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:

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

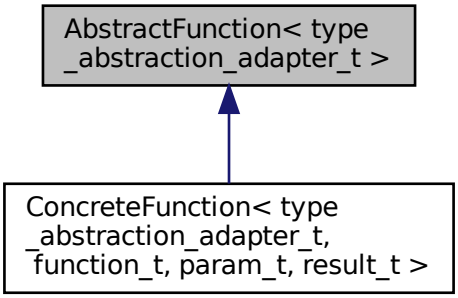
Class Documentation

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 >:



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:

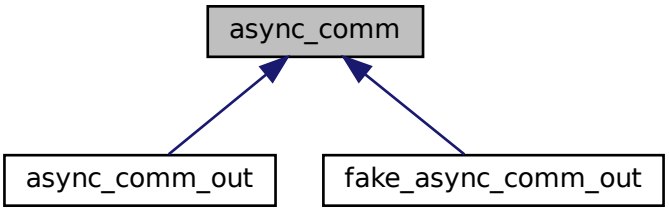
- distributed_computation/type_abstraction.hpp

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:

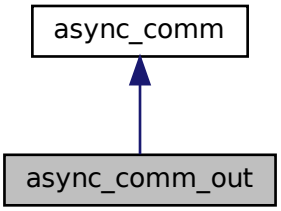
- util/mpi/mpi_async_communication.hpp
- util/mpi/mpi_async_communication.cpp

6.4 async_comm_out Class Reference

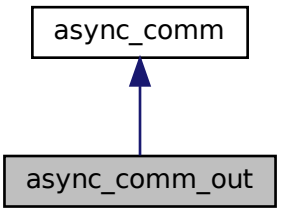
derived helper class providing async message sending

`#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]
```

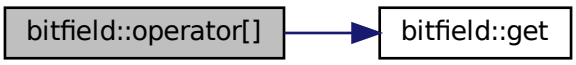
access position index by returning a boolean reference

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.

```
00198         {
00199             update();
00200 
00201             #if DEBUG_ASSERTIONS
00202             assert(index < length);
00203             #endif
00204 
00205             last_mod_index = index;
00206             last_mod_value = get(index);
00207             return last_mod_value;
00208         }
```

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]
```

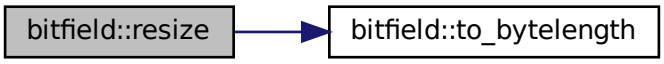
resize bitvector length

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.

```
00154         {
00155             update();
00156 
00157             if (to_bytelength(new_length) > to_bytelength(length)) {
00158                 assert(0 < to_bytelength(new_length));
00159 
00160                 uint8_t *old_array = array;
00161 
00162                 array = new uint8_t[to_bytelength(new_length)];
00163 
00164                 for (length_t i = 0; i < to_bytelength(length); i++) {
00165                     array[i] = old_array[i];
00166                 }
00167 
00168                 if (old_array) {
00169                     delete[] old_array;
00170                 }
00171             }
00172 
00173             length = new_length;
00174             last_mod_index = length;
00175         }
```

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:

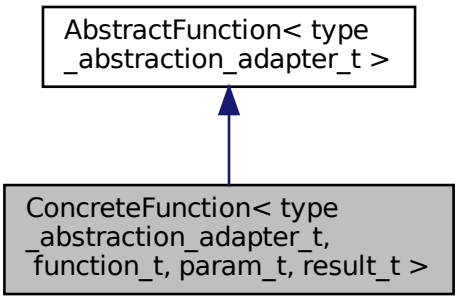
- metropolis_hastings/configuration/change_generator.hpp

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

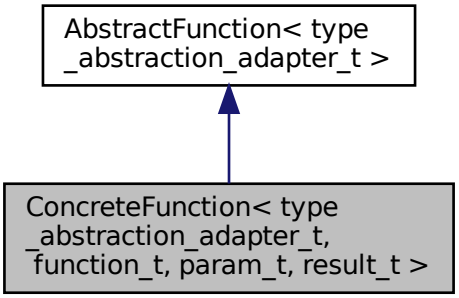
concrete implementation of an arbitrary function

`#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:

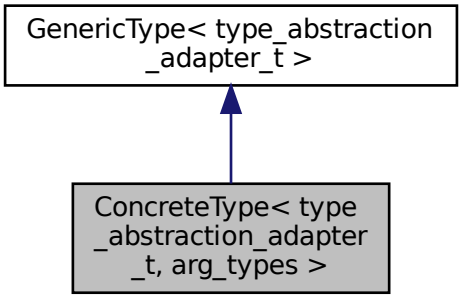
- distributed_computation/function_abstraction.hpp

6.11 ConcreteType< type_abstraction_adapter_t, arg_types > Class Template Reference

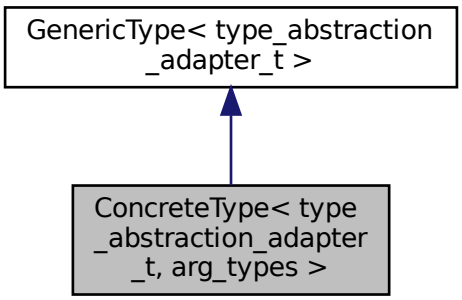
a concrete tuple of arbitrary types

`#include <rtti.hpp>`

Inheritance diagram for ConcreteType< type_abstraction_adapter_t, arg_types >:



Collaboration diagram for 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:

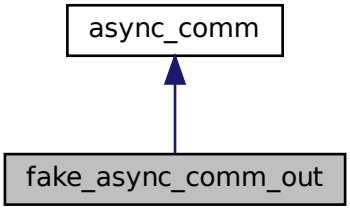
- metropolis_hastings/acceptance_computation/acceptance_probability.hpp

6.15 fake_async_comm_out Class Reference

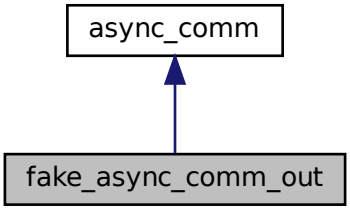
derived helper class for synchronous messaging (debug communication)

`#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:

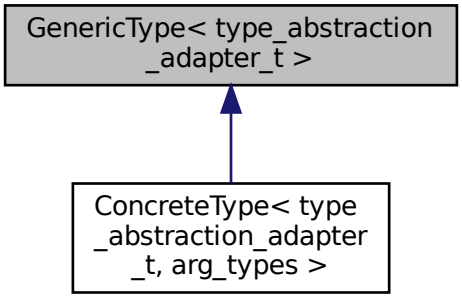
- distributed_computation/function_abstraction.hpp

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 >:



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:

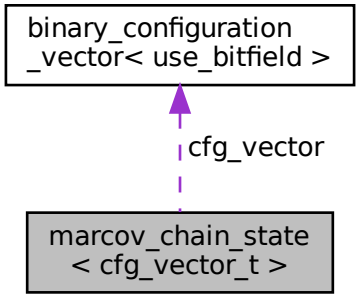
- util/basic_serialization.hpp

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 >:



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:

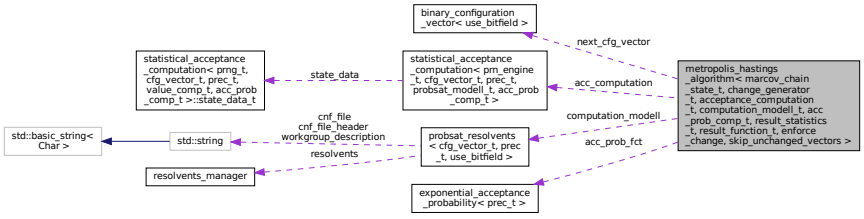
- metropolis_hastings/mcmc.hpp

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 >:



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:

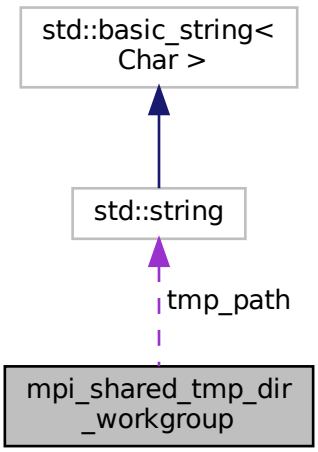
- metropolis_hastings/value_computation/modell.hpp

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

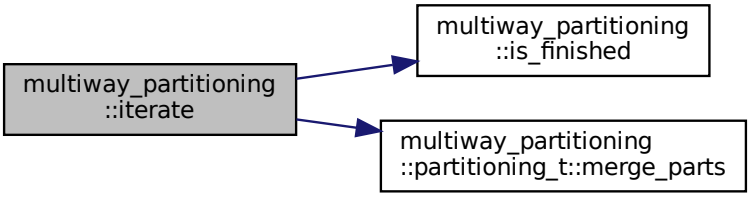
each iteration fuses two partitions in the dataset until one partition is left.

Definition at line 289 of file multiway_partitioning.hpp.

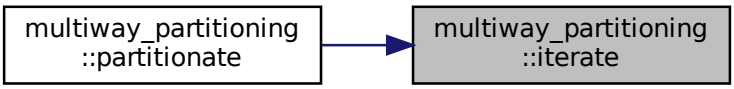
```
00290         {
00291             if (1 < dataset.size()) {
00292                 partitioning_t s1 = dataset.front();
00293                 dataset.pop_front();
00294                 partitioning_t s2 = dataset.front();
00295                 dataset.pop_front();
00296 
00297                 auto cmp_entry_f =
00298                     [this](const entry_t &l, const entry_t &r)
00299                 {
00300                     return sort_func(l.score, r.score);
```

```
00301                 };
00302
00303                 s1.merge_parts(s2, num_partitions, cmp_entry_f);
00304                 std::list<partitioning_t> nel = { s1 };
00305
00306                 auto cmp_f =
00307                 [this](const partitioning_t &l, const partitioning_t &r)
00308                 {
00309                     return sort_func(
00310                         l.difference(num_partitions),
00311                         r.difference(num_partitions));
00312                 };
00313                 dataset.merge(nel, cmp_f);
00314             }
00315
00316             return is_finished();
00317         }
```

Here is the call graph for this function:



Here is the caller graph for this function:



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

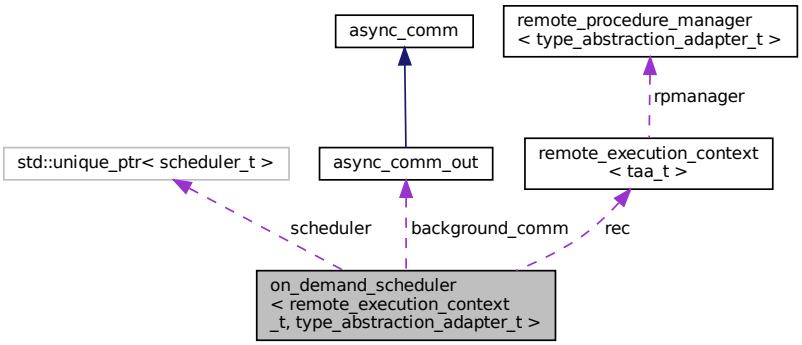
- distributed_computation/multiway_partitioning.hpp

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 >:



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]
```

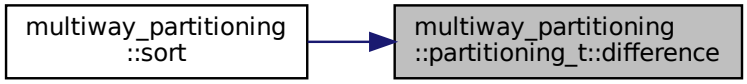
difference between minimum and maximum sum of scores in any subset contained

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.

