type(*)())
    {
        return ConcreteType<type_abstraction_adapter_t>(true);
    }

    template <typename fobject, typename return_type>
    static ConcreteType<type_abstraction_adapter_t>
        get_concrete_argument_type(return_type(fobject::*)())
    {
        return ConcreteType<type_abstraction_adapter_t>(true);
    }

    template <typename return_type>
    static ConcreteType<type_abstraction_adapter_t>
        get_concrete_argument_type(std::function<return_type()> f)
    {
        ignore(f);
        return ConcreteType<type_abstraction_adapter_t>(true);
    }


    template<typename first_arg_type, typename... tuple_tail_types>
    static auto create_concrete_type(
        first_arg_type *first_arg,
        std::tuple<tuple_tail_types...> &tail)
    {
        ignore(first_arg, tail);
        return ConcreteType<
            type_abstraction_adapter_t,
            first_arg_type,
            tuple_tail_types...>();
    }


    template <
        typename return_type,
        typename first_arg_type,
        typename... arg_types>
    static auto get_concrete_argument_type(
        return_type(*)(first_arg_type, arg_types...))
    {
        first_arg_type first_arg;
        std::function<return_type(arg_types...)> tail_func;
        typename decltype(
            get_concrete_argument_type(tail_func)
        )::tuple_t tail;
        return create_concrete_type(&first_arg, tail);
    }

    template <
        typename fobject,
        typename return_type,
        typename first_arg_type,
        typename... arg_types>
    static auto get_concrete_argument_type(
        return_type(fobject::*)(first_arg_type, arg_types...))
    {
        first_arg_type first_arg;
        std::function<return_type(arg_types...)> tail_func;
        typename decltype(
            get_concrete_argument_type(tail_func)
        )::tuple_t tail;
        return create_concrete_type(&first_arg, tail);
    }

    template <
        typename return_type,
        typename first_arg_type,
        typename... arg_types>
    static auto get_concrete_argument_type(
        std::function<return_type(first_arg_type, arg_types...)> f)
    {
        ignore(f);
        first_arg_type first_arg;
        std::function<return_type(arg_types...)> tail_func;
        typename decltype(
            get_concrete_argument_type(tail_func)
        )::tuple_t tail;
        return create_concrete_type(&first_arg, tail);
    }

    template <
        typename return_type,
        typename first_arg_type,
        typename... arg_types>
    static auto get_concrete_argument_type(
        return_type(*)(const first_arg_type &, arg_types...))
    {
        first_arg_type first_arg;
        std::function<return_type(arg_types...)> tail_func;
        typename decltype(
            get_concrete_argument_type(tail_func)
        )::tuple_t tail;
        return create_concrete_type(&first_arg, tail);
    }

    template <
        typename fobject,
        typename return_type,
        typename first_arg_type,
        typename... arg_types>
    static auto get_concrete_argument_type(
        return_type(fobject::*)(const first_arg_type &, arg_types...))
    {
        first_arg_type first_arg;
        std::function<return_type(arg_types...)> tail_func;
        typename decltype(
            get_concrete_argument_type(tail_func)
        )::tuple_t tail;
        return create_concrete_type(&first_arg, tail);
    }

    template <
        typename return_type,
        typename first_arg_type,
        typename... arg_types>
    static auto get_concrete_argument_type(
        std::function<return_type(const first_arg_type &, arg_types...)> f)
    {
        ignore(f);
        first_arg_type first_arg;
        std::function<return_type(arg_types...)> tail_func;
        typename decltype(
            get_concrete_argument_type(tail_func)
        )::tuple_t tail;
        return create_concrete_type(&first_arg, tail);
    }

/* old implementation: did not allow const or const reference as parameters
    template <typename return_type, typename... arg_types>
    static ConcreteType<type_abstraction_adapter_t, arg_types...>
        get_concrete_argument_type(return_type(*)(arg_types...))
    {
        return ConcreteType<type_abstraction_adapter_t, arg_types...>();
    }

    template <typename fobject, typename return_type, typename... arg_types>
    static ConcreteType<type_abstraction_adapter_t, arg_types...>
        get_concrete_argument_type(return_type(fobject::*)(arg_types...))
    {
        return ConcreteType<type_abstraction_adapter_t, arg_types...>();
    }

    template <typename return_type, typename... arg_types>
    static ConcreteType<type_abstraction_adapter_t, arg_types...>
        get_concrete_argument_type(std::function<return_type(arg_types...)> f)
    {
        ignore(f);
        return ConcreteType<type_abstraction_adapter_t, arg_types...>();
    }
*/

    // idea to get return type is based on:
    // https://stackoverflow.com/questions/41301536/get-function-return-type-in-template

    template <typename return_type, typename... arg_types>
    static auto get_concrete_return_type(return_type(*)(arg_types...))
    {
        if constexpr (std::is_same<void, return_type>::value) {
            return ConcreteType<type_abstraction_adapter_t>(true);
        } else {
            return ConcreteType<type_abstraction_adapter_t, return_type>();
        }
    }

    template <
        typename fobject, typename return_type, typename... arg_types>
    static auto get_concrete_return_type(
        return_type(fobject::*)(arg_types...))
    {
        if constexpr (std::is_same<void, return_type>::value) {
            return ConcreteType<type_abstraction_adapter_t>(true);
        } else {
            return ConcreteType<type_abstraction_adapter_t, return_type>();
        }
    }

    template <typename return_type, typename... arg_types>
    static ConcreteType<type_abstraction_adapter_t>
        get_concrete_return_type(
            std::function<return_type(arg_types...)> f)
    {
        ignore(f);
        if constexpr (std::is_same<void, return_type>::value) {
            return ConcreteType<type_abstraction_adapter_t>(true);
        } else {
            return ConcreteType<type_abstraction_adapter_t, return_type>();
        }
    }
};



template<typename type_abstraction_adapter_t>
    requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
class AbstractFunction
{
    public:
        type_abstraction_adapter_t::abstract_type operator()(
            type_abstraction_adapter_t &type_abstraction,
            const type_abstraction_adapter_t::abstract_type &abstract_param)
        {
            return this->execute(type_abstraction, abstract_param);
        }

        virtual type_abstraction_adapter_t::abstract_type execute(
            type_abstraction_adapter_t &type_abstraction,
            const type_abstraction_adapter_t::abstract_type &abstract_param)
        {
            ignore(type_abstraction, abstract_param);
            typename type_abstraction_adapter_t::abstract_type result;
            return result;
        }

        virtual ~AbstractFunction() { }
};


template<
    typename type_abstraction_adapter_t,
    typename function_t,
    typename param_t,
    typename result_t>
requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
class ConcreteFunction : public AbstractFunction<type_abstraction_adapter_t>
{
    private:
        function_t function;

        param_t parameters;

        result_t result;

    public:
        ConcreteFunction(auto function, auto param, auto result) :
            function(function), parameters(param), result(result) {}

        virtual type_abstraction_adapter_t::abstract_type execute(
            type_abstraction_adapter_t &type_abstraction,
            const type_abstraction_adapter_t::abstract_type &abstract_param
        ) override
        {
            assert(parameters.deserialize(type_abstraction, abstract_param)
                && "(most probably wrong parameters passed)");

            using ret_t = decltype(std::apply(function, parameters.values));

            if constexpr (std::is_same<void, ret_t>::value) {
                std::apply(function, parameters.values);
            } else {
                result.values = std::make_tuple(
                    std::apply(function, parameters.values));
            }

            return result.serialize(type_abstraction);
        }

        virtual ~ConcreteFunction() { }
};


template<typename type_abstraction_adapter_t, typename function_t>
    requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
AbstractFunction<type_abstraction_adapter_t> *
    getAbstractFunctionFor(function_t f)
{
    using taa_t = type_abstraction_adapter_t;

    auto params = function_typeinfo<taa_t>::get_concrete_argument_type(f);
    auto result = function_typeinfo<taa_t>::get_concrete_return_type(f);

    return new ConcreteFunction<
        type_abstraction_adapter_t,
        decltype(f),
        decltype(params),
        decltype(result)
    >(  f, params, result  );
}

#endif
```

7.4 lazy_round_robin_scheduler.hpp

```
#ifndef LAZY_ROUND_ROBIN_SCHEDULER_HPP
#define LAZY_ROUND_ROBIN_SCHEDULER_HPP

#include <cstddef>
#include <vector>
#include <map>
#include <cassert>

template<std::size_t workload_capacity>
class lazy_round_robin_scheduler
{
    private:
        const int size;

        std::vector<std::size_t> num_tasks_scheduled;

        std::multimap<std::size_t, int> ids_for_scheduled;

    public:

        bool is_process_without_work()
        {
            if (0 >= size) { return false; }

            auto first_it = ids_for_scheduled.begin();
            return (0 == first_it->first);
        }

        bool worker_available()
        {
            if (0 >= size) { return false; }

            auto first_it = ids_for_scheduled.begin();
            return (workload_capacity > first_it->first);
        }

        int get_id_to_schedule_task_on()
        {
            if (0 >= size) { return -1; }

            auto first_it = ids_for_scheduled.begin();
            if (workload_capacity == first_it->first) {
                return -1;
            }

            int id = first_it->second;
            const std::size_t num_scheduled_old =
                num_tasks_scheduled[id]++;
            assert(first_it->first == num_scheduled_old);

            ids_for_scheduled.erase(first_it);
            ids_for_scheduled.emplace(num_scheduled_old + 1, id);

            // std::cout « "get_id_to_schedule_task_on: "
                // « id « " is now busy with "
                // « (num_scheduled_old+1) « " tasks" « std::endl;

            return id;
        }

        void task_finished(int id)
        {
            if (0 >= size) { return; }

            assert(0 <= id);
            assert(id < size);

            // std::cout « "worker " « id
                // « " with " « num_tasks_scheduled[id]
                // « " tasks finished one" « std::endl;

            std::size_t num_scheduled_old = num_tasks_scheduled[id]--;
            assert(0 < num_scheduled_old);

            auto it = ids_for_scheduled.equal_range(num_scheduled_old);

            for (auto i = it.first; i != it.second; i++) {
                if (i->second == id) {
                    ids_for_scheduled.erase(i);
                    ids_for_scheduled.emplace(num_scheduled_old - 1, id);
                    return;
                }
            }

            assert(false);
        }

        lazy_round_robin_scheduler(int size) :
            size(size), num_tasks_scheduled(size), ids_for_scheduled()
        {
            static_assert(0 < workload_capacity);
            num_tasks_scheduled.resize(size);
            for (int id = 0; id < size; id++) {
                num_tasks_scheduled[id] = 0;
                ids_for_scheduled.emplace(0, id);
            }
        }
};

#endif
```

7.5 multiway_partitioning.hpp

```
#ifndef MULTIWAY_PARTITIONING_HPP
#define MULTIWAY_PARTITIONING_HPP

#include <cstddef>
#include <functional>
#include <list>
#include <iostream>
#include <cassert>
#include <random>

template<
    typename data_t,
    typename score_t = std::size_t,
    class sort_func_t = std::greater<score_t»
class multiway_partitioning
{
    public:

        struct entry_t {
            score_t score;

            data_t data;
        };


        struct subset_t {

            score_t sum;

            std::list<entry_t> elements;

            subset_t() : sum(0), elements() {}

            subset_t(entry_t first_entry) :
                sum(first_entry.score), elements({first_entry}) {}

            void merge_subsets(subset_t &other, auto cmp_function)
            {
                sum += other.sum;
                // merge_two_lists(elements, other.elements);
                elements.sort(cmp_function);
                other.elements.sort(cmp_function);
                elements.merge(other.elements, cmp_function);
            }

            friend std::ostream &operator«(
                std::ostream &os, const subset_t &s)
            {
                os « "{";
                if (!s.elements.empty()) {
                    bool first = true;
                    for (auto el : s.elements) {
                        if (!first) { os « ", "; }
                        os « el.score;
                        first = false;
                    }
                }
                os « "} [S: " « s.sum
                    « ", l: " « s.elements.size() « "]";
                return os;
            }
        };


        struct partitioning_t {
            std::vector<subset_t> subsets; // number of partitions = subset.size

            score_t min_sum;

            score_t max_sum;

            auto difference(std::size_t num_partitions) const
            {
                if (subsets.size() < num_partitions) {
                    return max_sum - std::min(min_sum, (std::size_t) 0);
                }
                return max_sum - min_sum;
            }

            partitioning_t(std::size_t num_partitions)
                : subsets(), min_sum(0), max_sum(0)
            {
                subsets.reserve(num_partitions);
            }

            partitioning_t(std::size_t num_partitions, auto first_subset) :
                subsets(),
                min_sum(std::min((score_t) 0, first_subset.sum)),
                max_sum(std::max((score_t) 0, first_subset.sum))
            {
                subsets.reserve(num_partitions);
                subsets.push_back(first_subset);
            }

            void merge_parts(
                partitioning_t &other,
                std::size_t num_partitions,
                auto cmp_function)
            {
                // simulate balanced partitioning, might be suboptimal!
                auto subset_cmp_f = [](subset_t &l, subset_t &r) {
                    if (l.elements.size() != r.elements.size()) {
                        return l.elements.size() > r.elements.size();
                    } else {
                        return l.sum < r.sum;
                    }
                };

                std::sort(subsets.begin(), subsets.end(), subset_cmp_f);
                std::sort(
                    other.subsets.begin(),
                    other.subsets.end(),
                    subset_cmp_f);

                std::size_t ossize = other.subsets.size();
                std::size_t tssize = subsets.size();
                std::size_t end =
                    std::min(num_partitions, subsets.size() + ossize);
                std::size_t start = end - ossize;
                std::size_t border = std::min(end, tssize);

                assert(other.subsets.size() <= end);
                assert(other.subsets.size() >= end-start);
                assert(subsets.size() >= border);

                subsets.resize(end);

                min_sum = std::numeric_limits<score_t>::max();
                max_sum = std::numeric_limits<score_t>::min();

                for (std::size_t i = 0; i < start; i++) {
                    min_sum = std::min(min_sum, subsets[i].sum);
                    max_sum = std::max(max_sum, subsets[i].sum);
                }

                for (std::size_t i = start; i < border; i++) {
                    std::size_t other_i = end - i - 1;
                    subsets[i].merge_subsets(
                        other.subsets[other_i], cmp_function);
                    min_sum = std::min(min_sum, subsets[i].sum);
                    max_sum = std::max(max_sum, subsets[i].sum);
                }

                for (std::size_t i = border; i < end; i++) {
                    std::size_t other_i = end - i - 1;
                    subsets[i] = other.subsets[other_i];
                    min_sum = std::min(min_sum, subsets[i].sum);
                    max_sum = std::max(max_sum, subsets[i].sum);
                }
            }

            void sort(auto sort_func)
            {
                auto cmp_f =
                    [sort_func](const subset_t &l, const subset_t &r)
                {
                    return sort_func(l.sum, r.sum);
                };

                std::sort(subsets.begin(), subsets.end(), cmp_f);
            }

            void final_sort(auto sort_func)
            {
                auto cmp_f =
                    [sort_func](const subset_t &l, const subset_t &r)
                {
                    if (l.elements.size() != r.elements.size()) {
                        return l.elements.size() < r.elements.size();
                    } else {
                        return sort_func(l.sum, r.sum);
                    }
                };

                std::sort(subsets.begin(), subsets.end(), cmp_f);
            }

            friend std::ostream &operator«(
                std::ostream &os, const partitioning_t &p)
            {
                os « "(";
                bool first = true;
                for (auto s : p.subsets) {
                    if (!first) { os « "; "; }
                    os « s;
                    first = false;
                }
                // warning: capacity should be correct,
                // but num_partitions is not available
                os « ") [D: "
                    « p.difference(p.subsets.capacity()) « "?]";
                return os;
            }
        };

        std::list<partitioning_t> dataset;

        std::size_t num_partitions;

        sort_func_t sort_func;

        multiway_partitioning(
            std::size_t num_partitions,
            auto sort_func = std::greater<score_t>()
        ) :
            dataset(),
            num_partitions(num_partitions),
            sort_func(sort_func)
        {
            assert(1 < num_partitions);
        }

        void reset(std::size_t n_partitions) {
            num_partitions = n_partitions;
            dataset.clear();
        }

        void add(auto score, auto data) {
            dataset.push_back(
                partitioning_t(num_partitions, subset_t({score, data})));
        }

        void sort()
        {
            auto cmp_f =
                [this](const partitioning_t &l, const partitioning_t &r)
            {
                return sort_func(
                    l.difference(num_partitions),
                    r.difference(num_partitions));
            };

            // std::sort(dataset.begin(), dataset.end(), cmp_f);
            dataset.sort(cmp_f);
        }

        bool is_finished() {
            return 1 == dataset.size();
        }

        bool iterate()
        {
            if (1 < dataset.size()) {
                partitioning_t s1 = dataset.front();
                dataset.pop_front();
                partitioning_t s2 = dataset.front();
                dataset.pop_front();

                auto cmp_entry_f =
                    [this](const entry_t &l, const entry_t &r)
                {
                    return sort_func(l.score, r.score);
                };

                s1.merge_parts(s2, num_partitions, cmp_entry_f);
                std::list<partitioning_t> nel = { s1 };

                auto cmp_f =
                [this](const partitioning_t &l, const partitioning_t &r)
                {
                    return sort_func(
                        l.difference(num_partitions),
                        r.difference(num_partitions));
                };
                dataset.merge(nel, cmp_f);
            }

            return is_finished();
        }

        void partitionate() {
            while (false == iterate()) {}
        }

        auto result() {
            assert(is_finished());
            return dataset.front();
        }

        friend std::ostream &operator«(
            std::ostream &os, const multiway_partitioning &mp)
        {
            os « "multiway_partitioning for "
                « mp.num_partitions « " partitions: ";

            os « "{\n";
            for (auto p : mp.dataset) {
                os « "\t" « p « "\n";
            }
            os « "}" « std::endl;

            return os;
        }
};

void test_multiway_partitioning()
{
    const bool print_steps_in_between = false;

    // config for first example
    std::size_t num_elements = 11;
    std::size_t num_partitions = 3;

    // followed by iterations-1 random tests
    std::size_t iterations = 1;

    // random test configuration
    std::mt19937 prng(42);
    std::uniform_int_distribution<std::size_t> udistr(1, 9);
    std::uniform_int_distribution<std::size_t> nedistr(10, 100);
    std::uniform_int_distribution<std::size_t> npdistr(2, 8);

    for (std::size_t it = 0; it < iterations; it++) {
        if (0 < it) {
            num_elements = nedistr(prng);
            num_partitions = npdistr(prng);
        }

        std::cout « "sorting: "
            « num_elements « " elements into "
            « num_partitions « " partitions" « std::endl;
        using sort_t = std::greater<std::size_t>;
        // using sort_t = std::less<std::size_t>;
        multiway_partitioning<std::size_t, std::size_t, sort_t>
            mp(num_partitions, sort_t());

        for (std::size_t i = 0; i < num_elements; i++) {
            std::size_t s = udistr(prng);
            mp.add(s, i);
        }

        mp.sort();

        std::cout « mp;

        if constexpr (print_steps_in_between) {
            bool is_finished;
            do {
                is_finished = mp.iterate();
                std::cout « mp;
            } while (!is_finished);
            // std::cout « mp;
        } else {
            mp.partitionate();
        }

        auto result = mp.result();
        // result.sort(sort_t());

        result.final_sort(sort_t());
        std::cout « result « std::endl;

        std::size_t min_length = std::numeric_limits<std::size_t>::max();
        std::size_t max_length = std::numeric_limits<std::size_t>::min();
        for (const auto &sset : result.subsets) {
            std::size_t length = sset.elements.size();
            max_length = std::max(max_length, length);
            min_length = std::min(min_length, length);
        }

        std::cout « "lengths [" « min_length « ", "
```

```
            « max_length « "]" « std::endl;
        assert(max_length >= min_length);
        assert(1 >= max_length - min_length);

        // result.improve_partitioning();
    }
}

#endif
```

7.6 on_demand_scheduler.hpp

```
#ifndef ON_DEMAND_SCHEDULER
#define ON_DEMAND_SCHEDULER

#include <cstddef>
#include <cassert>
#include <deque>
#include <memory>
#include <chrono>

#include <mpi.h>

#include "../util/mpi/mpi_util.hpp"
#include "../util/mpi/mpi_async_communication.hpp"

#include "lazy_round_robin_scheduler.hpp"


template<
    typename remote_execution_context_t,
    class type_abstraction_adapter_t =
        remote_execution_context_t::type_abstraction_adapter_t>
requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
class on_demand_scheduler
{
    public:
        using rpc_message_t = remote_execution_context_t::rpc_message_t;
        using buffer_t = type_abstraction_adapter_t::abstract_type;

    private:
        static constexpr const std::size_t workload_capacity = 2;

        using scheduler_t = lazy_round_robin_scheduler<workload_capacity>;

        using uptr_scheduler_t = std::unique_ptr<scheduler_t>;

        uptr_scheduler_t scheduler;

        std::deque<rpc_message_t> tasks;

        type_abstraction_adapter_t taa;

        std::size_t num_waiting_for_reply;

        async_comm_out background_comm;
        // fake_async_comm_out background_comm;



        void process_msg(MPI_Status &status)
        {
            assert(0 == env_comm_id);

            if (0 < get_message_size(&status)) {
                // std::cout « "process_msg" « std::endl;
                execute(get_msg(status));
            } else {
                assert(MPI_SUCCESS == MPI_Recv(
                    nullptr,
                    0,
                    MPI_BYTE,
                    status.MPI_SOURCE,
                    mpi_tag,
                    env_comm,
                    &status));
            }

            assert(0 < status.MPI_SOURCE);
            scheduler->task_finished(status.MPI_SOURCE - 1);
            num_waiting_for_reply--;
        }


        rpc_message_t get_msg(MPI_Status &status)
        {
            buffer_t b;
            get_message(env_comm, &status, b);
            // print_vector("get_msg:", b);
            rpc_message_t msg;
            assert(0 < b.size());
            assert(taa.deserialize(b, msg));

            // std::cout « "c msg from: " « status.MPI_SOURCE
                // « " with tag " « status.MPI_TAG « std::endl;
            return msg;
        }


        rpc_message_t wait_for_msg()
        {
            assert(0 != env_comm_id);
            MPI_Status status;
            wait_for_message(env_comm, &status, mpi_tag);
            // std::cout « "wait_for_msg" « std::endl;
            return get_msg(status);
        }


        void execute(auto msg)
        {
            // std::cout « "executing " « msg.function_name
                // « " (id: " « msg.id « ") with"
                // « (msg.answer_required ? "" : "out")
                // « " reply on " « env_comm_id
                // « " (crc: " « (int) msg.crc « ")" « std::endl;

            assert(msg.check_crc());

            rpc_message_t result = rec.execute_rpc(msg);
            assert(!result.answer_required);

            if (msg.answer_required) {
                result.compute_crc();
            }

            if (0 == env_comm_id) {
                if (msg.answer_required) {
                    rec.execute_rpc(result);
                }
            } else {
                if (msg.answer_required) {
                    buffer_t b = taa.serialize(result);
                    background_comm.send_message(
                        env_comm, 0, mpi_tag, std::move(b));
                    // print_vector("execute_send_reply:", b);
                } else {
                    ping(env_comm, 0, mpi_tag);
                }
            }
        }


        void feed_hungry_workers()
        {
            assert(0 == env_comm_id);

            while (scheduler->is_process_without_work())
            {
                if (tasks.empty()) { return; }

                rpc_message_t task = tasks.front();
                tasks.pop_front();
                schedule(task);
            }
        }


    public:
        MPI_Comm env_comm;

        const int env_comm_id;

        const int env_comm_size;

        const int mpi_tag;


        remote_execution_context_t &rec;



        on_demand_scheduler(
            MPI_Comm env_comm,
            int mpi_tag,
            remote_execution_context_t &rec
        ) :
            scheduler(nullptr),
            tasks(),
            taa(),
            num_waiting_for_reply(0),
            background_comm(),
            env_comm(env_comm),
            env_comm_id(mpi_get_comm_rank(env_comm)),
            env_comm_size(mpi_get_comm_size(env_comm)),
            mpi_tag(mpi_tag),
            rec(rec)
        {
            if (0 == env_comm_id) {
                scheduler =
                    uptr_scheduler_t(new scheduler_t(env_comm_size - 1));
            }
        }


        void schedule(auto task, int target = 0)
        {
            assert(0 == env_comm_id);

            int target_id = target;
            if (0 == target) {
                target_id = 1 + scheduler->get_id_to_schedule_task_on();
            }

            /*int crc = */ task.compute_crc();