```
00102             {
00103                 if (subsets.size() < num_partitions) {
00104                     return max_sum - std::min(min_sum, (std::size_t) 0);
00105                 }
00106                 return max_sum - min_sum;
00107             }
```

Here is the caller graph for this function:



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:

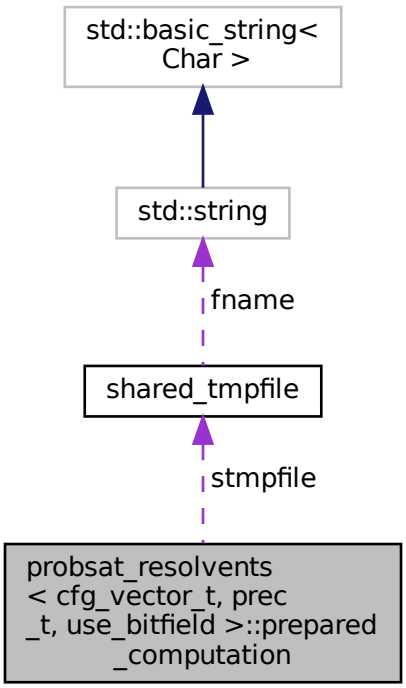
- metropolis_hastings/value_computation/modell.hpp

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:



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:

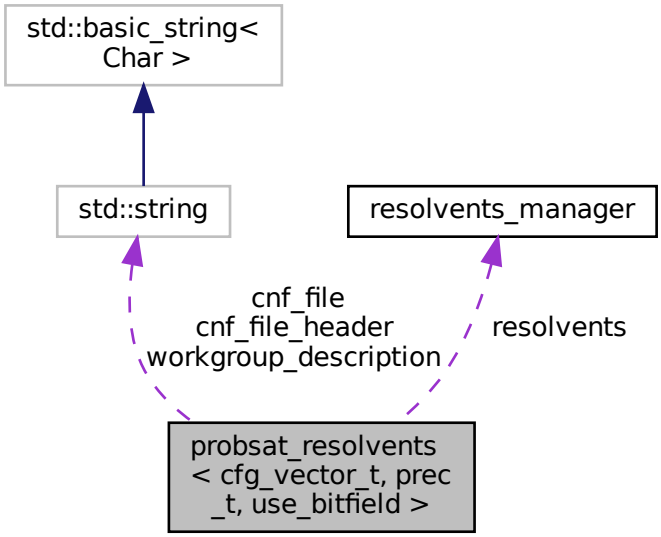
- metropolis_hastings/value_computation/probsat-with-resolvents.hpp

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:

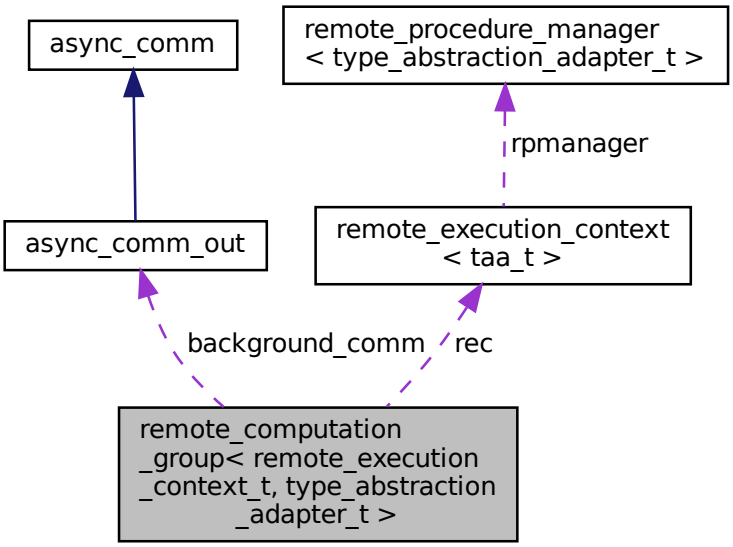
- metropolis_hastings/value_computation/probsat-with-resolvents.hpp

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 >:



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:

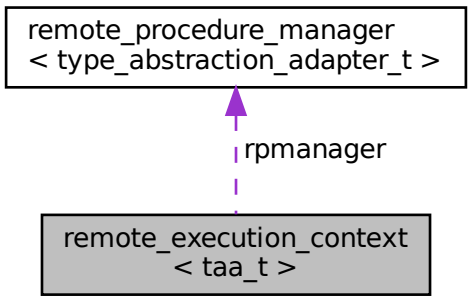
- distributed_computation/remote_computation_group.hpp

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:

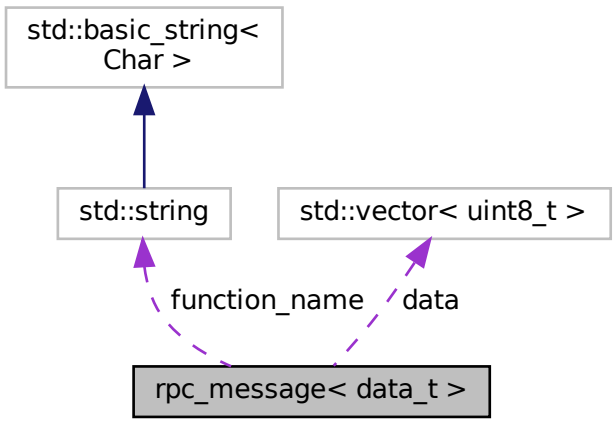
- metropolis_hastings/value_computation/utility/probsat-execution.hpp

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:

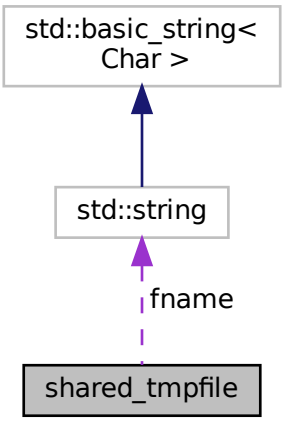
- util/seed.hpp

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:

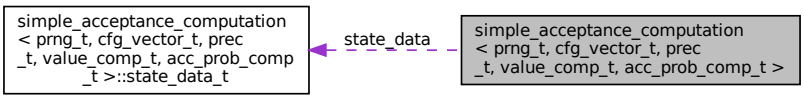
- util/shared_tmpfile.hpp
- util/shared_tmpfile.cpp

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 >:



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:

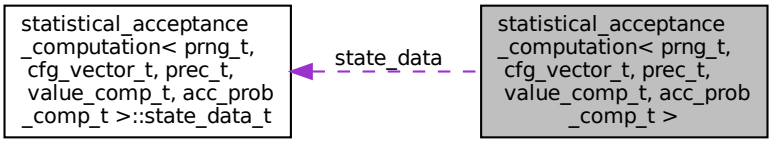
- metropolis_hastings/acceptance_computation/statistical_acceptance_computation.hpp

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 >:



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

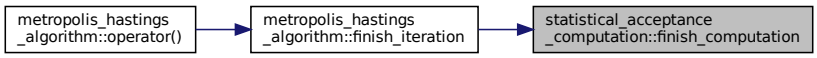
won't cleanup computation of accepted state, as it may be required to continue with it in the next iteration

Definition at line 468 of file statistical_acceptance_computation.hpp.

```
00470         {
00471             assert(0 == remaining_computations);
00472             assert(computation_scheduled_state);
```

```
00473             computation_scheduled_state = false;
00474
00475             ignore(&acc_prob_fct);
00476
00477             uint8_t old_index = 1 - new_index;
00478             if (is_first_cfg_vector) {
00479                 is_first_cfg_vector = false;
00480                 new_index = old_index;
00481                 return true;
00482             } else {
00483                 if (accept) {
00484                     // std::cout « "accept "
00485                         // « state_data[new_index].statistics.average
00486                         // « std::endl;
00487                     value_comp.finish_computation(
00488                         state_data[old_index].pcom);
00489                     new_index = old_index;
00490                 } else {
00491                     // std::cout « "decline "
00492                         // « state_data[new_index].statistics.average
00493                         // « std::endl;
00494                     // std::cout « "keep value "
00495                         // « state_data[old_index].statistics.average
00496                         // « std::endl;
00497                     value_comp.finish_computation(
00498                         state_data[new_index].pcom);
00499                 }
00500                 return accept;
00501             }
00502         }
```

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.

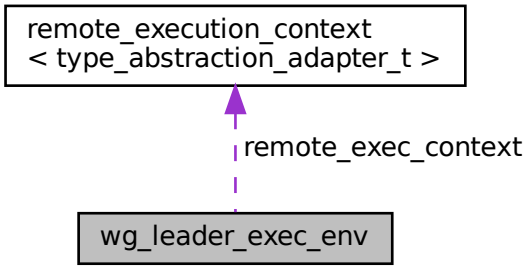
Definition at line 15 of file uuid.hpp.

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

```
00001
00002 #include "config.hpp"
00003
00004 // regarding the statistical acceptance computation
00006 double cfg_sac_confidence_interval_scaling_factor = 1.0;
00007
00009 double cfg_sac_testniveau = 0.01;
00010
00012 std::size_t cfg_sac_num_initial_values = 2400;
00013
00015 std::size_t cfg_sac_num_additional_values_per_iteration = 240;
00016
00018 std::size_t cfg_sac_max_iterations = 12;
```

7.2 config.hpp

```
00001 #ifndef CONFIG_HPP
00002 #define CONFIG_HPP
00003
00004 #include <cstddef>
00005 #include <string>
00006 #include <random>
00007 #include <chrono>
00008
00010 using prec_t = double;
00011 static_assert(std::numeric_limits<prec_t>::has_quiet_NaN);
00012
00013 // typedef std::mt19937 prn_engine_t;
00014 // using seed_t = seed_type<prn_engine_t::state_size, uint32_t>;
00015
00017 typedef std::mt19937 prn_engine_t;
00018 // using seed_t = seed_type<prn_engine_t::state_size, uint32_t>;
00019 // using seed_type_generator_t = seed_type_generator<typename seed_t::base_t>;
00020
00021
00022 // regarding probsat
00024 const std::string probsat_cmd
00025     = "./probSAT/probSAT --fct 0 --eps 0.9 --cb 2.06 --runs 1";
00026
00028 constexpr uint64_t probsat_max_flips = 20'000'000;
00029
00031 constexpr const std::chrono::minutes probsat_max_exec_time{1};
00032
00034 constexpr const bool interpret_timeout_as_max_flips_reached = false;
00035
00037 using probsat_seed_t = int32_t;
00038
00039 // regarding the statistical acceptance computation
00041 extern double cfg_sac_confidence_interval_scaling_factor;
00042
00044 extern double cfg_sac_testniveau;
00045
00047 extern std::size_t cfg_sac_num_initial_values;
00048
```

```
00050 extern std::size_t cfg_sac_num_additional_values_per_iteration;
00051
00053 extern std::size_t cfg_sac_max_iterations;
00054
00055 #endif
00056
```

7.3 function_abstraction.hpp

```
00001 #ifndef FUNCTION_ABSTRACTION_HPP
00002 #define FUNCTION_ABSTRACTION_HPP
00003
00004 #include <functional>
00005
00006 #include "rtti.hpp"
00007
00009 template <class type_abstraction_adapter_t>
00010     requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
00011 struct function_typeinfo
00012 {
00013     // functions without arguments
00014
00016     template <typename return_type>
00017     static ConcreteType<type_abstraction_adapter_t>
00018         get_concrete_argument_type(return_type(*)())
00019     {
00020         return ConcreteType<type_abstraction_adapter_t>(true);
00021     }
00022
00024     template <typename fobject, typename return_type>
00025     static ConcreteType<type_abstraction_adapter_t>
00026         get_concrete_argument_type(return_type(fobject::*)())
00027     {
00028         return ConcreteType<type_abstraction_adapter_t>(true);
00029     }
00030
00032     template <typename return_type>
00033     static ConcreteType<type_abstraction_adapter_t>
00034         get_concrete_argument_type(std::function<return_type()> f)
00035     {
00036         ignore(f);
00037         return ConcreteType<type_abstraction_adapter_t>(true);
00038     }
00039
00040
00042     template<typename first_arg_type, typename... tuple_tail_types>
00043     static auto create_concrete_type(
00044         first_arg_type *first_arg,
00045         std::tuple<tuple_tail_types...> &tail)
00046     {
00047         ignore(first_arg, tail);
00048         return ConcreteType<
00049             type_abstraction_adapter_t,
00050             first_arg_type,
00051             tuple_tail_types...>();
00052     }
00053
00054
00056     template <
00057         typename return_type,
00058         typename first_arg_type,
00059         typename... arg_types>
00060     static auto get_concrete_argument_type(
00061         return_type(*)(first_arg_type, arg_types...))
00062     {
00063         first_arg_type first_arg;
00064         std::function<return_type(arg_types...)> tail_func;
00065         typename decltype(
00066             get_concrete_argument_type(tail_func)
00067         )::tuple_t tail;
00068         return create_concrete_type(&first_arg, tail);
00069     }
00070
00072     template <
00073         typename fobject,
00074         typename return_type,
00075         typename first_arg_type,
00076         typename... arg_types>
00077     static auto get_concrete_argument_type(
00078         return_type(fobject::*)(first_arg_type, arg_types...))
00079     {
00080         first_arg_type first_arg;
00081         std::function<return_type(arg_types...)> tail_func;
```

```
00082         typename decltype(
00083             get_concrete_argument_type(tail_func)
00084         )::tuple_t tail;
00085         return create_concrete_type(&first_arg, tail);
00086     }
00087
00089     template <
00090         typename return_type,
00091         typename first_arg_type,
00092         typename... arg_types>
00093     static auto get_concrete_argument_type(
00094         std::function<return_type(first_arg_type, arg_types...)> f)
00095     {
00096         ignore(f);
00097         first_arg_type first_arg;
00098         std::function<return_type(arg_types...)> tail_func;
00099         typename decltype(
00100             get_concrete_argument_type(tail_func)
00101         )::tuple_t tail;
00102         return create_concrete_type(&first_arg, tail);
00103     }
00104
00106     template <
00107         typename return_type,
00108         typename first_arg_type,
00109         typename... arg_types>
00110     static auto get_concrete_argument_type(
00111         return_type(*)(const first_arg_type &, arg_types...))
00112     {
00113         first_arg_type first_arg;
00114         std::function<return_type(arg_types...)> tail_func;
00115         typename decltype(
00116             get_concrete_argument_type(tail_func)
00117         )::tuple_t tail;
00118         return create_concrete_type(&first_arg, tail);
00119     }
00120
00122     template <
00123         typename fobject,
00124         typename return_type,
00125         typename first_arg_type,
00126         typename... arg_types>
00127     static auto get_concrete_argument_type(
00128         return_type(fobject::*)(const first_arg_type &, arg_types...))
00129     {
00130         first_arg_type first_arg;
00131         std::function<return_type(arg_types...)> tail_func;
00132         typename decltype(
00133             get_concrete_argument_type(tail_func)
00134         )::tuple_t tail;
00135         return create_concrete_type(&first_arg, tail);
00136     }
00137
00139     template <
00140         typename return_type,
00141         typename first_arg_type,
00142         typename... arg_types>
00143     static auto get_concrete_argument_type(
00144         std::function<return_type(const first_arg_type &, arg_types...)> f)
00145     {
00146         ignore(f);
00147         first_arg_type first_arg;
00148         std::function<return_type(arg_types...)> tail_func;
00149         typename decltype(
00150             get_concrete_argument_type(tail_func)
00151         )::tuple_t tail;
00152         return create_concrete_type(&first_arg, tail);
00153     }
00154
00155 /* old implementation: did not allow const or const reference as parameters
00156     template <typename return_type, typename... arg_types>
00157     static ConcreteType<type_abstraction_adapter_t, arg_types...>
00158         get_concrete_argument_type(return_type(*)(arg_types...))
00159     {
00160         return ConcreteType<type_abstraction_adapter_t, arg_types...>();
00161     }
00162
00163     template <typename fobject, typename return_type, typename... arg_types>
00164     static ConcreteType<type_abstraction_adapter_t, arg_types...>
00165         get_concrete_argument_type(return_type(fobject::*)(arg_types...))
00166     {
00167         return ConcreteType<type_abstraction_adapter_t, arg_types...>();
00168     }
00169
00170     template <typename return_type, typename... arg_types>
00171     static ConcreteType<type_abstraction_adapter_t, arg_types...>
00172         get_concrete_argument_type(std::function<return_type(arg_types...)> f)
```

```
00173     {
00174         ignore(f);
00175         return ConcreteType<type_abstraction_adapter_t, arg_types...>();
00176     }
00177 */
00178
00179     // idea to get return type is based on:
00180     // https://stackoverflow.com/questions/41301536/get-function-return-type-in-template
00181
00183     template <typename return_type, typename... arg_types>
00184     static auto get_concrete_return_type(return_type(*)(arg_types...))
00185     {
00186         if constexpr (std::is_same<void, return_type>::value) {
00187             return ConcreteType<type_abstraction_adapter_t>(true);
00188         } else {
00189             return ConcreteType<type_abstraction_adapter_t, return_type>();
00190         }
00191     }
00192
00194     template <
00195         typename fobject, typename return_type, typename... arg_types>
00196     static auto get_concrete_return_type(
00197         return_type(fobject::*)(arg_types...))
00198     {
00199         if constexpr (std::is_same<void, return_type>::value) {
00200             return ConcreteType<type_abstraction_adapter_t>(true);
00201         } else {
00202             return ConcreteType<type_abstraction_adapter_t, return_type>();
00203         }
00204     }
00205
00207     template <typename return_type, typename... arg_types>
00208     static ConcreteType<type_abstraction_adapter_t>
00209         get_concrete_return_type(
00210             std::function<return_type(arg_types...)> f)
00211     {
00212         ignore(f);
00213         if constexpr (std::is_same<void, return_type>::value) {
00214             return ConcreteType<type_abstraction_adapter_t>(true);
00215         } else {
00216             return ConcreteType<type_abstraction_adapter_t, return_type>();
00217         }
00218     }
00219 };
00220
00221
00222
00224 template<typename type_abstraction_adapter_t>
00225     requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
00226 class AbstractFunction
00227 {
00228     public:
00230         type_abstraction_adapter_t::abstract_type operator()(
00231             type_abstraction_adapter_t &type_abstraction,
00232             const type_abstraction_adapter_t::abstract_type &abstract_param)
00233         {
00234             return this->execute(type_abstraction, abstract_param);
00235         }
00236
00238         virtual type_abstraction_adapter_t::abstract_type execute(
00239             type_abstraction_adapter_t &type_abstraction,
00240             const type_abstraction_adapter_t::abstract_type &abstract_param)
00241         {
00242             ignore(type_abstraction, abstract_param);
00243             typename type_abstraction_adapter_t::abstract_type result;
00244             return result;
00245         }
00246
00248         virtual ~AbstractFunction() { }
00249 };
00250
00251
00253 template<
00254     typename type_abstraction_adapter_t,
00255     typename function_t,
00256     typename param_t,
00257     typename result_t>
00258 requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
00259 class ConcreteFunction : public AbstractFunction<type_abstraction_adapter_t>
00260 {
00261     private:
00263         function_t function;
00264
00266         param_t parameters;
00267
00269         result_t result;
00270
```

```
00271     public:
00273         ConcreteFunction(auto function, auto param, auto result) :
00274             function(function), parameters(param), result(result) {}
00275
00277         virtual type_abstraction_adapter_t::abstract_type execute(
00278             type_abstraction_adapter_t &type_abstraction,
00279             const type_abstraction_adapter_t::abstract_type &abstract_param
00280         ) override
00281         {
00282             assert(parameters.deserialize(type_abstraction, abstract_param)
00283                 && "(most probably wrong parameters passed)");
00284
00285             using ret_t = decltype(std::apply(function, parameters.values));
00286
00287             if constexpr (std::is_same<void, ret_t>::value) {
00288                 std::apply(function, parameters.values);
00289             } else {
00290                 result.values = std::make_tuple(
00291                     std::apply(function, parameters.values));
00292             }
00293
00294             return result.serialize(type_abstraction);
00295         }
00296
00298         virtual ~ConcreteFunction() { }
00299 };
00300
00301
00303 template<typename type_abstraction_adapter_t, typename function_t>
00304     requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
00305 AbstractFunction<type_abstraction_adapter_t> *
00306     getAbstractFunctionFor(function_t f)
00307 {
00308     using taa_t = type_abstraction_adapter_t;
00309
00310     auto params = function_typeinfo<taa_t>::get_concrete_argument_type(f);
00311     auto result = function_typeinfo<taa_t>::get_concrete_return_type(f);
00312
00313     return new ConcreteFunction<
00314         type_abstraction_adapter_t,
00315         decltype(f),
00316         decltype(params),
00317         decltype(result)
00318     >(  f, params, result  );
00319 }
00320
00321 #endif
```

7.4 lazy_round_robin_scheduler.hpp

```
00001 #ifndef LAZY_ROUND_ROBIN_SCHEDULER_HPP
00002 #define LAZY_ROUND_ROBIN_SCHEDULER_HPP
00003
00004 #include <cstddef>
00005 #include <vector>
00006 #include <map>
00007 #include <cassert>
00008
00010 template<std::size_t workload_capacity>
00011 class lazy_round_robin_scheduler
00012 {
00013     private:
00015         const int size;
00016
00018         std::vector<std::size_t> num_tasks_scheduled;
00019
00021         std::multimap<std::size_t, int> ids_for_scheduled;
00022
00023     public:
00024
00026         bool is_process_without_work()
00027         {
00028             if (0 >= size) { return false; }
00029
00030             auto first_it = ids_for_scheduled.begin();
00031             return (0 == first_it->first);
00032         }
00033
00035         bool worker_available()
00036         {
00037             if (0 >= size) { return false; }
00038
00039             auto first_it = ids_for_scheduled.begin();
```

```
00040             return (workload_capacity > first_it->first);
00041         }
00042
00044         int get_id_to_schedule_task_on()
00045         {
00046             if (0 >= size) { return -1; }
00047
00048             auto first_it = ids_for_scheduled.begin();
00049             if (workload_capacity == first_it->first) {
00050                 return -1;
00051             }
00052
00053             int id = first_it->second;
00054             const std::size_t num_scheduled_old =
00055                 num_tasks_scheduled[id]++;
00056             assert(first_it->first == num_scheduled_old);
00057
00058             ids_for_scheduled.erase(first_it);
00059             ids_for_scheduled.emplace(num_scheduled_old + 1, id);
00060
00061             // std::cout « "get_id_to_schedule_task_on: "
00062                 // « id « " is now busy with "
00063                 // « (num_scheduled_old+1) « " tasks" « std::endl;
00064
00065             return id;
00066         }
00067
00069         void task_finished(int id)
00070         {
00071             if (0 >= size) { return; }
00072
00073             assert(0 <= id);
00074             assert(id < size);
00075
00076             // std::cout « "worker " « id
00077                 // « " with " « num_tasks_scheduled[id]
00078                 // « " tasks finished one" « std::endl;
00079
00080             std::size_t num_scheduled_old = num_tasks_scheduled[id]--;
00081             assert(0 < num_scheduled_old);
00082
00083             auto it = ids_for_scheduled.equal_range(num_scheduled_old);
00084
00085             for (auto i = it.first; i != it.second; i++) {
00086                 if (i->second == id) {
00087                     ids_for_scheduled.erase(i);
00088                     ids_for_scheduled.emplace(num_scheduled_old - 1, id);
00089                     return;
00090                 }
00091             }
00092
00093             assert(false);
00094         }
00095
00097         lazy_round_robin_scheduler(int size) :
00098             size(size), num_tasks_scheduled(size), ids_for_scheduled()
00099         {
00100             static_assert(0 < workload_capacity);
00101             num_tasks_scheduled.resize(size);
00102             for (int id = 0; id < size; id++) {
00103                 num_tasks_scheduled[id] = 0;
00104                 ids_for_scheduled.emplace(0, id);
00105             }
00106         }
00107 };
00108
00109 #endif
```

7.5 multiway_partitioning.hpp

```
00001 #ifndef MULTIWAY_PARTITIONING_HPP
00002 #define MULTIWAY_PARTITIONING_HPP
00003
00004 #include <cstddef>
00005 #include <functional>
00006 #include <list>
00007 #include <iostream>
00008 #include <cassert>
00009 #include <random>
00010
00019 template<
00020     typename data_t,
00021     typename score_t = std::size_t,
```

```
00022     class sort_func_t = std::greater<score_t>»
00023 class multiway_partitioning
00024 {
00025     public:
00026 
00028         struct entry_t {
00030             score_t score;
00031 
00033             data_t data;
00034         };
00035 
00036 
00038         struct subset_t {
00039 
00041             score_t sum;
00042 
00044             std::list<entry_t> elements;
00045 
00047             subset_t() : sum(0), elements() {}
00048 
00050             subset_t(entry_t first_entry) :
00051                 sum(first_entry.score), elements({first_entry}) {}
00052 
00054             void merge_subsets(subset_t &other, auto cmp_function)
00055             {
00056                 sum += other.sum;
00057                 // merge_two_lists(elements, other.elements);
00058                 elements.sort(cmp_function);
00059                 other.elements.sort(cmp_function);
00060                 elements.merge(other.elements, cmp_function);
00061             }
00062 
00064             friend std::ostream &operator«(
00065                 std::ostream &os, const subset_t &s)
00066             {
00067                 os « "{";
00068                 if (!s.elements.empty()) {
00069                     bool first = true;
00070                     for (auto el : s.elements) {
00071                         if (!first) { os « ", "; }
00072                         os « el.score;
00073                         first = false;
00074                     }
00075                 }
00076                 os « "} [S: " « s.sum
00077                     « ", l: " « s.elements.size() « "]";
00078                 return os;
00079             }
00080         };
00081 
00082 
00084         struct partitioning_t {
00086             std::vector<subset_t> subsets; // number of partitions = subset.size
00087 
00089             score_t min_sum;
00090 
00092             score_t max_sum;
00093 
00101             auto difference(std::size_t num_partitions) const
00102             {
00103                 if (subsets.size() < num_partitions) {
00104                     return max_sum - std::min(min_sum, (std::size_t) 0);
00105                 }
00106                 return max_sum - min_sum;
00107             }
00108 
00110             partitioning_t(std::size_t num_partitions)
00111                 : subsets(), min_sum(0), max_sum(0)
00112             {
00113                 subsets.reserve(num_partitions);
00114             }
00115 
00117             partitioning_t(std::size_t num_partitions, auto first_subset) :
00118                 subsets(),
00119                 min_sum(std::min((score_t) 0, first_subset.sum)),
00120                 max_sum(std::max((score_t) 0, first_subset.sum))
00121             {
00122                 subsets.reserve(num_partitions);
00123                 subsets.push_back(first_subset);
00124             }
00125 
00127             void merge_parts(
00128                 partitioning_t &other,
00129                 std::size_t num_partitions,
00130                 auto cmp_function)
00131             {
00132                 // simulate balanced partitioning, might be suboptimal!
```

```
00133                 auto subset_cmp_f = [](subset_t &l, subset_t &r) {
00134                     if (l.elements.size() != r.elements.size()) {
00135                         return l.elements.size() > r.elements.size();
00136                     } else {
00137                         return l.sum < r.sum;
00138                     }
00139                 };
00140 
00141                 std::sort(subsets.begin(), subsets.end(), subset_cmp_f);
00142                 std::sort(
00143                     other.subsets.begin(),
00144                     other.subsets.end(),
00145                     subset_cmp_f);
00146 
00147                 std::size_t ossize = other.subsets.size();
00148                 std::size_t tssize = subsets.size();
00149                 std::size_t end =
00150                     std::min(num_partitions, subsets.size() + ossize);
00151                 std::size_t start = end - ossize;
00152                 std::size_t border = std::min(end, tssize);
00153 
00154                 assert(other.subsets.size() <= end);
00155                 assert(other.subsets.size() >= end-start);
00156                 assert(subsets.size() >= border);
00157 
00158                 subsets.resize(end);
00159 
00160                 min_sum = std::numeric_limits<score_t>::max();
00161                 max_sum = std::numeric_limits<score_t>::min();
00162 
00163                 for (std::size_t i = 0; i < start; i++) {
00164                     min_sum = std::min(min_sum, subsets[i].sum);
00165                     max_sum = std::max(max_sum, subsets[i].sum);
00166                 }
00167 
00168                 for (std::size_t i = start; i < border; i++) {
00169                     std::size_t other_i = end - i - 1;
00170                     subsets[i].merge_subsets(
00171                         other.subsets[other_i], cmp_function);
00172                     min_sum = std::min(min_sum, subsets[i].sum);
00173                     max_sum = std::max(max_sum, subsets[i].sum);
00174                 }
00175 
00176                 for (std::size_t i = border; i < end; i++) {
00177                     std::size_t other_i = end - i - 1;
00178                     subsets[i] = other.subsets[other_i];
00179                     min_sum = std::min(min_sum, subsets[i].sum);
00180                     max_sum = std::max(max_sum, subsets[i].sum);
00181                 }
00182             }
00183 
00185             void sort(auto sort_func)
00186             {
00187                 auto cmp_f =
00188                     [sort_func](const subset_t &l, const subset_t &r)
00189                 {
00190                     return sort_func(l.sum, r.sum);
00191                 };
00192 
00193                 std::sort(subsets.begin(), subsets.end(), cmp_f);
00194             }
00195 
00197             void final_sort(auto sort_func)
00198             {
00199                 auto cmp_f =
00200                     [sort_func](const subset_t &l, const subset_t &r)
00201                 {
00202                     if (l.elements.size() != r.elements.size()) {
00203                         return l.elements.size() < r.elements.size();
00204                     } else {
00205                         return sort_func(l.sum, r.sum);
00206                     }
00207                 };
00208 
00209                 std::sort(subsets.begin(), subsets.end(), cmp_f);
00210             }
00211 
00213             friend std::ostream &operator«(
00214                 std::ostream &os, const partitioning_t &p)
00215             {
00216                 os « "(";
00217                 bool first = true;
00218                 for (auto s : p.subsets) {
00219                     if (!first) { os « "; "; }
00220                     os « s;
00221                     first = false;
00222                 }
```