            // std::cout « "ods scheduling " « task.function_name
                // « " (id: " « task.id « ") with"
                // « (task.answer_required ? "" : "out")
                // « " reply on " « target
                // « " (crc: " « crc « ")" « std::endl;

            if (0 == target_id) {
                tasks.push_back(task);
            } else {
                buffer_t b = taa.serialize(task);
                background_comm.send_message(
                    env_comm, target_id, mpi_tag, std::move(b));
                // print_vector("schedule:", b);
                num_waiting_for_reply++;
            }
        }


        std::size_t num_tasks_waiting() const {
            return tasks.size();
        }


        std::size_t num_tasks_pending() const {
            return num_waiting_for_reply;
        }


        bool is_work_outstanding() const {
            return ((0 < num_tasks_pending())
                || (0 < num_tasks_waiting()))
                || !background_comm.communication_is_done();
        }


        size_t do_work(bool wait_for_work = false)
        {
            if ((0 == env_comm_id) && (1 < env_comm_size))
            {
                feed_hungry_workers();

                MPI_Status status;
                while (is_message_available(env_comm, &status, mpi_tag))
                {
                    process_msg(status);
                    feed_hungry_workers();
                }

                while (scheduler->worker_available())
                {
                    if (tasks.empty()) { return 0; }

                    rpc_message_t task = tasks.front();
                    tasks.pop_front();
                    schedule(task);

                    if (is_message_available(env_comm, &status, mpi_tag))
                    {
                        process_msg(status);
                    }
                }

                background_comm.test_for_messages();

                // std::cout « "communications_in_queue: "
                    // « background_comm.communications_in_queue()
                    // « std::endl;

                if (wait_for_work) {
                    if (background_comm.communication_is_done()) {
                        if (!tasks.empty()) {
                            rpc_message_t task = tasks.front();
                            tasks.pop_front();
                            execute(task);
                        }
                    } else {
                        background_comm.wait_for_some_messages();
                    }
                }
            } else {
                MPI_Status status;
                while (is_message_available(env_comm, &status, mpi_tag))
                {
                    // std::cout « "message available" « std::endl;
                    tasks.push_back(get_msg(status));
                }

                background_comm.test_for_messages();

                /*
                if (tasks.empty()) {
                    if (wait_for_work && (1 < env_comm_size)) {
                        tasks.push_back(wait_for_msg());
                        background_comm.test_for_messages();
                    }
                }
                */

                if (wait_for_work && (1 < env_comm_size)) {
                    if (!background_comm.communication_is_done()) {
                        while (tasks.empty()) {
                            if (is_message_available(
                                env_comm, &status, mpi_tag))
                            {
                                tasks.push_back(get_msg(status));
                                break;
                            }

                            background_comm.test_for_one_message();
                            if (background_comm.communication_is_done())
                            {
                                break;
                            }
                        }
                    }

                    if (tasks.empty()) {
                        tasks.push_back(wait_for_msg());
                    }
                }

                if (!tasks.empty()) {
                    auto start =
                        std::chrono::high_resolution_clock::now();

                    execute(tasks.front());

                    auto end =
                        std::chrono::high_resolution_clock::now();
                    tasks.pop_front();

                    return std::chrono::duration_cast<
                        std::chrono::milliseconds>(end - start).count();
                }
            }

            return 0;
        }
```

```
};

#endif

```

7.7 remote_computation_group.hpp

```
#ifndef REMOTE_COMPUTATION_GROUP
#define REMOTE_COMPUTATION_GROUP

#include <cstddef>
#include <cassert>
#include <deque>

#include <mpi.h>

#include "../util/mpi/mpi_util.hpp"
#include "../util/mpi/mpi_async_communication.hpp"


template<
    typename remote_execution_context_t,
    class type_abstraction_adapter_t
        = remote_execution_context_t::type_abstraction_adapter_t>
requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
class remote_computation_group
{
    public:
        using rpc_message_t = remote_execution_context_t::rpc_message_t;

        using buffer_t = type_abstraction_adapter_t::abstract_type;

    private:

        std::deque<std::pair<int, rpc_message_t» tasks;

        type_abstraction_adapter_t taa;

        std::size_t num_waiting_for_reply;

        async_comm_out background_comm;
        // fake_async_comm_out background_comm;


        std::pair<int, rpc_message_t> get_msg(MPI_Status &status)
        {
            buffer_t b;
            get_message(env_comm, &status, b);
            rpc_message_t msg;
            assert(0 < b.size());
            assert(taa.deserialize(b, msg));

            assert(msg.check_crc());
            return std::make_pair(status.MPI_SOURCE, msg);
        }


        std::pair<int, rpc_message_t> wait_for_msg()
        {
            assert(0 != env_comm_id);
            MPI_Status status;
            while (1) {
                wait_for_message(env_comm, &status, mpi_tag);
                if (0 == get_message_size(&status)) {
                    num_waiting_for_reply--;
                } else {
                    break;
                }
            }

            // std::cout « "wait_for_msg" « std::endl;
            return get_msg(status);
        }


        void execute(const std::pair<int, rpc_message_t> &task)
        {
            const int origin = task.first;
            rpc_message_t msg = task.second;
            // std::cout « "executing " « msg.function_name
                // « " (id: " « msg.id « ") with"
                // « (msg.answer_required ? "" : "out")
                // « " reply and arg size " « msg.data.size()
                // « " on " « env_comm_id « std::endl;

            assert(msg.check_crc());

            rpc_message_t result = rec.execute_rpc(msg);
            assert(!result.answer_required);

            if (msg.answer_required) {
                result.compute_crc();
            }

            if (origin == env_comm_id) {
                if (msg.answer_required) {
                    rec.execute_rpc(result);
                }
            } else {
                if (msg.answer_required) {
                    buffer_t b = taa.serialize(result);
                    background_comm.send_message(
                        env_comm, origin, mpi_tag, std::move(b));
                    // print_vector("execute_send_reply:", b);
                } else {
                    // std::cout « "ping " « origin « std::endl;
                    ping(env_comm, origin, mpi_tag);
                }
            }
        }


    public:
        MPI_Comm env_comm;

        const int env_comm_id;

        const int env_comm_size;

        const int mpi_tag;


        remote_execution_context_t &rec;



        remote_computation_group(
            MPI_Comm env_comm,
            int mpi_tag,
            remote_execution_context_t &rec
        ) :
            tasks(),
            taa(),
            num_waiting_for_reply(0),
            background_comm(),
            env_comm(env_comm),
            env_comm_id(mpi_get_comm_rank(env_comm)),
            env_comm_size(mpi_get_comm_size(env_comm)),
            mpi_tag(mpi_tag),
            rec(rec)
        {}


        void schedule(auto task, int target_id)
        {
            // std::cout « "scheduling " « task.function_name
                // « " (id: " « task.id « ") with"
                // « (task.answer_required ? "" : "out")
                // « " reply and arg size " « task.data.size()
                // « " on " « target_id « " by "
                // « env_comm_id « std::endl;

            task.compute_crc();

            if (env_comm_id == target_id) {
                tasks.push_back(std::make_pair(target_id, task));
            } else {
                buffer_t b = taa.serialize(task);
                // int bcrc = crc8((uint8_t *) b.data(), b.size());
                // std::cout « "CCC schedule bcrc: " « bcrc « std::endl;
                // print_vector("CCC get_msg:", b);
                background_comm.send_message(
                    env_comm, target_id, mpi_tag, std::move(b));
                num_waiting_for_reply++;
                // std::cout « "num_waiting_for_reply increased to: "
                    // « num_waiting_for_reply « std::endl;
            }
        }

        std::size_t num_tasks_waiting() const {
            return tasks.size();
        }


        std::size_t num_tasks_pending() const {
            return num_waiting_for_reply;
        }


        bool is_work_outstanding() const {
            return ((0 < num_tasks_pending())
                || (0 < num_tasks_waiting()))
                || !background_comm.communication_is_done();
        }


        void do_work(bool wait_for_work = false)
        {
            MPI_Status status;
            while (is_message_available(env_comm, &status, mpi_tag)) {
                if (0 == get_message_size(&status)) {
                    accept_ping(env_comm, &status);
                    num_waiting_for_reply--;
                } else {
                    tasks.push_back(get_msg(status));
                }
            }

            background_comm.test_for_messages();

            if (wait_for_work && (1 < env_comm_size)) {
                if (!background_comm.communication_is_done()) {
                    while (tasks.empty()) {
                        if (is_message_available(
                            env_comm, &status, mpi_tag))
                        {
                            if (0 == get_message_size(&status)) {
                                accept_ping(env_comm, &status);
                                num_waiting_for_reply--;
                            } else {
                                tasks.push_back(get_msg(status));
                                break;
                            }
                        }

                        background_comm.test_for_one_message();
                        if (background_comm.communication_is_done()) {
                            break;
                        }
                    }
                }

                if (tasks.empty()) {
                    tasks.push_back(wait_for_msg());
                }
            }

            if (!tasks.empty()) {
                std::pair<int, rpc_message_t> p = tasks.front();
                execute(p);
                tasks.pop_front();
            }
        }

};

#endif
```

7.8 remote_execution_context.hpp

```
#ifndef REMOTE_EXECUTION_CONTEXT_HPP
#define REMOTE_EXECUTION_CONTEXT_HPP

#include "rpc.hpp"
#include "../util/uuid.hpp"

template<typename taa_t> requires (taa_t::is_TypeAbstractionAdapter)
class remote_execution_context
{
    public:
        using type_abstraction_adapter_t = taa_t;

    private:
        using uuid_t =
            uuid<remote_execution_context<type_abstraction_adapter_t>>;

    public:
        using data_t = type_abstraction_adapter_t::abstract_type;

        using rpc_message_t = rpc_message<data_t>;

        using remote_procedure_manager_t =
            remote_procedure_manager<type_abstraction_adapter_t>;

        remote_procedure_manager_t &rpmanager;


        remote_execution_context(remote_procedure_manager_t &rpmanager) :
            rpmanager(rpmanager) {}


        template<typename function_t>
        rpc_message_t on_result(auto msg, function_t function)
        {
            assert(0 == msg.id);

            std::size_t id = 1 + uuid_t::get();
            assert(0 != id);

            std::string rfname = msg.function_name
                + FCT_NAME_RESULT_TAG + "_" + std::to_string(id);
            rpmanager.add_function(rfname, function);

            msg.id = id;
            msg.answer_required = true;

            return msg;
        }


        auto execute_rpc(rpc_message_t &msg)
        {
            bool is_result =
                (false == msg.answer_required) && (0 != msg.id);

            if (is_result) {
                msg.function_name =
                    msg.function_name + "_" + std::to_string(msg.id);
            }

            rpc_message_t result_msg = rpmanager.execute_rpc(msg);

            if (is_result) {
                assert(1 == rpmanager.function_map.erase(msg.function_name));
                uuid_t::free(msg.id - 1);
                result_msg.id = 0;
                result_msg.function_name = "";
            }

            return result_msg;
        }
};


#endif
```

7.9 rescheduling.hpp

```
#ifndef RESCHEDULING_HPP
#define RESCHEDULING_HPP

#include <cstddef>
#include <functional>
#include <vector>
#include <cassert>
#include <iostream>

#include "../util/util.hpp"
#include "multiway_partitioning.hpp"

class rescheduling_manager
{
    private:
        bool rescheduling_required;

        int num_workgroups;

        std::size_t number_of_active_mcs;

        std::function<bool(std::size_t, std::size_t)> sort_func;

        multiway_partitioning<
            std::pair<std::size_t, int>,
            std::size_t,
            std::function<bool(std::size_t, std::size_t)» mwpart;

        std::vector<std::size_t> workload;

    public:
        rescheduling_manager(
            int num_workgroups,
            std::size_t num_mcs,
            bool use_mc_limit,
            bool rescheduling_required = false
        ) :
            rescheduling_required(rescheduling_required),
            num_workgroups(num_workgroups),
            number_of_active_mcs(num_mcs),
            sort_func(std::less<std::size_t>()),
            mwpart(num_workgroups, sort_func),
            workload(num_workgroups)
        { assert(0 < num_workgroups); ignore(use_mc_limit); }


        void start_rescheduling()
        {
            assert(!rescheduling_required);

            mwpart.reset(num_workgroups);

            workload.reserve(num_workgroups);
            for (int i = 0; i < num_workgroups; i++) {
                workload[i] = 0;
            }

            rescheduling_required = true;
        }


        bool is_rescheduling_in_progress() const
        {
            return rescheduling_required;
        }


        void add_progress_for_id(
            std::size_t id, std::size_t progress, int wg_id)
        {
            assert(rescheduling_required);
            assert(0 <= wg_id);

            std::cout « "adding reschedule progress " « progress
                « " for " « id « " on " « wg_id « std::endl;

            // progress_list.insert(id, std::make_pair(progress, wg_id));
            mwpart.add(progress, std::make_pair(id, wg_id));

            workload[wg_id] += progress;
        }


        void remove_mc(std::size_t id)
        {
            assert(rescheduling_required);

            ignore(id);
            if (!rescheduling_required) {
                number_of_active_mcs--;
            }
        }

        bool information_complete() const
        {
            assert(rescheduling_required);
            return mwpart.dataset.size() >= number_of_active_mcs;
        }

        bool is_rescheduling_unnecessary(
            std::size_t reschedule_requires_absolute_difference) const
        {
            assert(rescheduling_required);
            assert(information_complete());

            auto [min, max] =
                std::minmax_element(begin(workload), end(workload));

            std::size_t diff = max - min;
            bool got_required_abs_diff =
                diff >= reschedule_requires_absolute_difference;

            return !got_required_abs_diff;
        }

        void cancel_rescheduling(auto do_rescheduling)
        {
            // assert(rescheduling_required);
            // assert(information_complete());
            rescheduling_required = false;
            for (int target_wg = 0;
                target_wg < num_workgroups; target_wg++)
            {
                do_rescheduling(0, target_wg, target_wg);
            }
        }

        void reschedule(auto do_rescheduling)
        {
            assert(rescheduling_required);
            // assert(progress_list.size() == number_of_active_mcs);
            assert(mwpart.dataset.size() == number_of_active_mcs);

            // std::cout « "before rescheduling:\n" « mwpart;

            mwpart.sort();
            mwpart.partitionate();
            auto result = mwpart.result();
            // result.sort(sort_t());
            // todo: improve? (i.e. with complete Karmarkar-Karp algorithm)
            // an anytime algorithm would be possible as well
            result.final_sort(sort_func);

            // todo: find optimal mapping

            // std::cout « "after rescheduling:\n" « mwpart;

            rescheduling_required = false;

            const auto [min, max] =
                std::minmax_element(begin(workload), end(workload));
            std::size_t original_diff = *max - *min;

            // std::cout « "rescheduling from diff: "
                // « original_diff « " [" « *min « ", " « *max « "] "
                // « " to " « result.difference(num_workgroups)
                // « " [" « result.min_sum « ", "
                // « result.max_sum « "]" « std::endl;

            if (result.difference(num_workgroups) >= original_diff) {
                // std::cout « "rescheduling probably unnecessary!" « std::endl;
                cancel_rescheduling(do_rescheduling);
                return;
            }

            assert(result.subsets.size() == (std::size_t) num_workgroups);
            auto it = result.subsets.begin();
            for (int target_wg = 0;
                target_wg < num_workgroups; target_wg++)
            {
                std::size_t mcs_to_get = 0;
                for (auto entry : it->elements) {
                    if (entry.data.second != target_wg) {
                        std::size_t id = entry.data.first;
                        int original_wg = entry.data.second;
                        // std::cout « "move mc " « id
                            // « " with progress " « entry.score
                            // « " from workgroup " « original_wg
                            // « " to " « target_wg « std::endl;
                        do_rescheduling(id, original_wg, target_wg);
                        mcs_to_get++;
                    }
                }

                do_rescheduling(mcs_to_get, target_wg, target_wg);
                it++;
            }
            assert(it == result.subsets.end());
        }
};

```

```

#endif
```

7.10 rpc.hpp

```
#ifndef RPC_HPP
#define RPC_HPP

#include <iostream>
#include <limits>
#include <functional>
#include <type_traits>
#include <map>
#include <memory>

#include "../util/util.hpp"

#include "../util/basic_serialization.hpp"

#include "type_abstraction.hpp"
#include "rtti.hpp"
#include "function_abstraction.hpp"


const std::string FCT_NAME_RESULT_TAG = "__result";


template<typename data_t = std::vector<uint8_t»
class rpc_message
{
    public:
        using data_type = data_t;

        std::size_t id;

        std::string function_name;

        bool answer_required;

        data_t data;

        uint8_t crc;


        rpc_message() :
            id(std::numeric_limits<std::size_t>::max()),
            function_name(),
            answer_required(false),
            data(),
            crc(0)
        {}

        rpc_message(const rpc_message &other) :
            id(other.id),
            function_name(other.function_name),
            answer_required(other.answer_required),
            data(other.data),
            crc(other.crc)
        {}

        friend void swap(rpc_message &l, rpc_message &r)
        {
            std::swap(l.id, r.id);
            std::swap(l.function_name, r.function_name);
            std::swap(l.answer_required, r.answer_required);
            std::swap(l.data, r.data);
            std::swap(l.crc, r.crc);
        }


        rpc_message &operator=(const rpc_message &other) {
            id = other.id;
            function_name = other.function_name;
            answer_required = other.answer_required;
            data = other.data;
            crc = other.crc;
            return *this;
        }

/*
        rpc_message &operator=(const rpc_message &other) {
            swap(*this, other);
            return *this;
        }
*/

        rpc_message(rpc_message &&other) {
            swap(*this, other);
        }


    private:
        uint8_t compute_crc_() const
        {
            decltype(crc) crc_;
            crc_ = crc8((uint8_t *) &id, sizeof(decltype(id)));
            crc_ = crc8((uint8_t *) function_name.c_str(),
                function_name.length(), crc_);
            crc_ = crc8((uint8_t *) &answer_required, 1, crc_);
            crc_ = crc8((uint8_t *) data.data(), data.size(), crc_);
            return crc_;
        }

    public:
        // optional crc functionality

        uint8_t compute_crc() {
            crc = compute_crc_();
            return crc;
        }

        bool check_crc() const {
            return (compute_crc_() == crc);
        }
};


template<typename S>
void serialize(S &s, rpc_message<std::vector<uint8_t» &v) {
    std::size_t data_len = v.data.size();
    generic_serialize(s,
        v.id, v.function_name, v.answer_required, data_len);
    v.data.resize(data_len);
    s.container1b(v.data, data_len);
    assert(v.data.size() == data_len);
    generic_serialize(s, v.crc);
}


template<typename type_abstraction_adapter_t>
    requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
class remote_procedure_manager
{
    private:

        type_abstraction_adapter_t type_abstraction;

        using abstract_function_t =
            AbstractFunction<type_abstraction_adapter_t>;

    public:

        using rpc_message_t =
            rpc_message<typename type_abstraction_adapter_t::abstract_type>;

        std::map<std::string, std::unique_ptr<abstract_function_t»
            function_map;


        template<typename function_t>
        void add_function(std::string name, function_t function)
        {
            assert(!name.empty());

            AbstractFunction<type_abstraction_adapter_t> *afptr
                = getAbstractFunctionFor<
                    type_abstraction_adapter_t,
                    function_t> (function);
            assert(afptr);

            auto result = function_map.try_emplace(
                name,
                std::unique_ptr<abstract_function_t>(afptr)
            );
            assert(result.second &&
                "there is already a function with this name registered!");
        }

        template<typename... param_types>
        auto prepare_call(std::string name, param_types... params)
        {
            assert(!name.empty());

            rpc_message_t msg;
            msg.id = 0;
            msg.function_name = name;
            msg.answer_required = false;

            ConcreteType<type_abstraction_adapter_t, param_types...>
                param_container = {params..., true};

            msg.data = param_container.serialize(type_abstraction);
            return msg;
        }

        auto execute_rpc(const rpc_message_t &msg)
        {
            // std::cout « "execute_rpc: " «
                // msg.function_name « std::endl;

            assert(!msg.function_name.empty());

            auto fsearch = function_map.find(msg.function_name);
            if (function_map.end() == fsearch) {
                std::cerr « "Unknown remote procedure call: "
                    « msg.function_name « std::endl;
                assert(false && "Unknown remote procedure call!");
            } else {
                // std::cout « "rpc executing: "
                    // « msg.function_name « std::endl;
            }

            rpc_message_t result_msg;
            result_msg.id = msg.id;
            result_msg.function_name =
                msg.function_name + FCT_NAME_RESULT_TAG;
            result_msg.answer_required = false;

            abstract_function_t *fptr = (fsearch->second).get();

            try {
                result_msg.data =
                    fptr->execute(type_abstraction, msg.data);
            } catch (...) {
                std::cout « "exception executing function: "
                    « result_msg.function_name « std::endl;
                abort();
            }

            return result_msg;
        }
};


#endif
```

7.11 rtti.hpp

```
#ifndef RTTI_HPP
#define RTTI_HPP

#include <tuple>

#include "../util/util.hpp"


template<typename type_abstraction_adapter_t> requires
    (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
class GenericType
{
    public:
        virtual type_abstraction_adapter_t::abstract_type serialize
            (type_abstraction_adapter_t &adapter) const
        {
            ignore(adapter);
            return typename type_abstraction_adapter_t::abstract_type();
        }

        virtual bool deserialize(
            type_abstraction_adapter_t &adapter,
            const type_abstraction_adapter_t::abstract_type &v)
        {
            ignore(adapter, v);
            return false;
        }

        virtual ~GenericType() {}
};

template<typename type_abstraction_adapter_t, typename... arg_types>
    requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
class ConcreteType : public GenericType<type_abstraction_adapter_t>
{
    public:
        using tuple_t = std::tuple<arg_types ...>;

        tuple_t values;

        ConcreteType() : values() {}

        ConcreteType(
            arg_types... args,
            bool use_value_constructor = true
        ) :
            values(args...)
        {
            ignore(use_value_constructor);
        }

        ConcreteType(std::tuple<arg_types ...> &values) : values(values) {}

        ConcreteType(const ConcreteType &other) : values(other.values) {}

        template<typename other_type_abstraction_adapter_t>
        operator ConcreteType<
            other_type_abstraction_adapter_t, arg_types...>() const
        {
            return ConcreteType<
                other_type_abstraction_adapter_t, arg_types...>(values);
        }

        virtual type_abstraction_adapter_t::abstract_type
            serialize(type_abstraction_adapter_t &adapter) const override
        {
            return adapter.serialize(values);
        }

        virtual bool deserialize(
            type_abstraction_adapter_t &adapter,
            const type_abstraction_adapter_t::abstract_type &v) override
        {
            return adapter.deserialize(v, values);
        }
};


#endif


```

7.12 type_abstraction.hpp

```
#ifndef TYPE_ABSTRACTION_HPP
#define TYPE_ABSTRACTION_HPP

#include <any>

#include "../util/include_bitsery.hpp"

class Any_TypeAbstractionAdapter
{
    public:

        static const bool is_TypeAbstractionAdapter = true;

        using abstract_type = std::any;

        template<typename t>
        static abstract_type serialize(t &v) {
            return v;
        }

        template<typename t>
        static bool deserialize(const abstract_type &v, t &result) {
            try {
                result = std::any_cast<t>(v);
                return true;
            } catch (...) {
                return false;
            }
        }
};

template<
    typename buffer_t,
    typename input_adapter_t,
    typename output_adapter_t
>
class Bitsery_TypeAbstractionAdapter
{
    public:

        static const bool is_TypeAbstractionAdapter = true;

        using abstract_type = buffer_t;

        template<typename t>
        static abstract_type serialize(t &v)
        {
            buffer_t buffer;
            auto length = bitsery::quickSerialization(
                std::move(output_adapter_t{buffer}), v);
            buffer.resize(length);
            return buffer;
        }

        template<typename t>
        static bool deserialize(const abstract_type &buffer, t &result)
        {
            auto state = bitsery::quickDeserialization(
                std::move(input_adapter_t{buffer.begin(), buffer.size()}),
                result);

            if ((state.first != bitsery::ReaderError::NoError)
                || (false == state.second))
            {
                return false;
            }

            return true;
        }
};

#endif

```

7.13 main.cpp

```

// Due to debugging at the end of the development phase, this file
// was not yet cleaned up and well documented (work in progress state)
// There was a bug after rescheduling when the Markov chains were transferred

// mpirun -np 6 --use-hwthread-cpus --oversubscribe main
// seq 1 6 |parallel -I{} -j 1 "bash -c 'mpirun -np {} main > {}.txt'"
// seq 7 12 |parallel -I{} -j 1 "bash -c 'mpirun -np {} --use-hwthread-cpus --oversubscribe main > {}.txt'"
// for n in $(seq 9); do mv $n.txt 0$n.txt; done;
// grep -m 1 duration *.txt
// grep -m 1 duration *.txt |cut -d " " -f 3 |gnuplot -p -e 'plot "/dev/stdin" with lines'
// grep -m 1 duration *.txt |awk '{print $i*$3; $i++}' |gnuplot -p -e 'plot "/dev/stdin" with lines'