```
00223                 // warning: capacity should be correct,
00224                 // but num_partitions is not available
00225                 os « ") [D: "
00226                     « p.difference(p.subsets.capacity()) « "?]";
00227                 return os;
00228             }
00229         };
00230
00232         std::list<partitioning_t> dataset;
00233
00235         std::size_t num_partitions;
00236
00238         sort_func_t sort_func;
00239
00241         multiway_partitioning(
00242             std::size_t num_partitions,
00243             auto sort_func = std::greater<score_t>()
00244         ) :
00245             dataset(),
00246             num_partitions(num_partitions),
00247             sort_func(sort_func)
00248         {
00249             assert(1 < num_partitions);
00250         }
00251
00253         void reset(std::size_t n_partitions) {
00254             num_partitions = n_partitions;
00255             dataset.clear();
00256         }
00257
00259         void add(auto score, auto data) {
00260             dataset.push_back(
00261                 partitioning_t(num_partitions, subset_t({score, data})));
00262         }
00263
00265         void sort()
00266         {
00267             auto cmp_f =
00268                 [this](const partitioning_t &l, const partitioning_t &r)
00269             {
00270                 return sort_func(
00271                     l.difference(num_partitions),
00272                     r.difference(num_partitions));
00273             };
00274
00275             // std::sort(dataset.begin(), dataset.end(), cmp_f);
00276             dataset.sort(cmp_f);
00277         }
00278
00280         bool is_finished() {
00281             return 1 == dataset.size();
00282         }
00283
00289         bool iterate()
00290         {
00291             if (1 < dataset.size()) {
00292                 partitioning_t s1 = dataset.front();
00293                 dataset.pop_front();
00294                 partitioning_t s2 = dataset.front();
00295                 dataset.pop_front();
00296
00297                 auto cmp_entry_f =
00298                     [this](const entry_t &l, const entry_t &r)
00299                 {
00300                     return sort_func(l.score, r.score);
00301                 };
00302
00303                 s1.merge_parts(s2, num_partitions, cmp_entry_f);
00304                 std::list<partitioning_t> nel = { s1 };
00305
00306                 auto cmp_f =
00307                 [this](const partitioning_t &l, const partitioning_t &r)
00308                 {
00309                     return sort_func(
00310                         l.difference(num_partitions),
00311                         r.difference(num_partitions));
00312                 };
00313                 dataset.merge(nel, cmp_f);
00314             }
00315
00316             return is_finished();
00317         }
00318
00320         void partitionate() {
00321             while (false == iterate()) {}
00322         }
00323
```

```
00325         auto result() {
00326             assert(is_finished());
00327             return dataset.front();
00328         }
00329
00331         friend std::ostream &operator«(
00332             std::ostream &os, const multiway_partitioning &mp)
00333         {
00334             os « "multiway_partitioning for "
00335                 « mp.num_partitions « " partitions: ";
00336
00337             os « "{\n";
00338             for (auto p : mp.dataset) {
00339                 os « "\t" « p « "\n";
00340             }
00341             os « "}" « std::endl;
00342
00343             return os;
00344         }
00345 };
00346
00348 void test_multiway_partitioning()
00349 {
00350     const bool print_steps_in_between = false;
00351
00352     // config for first example
00353     std::size_t num_elements = 11;
00354     std::size_t num_partitions = 3;
00355
00356     // followed by iterations-1 random tests
00357     std::size_t iterations = 1;
00358
00359     // random test configuration
00360     std::mt19937 prng(42);
00361     std::uniform_int_distribution<std::size_t> udistr(1, 9);
00362     std::uniform_int_distribution<std::size_t> nedistr(10, 100);
00363     std::uniform_int_distribution<std::size_t> npdistr(2, 8);
00364
00365     for (std::size_t it = 0; it < iterations; it++) {
00366         if (0 < it) {
00367             num_elements = nedistr(prng);
00368             num_partitions = npdistr(prng);
00369         }
00370
00371         std::cout « "sorting: "
00372             « num_elements « " elements into "
00373             « num_partitions « " partitions" « std::endl;
00374         using sort_t = std::greater<std::size_t>;
00375         // using sort_t = std::less<std::size_t>;
00376         multiway_partitioning<std::size_t, std::size_t, sort_t>
00377             mp(num_partitions, sort_t());
00378
00379         for (std::size_t i = 0; i < num_elements; i++) {
00380             std::size_t s = udistr(prng);
00381             mp.add(s, i);
00382         }
00383
00384         mp.sort();
00385
00386         std::cout « mp;
00387
00388         if constexpr (print_steps_in_between) {
00389             bool is_finished;
00390             do {
00391                 is_finished = mp.iterate();
00392                 std::cout « mp;
00393             } while (!is_finished);
00394             // std::cout « mp;
00395         } else {
00396             mp.partitionate();
00397         }
00398
00399         auto result = mp.result();
00400         // result.sort(sort_t());
00401
00402         result.final_sort(sort_t());
00403         std::cout « result « std::endl;
00404
00405         std::size_t min_length = std::numeric_limits<std::size_t>::max();
00406         std::size_t max_length = std::numeric_limits<std::size_t>::min();
00407         for (const auto &sset : result.subsets) {
00408             std::size_t length = sset.elements.size();
00409             max_length = std::max(max_length, length);
00410             min_length = std::min(min_length, length);
00411         }
00412
00413         std::cout « "lengths [" « min_length « ", "
```

```
00414             « max_length « "]" « std::endl;
00415         assert(max_length >= min_length);
00416         assert(1 >= max_length - min_length);
00417 
00418         // result.improve_partitioning();
00419     }
00420 }
00421 
00422 #endif
```

7.6 on_demand_scheduler.hpp

```
00001 #ifndef ON_DEMAND_SCHEDULER
00002 #define ON_DEMAND_SCHEDULER
00003 
00004 #include <cstddef>
00005 #include <cassert>
00006 #include <deque>
00007 #include <memory>
00008 #include <chrono>
00009 
00010 #include <mpi.h>
00011 
00012 #include "../util/mpi/mpi_util.hpp"
00013 #include "../util/mpi/mpi_async_communication.hpp"
00014 
00015 #include "lazy_round_robin_scheduler.hpp"
00016 
00017 
00022 template<
00023     typename remote_execution_context_t,
00024     class type_abstraction_adapter_t =
00025         remote_execution_context_t::type_abstraction_adapter_t>
00026 requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
00027 class on_demand_scheduler
00028 {
00029     public:
00031         using rpc_message_t = remote_execution_context_t::rpc_message_t;
00033         using buffer_t = type_abstraction_adapter_t::abstract_type;
00034 
00035     private:
00037         static constexpr const std::size_t workload_capacity = 2;
00038 
00040         using scheduler_t = lazy_round_robin_scheduler<workload_capacity>;
00041 
00043         using uptr_scheduler_t = std::unique_ptr<scheduler_t>;
00044 
00046         uptr_scheduler_t scheduler;
00047 
00049         std::deque<rpc_message_t> tasks;
00050 
00052         type_abstraction_adapter_t taa;
00053 
00055         std::size_t num_waiting_for_reply;
00056 
00058         async_comm_out background_comm;
00059         // fake_async_comm_out background_comm;
00060 
00061 
00062 
00064         void process_msg(MPI_Status &status)
00065         {
00066             assert(0 == env_comm_id);
00067 
00068             if (0 < get_message_size(&status)) {
00069                 // std::cout « "process_msg" « std::endl;
00070                 execute(get_msg(status));
00071             } else {
00072                 assert(MPI_SUCCESS == MPI_Recv(
00073                     nullptr,
00074                     0,
00075                     MPI_BYTE,
00076                     status.MPI_SOURCE,
00077                     mpi_tag,
00078                     env_comm,
00079                     &status));
00080             }
00081 
00082             assert(0 < status.MPI_SOURCE);
00083             scheduler->task_finished(status.MPI_SOURCE - 1);
00084             num_waiting_for_reply--;
00085         }
00086 
```

```
00087
00089         rpc_message_t get_msg(MPI_Status &status)
00090         {
00091             buffer_t b;
00092             get_message(env_comm, &status, b);
00093             // print_vector("get_msg:", b);
00094             rpc_message_t msg;
00095             assert(0 < b.size());
00096             assert(taa.deserialize(b, msg));
00097
00098             // std::cout « "c msg from: " « status.MPI_SOURCE
00099                 // « " with tag " « status.MPI_TAG « std::endl;
00100             return msg;
00101         }
00102
00103
00105         rpc_message_t wait_for_msg()
00106         {
00107             assert(0 != env_comm_id);
00108             MPI_Status status;
00109             wait_for_message(env_comm, &status, mpi_tag);
00110             // std::cout « "wait_for_msg" « std::endl;
00111             return get_msg(status);
00112         }
00113
00114
00116         void execute(auto msg)
00117         {
00118             // std::cout « "executing " « msg.function_name
00119                 // « " (id: " « msg.id « ") with"
00120                 // « (msg.answer_required ? "" : "out")
00121                 // « " reply on " « env_comm_id
00122                 // « " (crc: " « (int) msg.crc « ")" « std::endl;
00123
00124             assert(msg.check_crc());
00125
00126             rpc_message_t result = rec.execute_rpc(msg);
00127             assert(!result.answer_required);
00128
00129             if (msg.answer_required) {
00130                 result.compute_crc();
00131             }
00132
00133             if (0 == env_comm_id) {
00134                 if (msg.answer_required) {
00135                     rec.execute_rpc(result);
00136                 }
00137             } else {
00138                 if (msg.answer_required) {
00139                     buffer_t b = taa.serialize(result);
00140                     background_comm.send_message(
00141                         env_comm, 0, mpi_tag, std::move(b));
00142                     // print_vector("execute_send_reply:", b);
00143                 } else {
00144                     ping(env_comm, 0, mpi_tag);
00145                 }
00146             }
00147         }
00148
00149
00151         void feed_hungry_workers()
00152         {
00153             assert(0 == env_comm_id);
00154
00155             while (scheduler->is_process_without_work())
00156             {
00157                 if (tasks.empty()) { return; }
00158
00159                 rpc_message_t task = tasks.front();
00160                 tasks.pop_front();
00161                 schedule(task);
00162             }
00163         }
00164
00165
00166     public:
00168         MPI_Comm env_comm;
00169
00171         const int env_comm_id;
00172
00174         const int env_comm_size;
00175
00177         const int mpi_tag;
00178
00179
00181         remote_execution_context_t &rec;
00182
```

```
00183
00184
00186         on_demand_scheduler(
00187             MPI_Comm env_comm,
00188             int mpi_tag,
00189             remote_execution_context_t &rec
00190         ) :
00191             scheduler(nullptr),
00192             tasks(),
00193             taa(),
00194             num_waiting_for_reply(0),
00195             background_comm(),
00196             env_comm(env_comm),
00197             env_comm_id(mpi_get_comm_rank(env_comm)),
00198             env_comm_size(mpi_get_comm_size(env_comm)),
00199             mpi_tag(mpi_tag),
00200             rec(rec)
00201         {
00202             if (0 == env_comm_id) {
00203                 scheduler =
00204                     uptr_scheduler_t(new scheduler_t(env_comm_size - 1));
00205             }
00206         }
00207
00208
00210         void schedule(auto task, int target = 0)
00211         {
00212             assert(0 == env_comm_id);
00213
00214             int target_id = target;
00215             if (0 == target) {
00216                 target_id = 1 + scheduler->get_id_to_schedule_task_on();
00217             }
00218
00219             /*int crc = */ task.compute_crc();
00220
00221             // std::cout « "ods scheduling " « task.function_name
00222                 // « " (id: " « task.id « ") with"
00223                 // « (task.answer_required ? "" : "out")
00224                 // « " reply on " « target
00225                 // « " (crc: " « crc « ")" « std::endl;
00226
00227             if (0 == target_id) {
00228                 tasks.push_back(task);
00229             } else {
00230                 buffer_t b = taa.serialize(task);
00231                 background_comm.send_message(
00232                     env_comm, target_id, mpi_tag, std::move(b));
00233                 // print_vector("schedule:", b);
00234                 num_waiting_for_reply++;
00235             }
00236         }
00237
00238
00240         std::size_t num_tasks_waiting() const {
00241             return tasks.size();
00242         }
00243
00244
00246         std::size_t num_tasks_pending() const {
00247             return num_waiting_for_reply;
00248         }
00249
00250
00252         bool is_work_outstanding() const {
00253             return ((0 < num_tasks_pending())
00254                 || (0 < num_tasks_waiting()))
00255                 || !background_comm.communication_is_done();
00256         }
00257
00258
00260         size_t do_work(bool wait_for_work = false)
00261         {
00262             if ((0 == env_comm_id) && (1 < env_comm_size))
00263             {
00264                 feed_hungry_workers();
00265
00266                 MPI_Status status;
00267                 while (is_message_available(env_comm, &status, mpi_tag))
00268                 {
00269                     process_msg(status);
00270                     feed_hungry_workers();
00271                 }
00272
00273                 while (scheduler->worker_available())
00274                 {
00275                     if (tasks.empty()) { return 0; }
```

```
00276
00277                   rpc_message_t task = tasks.front();
00278                   tasks.pop_front();
00279                   schedule(task);
00280
00281                   if (is_message_available(env_comm, &status, mpi_tag))
00282                   {
00283                       process_msg(status);
00284                   }
00285               }
00286
00287               background_comm.test_for_messages();
00288
00289               // std::cout « "communications_in_queue: "
00290                   // « background_comm.communications_in_queue()
00291                   // « std::endl;
00292
00293               if (wait_for_work) {
00294                   if (background_comm.communication_is_done()) {
00295                       if (!tasks.empty()) {
00296                           rpc_message_t task = tasks.front();
00297                           tasks.pop_front();
00298                           execute(task);
00299                       }
00300                   } else {
00301                       background_comm.wait_for_some_messages();
00302                   }
00303               }
00304           } else {
00305               MPI_Status status;
00306               while (is_message_available(env_comm, &status, mpi_tag))
00307               {
00308                   // std::cout « "message available" « std::endl;
00309                   tasks.push_back(get_msg(status));
00310               }
00311
00312               background_comm.test_for_messages();
00313
00314               /*
00315               if (tasks.empty()) {
00316                   if (wait_for_work && (1 < env_comm_size)) {
00317                       tasks.push_back(wait_for_msg());
00318                       background_comm.test_for_messages();
00319                   }
00320               }
00321               */
00322
00323               if (wait_for_work && (1 < env_comm_size)) {
00324                   if (!background_comm.communication_is_done()) {
00325                       while (tasks.empty()) {
00326                           if (is_message_available(
00327                               env_comm, &status, mpi_tag))
00328                           {
00329                               tasks.push_back(get_msg(status));
00330                               break;
00331                           }
00332
00333                           background_comm.test_for_one_message();
00334                           if (background_comm.communication_is_done())
00335                           {
00336                               break;
00337                           }
00338                       }
00339                   }
00340
00341                   if (tasks.empty()) {
00342                       tasks.push_back(wait_for_msg());
00343                   }
00344               }
00345
00346               if (!tasks.empty()) {
00347                   auto start =
00348                       std::chrono::high_resolution_clock::now();
00349
00350                   execute(tasks.front());
00351
00352                   auto end =
00353                       std::chrono::high_resolution_clock::now();
00354                   tasks.pop_front();
00355
00356                   return std::chrono::duration_cast<
00357                       std::chrono::milliseconds>(end - start).count();
00358               }
00359           }
00360
00361           return 0;
00362       }
```

```
00363 };
00364
00365 #endif
00366
```

7.7 remote_computation_group.hpp

```
00001 #ifndef REMOTE_COMPUTATION_GROUP
00002 #define REMOTE_COMPUTATION_GROUP
00003
00004 #include <cstddef>
00005 #include <cassert>
00006 #include <deque>
00007
00008 #include <mpi.h>
00009
00010 #include "../util/mpi/mpi_util.hpp"
00011 #include "../util/mpi/mpi_async_communication.hpp"
00012
00013
00015 template<
00016     typename remote_execution_context_t,
00017     class type_abstraction_adapter_t
00018         = remote_execution_context_t::type_abstraction_adapter_t>
00019 requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
00020 class remote_computation_group
00021 {
00022     public:
00024         using rpc_message_t = remote_execution_context_t::rpc_message_t;
00025
00027         using buffer_t = type_abstraction_adapter_t::abstract_type;
00028
00029     private:
00030
00032         std::deque<std::pair<int, rpc_message_t» tasks;
00033
00035         type_abstraction_adapter_t taa;
00036
00038         std::size_t num_waiting_for_reply;
00039
00041         async_comm_out background_comm;
00042         // fake_async_comm_out background_comm;
00043
00044
00046         std::pair<int, rpc_message_t> get_msg(MPI_Status &status)
00047         {
00048             buffer_t b;
00049             get_message(env_comm, &status, b);
00050             rpc_message_t msg;
00051             assert(0 < b.size());
00052             assert(taa.deserialize(b, msg));
00053
00054             assert(msg.check_crc());
00055             return std::make_pair(status.MPI_SOURCE, msg);
00056         }
00057
00058
00060         std::pair<int, rpc_message_t> wait_for_msg()
00061         {
00062             assert(0 != env_comm_id);
00063             MPI_Status status;
00064             while (1) {
00065                 wait_for_message(env_comm, &status, mpi_tag);
00066                 if (0 == get_message_size(&status)) {
00067                     num_waiting_for_reply--;
00068                 } else {
00069                     break;
00070                 }
00071             }
00072
00073             // std::cout « "wait_for_msg" « std::endl;
00074             return get_msg(status);
00075         }
00076
00077
00079         void execute(const std::pair<int, rpc_message_t> &task)
00080         {
00081             const int origin = task.first;
00082             rpc_message_t msg = task.second;
00083             // std::cout « "executing " « msg.function_name
00084                 // « " (id: " « msg.id « ") with"
00085                 // « (msg.answer_required ? "" : "out")
00086                 // « " reply and arg size " « msg.data.size()
```

```
00087                 // « " on " « env_comm_id « std::endl;
00088
00089             assert(msg.check_crc());
00090
00091             rpc_message_t result = rec.execute_rpc(msg);
00092             assert(!result.answer_required);
00093
00094             if (msg.answer_required) {
00095                 result.compute_crc();
00096             }
00097
00098             if (origin == env_comm_id) {
00099                 if (msg.answer_required) {
00100                     rec.execute_rpc(result);
00101                 }
00102             } else {
00103                 if (msg.answer_required) {
00104                     buffer_t b = taa.serialize(result);
00105                     background_comm.send_message(
00106                         env_comm, origin, mpi_tag, std::move(b));
00107                     // print_vector("execute_send_reply:", b);
00108                 } else {
00109                     // std::cout « "ping " « origin « std::endl;
00110                     ping(env_comm, origin, mpi_tag);
00111                 }
00112             }
00113         }
00114
00115
00116     public:
00118         MPI_Comm env_comm;
00119
00121         const int env_comm_id;
00122
00124         const int env_comm_size;
00125
00127         const int mpi_tag;
00128
00129
00131         remote_execution_context_t &rec;
00132
00133
00134
00136         remote_computation_group(
00137             MPI_Comm env_comm,
00138             int mpi_tag,
00139             remote_execution_context_t &rec
00140         ) :
00141             tasks(),
00142             taa(),
00143             num_waiting_for_reply(0),
00144             background_comm(),
00145             env_comm(env_comm),
00146             env_comm_id(mpi_get_comm_rank(env_comm)),
00147             env_comm_size(mpi_get_comm_size(env_comm)),
00148             mpi_tag(mpi_tag),
00149             rec(rec)
00150         {}
00151
00152
00154         void schedule(auto task, int target_id)
00155         {
00156             // std::cout « "scheduling " « task.function_name
00157                 // « " (id: " « task.id « ") with"
00158                 // « (task.answer_required ? "" : "out")
00159                 // « " reply and arg size " « task.data.size()
00160                 // « " on " « target_id « " by "
00161                 // « env_comm_id « std::endl;
00162
00163             task.compute_crc();
00164
00165             if (env_comm_id == target_id) {
00166                 tasks.push_back(std::make_pair(target_id, task));
00167             } else {
00168                 buffer_t b = taa.serialize(task);
00169                 // int bcrc = crc8((uint8_t *) b.data(), b.size());
00170                 // std::cout « "CCC schedule bcrc: " « bcrc « std::endl;
00171                 // print_vector("CCC get_msg:", b);
00172                 background_comm.send_message(
00173                     env_comm, target_id, mpi_tag, std::move(b));
00174                 num_waiting_for_reply++;
00175                 // std::cout « "num_waiting_for_reply increased to: "
00176                     // « num_waiting_for_reply « std::endl;
00177             }
00178         }
00179
00180
```

```
00182         std::size_t num_tasks_waiting() const {
00183             return tasks.size();
00184         }
00185 
00186 
00188         std::size_t num_tasks_pending() const {
00189             return num_waiting_for_reply;
00190         }
00191 
00192 
00194         bool is_work_outstanding() const {
00195             return ((0 < num_tasks_pending())
00196                 || (0 < num_tasks_waiting()))
00197                 || !background_comm.communication_is_done();
00198         }
00199 
00200 
00202         void do_work(bool wait_for_work = false)
00203         {
00204             MPI_Status status;
00205             while (is_message_available(env_comm, &status, mpi_tag)) {
00206                 if (0 == get_message_size(&status)) {
00207                     accept_ping(env_comm, &status);
00208                     num_waiting_for_reply--;
00209                 } else {
00210                     tasks.push_back(get_msg(status));
00211                 }
00212             }
00213 
00214             background_comm.test_for_messages();
00215 
00216             if (wait_for_work && (1 < env_comm_size)) {
00217                 if (!background_comm.communication_is_done()) {
00218                     while (tasks.empty()) {
00219                         if (is_message_available(
00220                             env_comm, &status, mpi_tag))
00221                         {
00222                             if (0 == get_message_size(&status)) {
00223                                 accept_ping(env_comm, &status);
00224                                 num_waiting_for_reply--;
00225                             } else {
00226                                 tasks.push_back(get_msg(status));
00227                                 break;
00228                             }
00229                         }
00230 
00231                         background_comm.test_for_one_message();
00232                         if (background_comm.communication_is_done()) {
00233                             break;
00234                         }
00235                     }
00236                 }
00237 
00238                 if (tasks.empty()) {
00239                     tasks.push_back(wait_for_msg());
00240                 }
00241             }
00242 
00243             if (!tasks.empty()) {
00244                 std::pair<int, rpc_message_t> p = tasks.front();
00245                 execute(p);
00246                 tasks.pop_front();
00247             }
00248         }
00249 
00250 };
00251 
00252 #endif
```

7.8 remote_execution_context.hpp

```
00001 #ifndef REMOTE_EXECUTION_CONTEXT_HPP
00002 #define REMOTE_EXECUTION_CONTEXT_HPP
00003 
00004 #include "rpc.hpp"
00005 #include "../util/uuid.hpp"
00006 
00008 template<typename taa_t> requires (taa_t::is_TypeAbstractionAdapter)
00009 class remote_execution_context
00010 {
00011     public:
00013         using type_abstraction_adapter_t = taa_t;
00014 
```

```
00015     private:
00017         using uuid_t =
00018             uuid<remote_execution_context<type_abstraction_adapter_t>>;
00019
00020     public:
00022         using data_t = type_abstraction_adapter_t::abstract_type;
00023
00025         using rpc_message_t = rpc_message<data_t>;
00026
00028         using remote_procedure_manager_t =
00029             remote_procedure_manager<type_abstraction_adapter_t>;
00030
00032         remote_procedure_manager_t &rpmanager;
00033
00034
00036         remote_execution_context(remote_procedure_manager_t &rpmanager) :
00037             rpmanager(rpmanager) {}
00038
00039
00041         template<typename function_t>
00042         rpc_message_t on_result(auto msg, function_t function)
00043         {
00044             assert(0 == msg.id);
00045
00046             std::size_t id = 1 + uuid_t::get();
00047             assert(0 != id);
00048
00049             std::string rfname = msg.function_name
00050                 + FCT_NAME_RESULT_TAG + "_" + std::to_string(id);
00051             rpmanager.add_function(rfname, function);
00052
00053             msg.id = id;
00054             msg.answer_required = true;
00055
00056             return msg;
00057         }
00058
00059
00061         auto execute_rpc(rpc_message_t &msg)
00062         {
00063             bool is_result =
00064                 (false == msg.answer_required) && (0 != msg.id);
00065
00066             if (is_result) {
00067                 msg.function_name =
00068                     msg.function_name + "_" + std::to_string(msg.id);
00069             }
00070
00071             rpc_message_t result_msg = rpmanager.execute_rpc(msg);
00072
00073             if (is_result) {
00074                 assert(1 == rpmanager.function_map.erase(msg.function_name));
00075                 uuid_t::free(msg.id - 1);
00076                 result_msg.id = 0;
00077                 result_msg.function_name = "";
00078             }
00079
00080             return result_msg;
00081         }
00082 };
00083
00084
00085 #endif
```

7.9 rescheduling.hpp

```
00001 #ifndef RESCHEDULING_HPP
00002 #define RESCHEDULING_HPP
00003
00004 #include <cstddef>
00005 #include <functional>
00006 #include <vector>
00007 #include <cassert>
00008 #include <iostream>
00009
00010 #include "../util/util.hpp"
00011 #include "multiway_partitioning.hpp"
00012
00014 class rescheduling_manager
00015 {
00016     private:
00018         bool rescheduling_required;
00019
```

```
00021         int num_workgroups;
00022
00024         std::size_t number_of_active_mcs;
00025
00026         std::function<bool(std::size_t, std::size_t)> sort_func;
00027
00029         multiway_partitioning<
00030             std::pair<std::size_t, int>,
00031             std::size_t,
00032             std::function<bool(std::size_t, std::size_t)» mwpart;
00033
00035         std::vector<std::size_t> workload;
00036
00037     public:
00039         rescheduling_manager(
00040             int num_workgroups,
00041             std::size_t num_mcs,
00042             bool use_mc_limit,
00043             bool rescheduling_required = false
00044         ) :
00045             rescheduling_required(rescheduling_required),
00046             num_workgroups(num_workgroups),
00047             number_of_active_mcs(num_mcs),
00048             sort_func(std::less<std::size_t>()),
00049             mwpart(num_workgroups, sort_func),
00050             workload(num_workgroups)
00051         { assert(0 < num_workgroups); ignore(use_mc_limit); }
00052
00053
00055         void start_rescheduling()
00056         {
00057             assert(!rescheduling_required);
00058
00059             mwpart.reset(num_workgroups);
00060
00061             workload.reserve(num_workgroups);
00062             for (int i = 0; i < num_workgroups; i++) {
00063                 workload[i] = 0;
00064             }
00065
00066             rescheduling_required = true;
00067         }
00068
00069
00071         bool is_rescheduling_in_progress() const
00072         {
00073             return rescheduling_required;
00074         }
00075
00076
00078         void add_progress_for_id(
00079             std::size_t id, std::size_t progress, int wg_id)
00080         {
00081             assert(rescheduling_required);
00082             assert(0 <= wg_id);
00083
00084             std::cout « "adding reschedule progress " « progress
00085                 « " for " « id « " on " « wg_id « std::endl;
00086
00087             // progress_list.insert(id, std::make_pair(progress, wg_id));
00088             mwpart.add(progress, std::make_pair(id, wg_id));
00089
00090             workload[wg_id] += progress;
00091         }
00092
00093
00095         void remove_mc(std::size_t id)
00096         {
00097             assert(rescheduling_required);
00098
00099             ignore(id);
00100             if (!rescheduling_required) {
00101                 number_of_active_mcs--;
00102             }
00103         }
00104
00106         bool information_complete() const
00107         {
00108             assert(rescheduling_required);
00109             return mwpart.dataset.size() >= number_of_active_mcs;
00110         }
00111
00113         bool is_rescheduling_unnecessary(
00114             std::size_t reschedule_requires_absolute_difference) const
00115         {
00116             assert(rescheduling_required);
00117             assert(information_complete());
```