// grep -m 1 duration *.txt |awk '{print '$(grep duration 01.txt|cut -d " " -f 3)'/($3/$i) - 1; $i++}' |gnuplot -p -e 'plot "/dev/stdin" with lines'
// grep -m 1 duration *.txt |awk '{print '$(grep duration 01.txt|cut -d " " -f 3)'/$3; $i++}' |gnuplot -p -e 'plot "/dev/stdin" with lines'

// echo ... |sed "s/ / + /g" |bc
// echo ... |sed "s/ /\n/g" |awk '{sum+=$1; sumsq+=$1*$1}END{print NR; print sum/NR; print sqrt(sumsq/(NR-1) - (sum/NR)**2)}'
// echo ... |sed "s/ /\n/g" |sort -nk1 |gnuplot -p -e 'set logscale y 2.718281828459045; plot "/dev/stdin" with lines'
// echo ... |sed "s/ /\n/g" |awk '{print log($1)}' |sort -g |gnuplot -p -e 'plot "/dev/stdin" with lines'

// grep "c value" out-1000.txt |cut -d " " -f 3 |gnuplot -p -e 'plot "/dev/stdin" with lines'
// grep "accepted" out-1000.txt |sed "s/)//g" |cut -d "(" -f 2 |gnuplot -p -e 'plot "/dev/stdin" with lines'

```

```
#include <set>
#include <queue>

#include "config.hpp"

std::size_t num_reference_chains = 1;

std::size_t num_markov_chains = 4;

std::size_t markov_chain_offset = 0;

std::size_t markov_chain_target_length = 4;

std::size_t reschedule_after_seconds = 0;

std::size_t reschedule_requires_absolute_difference = 0;

std::size_t max_runtime_in_seconds = 0;

bool checkpoint_on_termination = true;

std::size_t checkpoint_after_iterations = 0;

std::size_t checkpoint_after_seconds = 0;

uint8_t mcmc_num_changes = 3;

// prec_t mcmc_T = 0.05;
// prec_t mcmc_T = 0.005;
// 300 200 100 75 50 40 30 20 10 5

prec_t mcmc_T = 30;

// wenn neu - alt = mcmc_T => p(accept) = 1/e
// wenn neu - alt = 3*mcmc_T => p(accept) = 5 %


// #include "util/util.hpp"
// #include "util/mpi/mpi_util.hpp"
#include "util/mpi/mpi_shared_tmp_workgroup.hpp"
#include "util/mpi/mpi_types.hpp"
// #include "util/uuid.hpp"
// #include "util/bitfield.hpp"

const mpi_shared_tmp_dir_workgroup *workgroup_ptr = nullptr;

// #include "distributed_computation/rtti.hpp"
// #include "distributed_computation/type_abstraction.hpp"
// #include "distributed_computation/function_abstraction.hpp"
#include "distributed_computation/rpc.hpp"
#include "distributed_computation/remote_execution_context.hpp"
#include "distributed_computation/on_demand_scheduler.hpp"
#include "distributed_computation/remote_computation_group.hpp"

using buffer_t = std::vector<uint8_t>;
using output_adapter_t = bitsery::OutputBufferAdapter<buffer_t>;
using input_adapter_t = bitsery::InputBufferAdapter<buffer_t>;
using type_abstraction_adapter_t =
 Bitsery_TypeAbstractionAdapter<buffer_t, input_adapter_t, output_adapter_t>;

using remote_procedure_manager_t = remote_procedure_manager<type_abstraction_adapter_t>;
using remote_execution_context_t = remote_execution_context<type_abstraction_adapter_t>;
using rpc_message_t = remote_procedure_manager_t::rpc_message_t;

remote_procedure_manager_t wg_rpmanager;

struct wg_leader_exec_env {
    remote_procedure_manager_t rpmanager;
    remote_execution_context_t remote_exec_context;
    remote_computation_group<remote_execution_context_t> rcg;
    wg_leader_exec_env(MPI_Comm env_comm, int mpi_tag) :
        rpmanager(),
        remote_exec_context(rpmanager),
        rcg(env_comm, mpi_tag, remote_exec_context)
    {}
};

std::unique_ptr<wg_leader_exec_env> wg_leaders = nullptr;


#include "util/seed.hpp"

// typedef std::mt19937 prn_engine_t;
using seed_t = seed_type<prn_engine_t::state_size, uint32_t>;
using seed_type_generator_t = seed_type_generator<typename seed_t::base_t>;

seed_t global_seed;
```

```


#include "metropolis_hastings/configuration/configuration_vector.hpp"
#include "metropolis_hastings/configuration/configuration_generator.hpp"

constexpr const bool use_bitfield = false;
using cfg_vector_t = binary_configuration_vector<use_bitfield>;
// using cfg_vector_t = binary_configuration_vector;

std::size_t cfg_vector_length;

using cfg_generator_t = std::function<void(prn_engine_t &, cfg_vector_t &)>;

uniform_configuration_generator<cfg_vector_t> ugen;
cfg_generator_t default_cfg_generator = cfg_generator_t(ugen);

std::map<std::size_t, cfg_generator_t> mc_starting_with_special_cfg_vector;

#include "metropolis_hastings/configuration/change_generator.hpp"
constexpr const bool enforce_change = false;
constexpr const bool skip_unchanged_vectors = true;
using change_generator_t = simple_change_generator<cfg_vector_t, enforce_change>;

#include "metropolis_hastings/value_computation/utility/probsat-execution.hpp"
#include "metropolis_hastings/value_computation/modell.hpp"
#include "metropolis_hastings/value_computation/probsat-with-resolvents.hpp"

using probsat_modell_t = probsat_resolvents<cfg_vector_t, prec_t, use_bitfield>;
using computation_modell_t = probsat_modell_t;

#include "metropolis_hastings/mcmc.hpp"
using marcov_chain_state_t = marcov_chain_state<cfg_vector_t>;

std::map<std::size_t, marcov_chain_state_t> mc_chains_to_add;

#include "metropolis_hastings/acceptance_computation/simple_acceptance_computation.hpp"
#include "metropolis_hastings/acceptance_computation/statistical_acceptance_computation.hpp"
using acc_prob_comp_t = exponential_acceptance_probability<prec_t>;
// using acceptance_computation_t =
    // simple_acceptance_computation<prn_engine_t, cfg_vector_t, prec_t, probsat_modell_t, acc_prob_comp_t>;
using acceptance_computation_t =
    statistical_acceptance_computation<prn_engine_t, cfg_vector_t, prec_t, probsat_modell_t, acc_prob_comp_t>;

#include "metropolis_hastings/algorithm.hpp"
using metropolis_hastings_algorithm_t =
    metropolis_hastings_algorithm<
        marcov_chain_state_t,
        change_generator_t,
        acceptance_computation_t, computation_modell_t, acc_prob_comp_t,
        result_statistics<prec_t>,
        std::function<void(std::pair<uint64_t, probsat_return_cause::reason>)>,
        enforce_change,
        skip_unchanged_vectors
    >;


#include "distributed_computation/rescheduling.hpp"
std::unique_ptr<rescheduling_manager> reschedule_manager = nullptr;

std::map<std::size_t, int> marcov_chains_to_move = {};

std::set<std::size_t> marcov_chains_finished;

std::ptrdiff_t markov_chains_to_get = 0;
bool rescheduling_in_progress = false;


bool running = true;

void stop_running() {
    running = false;
}

// void test_rcg(int from, int to) {
    // std::cout « "test_rcg: hello from " « from « " to " « to « std::endl;
// }

void got_markov_chain(std::ptrdiff_t n = -1) {
    markov_chains_to_get += n;
    std::cout « "got_markov_chain + " « n « " = " « markov_chains_to_get
        « " on " « workgroup_ptr->workgroup_id « std::endl;
    if (0 == markov_chains_to_get) {
        assert(rescheduling_in_progress);
        rescheduling_in_progress = false;
        std::cout « "rescheduling finished on wg " « workgroup_ptr->workgroup_id « std::endl;
    }
}

/*
auto serialize_mcs(auto mc_state) {
    type_abstraction_adapter_t bitsery_taa;
    buffer_t b = bitsery_taa.serialize(mc_state);
    // std::cout « "serialize mcs(" « mc_state.iteration « ", ["
        // « mc_state.cfg_vector.data.size() « "], "
        // « mc_state.prn_engine « "}) on " « workgroup_ptr->workgroup_id « std::endl;

    std::size_t checksum = 0;
    for (auto v : b) { checksum += (v?1:0); checksum *= 3; }
    std::cout « "serialize: " « checksum « " on " « workgroup_ptr->workgroup_id « std::endl;
    // std::cout « "serialize:"; for (auto v : b) { std::cout « " " « (v?1:0); } std::cout «
 std::endl;
    return b;
}

void recieve_markov_chain(std::size_t id, buffer_t mcs) {
    ignore(id, mcs);
    std::cout « "TEST recieve" « std::endl;

    std::size_t checksum = 0;
    for (auto v : mcs) { checksum += (v?1:0); checksum *= 3; }
    std::cout « "recieve: " « checksum « " on " « workgroup_ptr->workgroup_id « std::endl;
    // std::cout « "serialize:"; for (auto v : mcs) { std::cout « " " « (v?1:0); } std::cout «
 std::endl;

    // type_abstraction_adapter_t bitsery_taa;
    // marcov_chain_state_t mc_state(cfg_vector_length, prn_engine_t());
    // assert(bitsery_taa.deserialize(mcs, mc_state));

    // std::cout « "recieve_markov_chain(" « id « ", {" « mc_state.iteration « ", ["
        // « mc_state.cfg_vector.data.size() « "], "
        // « mc_state.prn_engine « "}) on " « workgroup_ptr->workgroup_id « std::endl;

    got_markov_chain();
}
*/

/*
void recieve_markov_chain(const std::size_t &id, const marcov_chain_state_t &mc_state) {
    std::cout « "TEST recieve" « std::endl;

    std::cout « "recieve_markov_chain(" « id « ", {" « mc_state.iteration « ", ["
        « mc_state.cfg_vector.data.size() « ", " « mc_state.cfg_vector.count_ones() « "], "
        « mc_state.prn_engine « "}) on " « workgroup_ptr->workgroup_id « std::endl;

    got_markov_chain();
}
*/

void recieve_markov_chain(std::size_t id, const marcov_chain_state_t &mcs) {
    std::cout « "recieve_markov_chain(" « id « ", {" « mcs.iteration « ", ["
        « mcs.cfg_vector.data.size() « "], ...}) on " « workgroup_ptr->workgroup_id « std::endl;
    mc_chains_to_add.emplace(id, mcs);
    got_markov_chain();
}

void move_markov_chain(std::size_t id, int to) {
    std::cout « "move_markov_chain(" « id « ", " « to « ") on " « workgroup_ptr->workgroup_id «
 std::endl;
    if (workgroup_ptr->workgroup_id == to) {
        assert(0 == markov_chains_to_get);
        got_markov_chain(id);
    } else {
        std::cout « "XXX marcov_chains_to_move[" « id « "] = " « to « std::endl;
        marcov_chains_to_move[id] = to;
    }
}

bool send_progress_info = false;
void request_progress_info() {
    // std::cout « "request_progress_info() on " « workgroup_ptr->workgroup_id « std::endl;
    send_progress_info = true;
    assert(0 == markov_chains_to_get);
    markov_chains_to_get = 0;
    rescheduling_in_progress = true;
    // std::cout « "rescheduling started on wg " « workgroup_ptr->workgroup_id « std::endl;
}

void pass_progress_info(std::size_t id, std::size_t progress, int wg_id) {
    // std::cout « "request_progress_info(" « id « ", " « progress « ", " « wg_id
        // « ") on " « workgroup_ptr->workgroup_id « std::endl;
    assert(reschedule_manager);
    reschedule_manager->add_progress_for_id(id, progress, wg_id);
```

```
}

// TODO: rescheduling done

bool request_rescheduling()
{
    assert(reschedule_manager);
    assert(wg_leaders);

    if (reschedule_manager->is_rescheduling_in_progress()) {
        return false;
    }

    reschedule_manager->start_rescheduling();

    for (int i = 0; i < workgroup_ptr->workgroup_count; i++) {
        auto cmd = wg_leaders->rpmanager.prepare_call("request_progress_info");
        wg_leaders->rcg.schedule(cmd, i);
    }

    return true;
}


void add_new_mc_chain(std::size_t id, seed_t &mc_seed)
{
    auto mc_sseq = mc_seed.get_seed_seq();
    marcov_chain_state_t mc_state(cfg_vector_length, prn_engine_t(mc_sseq));

    auto search = mc_starting_with_special_cfg_vector.find(id);
    if (mc_starting_with_special_cfg_vector.end() != search) {
        (search->second)(mc_state.prn_engine, mc_state.cfg_vector);
    } else {
        default_cfg_generator(mc_state.prn_engine, mc_state.cfg_vector);
    }

    mc_chains_to_add.emplace(id, mc_state);

    {
        // type_abstraction_adapter_t bitsery_taa;
        // buffer_t b = bitsery_taa.serialize(mc_state);

        // marcov_chain_state_t mc_state_(cfg_vector_length, prn_engine_t());
        // assert(bitsery_taa.deserialize(b, mc_state_));
        // assert(mc_state == mc_state_);

        // reschedule_required = true;
        // auto id_fwidth = std::setw(std::to_string(num_markov_chains).length());
        // std::cout « "c id: " « id_fwidth « id « " with " « cfg_vector « std::endl;

        // type_abstraction_adapter_t bitsery_taa;
        // buffer_t b = bitsery_taa.serialize(cfg_vector);
        // cfg_vector_t cmp_cfg_vector(cfg_vector_length);
        // assert(bitsery_taa.deserialize(b, cmp_cfg_vector));
        // assert(cmp_cfg_vector == cfg_vector);
        // std::cout « "c id: " « id_fwidth « id « " with " « cmp_cfg_vector « std::endl;
    }

    {
        // auto prn1 = mc_state.prn_engine;
        // auto prn2 = mc_state.prn_engine;
        // simple_change_generator_t change_gen(mc_state.cfg_vector);

        // type_abstraction_adapter_t bitsery_taa;
        // buffer_t b = bitsery_taa.serialize(change_gen);
        // auto c1 = change_gen(prn1, mc_state.cfg_vector);
        // auto c2 = change_gen(prn1, mc_state.cfg_vector);
        // auto c3 = change_gen(prn1, mc_state.cfg_vector);
        // std::cout « c1.index « ": " « c1.new_value « std::endl;
        // std::cout « c2.index « ": " « c2.new_value « std::endl;
        // std::cout « c3.index « ": " « c3.new_value « std::endl;

        // assert(bitsery_taa.deserialize(b, change_gen));
        // assert(c1 == change_gen(prn2, mc_state.cfg_vector));
        // assert(c2 == change_gen(prn2, mc_state.cfg_vector));
        // assert(c3 == change_gen(prn2, mc_state.cfg_vector));
    }
}

std::string filename_cnf_formula;
std::string filename_resolvents;
std::string data_directory;

int main(int argc, char **argv)
{
    auto start = std::chrono::high_resolution_clock::now();

    try {
        MPI_Init(&argc, &argv);

        {
            int is_initialized;
            MPI_Initialized(&is_initialized);
            assert(0 != is_initialized);
        }

        int version = 0;
        int subversion = 0;
        assert(MPI_SUCCESS == MPI_Get_version(&version, &subversion));
        if (0 == mpi_get_comm_rank(MPI_COMM_WORLD)) {
            std::cout « "c MPI Version " « version « "." « subversion « std::endl;
        } else {
            MPI_Barrier(MPI_COMM_WORLD);
        }

        filename_cnf_formula = "data/three_color_gnp_50vertices_p0.092_seed37_cnfgen.cnf";
        filename_resolvents = "data/three_color_gnp_50vertices_p0.092_seed37_cnfgen.resolvents";
        data_directory = "out/three_color_gnp_50vertices_p0.092_seed37_cnfgen/";

        std::deque<std::string> args;
        for (int i = 1; i < argc; i++) {
            args.push_back(argv[i]);
        }
        while (!args.empty()) {
            if (args.front() == "-r") {
                args.pop_front();
                assert(!args.empty() && "missing resolvents filename");
                filename_resolvents = args.front();
                args.pop_front();
            }
            else if (args.front() == "-c") {
                args.pop_front();
                assert(!args.empty() && "missing cnf_formula filename");
                filename_cnf_formula = args.front();
                args.pop_front();
            }
            else if (args.front() == "-d") {
                args.pop_front();
                assert(!args.empty() && "missing data_directory filename");
                data_directory = args.front();
                args.pop_front();
            }
            else if (args.front() == "-of") {
                args.pop_front();
                assert(!args.empty() && "missing offset factor");
                markov_chain_offset = std::stoi(args.front()) * num_markov_chains;
                args.pop_front();
            }
            else if (args.front() == "-o") {
                args.pop_front();
                assert(!args.empty() && "missing offset value");
                markov_chain_offset = std::stoi(args.front());
                args.pop_front();
            }
            else if (args.front() == "-l") {
                args.pop_front();
                assert(!args.empty() && "missing target length");
                markov_chain_target_length = std::stoi(args.front());
                args.pop_front();
            }
            else if (args.front() == "-n") {
                args.pop_front();
                assert(!args.empty() && "missing num markov chains");
                num_markov_chains = std::stoi(args.front());
                args.pop_front();
            }
            else if (args.front() == "-t") {
                args.pop_front();
                assert(!args.empty() && "missing T value");
                mcmc_T = (double) std::stoi(args.front());
                args.pop_front();
            }
            else if (args.front() == "-f") {
                args.pop_front();
                assert(!args.empty() && "missing num_reference_chains");
                num_reference_chains = std::stoi(args.front());
                args.pop_front();
            }
            else {
                std::cout « "unknown parameter: " « args.front() « std::endl;
                exit(EXIT_FAILURE);
            }
        }

        if (0 == mpi_get_comm_rank(MPI_COMM_WORLD)) {
            std::cout « "cfg filename_cnf_formula: " « filename_cnf_formula « std::endl;
            std::cout « "cfg filename_resolvents: " « filename_resolvents « std::endl;
            std::cout « "cfg data_directory: " « data_directory « std::endl;
            std::cout « "cfg num_reference_chains: " « num_reference_chains « std::endl;
            std::cout « "cfg num_markov_chains: " « num_markov_chains « std::endl;
            std::cout « "cfg markov_chain_offset: " « markov_chain_offset « std::endl;
            std::cout « "cfg markov_chain_target_length: " « markov_chain_target_length « std::endl;
            std::cout « "cfg mcmc_T: " « mcmc_T « std::endl;
            std::cout « "num_reference_chains: " « num_reference_chains « std::endl;

            MPI_Barrier(MPI_COMM_WORLD);
        }

        // load sat and resolvents file
        probsat_modell_t probsat_modell(filename_cnf_formula, filename_resolvents);
        cfg_vector_length = probsat_modell.resolvents.get_num_resolvents();
        // cfg_vector_length = 10;

        // build mpi workgroups based on shared tmp directory
        const auto workgroup = mpi_shared_tmp_dir_workgroup(MPI_COMM_WORLD, "mcmc-sat");
        // const auto workgroup = mpi_shared_tmp_dir_workgroup(MPI_COMM_WORLD, "mcmc-sat" + std::to_string(mpi_get_comm_rank(MPI_COMM_WORLD) % 2));
        workgroup_ptr = &workgroup;
        // const auto workgroup = mpi_shared_tmp_dir_workgroup(MPI_COMM_WORLD, "numa" + std::to_string(mpi_get_comm_rank(MPI_COMM_WORLD)));
        // const auto workgroup = mpi_shared_tmp_dir_workgroup(MPI_COMM_WORLD, mpi_get_comm_rank(MPI_COMM_WORLD) < 3 ? "numa1" : "numa2");
        std::cout « "c parent node " « workgroup.parent_comm_id + 1 « " / " « workgroup.parent_comm_size
                  « " has workgroup rank " « workgroup.workgroup_comm_id + 1 « " / " « workgroup.workgroup_comm_size
                  « " in workgoup " « workgroup.workgroup_id + 1 « " / " « workgroup.workgroup_count « std::endl;

        std::random_device rdev; std::size_t random_nonce = rdev();
        probsat_modell.workgroup_description = "wg" + std::to_string(workgroup.workgroup_id + 1) + "_r" + std::to_string(random_nonce);
        //probsat_modell.workgroup_description = "wg" + std::to_string(workgroup.workgroup_id + 1);

        // register functions
        wg_rpmanager.add_function("stop_running", stop_running);
        wg_rpmanager.add_function("execute_probsat", execute_probsat);
        wg_rpmanager.add_function("modell_identity", modell_identity<prec_t>);
        wg_rpmanager.add_function("modell_multiply", modell_multiply<prec_t>);


        // TODO parse config
        constant_configuration_generator<cfg_vector_t> cgen_false(false);
        constant_configuration_generator<cfg_vector_t> cgen_true(true);
        cfg_generator_t cfg_generator_const_false = cfg_generator_t(cgen_false);
        cfg_generator_t cfg_generator_const_true = cfg_generator_t(cgen_true);
        // mc_starting_with_special_cfg_vector[0] = cfg_generator_const_false;
        //mc_starting_with_special_cfg_vector[1] = cfg_generator_const_false;
        //mc_starting_with_special_cfg_vector[2] = cfg_generator_const_true;
        // mc_starting_with_special_cfg_vector[3] = cfg_generator_const_true;
        for (std::size_t i = 0; i < num_reference_chains; i++) {
            mc_starting_with_special_cfg_vector[i] = cfg_generator_const_false;
            mc_starting_with_special_cfg_vector[num_reference_chains+i] = cfg_generator_const_true;
        }

        acc_prob_comp_t acc_prob_fct(mcmc_T);

        // global seed
        if (0 == workgroup.parent_comm_id)
        {
            std::string seed_filename = data_directory + "/global_seed.txt";
            global_seed = get_persistent_global_seed<seed_t>(seed_filename);
            std::cout « "c global seed: " « global_seed.short_rep() « "..." « std::endl;
        }
        assert(MPI_SUCCESS == MPI_Bcast(
            global_seed.seed_data.data(), global_seed.seed_data.size(),
            get_mpi<seed_t::base_t>::type(), 0, workgroup.parent_comm));


        if (workgroup.is_workgroup_head())
        { // distributed generation of markov chain seeds
            auto sseq = global_seed.get_seed_seq();
            prn_engine_t seed_engine(sseq);
            seed_type_generator_t sgen;

            const std::size_t num_workgroups = workgroup.workgroup_count;
            const std::size_t workgroup_id = workgroup.workgroup_id;

            std::size_t num_markov_chains_per_work_group = num_markov_chains / num_workgroups;
            std::size_t num_markov_chains_in_this_work_group = num_markov_chains_per_work_group
                + ((workgroup_id < num_markov_chains % num_workgroups) ? 1 : 0);
            std::size_t markov_chains_id_offset = ((workgroup_id > (num_markov_chains % num_workgroups))
                ? workgroup_id - (num_markov_chains % num_workgroups) : 0) * num_markov_chains_per_work_group
                + std::min(workgroup_id, num_markov_chains % num_workgroups) * (num_markov_chains_per_work_group + 1);
            std::size_t local_markov_chain_offset = markov_chain_offset + markov_chains_id_offset;

            std::cout « "c work group " « workgroup_id « " has " « num_markov_chains_in_this_work_group
                « " markov chains at id " « markov_chains_id_offset « " (offset " « local_markov_chain_offset « ")" « std::endl;

            // skip seeds for markov_chain_offset markov chains
            for (std::size_t i = 0; i < local_markov_chain_offset; i++) {
                seed_t mc_seed = seed_t(seed_engine, &sgen);
                ignore(mc_seed);
            }

            auto id_fwidth = std::setw(std::to_string(num_markov_chains).length());
            std::size_t mv_id_start = local_markov_chain_offset; // markov_chains_id_offset
            std::size_t mv_id_end = mv_id_start + num_markov_chains_in_this_work_group;
            for (std::size_t id = mv_id_start; id < mv_id_end; id++) {
                seed_t mc_seed = seed_t(seed_engine, &sgen);
                std::cout « "c markov chain " « id_fwidth « id « " with seed "
                    « std::setw(13) « mc_seed.short_rep() « "..."
                    « "\t" « "(on workgroup " « workgroup_id « ")" « std::endl;

                add_new_mc_chain(id, mc_seed);
            }
        }

        // if (0 == workgroup.parent_comm_id) {
            // std::cout « "continue?" « std::endl;
            // std::string s;
            // std::cin » s;
            // MPI_Barrier(MPI_COMM_WORLD);
        // }

        const bool use_mc_limit = 0 < markov_chain_target_length;

        using priority_id_pair_t = std::pair<std::size_t, std::size_t>;
        auto priority_cmp = [use_mc_limit](priority_id_pair_t left, priority_id_pair_t right) {
            // in case limit is set: sort by work to do, else sort after work done
            return use_mc_limit ? left.first < right.first : left.first > right.first;
        };
        std::priority_queue<
            priority_id_pair_t,
            std::vector<priority_id_pair_t>,
            decltype(priority_cmp)
        > work_queue(priority_cmp);

        std::map<std::size_t, metropolis_hastings_algorithm_t> mh_algorithms;




        remote_execution_context_t wg_remote_exec_context(wg_rpmanager);
        on_demand_scheduler wg_scheduler(workgroup.workgroup_comm, 3, wg_remote_exec_context);

        if (workgroup.is_workgroup_head()) {
            assert(0 == wg_scheduler.env_comm_id); // otherwise renumbering would be required

            wg_leaders = std::make_unique<wg_leader_exec_env>(workgroup.mpi_comm_leaders, 4);

            if (1 < workgroup.workgroup_count) {
                if (0 == workgroup.workgroup_id) {
                    reschedule_manager = std::make_unique<rescheduling_manager>(
                        workgroup.workgroup_count, num_markov_chains, use_mc_limit);
                    wg_leaders->rpmanager.add_function("request_rescheduling", request_rescheduling);
                    wg_leaders->rpmanager.add_function("pass_progress_info", pass_progress_info);
                }

                wg_leaders->rpmanager.add_function("request_progress_info", request_progress_info);
                wg_leaders->rpmanager.add_function("move_markov_chain", move_markov_chain);
                wg_leaders->rpmanager.add_function("recieve_markov_chain", recieve_markov_chain);
            }