```
00118
00119             auto [min, max] =
00120                 std::minmax_element(begin(workload), end(workload));
00121
00122             std::size_t diff = max - min;
00123             bool got_required_abs_diff =
00124                 diff >= reschedule_requires_absolute_difference;
00125
00126             return !got_required_abs_diff;
00127         }
00128
00130         void cancel_rescheduling(auto do_rescheduling)
00131         {
00132             // assert(rescheduling_required);
00133             // assert(information_complete());
00134             rescheduling_required = false;
00135             for (int target_wg = 0;
00136                 target_wg < num_workgroups; target_wg++)
00137             {
00138                 do_rescheduling(0, target_wg, target_wg);
00139             }
00140         }
00141
00143         void reschedule(auto do_rescheduling)
00144         {
00145             assert(rescheduling_required);
00146             // assert(progress_list.size() == number_of_active_mcs);
00147             assert(mwpart.dataset.size() == number_of_active_mcs);
00148
00149             // std::cout « "before rescheduling:\n" « mwpart;
00150
00151             mwpart.sort();
00152             mwpart.partitionate();
00153             auto result = mwpart.result();
00154             // result.sort(sort_t());
00155             // todo: improve? (i.e. with complete Karmarkar-Karp algorithm)
00156             // an anytime algorithm would be possible as well
00157             result.final_sort(sort_func);
00158
00159             // todo: find optimal mapping
00160
00161             // std::cout « "after rescheduling:\n" « mwpart;
00162
00163             rescheduling_required = false;
00164
00165             const auto [min, max] =
00166                 std::minmax_element(begin(workload), end(workload));
00167             std::size_t original_diff = *max - *min;
00168
00169             // std::cout « "rescheduling from diff: "
00170                 // « original_diff « " [" « *min « ", " « *max « "] "
00171                 // « " to " « result.difference(num_workgroups)
00172                 // « " [" « result.min_sum « ", "
00173                 // « result.max_sum « "]" « std::endl;
00174
00175             if (result.difference(num_workgroups) >= original_diff) {
00176                 // std::cout « "rescheduling probably unnecessary!" « std::endl;
00177                 cancel_rescheduling(do_rescheduling);
00178                 return;
00179             }
00180
00181             assert(result.subsets.size() == (std::size_t) num_workgroups);
00182             auto it = result.subsets.begin();
00183             for (int target_wg = 0;
00184                 target_wg < num_workgroups; target_wg++)
00185             {
00186                 std::size_t mcs_to_get = 0;
00187                 for (auto entry : it->elements) {
00188                     if (entry.data.second != target_wg) {
00189                         std::size_t id = entry.data.first;
00190                         int original_wg = entry.data.second;
00191                         // std::cout « "move mc " « id
00192                             // « " with progress " « entry.score
00193                             // « " from workgroup " « original_wg
00194                             // « " to " « target_wg « std::endl;
00195                         do_rescheduling(id, original_wg, target_wg);
00196                         mcs_to_get++;
00197                     }
00198                 }
00199
00200                 do_rescheduling(mcs_to_get, target_wg, target_wg);
00201                 it++;
00202             }
00203             assert(it == result.subsets.end());
00204         }
00205 };
00206
```

```
00207
00208 #endif
```

7.10 rpc.hpp

```
00001 #ifndef RPC_HPP
00002 #define RPC_HPP
00003
00004 #include <iostream>
00005 #include <limits>
00006 #include <functional>
00007 #include <type_traits>
00008 #include <map>
00009 #include <memory>
00010
00011 #include "../util/util.hpp"
00012
00013 #include "../util/basic_serialization.hpp"
00014
00015 #include "type_abstraction.hpp"
00016 #include "rtti.hpp"
00017 #include "function_abstraction.hpp"
00018
00019
00021 const std::string FCT_NAME_RESULT_TAG = "__result";
00022
00023
00025 template<typename data_t = std::vector<uint8_t»
00026 class rpc_message
00027 {
00028     public:
00030         using data_type = data_t;
00031
00033         std::size_t id;
00034
00036         std::string function_name;
00037
00039         bool answer_required;
00040
00042         data_t data;
00043
00045         uint8_t crc;
00046
00047
00049         rpc_message() :
00050             id(std::numeric_limits<std::size_t>::max()),
00051             function_name(),
00052             answer_required(false),
00053             data(),
00054             crc(0)
00055         {}
00056
00058         rpc_message(const rpc_message &other) :
00059             id(other.id),
00060             function_name(other.function_name),
00061             answer_required(other.answer_required),
00062             data(other.data),
00063             crc(other.crc)
00064         {}
00065
00067         friend void swap(rpc_message &l, rpc_message &r)
00068         {
00069             std::swap(l.id, r.id);
00070             std::swap(l.function_name, r.function_name);
00071             std::swap(l.answer_required, r.answer_required);
00072             std::swap(l.data, r.data);
00073             std::swap(l.crc, r.crc);
00074         }
00075
00076
00078         rpc_message &operator=(const rpc_message &other) {
00079             id = other.id;
00080             function_name = other.function_name;
00081             answer_required = other.answer_required;
00082             data = other.data;
00083             crc = other.crc;
00084             return *this;
00085         }
00086
00087 /*
00089         rpc_message &operator=(const rpc_message &other) {
00090             swap(*this, other);
00091             return *this;
```

```
00092         }
00093 */
00094
00096         rpc_message(rpc_message &&other) {
00097             swap(*this, other);
00098         }
00099
00100
00101     private:
00103         uint8_t compute_crc_() const
00104         {
00105             decltype(crc) crc_;
00106             crc_ = crc8((uint8_t *) &id, sizeof(decltype(id)));
00107             crc_ = crc8((uint8_t *) function_name.c_str(),
00108                 function_name.length(), crc_);
00109             crc_ = crc8((uint8_t *) &answer_required, 1, crc_);
00110             crc_ = crc8((uint8_t *) data.data(), data.size(), crc_);
00111             return crc_;
00112         }
00113
00114     public:
00115         // optional crc functionality
00116
00118         uint8_t compute_crc() {
00119             crc = compute_crc_();
00120             return crc;
00121         }
00122
00124         bool check_crc() const {
00125             return (compute_crc_() == crc);
00126         }
00127 };
00128
00129
00131 template<typename S>
00132 void serialize(S &s, rpc_message<std::vector<uint8_t>> &v) {
00133     std::size_t data_len = v.data.size();
00134     generic_serialize(s,
00135         v.id, v.function_name, v.answer_required, data_len);
00136     v.data.resize(data_len);
00137     s.container1b(v.data, data_len);
00138     assert(v.data.size() == data_len);
00139     generic_serialize(s, v.crc);
00140 }
00141
00142
00144 template<typename type_abstraction_adapter_t>
00145     requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
00146 class remote_procedure_manager
00147 {
00148     private:
00149
00151         type_abstraction_adapter_t type_abstraction;
00152
00154         using abstract_function_t =
00155             AbstractFunction<type_abstraction_adapter_t>;
00156
00157     public:
00158
00160         using rpc_message_t =
00161             rpc_message<typename type_abstraction_adapter_t::abstract_type>;
00162
00164         std::map<std::string, std::unique_ptr<abstract_function_t>>
00165             function_map;
00166
00167
00169         template<typename function_t>
00170         void add_function(std::string name, function_t function)
00171         {
00172             assert(!name.empty());
00173
00174             AbstractFunction<type_abstraction_adapter_t> *afptr
00175                 = getAbstractFunctionFor<
00176                     type_abstraction_adapter_t,
00177                     function_t> (function);
00178             assert(afptr);
00179
00180             auto result = function_map.try_emplace(
00181                 name,
00182                 std::unique_ptr<abstract_function_t>(afptr)
00183             );
00184             assert(result.second &&
00185                 "there is already a function with this name registered!");
00186         }
00187
00189         template<typename... param_types>
00190         auto prepare_call(std::string name, param_types... params)
```

```
00191         {
00192             assert(!name.empty());
00193 
00194             rpc_message_t msg;
00195             msg.id = 0;
00196             msg.function_name = name;
00197             msg.answer_required = false;
00198 
00199             ConcreteType<type_abstraction_adapter_t, param_types...>
00200                 param_container = {params..., true};
00201 
00202             msg.data = param_container.serialize(type_abstraction);
00203             return msg;
00204         }
00205 
00207         auto execute_rpc(const rpc_message_t &msg)
00208         {
00209             // std::cout « "execute_rpc: " «
00210                 // msg.function_name « std::endl;
00211 
00212             assert(!msg.function_name.empty());
00213 
00214             auto fsearch = function_map.find(msg.function_name);
00215             if (function_map.end() == fsearch) {
00216                 std::cerr « "Unknown remote procedure call: "
00217                     « msg.function_name « std::endl;
00218                 assert(false && "Unknown remote procedure call!");
00219             } else {
00220                 // std::cout « "rpc executing: "
00221                     // « msg.function_name « std::endl;
00222             }
00223 
00224             rpc_message_t result_msg;
00225             result_msg.id = msg.id;
00226             result_msg.function_name =
00227                 msg.function_name + FCT_NAME_RESULT_TAG;
00228             result_msg.answer_required = false;
00229 
00230             abstract_function_t *fptr = (fsearch->second).get();
00231 
00232             try {
00233                 result_msg.data =
00234                     fptr->execute(type_abstraction, msg.data);
00235             } catch (...) {
00236                 std::cout « "exception executing function: "
00237                     « result_msg.function_name « std::endl;
00238                 abort();
00239             }
00240 
00241             return result_msg;
00242         }
00243 };
00244 
00245 
00246 #endif
```

7.11 rtti.hpp

```
00001 #ifndef RTTI_HPP
00002 #define RTTI_HPP
00003 
00004 #include <tuple>
00005 
00006 #include "../util/util.hpp"
00007 
00008 
00010 template<typename type_abstraction_adapter_t> requires
00011     (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
00012 class GenericType
00013 {
00014     public:
00016         virtual type_abstraction_adapter_t::abstract_type serialize
00017             (type_abstraction_adapter_t &adapter) const
00018         {
00019             ignore(adapter);
00020             return typename type_abstraction_adapter_t::abstract_type();
00021         }
00022 
00024         virtual bool deserialize(
00025             type_abstraction_adapter_t &adapter,
00026             const type_abstraction_adapter_t::abstract_type &v)
00027         {
00028             ignore(adapter, v);
```

```
00029             return false;
00030         }
00031
00033         virtual ~GenericType() {}
00034 };
00035
00037 template<typename type_abstraction_adapter_t, typename... arg_types>
00038     requires (type_abstraction_adapter_t::is_TypeAbstractionAdapter)
00039 class ConcreteType : public GenericType<type_abstraction_adapter_t>
00040 {
00041     public:
00043         using tuple_t = std::tuple<arg_types ...>;
00044
00046         tuple_t values;
00047
00049         ConcreteType() : values() {}
00050
00052         ConcreteType(
00053             arg_types... args,
00054             bool use_value_constructor = true
00055         ) :
00056             values(args...)
00057         {
00058             ignore(use_value_constructor);
00059         }
00060
00062         ConcreteType(std::tuple<arg_types ...> &values) : values(values) {}
00063
00065         ConcreteType(const ConcreteType &other) : values(other.values) {}
00066
00068         template<typename other_type_abstraction_adapter_t>
00069         operator ConcreteType<
00070             other_type_abstraction_adapter_t, arg_types...>() const
00071         {
00072             return ConcreteType<
00073                 other_type_abstraction_adapter_t, arg_types...>(values);
00074         }
00075
00077         virtual type_abstraction_adapter_t::abstract_type
00078             serialize(type_abstraction_adapter_t &adapter) const override
00079         {
00080             return adapter.serialize(values);
00081         }
00082
00084         virtual bool deserialize(
00085             type_abstraction_adapter_t &adapter,
00086             const type_abstraction_adapter_t::abstract_type &v) override
00087         {
00088             return adapter.deserialize(v, values);
00089         }
00090 };
00091
00092
00093 #endif
00094
00095
```

7.12 type_abstraction.hpp

```
00001 #ifndef TYPE_ABSTRACTION_HPP
00002 #define TYPE_ABSTRACTION_HPP
00003
00004 #include <any>
00005
00006 #include "../util/include_bitsery.hpp"
00007
00009 class Any_TypeAbstractionAdapter
00010 {
00011     public:
00012
00014         static const bool is_TypeAbstractionAdapter = true;
00015
00017         using abstract_type = std::any;
00018
00020         template<typename t>
00021         static abstract_type serialize(t &v) {
00022             return v;
00023         }
00024
00026         template<typename t>
00027         static bool deserialize(const abstract_type &v, t &result) {
00028             try {
00029                 result = std::any_cast<t>(v);
```

```
00030                 return true;
00031             } catch (...) {
00032                 return false;
00033             }
00034         }
00035 };
00036
00038 template<
00039     typename buffer_t,
00040     typename input_adapter_t,
00041     typename output_adapter_t
00042 >
00043 class Bitsery_TypeAbstractionAdapter
00044 {
00045     public:
00046
00048         static const bool is_TypeAbstractionAdapter = true;
00049
00051         using abstract_type = buffer_t;
00052
00054         template<typename t>
00055         static abstract_type serialize(t &v)
00056         {
00057             buffer_t buffer;
00058             auto length = bitsery::quickSerialization(
00059                 std::move(output_adapter_t{buffer}), v);
00060             buffer.resize(length);
00061             return buffer;
00062         }
00063
00065         template<typename t>
00066         static bool deserialize(const abstract_type &buffer, t &result)
00067         {
00068             auto state = bitsery::quickDeserialization(
00069                 std::move(input_adapter_t{buffer.begin(), buffer.size()}),
00070                 result);
00071
00072             if ((state.first != bitsery::ReaderError::NoError)
00073                 || (false == state.second))
00074             {
00075                 return false;
00076             }
00077
00078             return true;
00079         }
00080 };
00081
00082 #endif
00083
```

7.13 main.cpp