/* WG Leaders Test
            wg_leaders->rpmanager.add_function("test_rcg", test_rcg);

            for (int i = 0; i < workgroup.workgroup_count; i++) {
                auto cmd = wg_leaders->rpmanager.prepare_call("test_rcg", workgroup.workgroup_id, i);

                std::function<void()> result_fct = [i, &workgroup]() {
                    std::cout « "wg " « workgroup.workgroup_id « " got ok for " « i « std::endl;
                };
                cmd = wg_leaders->remote_exec_context.on_result(cmd, result_fct);

                wg_leaders->rcg.schedule(cmd, i);
            }

            while (wg_leaders->rcg.is_work_outstanding()) {
                wg_leaders->rcg.do_work();
            }

            MPI_Barrier(workgroup.mpi_comm_leaders);
            if (0 == workgroup.workgroup_id) {
                std::cout « "wg leader test finished" « std::endl;
            }
*/
        }

        std::function<void(std::size_t, int, int)> do_rescheduling =
            [](std::size_t id, int from, int to) {
                auto cmd = wg_leaders->rpmanager.prepare_call("move_markov_chain", id, to);
                wg_leaders->rcg.schedule(cmd, from);
        };


/*
        for (auto & [id, mc_state] : mc_chains_to_add) {
            const bool already_done = use_mc_limit && (mc_state.iteration >= markov_chain_target_length);

            if (already_done) {
                std::cout « "c markov chain " « id « " has already reached"
                    « " computation limit of length " « markov_chain_target_length « std::endl;
            } else {
                std::size_t priority = use_mc_limit
                    ? markov_chain_target_length - mc_state.iteration
                    : mc_state.iteration;

                mh_algorithms.emplace(id, metropolis_hastings_algorithm_t(id, mc_state, probsat_modell, acc_prob_fct));
                // assert(mc_state == mh_algorithms.at(id).get_last_state());

                work_queue.push(std::make_pair(priority, id));
            }
        }*/



        std::size_t actual_working_time = 0;
        decltype(std::chrono::high_resolution_clock::now()) start_work, end_work;
        if (workgroup.is_workgroup_head())
        {
            // bool restarted[2] = {false};
            // bool accepted[8] = {true, true, false, true, true, true, true, true};
            // std::size_t skip_from = 1;
            // std::size_t skip_amount = 4;
            do {
                // std::cout « workgroup.workgroup_id « " XXX " « "main loop start" « std::endl;

                if (!mc_chains_to_add.empty()) {
                    // std::cout « workgroup.workgroup_id « " XXX " « "mc_chains_to_add" « std::endl;

                    for (auto & [id, mc_state] : mc_chains_to_add) {
                        const bool already_done = use_mc_limit && (mc_state.iteration >= markov_chain_target_length);

                        if (already_done) {
                            // std::cout « "c moved markov chain " « id « " has already reached"
                                // « " computation limit of length " « markov_chain_target_length « std::endl;
                        } else {
                            assert(mc_state.cfg_vector.data.size() == cfg_vector_length);
                            // std::cout « "c adding mc " « id « " in iteration " « mc_state.iteration « std::endl;

                            std::size_t priority = use_mc_limit
                                ? markov_chain_target_length - mc_state.iteration
                                : mc_state.iteration;

                            mh_algorithms.emplace(id, metropolis_hastings_algorithm_t(id, mc_state, probsat_modell, acc_prob_fct));

                            work_queue.push(std::make_pair(priority, id));
                        }
                    }

                    mc_chains_to_add.clear();
                }

                // std::cout « workgroup.workgroup_id « " XXX " « "main loop p1" « std::endl;

                if (!work_queue.empty()) {
                    // std::cout « workgroup.workgroup_id « " XXX " « "!work_queue empty" « std::endl;
                    bool was_moved = false;
                    do {
                        const auto [priority, id] = work_queue.top();
                        was_moved = !mh_algorithms.contains(id);
                        if (was_moved) { work_queue.pop(); }
                    } while ((!work_queue.empty()) && (was_moved));
                }

                std::size_t iteration = 0;
                if (work_queue.empty()) {
                    // std::cout « workgroup.workgroup_id « " XXX " « "work_queue empty" « std::endl;
                    running = rescheduling_in_progress || wg_leaders->rcg.is_work_outstanding() || !marcov_chains_to_move.empty();
                } else {
                    // std::cout « workgroup.workgroup_id « " XXX " « "process work" « std::endl;
                    /*
                    const auto [priority, id] = work_queue.top();
                    work_queue.pop();

                    // std::cout « "c processing mc " « id « " with priority " « priority « std::endl;

                    bool is_last_iteration = use_mc_limit && (1 == priority);

                    assert(mh_algorithms.contains(id));
                    metropolis_hastings_algorithm_t &mha = mh_algorithms.at(id);
                    */
                    std::size_t priority, id;
                    std::queue<std::pair<std::size_t, std::size_t» tmp_queue;
                    while (true) {
                        const auto [priority_, id_] = work_queue.top();
                        priority = priority_; id = id_;
                        work_queue.pop();
                        if (work_queue.empty()) {
                            break;
                        }

                        assert(mh_algorithms.contains(id));
                        metropolis_hastings_algorithm_t &mha = mh_algorithms.at(id);
                        if (mha.still_waiting_for_computation()) {
                            tmp_queue.push(std::make_pair(priority, id));
                        } else {
                            break;
                        }
                    }

                    while (!tmp_queue.empty()) {
                        work_queue.push(tmp_queue.front());
                        tmp_queue.pop();
                    }

                    // if (mha.can_start_next_iteration()) {
                        // if (restarted[id]) {
                            // if (skip_from == mha.get_iteration()) {
                                // for (std::size_t i = skip_from; i < skip_from + skip_amount; i++) {
                                    // std::cout « "skipping " « i « std::endl;
                                    // mha.skip_computation(accepted[i]);
                                // }
                            // }
                        // }
                    // }

                    //iteration = mha(wg_scheduler, is_last_iteration);
                    //wg_scheduler.do_work();

                    assert(mh_algorithms.contains(id));
                    metropolis_hastings_algorithm_t &mha = mh_algorithms.at(id);

                    bool is_last_iteration = use_mc_limit && (1 == priority);
                    iteration = mha(wg_scheduler, is_last_iteration);
                    wg_scheduler.do_work();

                    // if (mha.can_start_next_iteration()) {
                        // marcov_chain_state_t mcs = mha.get_last_state();
                        // if (1 == mcs.iteration) {
                            // if (!restarted[id]) {
                                // std::cout « "restart " « id « std::endl;
                                // restarted[id] = true;
                                // mha.cleanup();
                                // mh_algorithms.erase(id);
                                // mh_algorithms.emplace(id, metropolis_hastings_algorithm_t(id, mcs, probsat_modell, acc_prob_fct));
                            // }
                        // }
                    // }

                    std::size_t new_priority = use_mc_limit
                        ? markov_chain_target_length - iteration
                        : iteration;

                    bool is_done = use_mc_limit && (0 == new_priority);
                    if (is_done) {
                        mha.cleanup();
                        mh_algorithms.erase(id);
                        marcov_chains_finished.insert(id);
                        // TODO: inform wg 0
                    } else {
                        work_queue.push(std::make_pair(new_priority, id));
                    }
                }

                // std::cout « workgroup.workgroup_id « " XXX " « "done with work, communicate" « std::endl;

                wg_leaders->rcg.do_work();

                if (!marcov_chains_to_move.empty()) {
                    // std::cout « workgroup.workgroup_id « " XXX " « "mc to move" « std::endl;
                    for (auto const& [id, new_wg] : marcov_chains_to_move) {
                        // std::cout « "MC TO MOVE: " « id « " on " « workgroup.workgroup_id « std::endl;
                        auto search = marcov_chains_finished.find(id);
                        if (search != marcov_chains_finished.end()) {
                            std::cout « "moving dummy for id " « id « " id to " « new_wg « std::endl;

                            marcov_chain_state_t mcs(cfg_vector_length);
                            // buffer_t b = serialize_mcs(mcs);
                            auto cmd = wg_leaders->rpmanager.prepare_call("recieve_markov_chain", id, mcs);
                            wg_leaders->rcg.schedule(cmd, new_wg);
                        } else {
                            std::cout « "prepare movement for " « id « " on " « workgroup.workgroup_id « std::endl;
                            assert(mh_algorithms.contains(id));
                            metropolis_hastings_algorithm_t &mha = mh_algorithms.at(id);
                            while (!mha.can_start_next_iteration()) {
                                bool is_last_iteration = use_mc_limit && (markov_chain_target_length == iteration);
                                iteration = mha(wg_scheduler, is_last_iteration);
                                wg_scheduler.do_work();
                                wg_leaders->rcg.do_work();
                            }

                            // std::cout « "get last state for " « id « " on " « workgroup.workgroup_id « std::endl;
                            marcov_chain_state_t mcs = mha.get_last_state();

                            {
                                // buffer_t b = serialize_mcs(mcs);
                                auto cmd = wg_leaders->rpmanager.prepare_call("recieve_markov_chain", id, mcs);
                                wg_leaders->rcg.schedule(cmd, new_wg);
                            }

                            {
                                mha.cleanup();
                                mh_algorithms.erase(id);
                            }
                        }
                    }
                    marcov_chains_to_move.clear();
                }

                // std::cout « workgroup.workgroup_id « " XXX " « "loop p 622" « std::endl;

                if (send_progress_info) {
                    // std::cout « workgroup.workgroup_id « " XXX " « "send_progress_info" « std::endl;
                    for (auto const& [id, mh_alg] : mh_algorithms) {
                        std::size_t num_it = mh_alg.get_iteration();
                        std::size_t progress = num_it; // use_mc_limit ? markov_chain_target_length - num_it : num_it;

                        // if (0 == workgroup.workgroup_id) {
                            // pass_progress_info(id, progress, workgroup.workgroup_id);
                        // } else {
                        auto cmd = wg_leaders->rpmanager.prepare_call("pass_progress_info", id, progress, workgroup.workgroup_id);
                        wg_leaders->rcg.schedule(cmd, 0);
                        // }
                    }

                    send_progress_info = false;
                }

                // std::cout « workgroup.workgroup_id « " XXX " « "loop p641" « std::endl;

                if ((0 == workgroup.workgroup_id) && (1 < workgroup.workgroup_count)) {
                    static bool move_requested = false;
                    if ((!move_requested) && (2 == iteration)) {
                        request_rescheduling();
                        move_requested = true;
                    }

                    assert(reschedule_manager);
                    if (reschedule_manager->is_rescheduling_in_progress()) {
                        if (reschedule_manager->information_complete()) {
                            std::cout « "rescheduling algorithm start" « std::endl;
                            reschedule_manager->reschedule(do_rescheduling);
                            std::cout « "rescheduling algorithm end" « std::endl;
                        }
                    }
                }

                wg_leaders->rcg.do_work();
                // std::cout « workgroup.workgroup_id « " XXX " « "main loop end" « std::endl;

            } while (running);

            std::cout « "rescheduling_in_progress: " « rescheduling_in_progress « std::endl;
            std::cout « "marcov_chains_to_move.size(): " « marcov_chains_to_move.size() « std::endl;

            std::cout « "wg " « workgroup.workgroup_id « " is done running" « std::endl;

            while (wg_scheduler.is_work_outstanding() || wg_leaders->rcg.is_work_outstanding()) {
                wg_leaders->rcg.do_work();
                wg_scheduler.do_work();
            }

            auto stop_call = wg_rpmanager.prepare_call("stop_running");
            for (int i = 1; i < workgroup.workgroup_comm_size; i++) {
                wg_scheduler.schedule(stop_call, i);
            }
        } else {
            start_work = std::chrono::high_resolution_clock::now();
            while (running) {
                actual_working_time += wg_scheduler.do_work(true);
            }
            end_work = std::chrono::high_resolution_clock::now();
        }


        auto end = std::chrono::high_resolution_clock::now();
        auto overall_duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
        std::cout « "c duration: " « overall_duration.count() « " milliseconds" « std::endl;

        if (!workgroup.is_workgroup_head()) {
            auto work_duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_work - start_work);
            auto init_duration = overall_duration.count() - work_duration.count();
            auto efficiency = 100.0 * (double) actual_working_time / (double) work_duration.count();

            std::cout « "c init: " « init_duration « " ms, work: " « work_duration.count() « " ms with "
                « std::setprecision(2) « std::fixed « efficiency « " % efficiency" « std::endl;
        }

        MPI_Barrier(MPI_COMM_WORLD);

        /*
    } catch (const std::exception& ex) {
        std::cout « ex.what() « std::endl;
    }

    try {
    */
        {
            int is_initialized;
            MPI_Initialized(&is_initialized);
            assert(is_initialized);

            MPI_Barrier(MPI_COMM_WORLD);
            MPI_Finalize();

            int is_finalized;
            MPI_Finalized(&is_finalized);
            assert(is_finalized);
        }
    } catch (const std::exception& ex) {
        std::cout « ex.what() « std::endl;
    }

    return EXIT_SUCCESS;
}
```

7.14 acceptance_probability.hpp

```
#ifndef ACCEPTANCE_PROPABILITY_HPP
#define ACCEPTANCE_PROPABILITY_HPP

#include <string>

template <typename prec_t>
class exponential_acceptance_probability
{
    private:
        prec_t T;

    public:
        using prec_type = prec_t;
        constexpr static bool is_acceptance_probability_implementation = true;

        std::string get_description() {
            return std::string("exp-T-") + std::to_string(T);
        }

        exponential_acceptance_probability(prec_t T_) : T(T_) {}

        prec_t operator()(prec_t old_value, prec_t new_value) const {
            prec_t result = exp( -(new_value - old_value) / T );
            result = std::max(0.0, std::min(result, 1.0));

            // std::cout « "new: " « new_value « std::endl;
            // std::cout « "old: " « old_value « std::endl;
            // std::cout « "x: " « (new_value - old_value) « std::endl;
            // std::cout « "result: " « result « std::endl;

            return result;
        }
};

#endif
```

7.15 simple_acceptance_computation.hpp

```
#ifndef SIMPLE_ACCEPTANCE_COMPUTATION_HPP
#define SIMPLE_ACCEPTANCE_COMPUTATION_HPP


#include <random>
#include <cassert>
#include <functional>

#include "acceptance_probability.hpp"

#include "../../util/seed.hpp"
#include "../../config.hpp"

template <
    typename prng_t,
    typename cfg_vector_t,
    typename prec_t,
    typename value_comp_t,
    typename acc_prob_comp_t
        = exponential_acceptance_probability<prec_t>>
requires (
    value_comp_t::is_value_computation_implementation
    && acc_prob_comp_t::is_acceptance_probability_implementation
)
class simple_acceptance_computation
{
    private:
        std::uniform_real_distribution<prec_t> prob_distribution;

        typedef struct {
            value_comp_t::prepared_computation pcom;

            prng_t prng;

            prec_t value;
        } state_data_t;

        state_data_t state_data[2];

        bool is_first_cfg_vector;

        bool accept;

        uint8_t new_index;

        bool computation_scheduled_state;

        std::size_t remaining_computations;


        auto derive_prng(prng_t &main_prng) {
            seed_t seed = seed_t(main_prng);
            auto sseq = seed.get_seed_seq();
            return prng_t(sseq);
        }

    public:
        constexpr static bool is_acceptance_computation = true;

        std::string get_description() {
            return std::string("sac");
        }

        simple_acceptance_computation() :
            prob_distribution(0.0, 1.0),
            state_data(),
            is_first_cfg_vector(true),
            accept(false),
            new_index(0),
            computation_scheduled_state(false),
            remaining_computations(0)
        {}


        void skip_computation(prng_t &main_prng) {
            derive_prng(main_prng);
        }


        template<typename scheduler_t,
            typename result_projection_t = std::identity,
            typename result_function_t = std::function<void(prec_t)>>
        void start_computation(
            prng_t &main_prng,
            const cfg_vector_t &v,
            scheduler_t &scheduler,
            value_comp_t &value_comp,
            result_projection_t &result_projection = {})
        {
            assert(!computation_scheduled_state);
            computation_scheduled_state = true;

            state_data[new_index].prng = derive_prng(main_prng);

            result_function_t result_fct = [&](auto... results){
                remaining_computations--;
                auto result = result_projection(results...);
                static_assert(
                    std::is_same<prec_t, decltype(result)>::value
                    && "result_projection has wrong result type!");
                state_data[new_index].value = result;
            };

            value_comp.prepare_computation_with_cfg_vector(
                v, state_data[new_index].pcom);

            auto cmd = value_comp(
                state_data[new_index].prng,
                scheduler.rec.rpmanager,
                state_data[new_index].pcom);
            cmd = scheduler.rec.on_result(cmd, result_fct);

            remaining_computations++;
            scheduler.schedule(cmd);
        }


        bool still_waiting_for_computation() {
            return (0 < remaining_computations);
        }


        template<typename scheduler_t>
        bool continue_computation(
            scheduler_t &scheduler,
            value_comp_t &value_comp,
            acc_prob_comp_t &acc_prob_fct)
        {
            assert(computation_scheduled_state);
            ignore(&scheduler, &value_comp, &acc_prob_fct);
            return 0 == remaining_computations;
        }


        bool finish_computation(
            value_comp_t &value_comp, acc_prob_comp_t &acc_prob_fct)
        {
            assert(0 == remaining_computations);
            assert(computation_scheduled_state);
            computation_scheduled_state = false;

            value_comp.finish_computation(state_data[new_index].pcom);

            uint8_t old_index = 1 - new_index;
            if (is_first_cfg_vector) {
                is_first_cfg_vector = false;
                new_index = old_index;
                accept = true;
                return true;
            } else {
                prec_t z = prob_distribution(state_data[new_index].prng);

                prec_t p = acc_prob_fct(
                    state_data[old_index].value,
                    state_data[new_index].value);

                bool accept = (z <= p);
                if (accept) {
                    new_index = old_index;
                }

                return accept;
            }
        }


        void cleanup(value_comp_t &value_comp) {
            ignore(value_comp);
        }


        prec_t get_current_value() {
            assert(!computation_scheduled_state);
            uint8_t index = 1 - new_index;
            return state_data[index].value;
        }

        prec_t get_last_computed_value() {
            assert(!computation_scheduled_state);
            uint8_t index = accept ? 1 - new_index : new_index;
            return state_data[index].value;
        }

        prec_t get_previous_computed_value() {
            assert(!computation_scheduled_state);
            uint8_t index = new_index;
            return state_data[index].value;
        }

/*
        prec_t get_value() {
            assert(!computation_scheduled_state);
            return state_data[1 - new_index].value;
        }
*/
};

#endif
```

7.16 statistical_acceptance_computation.hpp

```
#ifndef STATISTICAL_ACCEPTANCE_COMPUTATION_HPP
#define STATISTICAL_ACCEPTANCE_COMPUTATION_HPP

#include <random>
#include <cassert>
#include <functional>

#include "acceptance_probability.hpp"

#include "../../util/statistics.hpp"
#include "../../util/seed.hpp"
#include "../../config.hpp"

template<typename prec_t, class acc_prob_comp_t>
auto compute_p_range(
    const prec_t cif,
    acc_prob_comp_t &acc_prob_fct,
    const basic_statistical_metrics<prec_t> &new_statistics,
    const basic_statistical_metrics<prec_t> &old_statistics)
{
    prec_t new_value = new_statistics.average;
    prec_t old_value = old_statistics.average;

    prec_t new_sigma = new_statistics.stddev;
    prec_t old_sigma = old_statistics.stddev;

    prec_t new_iv[2] = {
        new_value + cif*new_sigma, new_value - cif*new_sigma};
    prec_t old_iv[2] = {
        old_value + cif*old_sigma, old_value - cif*old_sigma};

    // std::cout « "\t" « "statistics:" « std::endl;
    // std::cout « "\t" « "new_iv: [" « new_iv[0] « ", "
        // « new_value « ", " « new_iv[1] « "]" « std::endl;
    // std::cout « "\t" « "old_iv: [" « old_iv[0] « ", "
        // « old_value « ", " « old_iv[1] « "]" « std::endl;

    prec_t max_x[2] = {
        std::max(new_iv[0], new_iv[1]),
        std::min(-old_iv[0], -old_iv[1])};
    prec_t min_x[2] = {
        std::min(new_iv[0], new_iv[1]),
        std::max(-old_iv[0], -old_iv[1])};

    prec_t p[2] = {
        acc_prob_fct(-max_x[1], max_x[0]),
        acc_prob_fct(-min_x[1], min_x[0])};

    prec_t p_max = std::max(p[0], p[1]);
    prec_t p_min = std::min(p[0], p[1]);
    prec_t p_avg = acc_prob_fct(old_value, new_value);

    /* debug output
    std::cout « "\t" « "stddev: [" « new_statistics.stddev « ", "
        « old_statistics.stddev « "]" « std::endl;
    std::cout « "\t" « "new range: [" « new_statistics.min « ", "
        « new_statistics.max « "]" « std::endl;
    std::cout « "\t" « "old range: [" « old_statistics.min « ", "
        « old_statistics.max « "]" « std::endl;
    std::cout « "\t" « "n: [" « new_statistics.n « ", "
        « old_statistics.n « "]" « std::endl;
    std::cout « "\t" « "new_iv: [" « new_iv[0] « ", "
        « new_value « ", " « new_iv[1] « "]" « std::endl;
    std::cout « "\t" « "old_iv: [" « old_iv[0] « ", "
        « old_value « ", " « old_iv[1] « "]" « std::endl;
    std::cout « "\t" « "max_x: [" « max_x[0] « ", "
        « max_x[1] « "]" « std::endl;
    std::cout « "\t" « "min_x: [" « min_x[0] « ", "
        « min_x[1] « "]" « std::endl;
    std::cout « "\t" « "x: [" « max_x[0] + max_x[1] « ", "
        « new_value - old_value « ", "
        « min_x[0] + min_x[1] « "]" « std::endl;
    std::cout « "\t" « "r: [" « p[0] « ", "
        « p[1] « "]" « std::endl;
    std::cout « "\t" « "new: [" « new_value « ", "
        « new_sigma « "]" « std::endl;
    std::cout « "\t" « "old: [" « old_value « ", "
        « old_sigma « "]" « std::endl;
    std::cout « "\t" « "p: [" « p_min « ", "
        « p_avg « ", " « p_max « "]" « std::endl;
    std::cout « std::endl;
    */

    assert(p_min <= p_avg);
    assert(p_avg <= p_max);

    return std::make_tuple(p_avg, p_min, p_max);
}


template <
    typename prng_t,
    typename cfg_vector_t,
    typename prec_t,
    typename value_comp_t,
    typename acc_prob_comp_t
        = exponential_acceptance_probability<prec_t>>
requires (
    value_comp_t::is_value_computation_implementation
    && acc_prob_comp_t::is_acceptance_probability_implementation
)
class statistical_acceptance_computation
{
    private:
        std::uniform_real_distribution<prec_t> prob_distribution;

        typedef struct {
            value_comp_t::prepared_computation pcom;

            prng_t prng;

            std::vector<prec_t> values;

            std::size_t iteration;

            basic_statistical_metrics<prec_t> statistics;
        } state_data_t;

        prec_t nextz;

        prec_t z;


        state_data_t state_data[2];

        bool is_first_cfg_vector;

        bool accept;

        uint8_t new_index;


        bool computation_scheduled_state;

        std::size_t remaining_computations;


        double coincidence_interval_factor;

        double testniveau;

        std::size_t values_per_iteration;

        std::size_t max_iterations;

        std::size_t values_required;

        std::function<
            void(void *scheduler_ptr, std::size_t i, uint8_t index)
        > schedule_computation;


        auto derive_prng(prng_t &main_prng) {
            // std::cout « "derive prng: " « main_prng « std::endl;
            seed_t seed = seed_t(main_prng);
            // std::cout « "seed: " « seed « std::endl;
            auto sseq = seed.get_seed_seq();
            // std::cout « "sseq: ";
            // sseq.param(std::ostream_iterator<int>(std::cout, ", "));
            // std::cout « std::endl;
            return prng_t(sseq);
        }

    public:
        constexpr static bool is_acceptance_computation = true;

        std::string get_description() {
            return std::string("cac");
        }

        statistical_acceptance_computation(
                // prng_t &prng,
                // double coincidence_interval_factor = 1.0,
                // double testniveau = 0.005,
                // std::size_t values_per_iteration = 2400,
                // std::size_t max_iterations = 100,
                // std::size_t values_required = 10000

                // testing
                // double coincidence_interval_factor = 1.0,
                // double testniveau = 0.01,
                // std::size_t values_per_iteration = 240*2,
                // std::size_t max_iterations = 12,
                // std::size_t values_required = 2400

                // configurable
                double coincidence_interval_factor = cfg_sac_confidence_interval_scaling_factor,
                double testniveau = cfg_sac_testniveau,
                std::size_t values_per_iteration = cfg_sac_num_additional_values_per_iteration,
                std::size_t max_iterations = cfg_sac_max_iterations,
                std::size_t values_required = cfg_sac_num_initial_values

                // works
                // double coincidence_interval_factor = 1.0,
                // double testniveau = 0.01,
                // std::size_t values_per_iteration = 240*2,
                // std::size_t max_iterations = 100,
                // std::size_t values_required = 10000
        ) :
            prob_distribution(0.0, 1.0),
            nextz(0),
            z(0),
            state_data(),
            is_first_cfg_vector(true),
            accept(false),
            new_index(0),
            computation_scheduled_state(false),
            remaining_computations(0),
            // main_prng(prng),
            coincidence_interval_factor(coincidence_interval_factor),
            testniveau(testniveau),
            values_per_iteration(values_per_iteration),
            max_iterations(max_iterations),
            values_required(values_required),
            schedule_computation(nullptr)
        {}


        // statistical_acceptance_computation(
            // statistical_acceptance_computation &&other) = default;
        // statistical_acceptance_computation(
            // const statistical_acceptance_computation &other) = delete;

        // ~statistical_acceptance_computation() {}


        void skip_computation(prng_t &main_prng) {
            derive_prng(main_prng);
        }


        template<typename scheduler_t,
            typename result_projection_t = std::identity,
            typename result_function_t = std::function<void(prec_t)»
        void start_computation(
            prng_t &main_prng,
            const cfg_vector_t &v,
            scheduler_t &scheduler,
            value_comp_t &value_comp,
            result_projection_t &result_projection = {})
        {
            assert(!computation_scheduled_state);
            computation_scheduled_state = true;

            ignore(&scheduler);


            // std::cout « "main_prng: " « main_prng « std::endl;
            state_data[new_index].prng = derive_prng(main_prng);
            // std::cout « "derived prng: "
                // « state_data[new_index].prng() « ", "
                // « state_data[new_index].prng() « ", "
                // « state_data[new_index].prng() « ", "
                // « state_data[new_index].prng « std::endl;

            prob_distribution.reset();
            z = nextz;
            nextz = prob_distribution(state_data[new_index].prng);
            // std::cout « "start_computation: z: "
```

```
                // « z « ", v: " « v « std::endl;

            value_comp.prepare_computation_with_cfg_vector(
                v, state_data[new_index].pcom);

            state_data[new_index].iteration = 0;
            state_data[new_index].statistics =
                basic_statistical_metrics<prec_t>();

            // reset statistics but not already computed values
            uint8_t old_index = 1 - new_index;
            state_data[old_index].statistics =
                basic_statistical_metrics<prec_t>();
            accept = false;

            schedule_computation =
                [&](void *scheduler_ptr, std::size_t i, uint8_t index)
            {
                scheduler_t &scheduler =
                    *(static_cast<scheduler_t *>(scheduler_ptr));

                result_function_t result_fct
                    = [&, i, index](auto... results)
                {
                    remaining_computations--;
                    auto result = result_projection(results...);
                    static_assert(
                        std::is_same<prec_t, decltype(result)>::value
                        && "result_projection has wrong result type!");
                    // std::cout « "c R state_data[" « (int) index
                        // « "].values[" « i « "] = "
                        // « result « std::endl;
                    state_data[index].values[i] = result;
                };

                auto cmd = value_comp(
                    state_data[index].prng,
                    scheduler.rec.rpmanager,
                    state_data[index].pcom);
                cmd = scheduler.rec.on_result(cmd, result_fct);

                remaining_computations++;
                scheduler.schedule(cmd);
            };
        }


        bool still_waiting_for_computation() {
            return (0 < remaining_computations);
        }


        template<typename scheduler_t>
        bool continue_computation(
            scheduler_t &scheduler,
            value_comp_t &value_comp,
            acc_prob_comp_t &acc_prob_fct)
        {
            assert(computation_scheduled_state);
            ignore(&value_comp);

            if (is_first_cfg_vector) {
                return true; // nothing to do
            }

            if (0 < remaining_computations) {
                return false; // not yet done
            }

            uint8_t old_index = 1 - new_index;
            uint8_t indices[2] = {new_index, old_index};

            // update statistics
            std::size_t num_values =
                state_data[new_index].iteration * values_per_iteration;
            if (0 < num_values) {
                for (auto j = 0; j < 2; j++) {
                    uint8_t index = indices[j];

                    auto it_start =
                        state_data[index].values.begin()
                        + state_data[index].statistics.counted();
                    auto it_end =
                        state_data[index].values.begin() + num_values;
                    state_data[index].statistics(it_start, it_end);

                    #if DEBUG_ASSERTIONS
                    assert(num_values
                        == state_data[index].statistics.counted());
                    #endif
                }

                // test for evidence of result
                std::size_t num_values_available =
                    std::min(state_data[new_index].statistics.n,
                    state_data[old_index].statistics.n);

                if (num_values_available >= values_required) {
                    auto [p_avg, p_min, p_max] = compute_p_range(
                        coincidence_interval_factor,
                        acc_prob_fct,
                        state_data[new_index].statistics,
                        state_data[old_index].statistics);

                    // std::cout « "\t" « "p: [" « p_min « ", "
                        // « p_avg « ", " « p_max « "]" « std::endl;

                    // test for statistical evidence of accept/decline
                    if (z <= p_min) {
                        // std::cout « "clear accept ("
                            // « z « ", " « p_min « ")" « std::endl;
                        accept = true;
                        return true;
                    }

                    if (z > p_max) {
                        // std::cout « "clear decline ("
                            // « z « ", " « p_max « ")" « std::endl;
                        accept = false;
                        return true;
                    }

                    const bool max_iterations_reached =
                        max_iterations <= state_data[new_index].iteration;
                    if (max_iterations_reached) {
                        // std::cout « "max iterations reached ("
                            // « z « ", " « p_avg « ")" « std::endl;
                        accept = (z <= p_avg); // false
                        return true; // done with best guess
                    }

                    const bool testniveau_unterschritten =
                        std::abs(p_max - p_min) < testniveau;
                    if (testniveau_unterschritten) {
                        // std::cout « "testniveau_unterschritten ("
                            // « z « ", " « p_avg « ")" « std::endl;
                        accept = (z <= p_avg); // false
                        return true; // done with best guess
                    }
                }
            }

            // not yet done => schedule next iteration(s) for more data
            const bool new_values_for_old_index_are_required =
                state_data[new_index].iteration
                >= state_data[old_index].iteration;

            assert(state_data[new_index].iteration
                <= state_data[old_index].iteration);

            for (auto j = 0;
                j < (new_values_for_old_index_are_required ? 2 : 1); j++)
            {
                uint8_t index = indices[j];
                state_data[index].iteration++;

                std::size_t num_values =
                    state_data[index].iteration*values_per_iteration;
                if (num_values > state_data[index].values.size()) {
                    state_data[index].values.resize(num_values);
                }

                for (std::size_t i = num_values - values_per_iteration;
                    i < num_values; i++)
                {
                    schedule_computation(
                        static_cast<void *>(&scheduler), i, index);
                }
            }

            return false;
        }
        bool finish_computation(
            value_comp_t &value_comp, acc_prob_comp_t &acc_prob_fct)
        {
            assert(0 == remaining_computations);
            assert(computation_scheduled_state);
            computation_scheduled_state = false;

            ignore(&acc_prob_fct);

            uint8_t old_index = 1 - new_index;
            if (is_first_cfg_vector) {
                is_first_cfg_vector = false;
                new_index = old_index;
                return true;
            } else {
                if (accept) {
                    // std::cout « "accept "
                        // « state_data[new_index].statistics.average
                        // « std::endl;
                    value_comp.finish_computation(
                        state_data[old_index].pcom);
                    new_index = old_index;
                } else {
                    // std::cout « "decline "
                        // « state_data[new_index].statistics.average
                        // « std::endl;
                    // std::cout « "keep value "
                        // « state_data[old_index].statistics.average
                        // « std::endl;
                    value_comp.finish_computation(
                        state_data[new_index].pcom);
                }
                return accept;
            }
        }


        void cleanup(value_comp_t &value_comp) {
            if (!is_first_cfg_vector) {
                uint8_t old_index = 1 - new_index;
                value_comp.finish_computation(state_data[old_index].pcom);
            }
        }


        prec_t get_current_value() {
            assert(!computation_scheduled_state);
            uint8_t index = 1 - new_index;
            return (0 == state_data[index].statistics.n) ?
                NAN : state_data[index].statistics.average;
        }

        prec_t get_last_computed_value() {
            assert(!computation_scheduled_state);
            uint8_t index = accept ? 1 - new_index : new_index;
            return (0 == state_data[index].statistics.n) ?
                NAN : state_data[index].statistics.average;
        }

        prec_t get_previous_computed_value() {
            assert(!computation_scheduled_state);
            uint8_t index = new_index;
            return (0 == state_data[index].statistics.n) ?
                NAN : state_data[index].statistics.average;
        }

/*
        prec_t get_value() {
            assert(!computation_scheduled_state);
            uint8_t index = accept ? 1 - new_index : new_index;
            return (0 == state_data[index].statistics.n) ?
                NAN : state_data[index].statistics.average;
        }
*/
};


#endif
```

7.17 algorithm.hpp

```
#ifndef ALGORITHM_HPP
#define ALGORITHM_HPP
```

```


#include <set>
#include <iostream>
#include <cassert>

#include "../config.hpp"


template<
    typename marcov_chain_state_t,
    typename change_generator_t,
    typename acceptance_computation_t,
    typename computation_modell_t,
    typename acc_prob_comp_t,
    typename result_statistics_t,
    typename result_function_t,
    const bool enforce_change,
    const bool skip_unchanged_vectors
    >
class metropolis_hastings_algorithm
{
    private:
        using cfg_vector_t = marcov_chain_state_t::cfg_vector_type;

        std::size_t id;

        marcov_chain_state_t mc_state;

        decltype(mc_state.prn_engine) last_prng;

        change_generator_t change_gen;

        cfg_vector_t next_cfg_vector;

        acceptance_computation_t acc_computation;

        bool is_processing;

        computation_modell_t &computation_modell;

        acc_prob_comp_t &acc_prob_fct;

        result_statistics_t execution_statistics;


        void generate_next_cfg_vector()
        {
            next_cfg_vector = mc_state.cfg_vector;

            bool change_happened = enforce_change;
            do {
                using index_t = decltype(
                    change_gen(mc_state.prn_engine,
                    next_cfg_vector).index);

                // generate changes
                std::set<index_t> changed_positions;

                for (decltype(mcmc_num_changes) i = 0;
                    i < mcmc_num_changes; i++)
                {
                    auto cfg_change =
                        change_gen(mc_state.prn_engine, next_cfg_vector);
                    next_cfg_vector.data[cfg_change.index]
                        = cfg_change.new_value;

                    if ((!change_happened) && (skip_unchanged_vectors))
                    {
                        changed_positions.insert(cfg_change.index);
                    }
                }

                // check if config vector is changed
                if ((!change_happened) && (skip_unchanged_vectors)) {
                    for(auto p : changed_positions) {
                        if (next_cfg_vector.data[p]
                            != mc_state.cfg_vector.data[p])
                        {
                            change_happened = true;
                            break;
                        }
                    }
                }
            } while ((!change_happened) && (skip_unchanged_vectors));
        }

    public:

        metropolis_hastings_algorithm(
            std::size_t id,
            auto mc_state_,
            computation_modell_t &computation_modell,
            acc_prob_comp_t &acc_prob_fct
        ) :
            id(id),
            mc_state(mc_state_),
            last_prng(mc_state.prn_engine),
            change_gen(mc_state.cfg_vector),
            next_cfg_vector(mc_state.cfg_vector),
            acc_computation(/*mc_state.prn_engine*/),
            is_processing(false),
            computation_modell(computation_modell),
            acc_prob_fct(acc_prob_fct),
            execution_statistics()
        {}

        // metropolis_hastings_algorithm(
            // metropolis_hastings_algorithm &&other) = default;
        // metropolis_hastings_algorithm(
            // const metropolis_hastings_algorithm &other) = delete;

        // ~metropolis_hastings_algorithm() {}


        marcov_chain_state_t get_last_state() {
            assert(!is_processing);

            return marcov_chain_state_t(
                mc_state.iteration - 1, mc_state.cfg_vector, last_prng);
        }


        template<typename scheduler_t>
        void prepare_iteration(scheduler_t &scheduler) {
            // std::cout « "prepare" « std::endl;
            assert(!is_processing);
            is_processing = true;

            last_prng = mc_state.prn_engine;
            // std::cout « "test prepare_iteration "
                // « mc_state.iteration « ": " « last_prng « std::endl;


            acc_computation.template start_computation
                <scheduler_t, result_statistics_t, result_function_t>
                (mc_state.prn_engine,
                next_cfg_vector,
                scheduler,
                computation_modell,
                execution_statistics);
        }


        bool still_waiting_for_computation() {
            return acc_computation.still_waiting_for_computation();
        }


        template<typename scheduler_t>
        bool continue_iteration(scheduler_t &scheduler) {
            // std::cout « "continue" « std::endl;
            assert(is_processing);
            bool is_finished = acc_computation.continue_computation(
                scheduler, computation_modell, acc_prob_fct);
            // scheduler.do_work();
            return is_finished;
        }


        void finish_iteration(bool last_iteration = false) {
            // std::cout « "finish" « std::endl;
            assert(is_processing);
            is_processing = false;

            bool accept = acc_computation.finish_computation(
                computation_modell, acc_prob_fct);
            if (accept) {
                mc_state.cfg_vector = next_cfg_vector;
            }

            if (1 == mc_state.iteration) {
                auto initial_value = accept ?
                    acc_computation.get_previous_computed_value() :
```

```
                    acc_computation.get_current_value();
                std::cout « "mc " « id
                    « " iteration " « mc_state.iteration
                    « " value " « initial_value
                    « " accepted " « initial_value « std::endl;
            }

            if (0 < mc_state.iteration) {
                std::cout « "mc " « id
                    « " iteration " « (mc_state.iteration + 1)
                    « " value " « acc_computation.get_current_value()
                    « (accept ? " accepted " : " declined ")
                    « acc_computation.get_last_computed_value()
                    « std::endl;
            }

            #if DEBUG_ASSERTIONS
            if (accept) {
                assert(mc_state.cfg_vector == next_cfg_vector);
            } else {
                assert((mc_state.cfg_vector != next_cfg_vector)
                    || !(skip_unchanged_vectors || enforce_change));
            }
            #endif

            if (last_iteration) {
                is_processing = true; // prevent further execution
                cleanup();
            } else {
                generate_next_cfg_vector();
            }

            mc_state.iteration++;
        }


        void skip_computation(bool accept) {
            acc_computation.skip_computation(mc_state.prn_engine);
            if (accept) {
                mc_state.cfg_vector = next_cfg_vector;
            }
            generate_next_cfg_vector();
            mc_state.iteration++;
        }


        void cleanup() {
            acc_computation.cleanup(computation_modell);
        }


        bool can_start_next_iteration() const {
            return !is_processing;
        }


        std::size_t get_iteration() const {
            return mc_state.iteration;
        }


        void print_execution_statistic() {
            std::cout « "mc " « id « " execution_statistics: "
                « execution_statistics « std::endl;
        }


        template<typename scheduler_t>
        std::size_t operator()(
            scheduler_t &scheduler, bool last_iteration = false)
        {
            if (false == is_processing) {
                prepare_iteration(scheduler);
            } else {
                bool is_finished = continue_iteration(scheduler);
                if (is_finished) {
                    finish_iteration(last_iteration);
                }
            }

            return mc_state.iteration;
        }
};

#endif
```

7.18 change_generator.hpp

```cpp
#ifndef CHANGE_GENERATOR_HPP
#define CHANGE_GENERATOR_HPP


#include <random>
#include <cassert>
#include <sstream>

#include "../../util/basic_serialization.hpp"


template<typename size_type, typename value_t>
struct change {
    size_type index;

    value_t new_value;

    auto operator==(const change<size_type, value_t> &rhs) const {
        return (index == rhs.index) && (new_value == rhs.new_value);
    }

    auto operator!=(const change<size_type, value_t> &rhs) const {
        return (*this == rhs);
    }
};


template <
    typename cfg_vector_t,
    bool enforce_change,

    typename value_t = std::conditional<
        std::is_same<bool, typename cfg_vector_t::value_type>::value,
        short,
        typename cfg_vector_t::value_type
    >::type,

    typename value_distr_type = std::uniform_int_distribution<value_t>,

    decltype(value_distr_type().min()) value_distr_minv
        = cfg_vector_t::min_value,

    decltype(value_distr_type().max()) value_distr_maxv
        = enforce_change ?
            cfg_vector_t::max_value - 1 :
            cfg_vector_t::max_value
>
class simple_change_generator
{
    public:
        constexpr static bool is_change_generator = true;
        using index_t = cfg_vector_t::size_type;

    private:
        std::uniform_int_distribution<index_t> index_distr;

        value_distr_type value_distr;

    public:
        std::string get_description() {
            return enforce_change ? "scg-ec" : "scg-ac";
        }

        simple_change_generator(const cfg_vector_t &v) :
            index_distr(0, v.data.size() - 1),
            value_distr(value_distr_minv, value_distr_maxv)
        {
            assert(0 < v.data.size());
            static_assert(
                cfg_vector_t::min_value < cfg_vector_t::max_value);
        }

        auto operator()(auto &prng, const cfg_vector_t &v)
        {
            index_t index = index_distr(prng);
            value_t value = value_distr(prng);

            if constexpr(enforce_change) {
                if (value >= v.data[index]) {
                    value++;
                }
            }

            return change<index_t, typename cfg_vector_t::value_type>{
                index,
                static_cast<cfg_vector_t::value_type>(value)
            };
        }
};


template <typename S, typename simple_change_generator_t>
    requires (simple_change_generator_t::is_change_generator)
void serialize(S& s, simple_change_generator_t &change_gen) {
    std::string descr = change_gen.get_description();
    s.text1b(descr, 7);
    assert(6 == descr.length());
    assert(change_gen.get_description() == descr);

    std::stringstream sstr;
    sstr « change_gen.index_distr;
    sstr « change_gen.value_distr;
    std::string distributions = sstr.str();
    assert(!sstr.fail());

    generic_serialize(s, distributions);

    sstr.str(distributions);
    sstr » change_gen.index_distr;
    sstr » change_gen.value_distr;
    assert(!sstr.fail());
    assert(sstr.eof());
}


#endif
```

7.19 configuration_generator.hpp

```
#ifndef CONFIGURATION_MANAGER_HPP
#define CONFIGURATION_MANAGER_HPP


#include <string>
#include <random>
#include <cassert>


template <typename cfg_vector_t>
class constant_configuration_generator
{
    public:
        constexpr static bool is_configuration_generator = true;

    private:
        using value_t = cfg_vector_t::value_type;

        value_t cvalue;

    public:
        std::string get_description() {
            return std::string("ci-") + std::to_string(cvalue);
        }

        constant_configuration_generator(value_t constant_value)
            : cvalue(constant_value)
        {
            assert(cfg_vector_t::min_value <= constant_value);
            assert(cfg_vector_t::max_value >= constant_value);
        }

        void operator()(auto &prng, cfg_vector_t &v) const {
            ignore(prng);

            for (typename cfg_vector_t::size_type i = 0;
                i < v.data.size(); i++)
            {
                v.data[i] = cvalue;
            }
        }
};


template <typename cfg_vector_t>
class uniform_configuration_generator
{
    private:
        using value_t = std::conditional<
```

```
            std::is_same<bool, typename cfg_vector_t::value_type>::value,
            short,
            typename cfg_vector_t::value_type>::type;

    public:
        constexpr static bool is_configuration_generator = true;

        std::string get_description() {
            return std::string("ui");
        }

        uniform_configuration_generator() {}

        template <typename rng_t>
        void operator()(rng_t &prng, cfg_vector_t &v) const {
            std::uniform_int_distribution<value_t>
                distr(cfg_vector_t::min_value, cfg_vector_t::max_value);
            for (typename cfg_vector_t::size_type i = 0;
                i < v.data.size(); i++)
            {
                value_t value = distr(prng);
                v.data[i] = static_cast<cfg_vector_t::value_type>(value);
            }
        }
};


#endif
```

7.20 configuration_vector.hpp

```
#ifndef CONFIGURATION_VECTOR_HPP
#define CONFIGURATION_VECTOR_HPP


#include <vector>
#include <type_traits>

#include "../../util/bitfield.hpp"
#include "../../util/basic_serialization.hpp"


template<bool use_bitfield = false>
struct binary_configuration_vector
{
    public:
        using value_type = bool;

        static constexpr value_type min_value = false;

        static constexpr value_type max_value = true;

        using vector_type =
            std::conditional< use_bitfield,
                bitfield<std::size_t>,
                std::vector<value_type>
            >::type;
        // using vector_type = std::vector<value_type>;

        using size_type = decltype(vector_type().size());

        vector_type data;



        binary_configuration_vector() : data(0) {}

        binary_configuration_vector(size_type length) : data(length) {}


        binary_configuration_vector(
            const binary_configuration_vector &other)
        {
            data = other.data;
        }


        friend void swap(
            binary_configuration_vector<use_bitfield> &l,
            binary_configuration_vector<use_bitfield> &r)
        {
            std::swap(l.data, r.data);
        }


        binary_configuration_vector &operator=(
            const binary_configuration_vector<use_bitfield> &other)
        {
            data = other.data;
            return *this;
        }