```
00001
00002 // Due to debugging at the end of the development phase, this file
00003 // was not yet cleaned up and well documented (work in progress state)
00004 // There was a bug after rescheduling when the Markov chains were transferred
00005
00006 // mpirun -np 6 --use-hwthread-cpus --oversubscribe main
00007 // seq 1 6 |parallel -I{} -j 1 "bash -c 'mpirun -np {} main > {}.txt'"
00008 // seq 7 12 |parallel -I{} -j 1 "bash -c 'mpirun -np {} --use-hwthread-cpus --oversubscribe main >
       {}.txt'"
00009 // for n in $(seq 9); do mv $n.txt 0$n.txt; done;
00010 // grep -m 1 duration *.txt
00011 // grep -m 1 duration *.txt |cut -d " " -f 3 |gnuplot -p -e 'plot "/dev/stdin" with lines'
00012 // grep -m 1 duration *.txt |awk '{print $i*$3; $i++}' |gnuplot -p -e 'plot "/dev/stdin" with lines'
00013
00014 // grep -m 1 duration *.txt |awk '{print '$(grep duration 01.txt|cut -d " " -f 3)'/($3/$i) - 1; $i++}'
       |gnuplot -p -e 'plot "/dev/stdin" with lines'
00015 // grep -m 1 duration *.txt |awk '{print '$(grep duration 01.txt|cut -d " " -f 3)'/$3; $i++}' |gnuplot
       -p -e 'plot "/dev/stdin" with lines'
00016
00017 // echo ... |sed "s/ / + /g" |bc
00018 // 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)}'
00019 // echo ... |sed "s/ /\n/g" |sort -nk1 |gnuplot -p -e 'set logscale y 2.718281828459045; plot
       "/dev/stdin" with lines'
00020 // echo ... |sed "s/ /\n/g" |awk '{print log($1)}' |sort -g |gnuplot -p -e 'plot "/dev/stdin" with
       lines'
00021
00022 // grep "c value" out-1000.txt |cut -d " " -f 3 |gnuplot -p -e 'plot "/dev/stdin" with lines'
00023 // grep "accepted" out-1000.txt |sed "s/)//g" |cut -d "(" -f 2 |gnuplot -p -e 'plot "/dev/stdin" with
       lines'
00024
```

```
00025 #include <set>
00026 #include <queue>
00027
00028 #include "config.hpp"
00029
00031 std::size_t num_reference_chains = 1;
00032
00034 std::size_t num_markov_chains = 4;
00035
00037 std::size_t markov_chain_offset = 0;
00038
00040 std::size_t markov_chain_target_length = 4;
00041
00043 std::size_t reschedule_after_seconds = 0;
00044
00046 std::size_t reschedule_requires_absolute_difference = 0;
00047
00049 std::size_t max_runtime_in_seconds = 0;
00050
00052 bool checkpoint_on_termination = true;
00053
00055 std::size_t checkpoint_after_iterations = 0;
00056
00058 std::size_t checkpoint_after_seconds = 0;
00059
00061 uint8_t mcmc_num_changes = 3;
00062
00063 // prec_t mcmc_T = 0.05;
00064 // prec_t mcmc_T = 0.005;
00065 // 300 200 100 75 50 40 30 20 10 5
00066
00068 prec_t mcmc_T = 30;
00069
00070 // wenn neu - alt = mcmc_T => p(accept) = 1/e
00071 // wenn neu - alt = 3*mcmc_T => p(accept) = 5 %
00072
00073
00074 // #include "util/util.hpp"
00075 // #include "util/mpi/mpi_util.hpp"
00076 #include "util/mpi/mpi_shared_tmp_workgroup.hpp"
00077 #include "util/mpi/mpi_types.hpp"
00078 // #include "util/uuid.hpp"
00079 // #include "util/bitfield.hpp"
00080
00081 const mpi_shared_tmp_dir_workgroup *workgroup_ptr = nullptr;
00082
00083 // #include "distributed_computation/rtti.hpp"
00084 // #include "distributed_computation/type_abstraction.hpp"
00085 // #include "distributed_computation/function_abstraction.hpp"
00086 #include "distributed_computation/rpc.hpp"
00087 #include "distributed_computation/remote_execution_context.hpp"
00088 #include "distributed_computation/on_demand_scheduler.hpp"
00089 #include "distributed_computation/remote_computation_group.hpp"
00090
00091 using buffer_t = std::vector<uint8_t>;
00092 using output_adapter_t = bitsery::OutputBufferAdapter<buffer_t>;
00093 using input_adapter_t = bitsery::InputBufferAdapter<buffer_t>;
00094 using type_abstraction_adapter_t =
      Bitsery_TypeAbstractionAdapter<buffer_t, input_adapter_t, output_adapter_t>;
00095
00096 using remote_procedure_manager_t = remote_procedure_manager<type_abstraction_adapter_t>;
00097 using remote_execution_context_t = remote_execution_context<type_abstraction_adapter_t>;
00098 using rpc_message_t = remote_procedure_manager_t::rpc_message_t;
00099
00100 remote_procedure_manager_t wg_rpmanager;
00101
00102 struct wg_leader_exec_env {
00103     remote_procedure_manager_t rpmanager;
00104     remote_execution_context_t remote_exec_context;
00105     remote_computation_group<remote_execution_context_t> rcg;
00106     wg_leader_exec_env(MPI_Comm env_comm, int mpi_tag) :
00107         rpmanager(),
00108         remote_exec_context(rpmanager),
00109         rcg(env_comm, mpi_tag, remote_exec_context)
00110     {}
00111 };
00112
00113 std::unique_ptr<wg_leader_exec_env> wg_leaders = nullptr;
00114
00115
00116 #include "util/seed.hpp"
00117
00118 // typedef std::mt19937 prn_engine_t;
00119 using seed_t = seed_type<prn_engine_t::state_size, uint32_t>;
00120 using seed_type_generator_t = seed_type_generator<typename seed_t::base_t>;
00121
00122 seed_t global_seed;
```

```
00123
00124
00125 #include "metropolis_hastings/configuration/configuration_vector.hpp"
00126 #include "metropolis_hastings/configuration/configuration_generator.hpp"
00127
00128 constexpr const bool use_bitfield = false;
00129 using cfg_vector_t = binary_configuration_vector<use_bitfield>;
00130 // using cfg_vector_t = binary_configuration_vector;
00131
00132 std::size_t cfg_vector_length;
00133
00134 using cfg_generator_t = std::function<void(prn_engine_t &, cfg_vector_t &)>;
00135
00136 uniform_configuration_generator<cfg_vector_t> ugen;
00137 cfg_generator_t default_cfg_generator = cfg_generator_t(ugen);
00138
00139 std::map<std::size_t, cfg_generator_t> mc_starting_with_special_cfg_vector;
00140
00141 #include "metropolis_hastings/configuration/change_generator.hpp"
00142 constexpr const bool enforce_change = false;
00143 constexpr const bool skip_unchanged_vectors = true;
00144 using change_generator_t = simple_change_generator<cfg_vector_t, enforce_change>;
00145
00146 #include "metropolis_hastings/value_computation/utility/probsat-execution.hpp"
00147 #include "metropolis_hastings/value_computation/modell.hpp"
00148 #include "metropolis_hastings/value_computation/probsat-with-resolvents.hpp"
00149
00150 using probsat_modell_t = probsat_resolvents<cfg_vector_t, prec_t, use_bitfield>;
00151 using computation_modell_t = probsat_modell_t;
00152
00153 #include "metropolis_hastings/mcmc.hpp"
00154 using marcov_chain_state_t = marcov_chain_state<cfg_vector_t>;
00155
00156 std::map<std::size_t, marcov_chain_state_t> mc_chains_to_add;
00157
00158 #include "metropolis_hastings/acceptance_computation/simple_acceptance_computation.hpp"
00159 #include "metropolis_hastings/acceptance_computation/statistical_acceptance_computation.hpp"
00160 using acc_prob_comp_t = exponential_acceptance_probability<prec_t>;
00161 // using acceptance_computation_t =
00162     // simple_acceptance_computation<prn_engine_t, cfg_vector_t, prec_t, probsat_modell_t,
       acc_prob_comp_t>;
00163 using acceptance_computation_t =
00164
       statistical_acceptance_computation<prn_engine_t, cfg_vector_t, prec_t, probsat_modell_t, acc_prob_comp_t>;
00165
00166 #include "metropolis_hastings/algorithm.hpp"
00167 using metropolis_hastings_algorithm_t =
00168     metropolis_hastings_algorithm<
00169         marcov_chain_state_t,
00170         change_generator_t,
00171         acceptance_computation_t, computation_modell_t, acc_prob_comp_t,
00172         result_statistics<prec_t>,
00173         std::function<void(std::pair<uint64_t, probsat_return_cause::reason>)>,
00174         enforce_change,
00175         skip_unchanged_vectors
00176     >;
00177
00178
00179 #include "distributed_computation/rescheduling.hpp"
00180 std::unique_ptr<rescheduling_manager> reschedule_manager = nullptr;
00181
00182 std::map<std::size_t, int> marcov_chains_to_move = {};
00183
00184 std::set<std::size_t> marcov_chains_finished;
00185
00186 std::ptrdiff_t markov_chains_to_get = 0;
00187 bool rescheduling_in_progress = false;
00188
00189
00190 bool running = true;
00191
00192 void stop_running() {
00193     running = false;
00194 }
00195
00196 // void test_rcg(int from, int to) {
00197     // std::cout « "test_rcg: hello from " « from « " to " « to « std::endl;
00198 // }
00199
00200 void got_markov_chain(std::ptrdiff_t n = -1) {
00201     markov_chains_to_get += n;
00202     std::cout « "got_markov_chain + " « n « " = " « markov_chains_to_get
00203         « " on " « workgroup_ptr->workgroup_id « std::endl;
00204     if (0 == markov_chains_to_get) {
00205         assert(rescheduling_in_progress);
00206         rescheduling_in_progress = false;
00207         std::cout « "rescheduling finished on wg " « workgroup_ptr->workgroup_id « std::endl;
```

```
00208     }
00209 }
00210
00211 /*
00212 auto serialize_mcs(auto mc_state) {
00213     type_abstraction_adapter_t bitsery_taa;
00214     buffer_t b = bitsery_taa.serialize(mc_state);
00215     // std::cout « "serialize mcs(" « mc_state.iteration « ", ["
00216         // « mc_state.cfg_vector.data.size() « "], "
00217         // « mc_state.prn_engine « "}) on " « workgroup_ptr->workgroup_id « std::endl;
00218
00219     std::size_t checksum = 0;
00220     for (auto v : b) { checksum += (v?1:0); checksum *= 3; }
00221     std::cout « "serialize: " « checksum « " on " « workgroup_ptr->workgroup_id « std::endl;
00222     // std::cout « "serialize:"; for (auto v : b) { std::cout « " " « (v?1:0); } std::cout «
      std::endl;
00223     return b;
00224 }
00225
00226 void recieve_markov_chain(std::size_t id, buffer_t mcs) {
00227     ignore(id, mcs);
00228     std::cout « "TEST recieve" « std::endl;
00229
00230     std::size_t checksum = 0;
00231     for (auto v : mcs) { checksum += (v?1:0); checksum *= 3; }
00232     std::cout « "recieve: " « checksum « " on " « workgroup_ptr->workgroup_id « std::endl;
00233     // std::cout « "serialize:"; for (auto v : mcs) { std::cout « " " « (v?1:0); } std::cout «
      std::endl;
00234
00235     // type_abstraction_adapter_t bitsery_taa;
00236     // marcov_chain_state_t mc_state(cfg_vector_length, prn_engine_t());
00237     // assert(bitsery_taa.deserialize(mcs, mc_state));
00238
00239     // std::cout « "recieve_markov_chain(" « id « ", {" « mc_state.iteration « ", ["
00240         // « mc_state.cfg_vector.data.size() « "], "
00241         // « mc_state.prn_engine « "}) on " « workgroup_ptr->workgroup_id « std::endl;
00242
00243     got_markov_chain();
00244 }
00245 */
00246
00247 /*
00248 void recieve_markov_chain(const std::size_t &id, const marcov_chain_state_t &mc_state) {
00249     std::cout « "TEST recieve" « std::endl;
00250
00251     std::cout « "recieve_markov_chain(" « id « ", {" « mc_state.iteration « ", ["
00252         « mc_state.cfg_vector.data.size() « ", " « mc_state.cfg_vector.count_ones() « "], "
00253         « mc_state.prn_engine « "}) on " « workgroup_ptr->workgroup_id « std::endl;
00254
00255     got_markov_chain();
00256 }
00257 */
00258
00259 void recieve_markov_chain(std::size_t id, const marcov_chain_state_t &mcs) {
00260     std::cout « "recieve_markov_chain(" « id « ", {" « mcs.iteration « ", ["
00261         « mcs.cfg_vector.data.size() « "], ...}) on " « workgroup_ptr->workgroup_id « std::endl;
00262     mc_chains_to_add.emplace(id, mcs);
00263     got_markov_chain();
00264 }
00265
00266 void move_markov_chain(std::size_t id, int to) {
00267     std::cout « "move_markov_chain(" « id « ", " « to « ") on " « workgroup_ptr->workgroup_id «
      std::endl;
00268     if (workgroup_ptr->workgroup_id == to) {
00269         assert(0 == markov_chains_to_get);
00270         got_markov_chain(id);
00271     } else {
00272         std::cout « "XXX marcov_chains_to_move[" « id « "] = " « to « std::endl;
00273         marcov_chains_to_move[id] = to;
00274     }
00275 }
00276
00277 bool send_progress_info = false;
00278 void request_progress_info() {
00279     // std::cout « "request_progress_info() on " « workgroup_ptr->workgroup_id « std::endl;
00280     send_progress_info = true;
00281     assert(0 == markov_chains_to_get);
00282     markov_chains_to_get = 0;
00283     rescheduling_in_progress = true;
00284     // std::cout « "rescheduling started on wg " « workgroup_ptr->workgroup_id « std::endl;
00285 }
00286
00287 void pass_progress_info(std::size_t id, std::size_t progress, int wg_id) {
00288     // std::cout « "request_progress_info(" « id « ", " « progress « ", " « wg_id
00289         // « ") on " « workgroup_ptr->workgroup_id « std::endl;
00290     assert(reschedule_manager);
00291     reschedule_manager->add_progress_for_id(id, progress, wg_id);
```