        /*
        binary_configuration_vector &operator=(
            const binary_configuration_vector<use_bitfield> &other)
        {
            swap(*this, other);
            return *this;
        }
        */


        binary_configuration_vector(binary_configuration_vector &&other)
        {
            swap(*this, other);
        }


        friend std::ostream &operator«(
            std::ostream &os,
            const binary_configuration_vector<use_bitfield> &bf)
        {
            os « "binary_configuration_vector:";
            for (size_type i = 0; i < bf.data.size(); i++) {
                os « " " « (bf.data[i] ? "1" : "0");
            }
            return os;
        }


        std::size_t count_ones() const
        {
            std::size_t ones = 0;

            for (size_type i = 0; i < data.size(); i++) {
                if (data[i]) { ones++; }
            }

            return ones;
        }

        bool operator==(
            const binary_configuration_vector<use_bitfield> &rhs) const
        {
            if (data.size() == rhs.data.size()) {
                for (size_type i = 0; i < data.size(); i++) {
                    if (data[i] != rhs.data[i]) { return false; }
                }
                return true;
            }
            return false;
        }

        bool operator!=(
            const binary_configuration_vector<use_bitfield> &rhs) const
        {
            return !(*this == rhs);
        }
};


template<typename S, bool use_bitfield>
void serialize(
    S &s, binary_configuration_vector<use_bitfield> &cfg_vector)
{
    serialize_marker(s, "bcv", 4);

    bool ub = use_bitfield;
    s.value1b(ub);
    assert(use_bitfield == ub);

    serialize(s, cfg_vector.data);
}


#endif
```

7.21 mcmc.hpp

```
#ifndef MCMC_HPP
#define MCMC_HPP


#include <cstddef>
#include <random>
#include <sstream>

#include "../config.hpp"
#include "../util/basic_serialization.hpp"


template<typename cfg_vector_t>
class marcov_chain_state
{
    public:
        using cfg_vector_type = cfg_vector_t;

        std::size_t iteration;

        cfg_vector_t cfg_vector;

        prn_engine_t prn_engine;


        marcov_chain_state(std::size_t cfg_vector_length = 0) :
            iteration(std::numeric_limits<std::size_t>::max()),
            cfg_vector(cfg_vector_length),
            prn_engine() {}

        marcov_chain_state(
            std::size_t cfg_vector_length,
            auto prn_engine
        ) :
            iteration(0),
            cfg_vector(cfg_vector_length),
            prn_engine(prn_engine)
        {}

        marcov_chain_state(
            std::size_t iteration,
            cfg_vector_t &cfg_vector,
            prn_engine_t &prn_engine
        ) :
            iteration(iteration),
            cfg_vector(cfg_vector),
            prn_engine(prn_engine) {}

        marcov_chain_state(const marcov_chain_state &other) :
            iteration(other.iteration),
            cfg_vector(other.cfg_vector),
            prn_engine(other.prn_engine) {}

        marcov_chain_state(marcov_chain_state &&other) {
            swap(*this, other);
        }

        marcov_chain_state &operator=(
            const marcov_chain_state<cfg_vector_t> &other)
        {
            iteration = other.iteration;
            cfg_vector = other.cfg_vector;
            prn_engine = other.prn_engine;
            return *this;
        }

        friend void swap(
            marcov_chain_state<cfg_vector_t> &l,
            marcov_chain_state<cfg_vector_t> &r)
        {
            std::swap(l.iteration, r.iteration);
            std::swap(l.cfg_vector, r.cfg_vector);
            std::swap(l.prn_engine, r.prn_engine);
        }

        bool operator==(
            const marcov_chain_state<cfg_vector_t> &rhs) const
        {
            return (iteration == rhs.iteration)
                && (cfg_vector == rhs.cfg_vector)
                && (prn_engine == rhs.prn_engine);
        }

        bool operator!=(
            const marcov_chain_state<cfg_vector_t> &rhs) const
        {
            return !(*this == rhs);
        }

        friend std::ostream &operator«(
            std::ostream &os,
            const marcov_chain_state<cfg_vector_t> &mcs)
        {
            os  « "state of iteration " « mcs.iteration « ":\n"
                « mcs.cfg_vector « "\n" « mcs.prn_engine « std::endl;
            return os;
        }

        std::string get_prn_engine_as_string() const {
            std::stringstream isstr;
            isstr « prn_engine;
            assert(!isstr.fail());
            return isstr.str();
        }

        void set_prn_engine_from_string(auto prn_state) {
            std::stringstream osstr(prn_state);
            osstr » prn_engine;
            assert(!osstr.fail());
            assert(osstr.eof());
        }
};


template<typename S, typename cfg_vector_t>
void serialize(S &s, marcov_chain_state<cfg_vector_t> &mc)
{
    std::string prn_state = mc.get_prn_engine_as_string();

    // serialize_marker(s, "ms1", 4);
    serialize_basic_type<S, decltype(mc.iteration)>(s, mc.iteration);

    // serialize_marker(s, "ms2", 4);
    serialize(s, mc.cfg_vector);

    // serialize_marker(s, "ms3", 4);
    generic_serialize(s, prn_state);
    mc.set_prn_engine_from_string(prn_state);

    // serialize_marker(s, "ms4", 4);
}

/*
template<typename S, typename cfg_vector_t>
void serialize(S &s, marcov_chain_state<cfg_vector_t> &mc)
{
    std::cout « "start" « std::endl;
    std::string prn_state;

    std::stringstream isstr;
    isstr « mc.prn_engine;
    prn_state = isstr.str();

    // generic_serialize(s, mc.iteration, mc.cfg_vector, prn_state);
    // generic_serialize(s, mc.iteration);
    s.value8b(mc.iteration);

    // if (mc.cfg_vector.data.length()
    // cfg_vector(cfg_vector_length)

    // std::cout « "mc.cfg_vector size a: " « mc.cfg_vector.data.size() « std::endl;

    serialize(s, mc.cfg_vector);

    // std::cout « "mc.cfg_vector size b: " « mc.cfg_vector.data.size() « std::endl;

    std::string marker1 = "m42";
    s.text1b(marker1, 4);
    assert(std::string("m42") == marker1);

    std::size_t strlength = prn_state.size();
    generic_serialize(s, strlength);
    prn_state.reserve(strlength);
    s.text1b(prn_state, strlength);
    assert((prn_state.size() == strlength) || print_trace());

    std::string marker2 = "fm!";
    s.text1b(marker2, 4);
    assert(std::string("fm!") == marker2);

    std::stringstream osstr(prn_state);
    osstr » mc.prn_engine;

    std::cout « "ok" « std::endl;
}
*/

/*
// output statistic
class mcmc_statistic
{
    private:
        std::size_t mc_id;

    public:
        mcmc_statistic(std::size_t mc_id) : mc_id(mc_id) {}
};

// able to fast reload application state
class mcmc_history
{
    private:
        std::size_t mc_id;
        std::fstream hfile;
        uint8_t no_counter = 0;

Problem: do changes need consistent prn generator?

    full save:
        store "F"
        size of marcov_chain_state
        marcov_chain_state
            iteration
            cfg_vector
            prn_engine

        go to the start of the file and save last position of "F"

    incremental save:
        accepted?
            if yes:
                store "n" save no counter
                store "y"
                save changes in cfg vector (size + changes)
            if no:
                increment no counter
                prevent overflow (i.e. store "n" and save no counter)

    restore:
        load last full save
        restore current iteration cfg_vector and prn_engine state by fast forwarding
        continue as from beginning


        void save_no() {
            hfile.put("n");
            hfile.put(no_counter);
            no_counter = 0;
        }

        void save_full(marcov_chain_state &mc_state)
        {
            const std::size_t offset = hfile.tellp();

            // write "F", size, mc_state
            hfile.put("F");

            type_abstraction_adapter_t bitsery_taa;
            buffer_t data = bitsery_taa.serialize(mc_state);

            make_generic_serializeable<std::size_t> gssize = data.size();
            buffer_t size = bitsery_taa.serialize(gssize);
            assert(size.size() == sizeof(std::size_t)); // otherwise load won’t work!

            assert(!hfile.write(size.data(), size.size()).fail());
            assert(!hfile.write(data.data(), data.size()).fail());
            hfile.flush();
            assert(!hfile.fail());

            // write offset at start of file
            const std::size_t end = hfile.tellp();
            hfile.seekp(0, hfile.beg);
            make_generic_serializeable<std::size_t> gsoffset = offset;
            buffer_t offs = bitsery_taa.serialize(gsoffset);
            assert(offs.size() == sizeof(std::size_t));
            assert(!hfile.write(offs.data(), offs.size()).fail());
            hfile.seekp(end, hfile.beg);

            hfile.flush();
            assert(!hfile.fail());
        }

        void load_full(marcov_chain_state &mc_state)
        {
            char c = 0;
            assert(hfile.get(c) && c == ’F’);
            type_abstraction_adapter_t bitsery_taa;

            // read size
            make_generic_serializeable<std::size_t> gssize = 0;
            std::vector inbuf(sizeof(std::size_t));
            assert(!hfile.read(inbuf.data(), inbuf.size()).fail());
            assert(bitsery_taa.deserialize(inbuf.data(), gssize));

            // read mc_state
            inbuf.resize(gssize.data);
            assert(!hfile.read(inbuf.data(), inbuf.size()).fail());
            assert(bitsery_taa.deserialize(inbuf.data(), mc_state));
        }

    public:
        void add_changes(bool accepted, auto &changes) {
            if (accepted) {
                if (0 < no_counter) { save_no(); }
                hfile.put("y");
                // type_abstraction_adapter_t bitsery_taa;
                // buffer_t data = bitsery_taa.serialize(changes);
                // assert(!hfile.write(data.data(), data.size()).fail());
                changes.save_to(hfile);
                hfile.flush();
            } else {
                no_counter++;
                if (std::numeric_limits<decltype(no_counter)>::max() == no_counter) {
                    save_no();
                    hfile.flush();
                }
            }

            assert(!hfile.fail());
        }

        mcmc_history(std::size_t mc_id, marcov_chain_state &mc_state) :
            mc_id(mc_id)
            hfile(data_directory + "/" + std::to_string(mc_id) + ".mch",
                std::ios::in | std::ios::out | ios::binary | std::ios::app),
            no_counter(0)
        {
            assert(hfile.is_open());

            hfile.seekp(0, hfile.end);
            const auto fsize = hfile.tellp();

            if (0 == fsize) {
                // write offset and first full save
                std::size_t s = 0;
                assert(!hfile.write(&s, sizeof(std::size_t)).fail());
                save_full(mc_state);
            } else {
                // read offset of last full save
                hfile.seekg(0, hfile.beg);
                type_abstraction_adapter_t bitsery_taa;
                make_generic_serializeable<std::size_t> gsoffset = 0;
                std::vector inbuf(sizeof(std::size_t));
                assert(hfile.read(inbuf.data(), inbuf.size()).good());
                assert(bitsery_taa.deserialize(inbuf.data(), gsoffset));

                // read first save
                hfile.seekg(sizeof(std::size_t), hfile.beg);
                marcov_chain_state_t mc_state_(mc_state.cfg_vector.data.size(), prn_engine_t());
                load_full(mc_state_);
                assert(mc_state_ == mc_state && "initial state must match!");

                // read last full save
                hfile.seekg(gsoffset.data, seed_file.beg);
                load_full(mc_state);

                // fast forward
                while () {

                }
            }

            assert(!hfile.fail());
        }
};
*/
```

```

#endif
```

7.22 modell.hpp

```
#ifndef MODELL_HPP
#define MODELL_HPP

#include <random>
#include <cassert>


template<typename prec_t>
prec_t modell_identity(prec_t p) { return p; }


template<typename prec_t>
prec_t modell_multiply(prec_t a, prec_t b) { return a*b; }


template<typename cfg_vector_t, typename prec_t>
static prec_t bitvector_to_num(const cfg_vector_t &v)
{
    prec_t r = 0.0;
    using size_type = cfg_vector_t::size_type;
    for (size_type i = 0; i < v.data.size(); i++) {
        r *= 2.0;
        r += v[i];
    }

    return r / powl(2.0, v.data.size());
}


template<typename modell_t>
requires (modell_t::is_value_computation_implementation)
class inaccurate_modell
{
    public:
        using cfg_vector_t = modell_t::cfg_vector_type;
        using prec_type = modell_t::prec_type;
        using prec_t = prec_type;
        constexpr static bool is_value_computation_implementation
            = modell_t::is_value_computation_implementation;

        typedef struct {
            modell_t::prepared_computation pcom;

            std::lognormal_distribution<prec_t> error_distr;
        } prepared_computation;

    private:
        modell_t modell_;

        // lognormal distributed errors
        // std::lognormal_distribution<prec_t> error_distr; // (mu, sigma)

        const prec_t error_strength;

        const prec_t error_offset;

    public:

        inaccurate_modell(modell_t &modell_, prec_t error_strength_) :
            modell_(modell_),
            error_strength(error_strength_),
            // error_distr(0, error_strength_),
            error_offset(exp(error_strength_*error_strength_/2) - 1)
        {}

        void prepare_computation_with_cfg_vector(
            const cfg_vector_t &v, prepared_computation &pcom)
        {
            modell_.prepare_computation_with_cfg_vector(v, pcom.pcom);
            pcom.error_distr =
                std::lognormal_distribution<prec_t>(0, error_strength);
        }

        template<typename prng_t, typename rpmanager_t>
        auto operator()(
            prng_t &prng,
            rpmanager_t &rpmanager,
            prepared_computation &pcom) const
        {
            // rpmanager.execute_rpc(modell_(prng, rpmanager, pcom.pcom))

            prec_t f = pcom.pcom.f;
            prec_t errf = pcom.error_distr(prng) - error_offset;

            return rpmanager.prepare_call("modell_multiply", f, errf);
        }

        void finish_computation(prepared_computation &pcom) {
            modell_.finish_computation(pcom.pcom);
        }
};


template<typename cfg_vector_t, typename prec_t>
class modell
{
    public:
        using prec_type = prec_t;
        using cfg_vector_type = cfg_vector_t;
        constexpr static bool is_value_computation_implementation = true;

        typedef struct {
            prec_t f;
        } prepared_computation;

    private:
        const prec_t s;

        friend inaccurate_modell<modell<cfg_vector_t, prec_t>>;

    public:
        modell(prec_t s_) : s(s_) {}


        void prepare_computation_with_cfg_vector(
            const cfg_vector_t &v, prepared_computation &pcom)
        {
            prec_t x = bitvector_to_num<cfg_vector_t, prec_t>(v);
            assert(0 <= x);
            assert(x <= 1);

            static const prec_t nf = (cosh(s) - 3.*sinh(s) - 1);
            pcom.f = 1./2. + sinh(s - 2.*s*x) / nf;
        }

        template<typename prng_t, typename rpmanager_t>
        auto operator()(
            prng_t &prng,
            rpmanager_t &rpmanager,
            prepared_computation &pcom) const
        {
            ignore(prng);

            return rpmanager.prepare_call("modell_identity", pcom.f);
        }

        void finish_computation(prepared_computation &pcom) {
            ignore(pcom);
        }
};




#endif
```

7.23 probsat-with-resolvents.hpp

```
#ifndef PROBSAT_WITH_RESOLVENTS_HPP
#define PROBSAT_WITH_RESOLVENTS_HPP

#include <filesystem>
#include <string>
#include <cstddef>

#include "../configuration/configuration_vector.hpp"

#include "utility/resolvents_manager.hpp"
#include "utility/probsat-execution.hpp"

#include "../../util/util.hpp"
```

```
#include "../../util/shared_tmpfile.hpp"


template<typename cfg_vector_t, typename prec_t, bool use_bitfield>
requires (
    std::is_same<binary_configuration_vector<use_bitfield>,
    cfg_vector_t>::value
)
class probsat_resolvents
{
    public:
        using prec_type = prec_t;
        using cfg_vector_type = cfg_vector_t;
        constexpr static bool is_value_computation_implementation = true;

    private:
        std::size_t cnf_num_vars;

        std::size_t cnf_num_clauses;

        std::string cnf_file;

        std::string cnf_file_header;

        using seed_distr_t = std::uniform_int_distribution<probsat_seed_t>;

    public:
        std::string workgroup_description;


        typedef struct {
            shared_tmpfile stmpfile;

            seed_distr_t seed_distr;
        } prepared_computation;


        resolvents_manager resolvents;


        probsat_resolvents(
            std::string cnf_filename,
            std::string resolvents_fn
        ) :
            cnf_num_vars(0),
            cnf_num_clauses(0),
            cnf_file(),
            cnf_file_header(),
            resolvents()
        {
            std::ptrdiff_t fsize = slurp_file(cnf_file, cnf_filename);
            if ((0 > fsize) || ((std::size_t) fsize != cnf_file.size()))
            {
                throw std::runtime_error(
                    "cant read cnf file " + cnf_filename);
            }

            std::string cfg_marker = "p cnf ";
            std::size_t pos = cnf_file.find(cfg_marker);
            std::size_t eol = cnf_file.find("\n", pos+1, 1);
            if ((std::string::npos == pos) || (std::string::npos == eol))
            {
                throw std::runtime_error(
                    "can’t find config line starting with ’"
                    + cfg_marker + "’ in cnf file " + cnf_filename);
            }
            cnf_file[pos] = ’c’; // comment out the cnf line
            std::string buf = cnf_file.substr(
                pos+cfg_marker.length(), eol-pos-cfg_marker.length());
            std::istringstream iss(buf);
            iss » cnf_num_vars;
            iss » cnf_num_clauses;

            cnf_file_header = cnf_file.substr(0, eol+1);
            cnf_file = cnf_file.substr(eol+1);

            if (!resolvents.load(resolvents_fn)) {
                throw std::runtime_error(
                    "cant read resolvents file " + resolvents_fn);
            }
        }

        // ~probsat_resolvents() {
            // std::cout « "~probsat_resolvents" « std::endl;
        // }

        probsat_resolvents(const probsat_resolvents &other) = delete;


        void prepare_computation_with_cfg_vector(
            const cfg_vector_t &v, prepared_computation &pcom)
        {
            pcom.seed_distr.reset();

            pcom.stmpfile = std::move(shared_tmpfile(
                "sat-with-resolvents-" + workgroup_description));
            create_cnf_file_from_cfg_vector(v, pcom.stmpfile.fptr);
        }


        template<typename prng_t, typename rpmanager_t>
        auto operator()(
            prng_t &prng,
            rpmanager_t &rpmanager,
            prepared_computation &pcom) const
        {
            std::string satfile = pcom.stmpfile.fname;
            probsat_seed_t seed = pcom.seed_distr(prng);

            // std::cout « "c PWR preparing "
                // « satfile « " with seed " « seed « std::endl;

            return rpmanager.prepare_call(
                "execute_probsat", satfile, seed);
        }

        void finish_computation(prepared_computation &pcom) {
            pcom.stmpfile.remove();
        }

    private:

        FILE *create_cnf_file_from_cfg_vector(
            const cfg_vector_t &v, FILE *fptr) const
        {
            // TODO
            assert(v.data.size() <= resolvents.get_num_resolvents());
            // assert(v.data.size() == resolvents.get_num_resolvents());

            // FILE *fptr = tmpfile();
            assert(nullptr != fptr);

            std::size_t num_clauses = cnf_num_clauses;
            for (typename cfg_vector_t::size_type i = 0;
                i < v.data.size(); i++)
            {
                if (0 != v.data[i]) {
                    num_clauses++;
                }
            }

            assert(cnf_file_header.size() ==
                fwrite(
                    cnf_file_header.data(),
                    1,
                    cnf_file_header.size(),
                    fptr)
                || perror_("fwrite failed"));
            std::string cfg_line = std::string("p cnf ")
                + std::to_string(cnf_num_vars) + std::string(" ")
                + std::to_string(num_clauses) + std::string("\n");

            cfg_line = "c after adding resolvents:\n" + cfg_line;

            assert(cfg_line.size() ==
                fwrite(cfg_line.data(), 1, cfg_line.size(), fptr)
                || perror_("fwrite failed"));

            assert(cnf_file.size() ==
                fwrite(cnf_file.data(), 1, cnf_file.size(), fptr)
                || perror_("fwrite failed"));

            typename cfg_vector_t::size_type num_clauses_to_write
                = num_clauses - cnf_num_clauses;
            for (typename cfg_vector_t::size_type i = 0;
                i < v.data.size(); i++)
            {
                if (0 != v.data[i]) {
                    std::string rv = resolvents.get_resolvent(i);
                    if (1 < num_clauses_to_write) {
                        rv += "\n";
                    }

                    assert(rv.size() ==
                        fwrite(rv.data(), 1, rv.size(), fptr));
                    num_clauses_to_write--;
                }
            }

            assert(0 == num_clauses_to_write);

            assert(0 == fflush(fptr) || perror_("fflush failed"));

            return std::move(fptr);
        }
};

#endif

```

7.24 probsat-execution.cpp

```

#include <chrono>
#include <string>
#include <iostream>
#include <cstdio>
#include <cstddef>
#include <sys/select.h>
#include <stdio.h>

#include "probsat-execution.hpp"

#include "../../../config.hpp"
#include "../../../util/util.hpp"


uint8_t execute_probsat_(
    uint64_t &num_flips,
    const std::string &filename,
    const probsat_seed_t seed,
    const uint64_t max_flips,
    const std::chrono::seconds &max_exec_time,
    std::string *debug_probsat_output)
{
    uint8_t reason = 0;

    const std::string cmd =
        probsat_cmd +
        ((0 < max_flips) ?
            " --maxflips " + std::to_string(max_flips) : "")
        + " " + filename + " " + std::to_string(seed) + " 2>&1";

    num_flips = std::numeric_limits<uint64_t>::max();

    FILE *pipe_fh = nullptr;
    pipe_fh = popen(cmd.c_str(), "r");
    if (!pipe_fh) {
        perror("popen failed");
        return reason;
    }

    try {
        auto start = std::chrono::high_resolution_clock::now();

        int piped = fileno(pipe_fh);
        if (-1 == piped) {
            perror("fileno(pipe_fh)");
            throw std::runtime_error("fileno failed");
        }

        fcntl(piped, F_SETFL, O_NONBLOCK);
        if (-1 == piped) {
            perror("fcntl(piped, F_SETFL, O_NONBLOCK);");
            throw std::runtime_error("fcntl failed");
        }

        const std::string marker = "c numFlips";
        size_t bsize = 4096;
        assert(bsize <= std::numeric_limits<ssize_t>::max());
        char buffer[bsize];
        std::string bstr = "";

        fd_set readfds;
        FD_ZERO(&readfds);
        FD_SET(piped, &readfds);

        timespec timeout;
        timespec *timeout_ptr =
            (std::chrono::seconds::zero() == max_exec_time) ?
                nullptr : &timeout;
        assert(std::chrono::ceil<std::chrono::seconds>(
            max_exec_time - std::chrono::seconds::zero()).count()
            <= std::numeric_limits<decltype(timeout.tv_sec)>::max());

        bool running = fd_is_valid(piped);
        while (running)
        {
            auto now = std::chrono::high_resolution_clock::now();
            auto current_execution_time = now - start;

            if (nullptr != timeout_ptr) {
                timeout.tv_sec = (decltype(timeout.tv_sec)) std::max(0l,
                    std::chrono::ceil<std::chrono::seconds>(
                        max_exec_time - current_execution_time).count());
                timeout.tv_nsec = 1;
            }

            int rval = pselect(
                piped+1, &readfds, nullptr, nullptr, timeout_ptr, nullptr);
            switch (rval) {
                case 0: // timeout
                    now = std::chrono::high_resolution_clock::now();
                    current_execution_time =
                        std::chrono::duration_cast<std::chrono::seconds>
                        (now - start);
                    if (max_exec_time <= current_execution_time) {
                        std::cerr « "timeout after "
                            « current_execution_time.count()
                            « " seconds" « std::endl;
                        running = false;
                        reason = 1;
                    }
                    break;

                case 1: // data available
                    while (running)
                    {
                        ssize_t num_chars_read =
                            read(piped, &buffer[0], bsize);
                        buffer[num_chars_read] = 0; // null terminate

                        if (0 > num_chars_read)
                        {
                            if ((errno != EAGAIN)
                                and (errno != EWOULDBLOCK))
                            {
                                perror("read");
                                throw std::runtime_error("read failed");
                                running = false;
                            } else {
                                break;
                            }
                        } else {
                            if (nullptr != debug_probsat_output) {
                                *debug_probsat_output +=
                                    std::string(&buffer[0]) + "|BLOCK|";
                            }
                        }

                        if (0 < num_chars_read)
                        {
                            std::size_t offset = bstr.length();
                            std::string bfs = std::string(&buffer[0]);
                            bstr += bfs;
                            assert(bfs.length()
                                == (std::size_t) num_chars_read);

                            while (true) {
                                size_t pos = bstr.find("\n", offset);
                                if (pos == std::string::npos) {
                                    break;
                                }

                                std::string line = bstr.substr(0, pos);
                                bstr =
                                    bstr.substr(pos+1, std::string::npos);
                                offset = 0;

                                // std::cout « "c probsat: "
                                    // « line « std::endl;

                                if (line.starts_with(marker)) {
                                    size_t pos =
                                        line.find(": ", marker.length());
                                    assert(pos != std::string::npos);

                                    std::string num_flips_str =
                                        line.substr(pos + 2);
                                    assert(!num_flips_str.empty());

                                    num_flips = from_string<std::size_t>
                                        (num_flips_str);
                                    running = false;
                                    reason = 255;
                                    break;
                                }
                            }
                        }
                    }

                    if (running) {
                        running = fd_is_valid(piped);
                    }
                    break;

                default:
                    perror("pselect");
                    throw std::runtime_error("pselect failed");
                    running = false;
                    break;
            };
        }
    } catch(std::exception &ex) {
        std::cerr « "exception executing probsat:\n";
        if (!cmd.empty()) { std::cerr « cmd « "\n"; }
        std::cerr « ex.what() « std::endl;
    }

    if (pipe_fh) {
        if (nullptr != debug_probsat_output) {
            *debug_probsat_output += "|CLOSE|";
        }

        auto rval = pclose(pipe_fh);
        if (-1 == rval) {
            perror("pclose(pipe_fh)");
        }
        pipe_fh = nullptr;
    }

    return reason;
}


std::string execute_cmd(std::string cmd)
{
    FILE *pipe_fh = nullptr;
    pipe_fh = popen(cmd.c_str(), "r");
    if (!pipe_fh) {
        perror("popen failed");
        return "popen failed";
    }

    std::string content;
    try {
        char c;
        while ((c = fgetc(pipe_fh)) != EOF) {
            content += c;
        }
    } catch(std::exception &ex) {
        std::cerr « "exception executing probsat:\n";
        if (!cmd.empty()) { std::cerr « cmd « "\n"; }
        std::cerr « ex.what() « std::endl;
        return content + ex.what();
    }

    if (pipe_fh) {
        auto rval = pclose(pipe_fh);
        if (-1 == rval) {
            perror("pclose(pipe_fh)");
        }
        pipe_fh = nullptr;
    }

    return content;
}

std::pair<uint64_t, probsat_return_cause::reason> execute_probsat
    (std::string filename, probsat_seed_t seed)
{
    using reason_t = probsat_return_cause::reason;
    try {
        uint64_t num_flips = std::numeric_limits<uint64_t>::max();

        uint8_t reason = execute_probsat_(
            num_flips,
            filename,
            seed,
            probsat_max_flips,
            probsat_max_exec_time);

        if (1 == reason) {
            return std::make_pair<uint64_t, reason_t>(
                std::move(num_flips), probsat_return_cause::TIMEOUT);
        } else if (255 == reason) {
            if ((0 < num_flips) && (num_flips == probsat_max_flips)) {
                return std::make_pair<uint64_t, reason_t>(
                    std::move(num_flips), probsat_return_cause::MAX_FLIPS);
            } else {
                return std::make_pair<uint64_t, reason_t>(
                    std::move(num_flips), probsat_return_cause::SUCCESS);
            }
        }
    } catch(std::exception &ex) {
        std::cerr « "exception executing probsat:\n";
        std::cerr « ex.what() « std::endl;
    }

    return std::make_pair<uint64_t, reason_t>(
        std::numeric_limits<uint64_t>::max(), probsat_return_cause::ERROR);
}



```

7.25 probsat-execution.hpp

```
#ifndef PROBSAT_EXECUTION_HPP
#define PROBSAT_EXECUTION_HPP


#include <chrono>
#include <string>
#include <iostream>
#include <cstdio>
#include <cstddef>
#include <sys/select.h>

#include "../../../config.hpp"
#include "../../../util/util.hpp"

#include "../../../util/basic_serialization.hpp"


namespace probsat_return_cause {

    enum reason : uint8_t {
        ERROR = 0,
        TIMEOUT = 1,
        MAX_FLIPS= 2,
        SUCCESS = 3,
        NUM_REASONS = 4
    };

    const static std::string as_string[reason::NUM_REASONS] = {
        "ERROR",
        "TIMEOUT",
        "MAX_FLIPS",
        "SUCCESS"
    };

    template<typename S>
    void serialize(S &s, reason &r) {
        uint8_t v = r;
        generic_serialize(s, v);
        assert(v <= probsat_return_cause::NUM_REASONS);
        r = static_cast<reason>(v);
    }
};


uint8_t execute_probsat_(
    uint64_t &num_flips,
    const std::string &filename,
    const probsat_seed_t seed,
    const uint64_t max_flips = 0,
    const std::chrono::seconds &max_exec_time
        = std::chrono::seconds::zero(),
    std::string *debug_probsat_output = nullptr);


std::pair<uint64_t, probsat_return_cause::reason>
    execute_probsat(std::string filename, probsat_seed_t seed);


template<typename prec_t = prec_t>
struct result_statistics
{
    std::size_t probsat_executions = 0;

    std::size_t reasons[probsat_return_cause::NUM_REASONS] = {0};

    double total_flips_executed = 0;


    prec_t operator()(
        std::pair<uint64_t, probsat_return_cause::reason> result)
    {
        prec_t num_flips = static_cast<prec_t>(result.first);
        probsat_executions++;
        assert(result.second < probsat_return_cause::NUM_REASONS);
        reasons[result.second]++;

        if (std::numeric_limits<uint64_t>::max() == num_flips) {
            num_flips = std::numeric_limits<prec_t>::quiet_NaN();
        } else {
            total_flips_executed += num_flips;
        }

        if ((interpret_timeout_as_max_flips_reached)
            && (0 < probsat_max_flips))
        {
            if (result.second == probsat_return_cause::TIMEOUT) {
                num_flips = probsat_max_flips;
            }
        }

        if (result.second != probsat_return_cause::SUCCESS) {
            std::cerr « "probsat executed with " «
                probsat_return_cause::as_string[result.second]
                « " and " « num_flips « " flips" « std::endl;
        }

        // return std::log(num_flips);
        return num_flips;
    };

    friend std::ostream &operator«(
        std::ostream &os, const result_statistics &rs)
    {
        namespace prc = probsat_return_cause;
        // new version is easier parseable from the command line
        os « rs.probsat_executions « " executions, "
            « rs.reasons[prc::SUCCESS] « " successfull, "
            « rs.reasons[prc::MAX_FLIPS] « " max_flips_reached, "
            « rs.reasons[prc::TIMEOUT] « " terminated_by_timeout, "
            « rs.reasons[prc::ERROR] « " with_errors, "
            « rs.total_flips_executed « " flips_in_total_(at_least)";
/* old version
        os « rs.probsat_executions « " executions ("
            « rs.reasons[prc::SUCCESS] « " successfull, "
            « rs.reasons[prc::MAX_FLIPS] « " max flips reached, "
            « rs.reasons[prc::TIMEOUT] « " terminated by timeout, "
            « rs.reasons[prc::ERROR] « " with errors, "
            « "and at least " «
                rs.total_flips_executed « " flips in total)";
*/
        return os;
    }

};


#endif
```

7.26 resolvents_manager.cpp

```

#include <string>
#include <vector>
#include <fstream>
#include <iostream>
#include <cassert>

#include "resolvents_manager.hpp"

bool resolvents_manager::load(std::string filename) {
    std::ifstream file(filename, std::ifstream::in);

    if (false == file.is_open() ) {
        std::string errmsg = std::string("can’t open ") + filename;
        perror(errmsg.c_str());
        return false;
    }

    while (file.good()) {
        std::string line;
        std::getline(file, line);
        if ((false == line.starts_with("c ")) && (!line.empty())) {
            assert(line.ends_with(" 0"));
            resolvent.push_back(line);
        }
    }

    file.close();

    return true;
}

std::size_t resolvents_manager::get_num_resolvents() const {
    return resolvent.size();
}

std::string resolvents_manager::get_resolvent(std::size_t i) const {
    return resolvent[i];
}

```

7.27 resolvents_manager.hpp

```
#ifndef RESOLVENTS_MANAGER_HPP
#define RESOLVENTS_MANAGER_HPP

#include <string>
#include <vector>

class resolvents_manager {
    private:
        std::vector<std::string> resolvent;

    public:
        resolvents_manager() = default;

        bool load(std::string filename);

        std::size_t get_num_resolvents() const;

        std::string get_resolvent(std::size_t i) const;
};

#endif
```

7.28 basic_serialization.hpp

```
#ifndef BASIC_SERIALIZATION_HPP
#define BASIC_SERIALIZATION_HPP

#include <cassert>
#include <string>
#include <tuple>
#include <vector>

template <typename S, typename basic_t>
void serialize_basic_type(S& s, basic_t &v) {
    if constexpr (1 == sizeof(basic_t)) {
        s.value1b(v);
    } else if constexpr (2 == sizeof(basic_t)) {
        s.value2b(v);
    } else if constexpr (4 == sizeof(basic_t)) {
        s.value4b(v);
    } else if constexpr (8 == sizeof(basic_t)) {
        s.value8b(v);
    }
    #if DEBUG_ASSERTIONS
    else {
        assert(false);
    }
    #endif

    static_assert(
        (1 == sizeof(basic_t)) ||
        (2 == sizeof(basic_t)) ||
        (4 == sizeof(basic_t)) ||
        (8 == sizeof(basic_t)) );

    // std::cout « "basic type of size " « sizeof(basic_t) « ":" « v « std::endl;
}


template<typename S>
void serialize_marker(
    S &s, const std::string marker, const std::size_t length)
{
    // length must contain null byte!
    std::string marker_ = marker;
    s.text1b(marker_, length);
    assert(marker == marker_);
}


template<typename S, typename data_t> requires
((1 == sizeof(data_t)) && (false == std::is_same<bool, data_t>::value))
void serialize(S &s, std::vector<data_t> &v)
{
    std::size_t length = v.size();
    serialize_basic_type<S, std::size_t>(s, length);
    v.resize(length);
    s.container1b(v, length);
    assert(v.size() == length);
}


template<typename S>
void generic_serialize(S &s) { ignore(s); }

template<typename S, typename first_t, typename... arg_types>
void generic_serialize(S &s, first_t &a, arg_types &... args);

template<typename S, typename first_t, typename second_t>
void serialize(S &s, std::pair<first_t, second_t> &v) {
    generic_serialize(s, v.first, v.second);
}

template<typename S, typename first_t, typename... arg_types>
void generic_serialize(S &s, first_t &a, arg_types &... args)
{
    // serialize first argument
    if constexpr (std::is_fundamental<first_t>::value) {
        serialize_basic_type<S, first_t>(s, a);
    } else if constexpr (std::is_same<std::string, first_t>::value) {
        auto length = a.size();
        generic_serialize(s, length);
        a.reserve(length);
        s.text1b(a, length);
        assert(a.size() == length);
    } else if constexpr (std::is_same<const char *, first_t>::value) {
        // one way trap for serialization of const char *
        // note: when used with remote procedure calls
        // the function parameter must be std::string,
        // but the argument passed may const char *
        std::string tmp_str = a;
        generic_serialize(s, tmp_str);
    } else {
        serialize(s, a);
    }

    // recursively pass further arguments
    generic_serialize(s, args...);
}


```

```
template<typename S>
void serialize(S &s, std::tuple<> &v) {
    ignore(s, v);
}

/*
template<typename S, typename head_arg_type, typename... tail_arg_types>
void serialize(S &s, std::tuple<head_arg_type, tail_arg_types...> &v) {
    head_arg_type tmp = std::get<0>(v);
    generic_serialize<S, head_arg_type>(s, tmp);
    if constexpr (0 < std::tuple_size<std::tuple<tail_arg_types...»::value)
    {
        auto t = tail(v);
        serialize<S, tail_arg_types...>(s, t);
        v = std::tuple_cat(std::make_tuple(tmp), t);
    } else {
        v = std::make_tuple(tmp);
    }
}
*/

template<typename S, typename... arg_types>
void serialize(S &s, std::tuple<arg_types...> &t) {
    apply([&s](auto &... args) { generic_serialize<S, arg_types...>(s, args...); }, t);
}


template<typename t>
struct make_generic_serializeable {
    t data;
};

template<typename S, typename t>
void serialize(S &s, make_generic_serializeable<t> &o) {
    generic_serialize(s, o.data);
}


/*
#include "util.hpp"

template<typename S>
void serialize(S &s, std::vector<bool> &v) {
    auto content = vector_to_string(v);
    std::cout « "content length before: " « content.length() « std::endl;
    // ignore(content);
    generic_serialize<S, std::string>(s, content);
    std::cout « "content length after: " « content.length() « std::endl;
    v = vector_from_string(content);
}
*/

template<typename S>
void serialize(S &s, std::vector<bool> &v) {
    std::size_t length = v.size();
    serialize_basic_type<S, std::size_t>(s, length);
    v.resize(length);

    for (std::size_t i = 0; i < length; i++) {
        uint8_t value = v[i] ? 1 : 0;
        s.value1b(value);
        v[i] = (0 != value);
    }

    // s.container1b(v, length);
    assert(v.size() == length);
}

#endif
```

7.29 bitfield.hpp

```
#ifndef BITFIELD_HPP
#define BITFIELD_HPP

#include <cstdint>
#include <cassert>
#include <type_traits>
#include <ostream>

template<typename length_t>
class bitfield {
    private:
        length_t length;

        // helper values to allow returning references to memory

        length_t last_mod_index;

        bool last_mod_value;

        uint8_t *array;


        // helper functions

        static inline length_t to_bytelength(const length_t l) {
            return (l+7) » 3;
        }

        inline bool get(const length_t index) const
        {
            #if DEBUG_ASSERTIONS
            assert(index < length);
            #endif

            return 1 & (array[index » 3] » (index & 0b111));
        }

        inline void set(const length_t index, const bool value)
        {
            update();

            #if DEBUG_ASSERTIONS
            assert(index < length);
            #endif

            const uint8_t mask = 1 « (index & 0b111);
            if (value) {
                array[index » 3] |= mask;
            } else {
                array[index » 3] &= ~mask;
            }
        }

        // apply changes by returned reference to array
        inline void update() {
            if (last_mod_index < length) {
                // std::cout « "a["«(int)last_mod_index«"] = "
                    // « (int)last_mod_value « std::endl;
                set(last_mod_index, last_mod_value);
                last_mod_index = length;
            }
        }

    public:
        // constructors

        bitfield() :
            length(0),
            last_mod_index(length),
            last_mod_value(false),
            array(nullptr) {}

        bitfield(length_t n) :
            length(n), last_mod_index(length), last_mod_value(false),
            array(length > 0 ? new uint8_t[to_bytelength(length)] : nullptr) {}

        // bitfield(const bitfield &other) = delete;

        bitfield(const bitfield &other) :
            length(other.length),
            last_mod_index(other.last_mod_index),
            last_mod_value(other.last_mod_value),
            array(length > 0 ? new uint8_t[to_bytelength(length)] : nullptr)
        {
            for (length_t i = 0; i < to_bytelength(length); i++) {
                array[i] = other.array[i];
            }
        }

        ~bitfield() {
            if (nullptr != array) {
                #if DEBUG_ASSERTIONS
                    assert(0 < length);
                #endif
                delete[] array;
                array = nullptr;
            } else {
                #if DEBUG_ASSERTIONS
                    assert(0 == length);
                #endif
            }
        }

        // copy & swap idiom

        friend void swap(bitfield &a, bitfield &b)
            noexcept(std::is_nothrow_swappable_v<length_t>
                &&   std::is_nothrow_swappable_v<bool>
                &&   std::is_nothrow_swappable_v<uint8_t*>)
        {
            #if DEBUG_ASSERTIONS
            assert(a.length == b.length);
            #endif
            // or remove const from length
            // std::swap(a.length, b.length);
            std::swap(a.last_mod_index, b.last_mod_index);
            std::swap(a.last_mod_value, b.last_mod_value);
            std::swap(a.array, b.array);
        }

        bitfield(bitfield &&other) : bitfield() {
            swap(*this, other);
        }

        bitfield& operator=(bitfield other) {
            swap(*this, other);
            return *this;
        }

        void resize(length_t new_length)
        {
            update();

            if (to_bytelength(new_length) > to_bytelength(length)) {
                assert(0 < to_bytelength(new_length));

                uint8_t *old_array = array;

                array = new uint8_t[to_bytelength(new_length)];

                for (length_t i = 0; i < to_bytelength(length); i++) {
                    array[i] = old_array[i];
                }

                if (old_array) {
                    delete[] old_array;
                }
            }

            length = new_length;
            last_mod_index = length;
        }

        length_t get_bytelength() const {
            return to_bytelength(length);
        }

        uint8_t *data() {
            update();
            return array;
        }

        // access operators

        inline bool& operator[](const length_t index)
        {
            update();

            #if DEBUG_ASSERTIONS
            assert(index < length);
            #endif

            last_mod_index = index;
            last_mod_value = get(index);
            return last_mod_value;
        }

        inline bool operator[](const length_t index) const {
            // dont apply changes, so this function can be const
            if (index == last_mod_index) {
                return last_mod_value;
            }

            return get(index);
        }
```

```

        length_t size() const {
            return length;
        }

        // output

        std::ostream &print(std::ostream &os, const char *separator = " ")
        {
            update();

            const length_t bl = to_bytelength(length);

            auto sep = "";
            for (length_t i = 0; i < bl-1; i++) {
                os « sep;

                uint8_t d = array[i];
                for (uint8_t j = 0; j < 8; j++) {
                    os « ((d & 1) ? "1" : "0");
                    d »= 1;
                }

                sep = separator;
            }

            if (0 < bl) {
                os « sep;
                for (length_t i = (bl-1)*8; i < length; i++) {
                    os « (get(i) ? "1" : "0");
                }
            }

            return os;
        }
};

template<typename length_t>
std::ostream &operator«(std::ostream& os, bitfield<length_t> bf) {
    return bf.print(os);
}

template<typename S, typename length_t>
void serialize(S &s, bitfield<length_t> &bf) {
    bf.update();

    length_t size = bf.length;
    serialize_basic_type<S, length_t>(s, size);

    bf.resize(size);

    uint8_t *dataptr = bf.data();
    for (std::size_t i = 0; i < bf.get_bytelength(); i++) {
        s.value1b(dataptr[i]);
    }

    assert(bf.size() == size);
}

#endif
```

7.30 debug.hpp

```
#ifndef DEBUG_HPP
#define DEBUG_HPP

/*
    attach gdb in realtime to current process and print stack trace
    extremely useful for multi-process environments like mpi

    **complete code in this file is from**

    https://stackoverflow.com/questions/4636456/how-to-get-a-stack-trace-for-c-using-gcc-with-line-number-information
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
#include <sys/prctl.h>

bool print_trace() {
    char pid_buf[30];
    sprintf(pid_buf, "%d", getpid());
    char name_buf[512];
    name_buf[readlink("/proc/self/exe", name_buf, 511)]=0;
    prctl(PR_SET_PTRACER, PR_SET_PTRACER_ANY, 0, 0, 0);
    int child_pid = fork();
    if (!child_pid) {
        dup2(2,1); // redirect output to stderr - edit: unnecessary?
        execl("/usr/bin/gdb", "gdb", "--batch", "-n", "-ex", "thread", "-ex", "bt", name_buf, pid_buf, NULL);
        abort(); /* If gdb failed to start */
    } else {
        waitpid(child_pid,NULL,0);
    }

    return false;
}

#endif
```

7.31 include_bitsery.hpp

```
#ifndef INCLUDE_BITSERY_HPP
#define INCLUDE_BITSERY_HPP

#include <bitsery/bitsery.h>
#include <bitsery/adapter/buffer.h>
#include <bitsery/traits/vector.h>
#include <bitsery/traits/string.h>
#include <bitsery/ext/std_tuple.h>

#endif
```

7.32 mpi_async_communication.cpp

```

#include <cassert>
#include <algorithm>

#include "mpi_async_communication.hpp"


void async_comm::wait_for_one_message()
{
    std::size_t size = outstanding_requests.size();
    if (0 >= size) { return; }

    int index = MPI_UNDEFINED;
    assert(MPI_SUCCESS == MPI_Waitany(
        size, outstanding_requests.data(), &index, MPI_STATUSES_IGNORE));

    if (MPI_UNDEFINED != index) {
        assert((0 <= index) && ((std::size_t) index < size));
        outstanding_requests.erase(outstanding_requests.begin() + index);
        outstanding_data.erase(outstanding_data.begin() + index);
    }
}


void async_comm::wait_for_all_messages()
{
    std::size_t size = outstanding_requests.size();
    if (0 >= size) { return; }

    assert(MPI_SUCCESS == MPI_Waitall(
        size, outstanding_requests.data(), MPI_STATUSES_IGNORE));
}


void async_comm::wait_for_some_messages()
{
    std::size_t size = outstanding_requests.size();
    if (0 >= size) { return; }

    int count_finished = 0;
    std::vector<int> indices_finished(size);

    assert(MPI_SUCCESS == MPI_Waitsome(size, outstanding_requests.data(),
        &count_finished, indices_finished.data(), MPI_STATUS_IGNORE));

    indices_finished.resize(count_finished);
    sort(indices_finished.begin(), indices_finished.end(), std::greater<>());

    for (int i = 0; i < count_finished; i++) {
        auto index = indices_finished[i];
        assert(index != MPI_UNDEFINED);
        outstanding_requests.erase(outstanding_requests.begin() + index);
        outstanding_data.erase(outstanding_data.begin() + index);
    }
}


void async_comm::test_for_one_message()
{
    std::size_t size = outstanding_requests.size();
    if (0 >= size) { return; }

    int index = MPI_UNDEFINED;
    int completed = 0;

    assert(MPI_SUCCESS == MPI_Testany(
        size,
        outstanding_requests.data(),
        &index,
        &completed,
        MPI_STATUS_IGNORE));

    if (completed) {
        assert(index != MPI_UNDEFINED);
        assert(MPI_REQUEST_NULL == outstanding_requests[index]);
        // assert(MPI_SUCCESS == MPI_Wait(&outstanding_requests[index], MPI_STATUS_IGNORE));
        outstanding_requests.erase(outstanding_requests.begin() + index);
        outstanding_data.erase(outstanding_data.begin() + index);
    }
}


void async_comm::test_for_messages()
{
    std::size_t size = outstanding_requests.size();

    if (0 >= size) { return; }
    if (1 == size) { return test_for_one_message(); }

    int count_finished = 0;
    std::vector<int> indices_finished(size);
    assert(MPI_SUCCESS == MPI_Testsome(size, outstanding_requests.data(),
        &count_finished, indices_finished.data(), MPI_STATUS_IGNORE));

    indices_finished.resize(count_finished);
    sort(indices_finished.begin(), indices_finished.end(), std::greater<>());

    for (int i = 0; i < count_finished; i++) {
        auto index = indices_finished[i];
        assert(index != MPI_UNDEFINED);
        assert(MPI_REQUEST_NULL == outstanding_requests[index]);
        // assert(MPI_SUCCESS == MPI_Wait(
            // &outstanding_requests[index], MPI_STATUS_IGNORE));
        outstanding_requests.erase(outstanding_requests.begin() + index);
        outstanding_data.erase(outstanding_data.begin() + index);
    }
}


auto async_comm::communications_in_queue() const
{
    auto num_waiting = outstanding_requests.size();
    #if DEBUG_ASSERTIONS
    assert(num_waiting == outstanding_data.size());
    #endif
    return num_waiting;
}


bool async_comm::communication_is_done() const
{
    bool is_empty = outstanding_requests.empty();
    #if DEBUG_ASSERTIONS
    assert(is_empty == outstanding_data.empty());
    #endif
    return is_empty;
}


async_comm::~async_comm()
{
    // wait_for_all_messages();
    #if DEBUG_ASSERTIONS
    assert(outstanding_requests.empty() == outstanding_data.empty());
    assert(communication_is_done());
    #endif
}



void async_comm_out::send_message(
    const MPI_Comm &comm, int dest, int tag, data_t &&content)
{
    MPI_Request request;
    assert(MPI_SUCCESS == MPI_Isend(
        content.data(),
        content.size(),
        MPI_BYTE,
        dest,
        tag,
        comm,
        &request));

    int completed = 0;
    assert(MPI_SUCCESS == MPI_Test(
        &request, &completed, MPI_STATUS_IGNORE));

    if (!completed) {
        outstanding_requests.emplace_back(std::move(request));
        outstanding_data.emplace_back(std::move(content));
    } else {
        assert(MPI_REQUEST_NULL == request);
        // assert(MPI_SUCCESS == MPI_Wait(&request, MPI_STATUS_IGNORE));
    }
}



void fake_async_comm_out::send_message(
    const MPI_Comm &comm, int dest, int tag, auto content)
{
    assert(MPI_SUCCESS == MPI_Send(
        content.data(), content.size(), MPI_BYTE, dest, tag, comm));
}

```

7.33 mpi_async_communication.hpp

```
#ifndef MPI_ASYNC_COMMUNICATION_HPP
#define MPI_ASYNC_COMMUNICATION_HPP

#include <vector>
#include <mpi.h>

class async_comm
{
    protected:
        using data_t = std::vector<uint8_t>;

        std::vector<MPI_Request> outstanding_requests;

        std::vector<data_t> outstanding_data;

    public:
        void wait_for_one_message();

        void wait_for_all_messages();

        void wait_for_some_messages();

        void test_for_one_message();

        void test_for_messages();

        auto communications_in_queue() const;

        bool communication_is_done() const;

        ~async_comm();
};

class async_comm_out : public async_comm
{
    public:
        void send_message(
            const MPI_Comm &comm, int dest, int tag, data_t &&content);
};

class fake_async_comm_out : public async_comm
{
    public:
        void send_message(
            const MPI_Comm &comm, int dest, int tag, auto content);
};


#endif
```

7.34 mpi_shared_tmp_workgroup.cpp

```

#include <random>
#include <cassert>
#include <filesystem>

#include "mpi_shared_tmp_workgroup.hpp"

#include "mpi_types.hpp"
#include "mpi_util.hpp"
#include "../util.hpp"


std::string mpi_shared_tmp_dir_workgroup::
    build_tmp_dir_path(std::string app_name)
{
    std::string tmp_path = std::filesystem::temp_directory_path();
    tmp_path += "/mpi_split_by_shared_tmp/" + app_name + "-";

    // add shared random number to path
    uint32_t random = 0;

    if (0 == parent_comm_id) {
        std::random_device r;
        random = r();
    }

    assert(MPI_SUCCESS == MPI_Bcast(
        &random, 1, get_mpi<decltype(random)>::type(), 0, parent_comm));

    tmp_path += std::to_string(random);

    return tmp_path;
}


int mpi_shared_tmp_dir_workgroup::identify_workgroup_head()
{
    namespace fs = std::filesystem;

    // create random tmp path
    fs::create_directories(tmp_path);

    // ensure its empty
    assert(fs::is_empty(tmp_path));
    assert(MPI_SUCCESS == MPI_Barrier(parent_comm));

    // create dir with id inside
    assert(fs::create_directory(
        tmp_path + "/" + std::to_string(parent_comm_id)));
    assert(MPI_SUCCESS == MPI_Barrier(parent_comm));

    // find minimum id in local tmp dir
    int min_id = std::numeric_limits<int>::max();
    for (const auto & entry : fs::directory_iterator(tmp_path))
    {
        min_id = std::min(
            min_id, from_string<int>(entry.path().filename()));
    }

    // sanity checks
    assert(0 <= min_id);
    assert(min_id < parent_comm_size);

    // remove tmp directories if all are finished
    // note: /tmp/mpi_split_by_shared_tmp/ will persist
    MPI_Barrier(parent_comm);
    if (parent_comm_id == min_id) {
        std::filesystem::remove_all(tmp_path);
    }

    return min_id;
}


bool mpi_shared_tmp_dir_workgroup::is_workgroup_head() const {
    return parent_comm_id == parent_comm_workgroup_head_id;
}


mpi_shared_tmp_dir_workgroup::mpi_shared_tmp_dir_workgroup(
    MPI_Comm parent_comm,
    std::string app_name
) :
    parent_comm(parent_comm)
{
    parent_comm_id = mpi_get_comm_rank(parent_comm);
    parent_comm_size = mpi_get_comm_size(parent_comm);

    if (1 == parent_comm_size)
    { // shortcut in case of a single process
        assert(0 == parent_comm_id);
        parent_comm_workgroup_head_id = 0;
        workgroup_comm = parent_comm;
        workgroup_comm_id = 0;
        workgroup_comm_size = 1;
        mpi_comm_leaders = parent_comm;
        workgroup_count = 1;
        workgroup_id = 0;
        return;
    }

    tmp_path = build_tmp_dir_path(app_name);
    parent_comm_workgroup_head_id = identify_workgroup_head();

    assert(MPI_SUCCESS == MPI_Comm_split(
        parent_comm,
        parent_comm_workgroup_head_id,
        parent_comm_id,
        &workgroup_comm));

    workgroup_comm_id = mpi_get_comm_rank(workgroup_comm);
    workgroup_comm_size = mpi_get_comm_size(workgroup_comm);

    const bool is_head = is_workgroup_head();
    assert(MPI_SUCCESS == MPI_Comm_split(
        parent_comm,
        is_head ? 0 : 1,
        parent_comm_id,
        &mpi_comm_leaders));

    if (!is_head) {
        assert(MPI_SUCCESS == MPI_Comm_disconnect(&mpi_comm_leaders));
        mpi_comm_leaders = MPI_COMM_NULL;
    }
    assert(MPI_SUCCESS == MPI_Barrier(parent_comm));

    int bcast[2];
    if (is_head) {
        bcast[0] = mpi_get_comm_size(mpi_comm_leaders);
        bcast[1] = mpi_get_comm_rank(mpi_comm_leaders);
    }

    assert(MPI_SUCCESS == MPI_Bcast(
        &bcast, 2, get_mpi<int>::type(), 0, workgroup_comm));

    workgroup_count = bcast[0];
    workgroup_id = bcast[1];

    if (0 == parent_comm_id) { assert(is_head); }
}


mpi_shared_tmp_dir_workgroup::~mpi_shared_tmp_dir_workgroup()
{
    int was_finalized = 0;
    assert(MPI_SUCCESS == MPI_Finalized( &was_finalized ));
    if (!was_finalized) {
        assert(MPI_SUCCESS == MPI_Barrier(parent_comm));

        if (is_workgroup_head()) {
            assert(MPI_SUCCESS ==
                MPI_Comm_disconnect(&mpi_comm_leaders));
            mpi_comm_leaders = MPI_COMM_NULL;
        }

        assert(MPI_SUCCESS == MPI_Barrier(parent_comm));
        assert(MPI_SUCCESS == MPI_Comm_disconnect(&workgroup_comm));
        workgroup_comm = MPI_COMM_NULL;
    }
}

```

7.35 mpi_shared_tmp_workgroup.hpp

```
#ifndef MPI_SHARED_WORKGROUP_HPP
#define MPI_SHARED_WORKGROUP_HPP

#include <string>
#include <mpi.h>

class mpi_shared_tmp_dir_workgroup
{
    public:
        MPI_Comm parent_comm;

        int parent_comm_id;

        int parent_comm_size;

        int parent_comm_workgroup_head_id;


        MPI_Comm workgroup_comm;

        int workgroup_comm_id;

        int workgroup_comm_size;


        MPI_Comm mpi_comm_leaders;

        int workgroup_count;

        int workgroup_id;

    private:
        std::string tmp_path;

        std::string build_tmp_dir_path(std::string app_name);

        int identify_workgroup_head();

    public:
        bool is_workgroup_head() const;

        mpi_shared_tmp_dir_workgroup(
            MPI_Comm parent_comm = MPI_COMM_WORLD,
            std::string app_name = "main");

        ~mpi_shared_tmp_dir_workgroup();
};

#endif
```

7.36 mpi_types.hpp

```
#ifndef MPI_TYPES_HPP
#define MPI_TYPES_HPP

#include <mpi.h>

template<typename t>
struct get_mpi {
};

// exclude template specializations

// with definitions like the following all was working fine in the past... :/
// template<>
// struct get_mpi<std::byte> {
    // static constexpr auto type = MPI_BYTE;
// };

// but now hacks are required to make it work again, see:
```

```
// https://github.com/open-mpi/ompi/issues/10017
// therefore it was switched to runtime evaluation...


template<>
struct get_mpi<std::byte> {
    static auto type() { return MPI_BYTE; }
};

template<>
struct get_mpi<char> {
    static auto type() { return MPI_SIGNED_CHAR; }
};

template<>
struct get_mpi<unsigned char> {
    static auto type() { return MPI_UNSIGNED_CHAR; }
};

template<>
struct get_mpi<short> {
    static auto type() { return MPI_SHORT; }
};

template<>
struct get_mpi<unsigned short> {
    static auto type() { return MPI_UNSIGNED_SHORT; }
};

template<>
struct get_mpi<int> {
    static auto type() { return MPI_INT; }
};

template<>
struct get_mpi<unsigned int> {
    static auto type() { return MPI_UNSIGNED; }
};

template<>
struct get_mpi<long int> {
    static auto type() { return MPI_LONG; }
};

template<>
struct get_mpi<unsigned long int> {
    static auto type() { return MPI_UNSIGNED_LONG; }
};

template<>
struct get_mpi<long long int> {
    static auto type() { return MPI_LONG_LONG_INT; }
};

template<>
struct get_mpi<float> {
    static auto type() { return MPI_FLOAT; }
};

template<>
struct get_mpi<double> {
    static auto type() { return MPI_DOUBLE; }
};

template<>
struct get_mpi<long double> {
    static auto type() { return MPI_LONG_DOUBLE; }
};

// old code:


// template<>
// struct get_mpi<std::byte> {
    // static constexpr auto type = MPI_BYTE;
// };

// template<>
// struct get_mpi<char> {
    // static constexpr auto type = MPI_SIGNED_CHAR;
// };

// template<>
// struct get_mpi<unsigned char> {
    // static constexpr auto type = MPI_UNSIGNED_CHAR;
// };

// template<>
// struct get_mpi<short> {
    // static constexpr auto type = MPI_SHORT;
// };

// template<>
// struct get_mpi<unsigned short> {
    // static constexpr auto type = MPI_UNSIGNED_SHORT;
// };

// template<>
// struct get_mpi<int> {
    // static constexpr auto type = MPI_INT;
// };

// template<>
// struct get_mpi<unsigned int> {
    // static constexpr auto type = MPI_UNSIGNED;
// };

// template<>
// struct get_mpi<long int> {
    // static constexpr auto type = MPI_LONG;
// };

// template<>
// struct get_mpi<unsigned long int> {
    // static constexpr auto type = MPI_UNSIGNED_LONG;
// };

// template<>
// struct get_mpi<long long int> {
    // static constexpr auto type = MPI_LONG_LONG_INT;
// };

// template<>
// struct get_mpi<float> {
    // static constexpr auto type = MPI_FLOAT;
// };

// template<>
// struct get_mpi<double> {
    // static constexpr auto type = MPI_DOUBLE;
// };

// template<>
// struct get_mpi<long double> {
    // static constexpr auto type = MPI_LONG_DOUBLE;
// };


#endif
```

7.37 mpi_util.cpp

```

#include "mpi_util.hpp"
#include <cassert>

int mpi_get_comm_rank(const MPI_Comm &comm)
{
    int mpi_id = -1;
    assert(MPI_SUCCESS == MPI_Comm_rank(comm, &mpi_id));
    assert(0 <= mpi_id);
    return mpi_id;
}


int mpi_get_comm_size(const MPI_Comm &comm)
{
    int wg_size = -1;
    assert(MPI_SUCCESS == MPI_Comm_size(comm, &wg_size));
    assert(0 <= wg_size);
    return wg_size;
}

bool is_message_available(
    const MPI_Comm &comm, MPI_Status *status, int tag)
{
    int received_flag = 0;
    assert(MPI_SUCCESS ==
        MPI_Iprobe(MPI_ANY_SOURCE, tag, comm, &received_flag, status));
    return 0 != received_flag;
}

void wait_for_message(const MPI_Comm &comm, MPI_Status *status, int tag)
{
    assert(MPI_SUCCESS == MPI_Probe(MPI_ANY_SOURCE, tag, comm, status));
}

int get_message_size(MPI_Status *status)
{
    int length = -1;
    assert(MPI_SUCCESS == MPI_Get_count(status, MPI_BYTE, &length));
    assert(0 <= length);
    return length;
}

void get_message(
    const MPI_Comm &comm,
    MPI_Status *status,
    std::vector<uint8_t> &content)
{
    int length = get_message_size(status);
    content.resize(length);

    assert(MPI_SUCCESS == MPI_Recv(
        content.data(),
        length,
        MPI_BYTE,
        status->MPI_SOURCE,
        status->MPI_TAG,
        comm,
        status));

    // assert(get_message_size(status) == length);
}

void send_message(
    const MPI_Comm &comm,
    int dest,
    int tag,
    const std::vector<uint8_t> &content)
{
    assert(MPI_SUCCESS == MPI_Send(
        content.data(), content.size(), MPI_BYTE, dest, tag, comm));
}

void ping(const MPI_Comm &comm, int dest, int tag)
{
    assert(MPI_SUCCESS ==
        MPI_Send(nullptr, 0, MPI_BYTE, dest, tag, comm));
}

void accept_ping(const MPI_Comm &comm, MPI_Status *status)
{
    assert(MPI_SUCCESS == MPI_Recv(
        nullptr,
        0,
        MPI_BYTE,
        status->MPI_SOURCE,
        status->MPI_TAG,
        comm,
        status));
}

```

7.38 mpi_util.hpp

```
#ifndef MPI_UTIL_HPP
#define MPI_UTIL_HPP

#include <mpi.h>
#include <vector>

int mpi_get_comm_rank(const MPI_Comm &comm);

int mpi_get_comm_size(const MPI_Comm &comm);

bool is_message_available(
    const MPI_Comm &comm,
    MPI_Status *status,
    int tag = MPI_ANY_TAG);

void wait_for_message(
    const MPI_Comm &comm,
    MPI_Status *status,
    int tag = MPI_ANY_TAG);

int get_message_size(MPI_Status *status);

void get_message(
    const MPI_Comm &comm,
    MPI_Status *status,
    std::vector<uint8_t> &content);

void send_message(
    const MPI_Comm &comm,
    int dest,
    int tag,
    const std::vector<uint8_t> &content);

void ping(const MPI_Comm &comm, int dest, int tag);

void accept_ping(const MPI_Comm &comm, MPI_Status *status);

#endif
```

7.39 seed.hpp

```
#ifndef SEED_HPP
#define SEED_HPP


#include <iostream>
#include <random>
#include <array>
#include <iterator>
#include <cassert>
#include <fstream>


template<typename base_type>
struct seed_type_generator
{
    std::uniform_int_distribution<base_type> dist;

    seed_type_generator() :
        dist(
            std::numeric_limits<base_type>::min(),
            std::numeric_limits<base_type>::max()
        )
    {}
};


template<std::size_t size, typename base_type>
class seed_type
{
    private:
        template<typename rdev_t>
        void init(
            rdev_t &rng_dev,
            seed_type_generator<base_type> *sgen_ptr = nullptr)
        {
            seed_type_generator<base_type> sgen;
            sgen_ptr = sgen_ptr ? sgen_ptr : &sgen;
            for (std::size_t i = 0; i < seed_data.size(); i++) {
                seed_data[i] = sgen.dist(rng_dev);
            }
        }

    public:
        using base_t = base_type;

        std::array<base_type, size> seed_data;

        inline auto get_seed_seq() {
            return std::seed_seq(
                std::begin(seed_data), std::end(seed_data));
        }

        seed_type(bool generate_random_seed = false) : seed_data()
        {
            if (generate_random_seed) {
                std::random_device rdev;
                init(rdev);
            }
        }

        seed_type(std::string seed_str) : seed_data() {
            std::stringstream sstr(seed_str);
            sstr » *this;
        }

        template<typename rdev_t> requires // hack to check for RNG:
            (!std::is_same<void, typename rdev_t::result_type>::value)
        seed_type(
            rdev_t &rng_dev,
            seed_type_generator<base_type> *sgen = nullptr
        ) :
            seed_data()
        {
            init(rng_dev, sgen);
        }

        base_type short_rep() {
            return seed_data[0];
        }

        bool operator==(const seed_type<size, base_type> &rhs) const {
            return seed_data == rhs.seed_data;
        }

        bool operator!=(const seed_type<size, base_type> &rhs) const {
            return seed_data != rhs.seed_data;
        }

        friend std::ostream &operator«(
            std::ostream &os, const seed_type &st)
        {
            std::copy(
                st.seed_data.begin(),
                st.seed_data.end(),
                std::ostream_iterator<base_type>(os, " "));
            return os;
        }

        friend std::istream &operator»(std::istream &is, seed_type &st)
        {
            for (std::size_t i = 0; i < st.seed_data.size(); i++) {
                base_type v;
                is » v;
                st.seed_data[i] = v;
            }

            return is;
        }
};


template <typename S, std::size_t size, typename base_type>
void serialize(S& s, seed_type<size, base_type> &seed)
{
    std::size_t sz = size;
    std::size_t bts = sizeof(base_type);

    static_assert(8 == sizeof(std::size_t));

    s.value8b(sz);
    s.value8b(bts);
    s.container(seed.seed_data, size);

    assert(size == sz);
    assert(sizeof(base_type) == bts);
}


template<typename seed_t>
auto get_persistent_global_seed(std::string seed_filename)
{
    std::fstream seed_file(
        seed_filename, std::ios::in | std::ios::out | std::ios::app);
    assert(seed_file.is_open());

    seed_file.seekp(0, seed_file.end);
    const auto fsize = seed_file.tellp();

    // generate a random seed
    seed_t global_seed = seed_t(true);

    if (0 == fsize) {
        // and save it if none was available
        std::cout « "c writing seed" « std::endl;
        seed_file « global_seed;
        seed_file.flush();
        assert(seed_file.good());
    } else {
        // or load if one had been saved before
        seed_file.seekg(0, seed_file.beg);
        std::cout « "c reading seed" « std::endl;
        seed_file » global_seed;
        // assert(seed_file.eof());
        assert(seed_file.tellg() + 1 == fsize);
    }

    assert(!seed_file.bad());
    seed_file.close();

    return global_seed;
}


#endif
```

7.40 shared_tmpfile.cpp

```

#include "shared_tmpfile.hpp"
#include "uuid.hpp"

#include <filesystem>
#include <cassert>
#include <iostream>

shared_tmpfile::shared_tmpfile() : id(0), fname(), fptr(nullptr) {}

shared_tmpfile::shared_tmpfile(std::string purpose) :
    id(uuid<shared_tmpfile>::get()+1),
    fname(),
    fptr(nullptr)
{
    std::string tmp_path = std::filesystem::temp_directory_path();
    fname = tmp_path + "/tmpfile-" + purpose + "_" + std::to_string(id);

    // std::cout « "created shared_tmpfile: " « fname « std::endl;
    fptr = fopen(fname.c_str(), "wbx+");
    if (nullptr == fptr) {
        std::string errmsg = "fopen error (shared_tmpfile "
            + std::to_string(id) + "):";
        perror(errmsg.c_str());

        if (false) {
            fptr = fopen(fname.c_str(), "wb+");
            assert(fptr);
            std::cerr
                « "note: recovered by reopening with wb+" « std::endl;
        } else {
            assert(false);
        }
    }
}

void swap(shared_tmpfile &a, shared_tmpfile &b)
{
    std::swap(a.id, b.id);
    std::swap(a.fname, b.fname);
    std::swap(a.fptr, b.fptr);
}

shared_tmpfile::shared_tmpfile(shared_tmpfile &&other) noexcept :
    shared_tmpfile()
{
    swap(*this, other);
}

shared_tmpfile &shared_tmpfile::operator=(shared_tmpfile other)
{
    swap(*this, other);
    return *this;
}

void shared_tmpfile::remove()
{
    if (0 < id) {
        // std::cout « "removing stmpfile " « id
            // « ": " « fname « std::endl;
        uuid<shared_tmpfile>::free(id-1);
        id = 0;
    }

    if (nullptr != fptr) {
        assert(0 == fclose(fptr));
        fptr = nullptr;
        assert(std::filesystem::remove(fname));
    }

    fname = "";
}

shared_tmpfile::~shared_tmpfile() {
    remove();
}

```

7.41 shared_tmpfile.hpp

```
#ifndef SHARED_TMPFILE
#define SHARED_TMPFILE

#include <cstddef>
#include <string>
#include <stdio.h>

class shared_tmpfile
{
    private:
        std::size_t id;

    public:
        std::string fname;

        FILE *fptr;

        shared_tmpfile();

        shared_tmpfile(std::string purpose);

        shared_tmpfile(const shared_tmpfile &other) = delete;

        // copy & swap idiom

        friend void swap(shared_tmpfile &a, shared_tmpfile &b);

        shared_tmpfile(shared_tmpfile &&other) noexcept;

        shared_tmpfile &operator=(shared_tmpfile other);

        void remove();

        ~shared_tmpfile();
};

#endif
```

7.42 statistics.hpp

```
#ifndef STATISTICS_HPP
#define STATISTICS_HPP

#include <cmath>

template<typename prec_t>
class basic_statistical_metrics
{
    public:
        std::size_t n;

        prec_t sum_xi;

        prec_t sum_xi_sq;

        prec_t varianz;

        prec_t stddev;

        prec_t average;

        prec_t min;

        prec_t max;

        std::size_t not_finite;

        basic_statistical_metrics() :
            n(0),
            sum_xi(0),
            sum_xi_sq(0),
            varianz(0),
            stddev(0),
            average(0),
            min(std::numeric_limits<prec_t>::max()),
            max(std::numeric_limits<prec_t>::min()),
            not_finite(0) {}

        std::size_t counted() {
            return n + not_finite;
        }

        template <typename it_t>
        void operator ()(it_t begin, it_t end)
        {
            for (auto i = begin; i != end; i++)
            {
                if (std::isfinite(*i)) {
                    min = std::min(min, *i);
                    max = std::max(max, *i);
                    n++;
                    prec_t x = *i;
                    sum_xi += x;
                    sum_xi_sq += x*x;
                } else {
                    not_finite++;
                }
            }

            average = sum_xi / n;
            varianz = (sum_xi_sq - sum_xi*sum_xi / n) / (n - 1);
            stddev = sqrt(varianz);
        }
};

// prec_t mean = std::accumulate(list.begin(), list.end(), (prec_t) 0.0) / list.size();

#endif
```

7.43 util.cpp

```

#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

#include "util.hpp"

std::ptrdiff_t slurp_file(std::string &content_str, const std::string filename) {
    auto fd = open(filename.c_str(), O_RDONLY);
    if (0 > fd) {
        std::string errmsg = std::string("can't get file descriptor for ") + filename;
        perror(errmsg.c_str());
        return -1;
    }

    struct stat stat_buf;
    auto r = fstat(fd, &stat_buf);
    if (0 != r) {
        std::string errmsg = std::string("can't get file size with fstat on ") + filename;
        perror(errmsg.c_str());
        return -1;
    }

    content_str.resize(stat_buf.st_size);
    if (stat_buf.st_size != read(fd, content_str.data(), stat_buf.st_size)) {
        std::string errmsg = std::string("error reading ") + std::to_string(stat_buf.st_size) + std::string(" bytes from ") + filename;
        perror(errmsg.c_str());
        return -1;
    }

    close(fd);
    return stat_buf.st_size;
}


void print_vector(std::string msg, const std::vector<uint8_t> &v)
{
    std::cout « msg;
    for (uint8_t i : v) {
        std::cout « " " « (int) i;
    }
    std::cout « std::endl;
}


uint8_t crc8(const uint8_t *data, const uint32_t length, uint8_t crc)
{
    constexpr const uint16_t polynom = 0b100110001;
    constexpr const uint8_t p_mask = polynom & 0xFF;

    for (uint32_t i = 0; i < length; i++) {
        const uint8_t d = data[i];
        for (int8_t j = 7; j >= 0; j--) {
            uint8_t e = ((crc » 7)^(d » j)) & 1;
            crc = (crc « 1) ^ p_mask*e;
        }
    }

    return crc;
}


auto vector_to_string(const std::vector<bool> &v)
{
    std::stringstream isstr;
    isstr « v.size();
    for (auto b : v) {
        isstr « " " « b;
    }
    assert(!isstr.fail());
    return isstr.str();
}


auto vector_from_string(auto content)
{
    std::stringstream osstr(content);
    std::size_t size;
    osstr » size;

    std::vector<bool> v(size);

    for (std::size_t i = 0; i < size; i++) {
        bool value;
        osstr » value;
        v[i] = value;
    }

    assert(!osstr.fail());
    assert(osstr.eof());
    return v;
}


bool fd_is_valid(int fd)
{
    return fcntl(fd, F_GETFD) != -1 || errno != EBADF;
}

```

7.44 util.hpp

```
#ifndef UTIL_HPP
#define UTIL_HPP


#include <string>
#include <cstddef>
#include <sstream>
#include <tuple>
#include <iostream>
#include <vector>
#include <cassert>

#include <unistd.h>
```

```
#include <fcntl.h>


template<typename t, typename... args>
void ignore(const t&, args...) { }


std::ptrdiff_t slurp_file(
    std::string &content_str, const std::string filename);


template<typename t>
t from_string(std::string s) {
    t value;
    std::istringstream iss(s);
    iss » value;

    if (!iss) {
        std::string errmsg = "can’t parse ’" + s + "’";
        throw std::runtime_error(errmsg);
    }

    return value;
}


void print_vector(std::string msg, const std::vector<uint8_t> &v);


template <typename head_t, typename... tail_t>
auto tail(const std::tuple<head_t, tail_t...> &tuple) {
    return apply([](auto &head, auto &... tail) { ignore(head); return std::make_tuple(tail...); }, tuple);
}


bool perror_(auto msg) {
    perror(msg);
    return false;
}


uint8_t crc8(const uint8_t *data, const uint32_t length, uint8_t crc = 0);


auto vector_to_string(const std::vector<bool> &v);


auto vector_from_string(auto content);

bool fd_is_valid(int fd);


/*
// https://stackoverflow.com/questions/41301536/get-function-return-type-in-template
template<typename R, typename... A>
R get_return_type(R(*)(A...));

template<typename C, typename R, typename... A>
R get_return_type(R(C::*)(A...));
*/


#endif
```

7.45 uuid.hpp

```
#ifndef UUID_HPP
#define UUID_HPP

#include <cstddef>
#include <vector>
#include <cassert>

template<class crtp, typename uuid_type = std::size_t, uuid_type reserved_ids = 1>
class uuid
{
    private:
        static uuid_type new_uuid;

        static std::vector<uuid_type>
            unused;

    public:
        uuid() = delete;

        static uuid_type get()
        {
            uuid_type id;

            if (unused.empty()) {
                id = new_uuid;
                new_uuid++;
                assert(0 != new_uuid + reserved_ids);
            } else {
                id = unused.back();
                unused.pop_back();
            }

            // std::cout « "get " « id « std::endl;

            return id;
        }

        static void free(const uuid_type id) {
            // std::cout « "free " « id « std::endl;
            unused.push_back(id);
        }
};

// static parameter initialization

template<class crtp, typename uuid_type, uuid_type reserved_ids>
uuid_type uuid<crtp, uuid_type, reserved_ids>::new_uuid = 0;

template<class crtp, typename uuid_type, uuid_type reserved_ids>
std::vector<uuid_type> uuid<crtp, uuid_type, reserved_ids>::unused;

#endif
```

Index