```
00292 }
00293
00294 // TODO: rescheduling done
00295
00296 bool request_rescheduling()
00297 {
00298     assert(reschedule_manager);
00299     assert(wg_leaders);
00300
00301     if (reschedule_manager->is_rescheduling_in_progress()) {
00302         return false;
00303     }
00304
00305     reschedule_manager->start_rescheduling();
00306
00307     for (int i = 0; i < workgroup_ptr->workgroup_count; i++) {
00308         auto cmd = wg_leaders->rpmanager.prepare_call("request_progress_info");
00309         wg_leaders->rcg.schedule(cmd, i);
00310     }
00311
00312     return true;
00313 }
00314
00315
00316 void add_new_mc_chain(std::size_t id, seed_t &mc_seed)
00317 {
00318     auto mc_sseq = mc_seed.get_seed_seq();
00319     marcov_chain_state_t mc_state(cfg_vector_length, prn_engine_t(mc_sseq));
00320
00321     auto search = mc_starting_with_special_cfg_vector.find(id);
00322     if (mc_starting_with_special_cfg_vector.end() != search) {
00323         (search->second)(mc_state.prn_engine, mc_state.cfg_vector);
00324     } else {
00325         default_cfg_generator(mc_state.prn_engine, mc_state.cfg_vector);
00326     }
00327
00328     mc_chains_to_add.emplace(id, mc_state);
00329
00330     {
00331         // type_abstraction_adapter_t bitsery_taa;
00332         // buffer_t b = bitsery_taa.serialize(mc_state);
00333
00334         // marcov_chain_state_t mc_state_(cfg_vector_length, prn_engine_t());
00335         // assert(bitsery_taa.deserialize(b, mc_state_));
00336         // assert(mc_state == mc_state_);
00337
00338         // reschedule_required = true;
00339         // auto id_fwidth = std::setw(std::to_string(num_markov_chains).length());
00340         // std::cout « "c id: " « id_fwidth « id « " with " « cfg_vector « std::endl;
00341
00342         // type_abstraction_adapter_t bitsery_taa;
00343         // buffer_t b = bitsery_taa.serialize(cfg_vector);
00344         // cfg_vector_t cmp_cfg_vector(cfg_vector_length);
00345         // assert(bitsery_taa.deserialize(b, cmp_cfg_vector));
00346         // assert(cmp_cfg_vector == cfg_vector);
00347         // std::cout « "c id: " « id_fwidth « id « " with " « cmp_cfg_vector « std::endl;
00348     }
00349
00350     {
00351         // auto prn1 = mc_state.prn_engine;
00352         // auto prn2 = mc_state.prn_engine;
00353         // simple_change_generator_t change_gen(mc_state.cfg_vector);
00354
00355         // type_abstraction_adapter_t bitsery_taa;
00356         // buffer_t b = bitsery_taa.serialize(change_gen);
00357         // auto c1 = change_gen(prn1, mc_state.cfg_vector);
00358         // auto c2 = change_gen(prn1, mc_state.cfg_vector);
00359         // auto c3 = change_gen(prn1, mc_state.cfg_vector);
00360         // std::cout « c1.index « ": " « c1.new_value « std::endl;
00361         // std::cout « c2.index « ": " « c2.new_value « std::endl;
00362         // std::cout « c3.index « ": " « c3.new_value « std::endl;
00363
00364         // assert(bitsery_taa.deserialize(b, change_gen));
00365         // assert(c1 == change_gen(prn2, mc_state.cfg_vector));
00366         // assert(c2 == change_gen(prn2, mc_state.cfg_vector));
00367         // assert(c3 == change_gen(prn2, mc_state.cfg_vector));
00368     }
00369 }
00370
00371 std::string filename_cnf_formula;
00372 std::string filename_resolvents;
00373 std::string data_directory;
00374
00375 int main(int argc, char **argv)
00376 {
00377     auto start = std::chrono::high_resolution_clock::now();
00378
```

```
00379     try {
00380         MPI_Init(&argc, &argv);
00381 
00382         {
00383             int is_initialized;
00384             MPI_Initialized(&is_initialized);
00385             assert(0 != is_initialized);
00386         }
00387 
00388         int version = 0;
00389         int subversion = 0;
00390         assert(MPI_SUCCESS == MPI_Get_version(&version, &subversion));
00391         if (0 == mpi_get_comm_rank(MPI_COMM_WORLD)) {
00392             std::cout « "c MPI Version " « version « "." « subversion « std::endl;
00393         } else {
00394             MPI_Barrier(MPI_COMM_WORLD);
00395         }
00396 
00397         filename_cnf_formula = "data/three_color_gnp_50vertices_p0.092_seed37_cnfgen.cnf";
00398         filename_resolvents = "data/three_color_gnp_50vertices_p0.092_seed37_cnfgen.resolvents";
00399         data_directory = "out/three_color_gnp_50vertices_p0.092_seed37_cnfgen/";
00400 
00401         std::deque<std::string> args;
00402         for (int i = 1; i < argc; i++) {
00403             args.push_back(argv[i]);
00404         }
00405         while (!args.empty()) {
00406             if (args.front() == "-r") {
00407                 args.pop_front();
00408                 assert(!args.empty() && "missing resolvents filename");
00409                 filename_resolvents = args.front();
00410                 args.pop_front();
00411             }
00412             else if (args.front() == "-c") {
00413                 args.pop_front();
00414                 assert(!args.empty() && "missing cnf_formula filename");
00415                 filename_cnf_formula = args.front();
00416                 args.pop_front();
00417             }
00418             else if (args.front() == "-d") {
00419                 args.pop_front();
00420                 assert(!args.empty() && "missing data_directory filename");
00421                 data_directory = args.front();
00422                 args.pop_front();
00423             }
00424             else if (args.front() == "-of") {
00425                 args.pop_front();
00426                 assert(!args.empty() && "missing offset factor");
00427                 markov_chain_offset = std::stoi(args.front()) * num_markov_chains;
00428                 args.pop_front();
00429             }
00430             else if (args.front() == "-o") {
00431                 args.pop_front();
00432                 assert(!args.empty() && "missing offset value");
00433                 markov_chain_offset = std::stoi(args.front());
00434                 args.pop_front();
00435             }
00436             else if (args.front() == "-l") {
00437                 args.pop_front();
00438                 assert(!args.empty() && "missing target length");
00439                 markov_chain_target_length = std::stoi(args.front());
00440                 args.pop_front();
00441             }
00442             else if (args.front() == "-n") {
00443                 args.pop_front();
00444                 assert(!args.empty() && "missing num markov chains");
00445                 num_markov_chains = std::stoi(args.front());
00446                 args.pop_front();
00447             }
00448             else if (args.front() == "-t") {
00449                 args.pop_front();
00450                 assert(!args.empty() && "missing T value");
00451                 mcmc_T = (double) std::stoi(args.front());
00452                 args.pop_front();
00453             }
00454             else if (args.front() == "-f") {
00455                 args.pop_front();
00456                 assert(!args.empty() && "missing num_reference_chains");
00457                 num_reference_chains = std::stoi(args.front());
00458                 args.pop_front();
00459             }
00460             else {
00461                 std::cout « "unknown parameter: " « args.front() « std::endl;
00462                 exit(EXIT_FAILURE);
00463             }
00464         }
00465 
```

```
00466         if (0 == mpi_get_comm_rank(MPI_COMM_WORLD)) {
00467             std::cout « "cfg filename_cnf_formula: " « filename_cnf_formula « std::endl;
00468             std::cout « "cfg filename_resolvents: " « filename_resolvents « std::endl;
00469             std::cout « "cfg data_directory: " « data_directory « std::endl;
00470             std::cout « "cfg num_reference_chains: " « num_reference_chains « std::endl;
00471             std::cout « "cfg num_markov_chains: " « num_markov_chains « std::endl;
00472             std::cout « "cfg markov_chain_offset: " « markov_chain_offset « std::endl;
00473             std::cout « "cfg markov_chain_target_length: " « markov_chain_target_length « std::endl;
00474             std::cout « "cfg mcmc_T: " « mcmc_T « std::endl;
00475             std::cout « "num_reference_chains: " « num_reference_chains « std::endl;
00476 
00477             MPI_Barrier(MPI_COMM_WORLD);
00478         }
00479 
00480         // load sat and resolvents file
00481         probsat_modell_t probsat_modell(filename_cnf_formula, filename_resolvents);
00482         cfg_vector_length = probsat_modell.resolvents.get_num_resolvents();
00483         // cfg_vector_length = 10;
00484 
00485         // build mpi workgroups based on shared tmp directory
00486         const auto workgroup = mpi_shared_tmp_dir_workgroup(MPI_COMM_WORLD, "mcmc-sat");
00487         // const auto workgroup = mpi_shared_tmp_dir_workgroup(MPI_COMM_WORLD, "mcmc-sat" +
    std::to_string(mpi_get_comm_rank(MPI_COMM_WORLD) % 2));
00488         workgroup_ptr = &workgroup;
00489         // const auto workgroup = mpi_shared_tmp_dir_workgroup(MPI_COMM_WORLD, "numa" +
    std::to_string(mpi_get_comm_rank(MPI_COMM_WORLD)));
00490         // const auto workgroup = mpi_shared_tmp_dir_workgroup(MPI_COMM_WORLD,
    mpi_get_comm_rank(MPI_COMM_WORLD) < 3 ? "numa1" : "numa2");
00491         std::cout « "c parent node " « workgroup.parent_comm_id + 1 « " / " «
    workgroup.parent_comm_size
00492                  « " has workgroup rank " « workgroup.workgroup_comm_id + 1 « " / " «
    workgroup.workgroup_comm_size
00493                  « " in workgoup " « workgroup.workgroup_id + 1 « " / " « workgroup.workgroup_count «
    std::endl;
00494 
00495         std::random_device rdev; std::size_t random_nonce = rdev();
00496         probsat_modell.workgroup_description = "wg" + std::to_string(workgroup.workgroup_id + 1) +
    "_r" + std::to_string(random_nonce);
00497         //probsat_modell.workgroup_description = "wg" + std::to_string(workgroup.workgroup_id + 1);
00498 
00499         // register functions
00500         wg_rpmanager.add_function("stop_running", stop_running);
00501         wg_rpmanager.add_function("execute_probsat", execute_probsat);
00502         wg_rpmanager.add_function("modell_identity", modell_identity<prec_t>);
00503         wg_rpmanager.add_function("modell_multiply", modell_multiply<prec_t>);
00504 
00505 
00506         // TODO parse config
00507         constant_configuration_generator<cfg_vector_t> cgen_false(false);
00508         constant_configuration_generator<cfg_vector_t> cgen_true(true);
00509         cfg_generator_t cfg_generator_const_false = cfg_generator_t(cgen_false);
00510         cfg_generator_t cfg_generator_const_true = cfg_generator_t(cgen_true);
00511         // mc_starting_with_special_cfg_vector[0] = cfg_generator_const_false;
00512         //mc_starting_with_special_cfg_vector[1] = cfg_generator_const_false;
00513         //mc_starting_with_special_cfg_vector[2] = cfg_generator_const_true;
00514         // mc_starting_with_special_cfg_vector[3] = cfg_generator_const_true;
00515         for (std::size_t i = 0; i < num_reference_chains; i++) {
00516             mc_starting_with_special_cfg_vector[i] = cfg_generator_const_false;
00517             mc_starting_with_special_cfg_vector[num_reference_chains+i] = cfg_generator_const_true;
00518         }
00519 
00520         acc_prob_comp_t acc_prob_fct(mcmc_T);
00521 
00522         // global seed
00523         if (0 == workgroup.parent_comm_id)
00524         {
00525             std::string seed_filename = data_directory + "/global_seed.txt";
00526             global_seed = get_persistent_global_seed<seed_t>(seed_filename);
00527             std::cout « "c global seed: " « global_seed.short_rep() « "..." « std::endl;
00528         }
00529         assert(MPI_SUCCESS == MPI_Bcast(
00530             global_seed.seed_data.data(), global_seed.seed_data.size(),
00531             get_mpi<seed_t::base_t>::type(), 0, workgroup.parent_comm));
00532 
00533 
00534         if (workgroup.is_workgroup_head())
00535         { // distributed generation of markov chain seeds
00536             auto sseq = global_seed.get_seed_seq();
00537             prn_engine_t seed_engine(sseq);
00538             seed_type_generator_t sgen;
00539 
00540             const std::size_t num_workgroups = workgroup.workgroup_count;
00541             const std::size_t workgroup_id = workgroup.workgroup_id;
00542 
00543             std::size_t num_markov_chains_per_work_group = num_markov_chains / num_workgroups;
00544             std::size_t num_markov_chains_in_this_work_group = num_markov_chains_per_work_group
00545                 + ((workgroup_id < num_markov_chains % num_workgroups) ? 1 : 0);
```

```
00546
00547             std::size_t markov_chains_id_offset = ((workgroup_id > (num_markov_chains %
    num_workgroups))
00548                 ? workgroup_id - (num_markov_chains % num_workgroups) : 0) *
    num_markov_chains_per_work_group
00549                 + std::min(workgroup_id, num_markov_chains % num_workgroups) *
    (num_markov_chains_per_work_group + 1);
00550             std::size_t local_markov_chain_offset = markov_chain_offset + markov_chains_id_offset;
00551
00552             std::cout « "c work group " « workgroup_id « " has " «
    num_markov_chains_in_this_work_group
00553                 « " markov chains at id " « markov_chains_id_offset « " (offset " «
    local_markov_chain_offset « ")" « std::endl;
00554
00555             // skip seeds for markov_chain_offset markov chains
00556             for (std::size_t i = 0; i < local_markov_chain_offset; i++) {
00557                 seed_t mc_seed = seed_t(seed_engine, &sgen);
00558                 ignore(mc_seed);
00559             }
00560
00561             auto id_fwidth = std::setw(std::to_string(num_markov_chains).length());
00562             std::size_t mv_id_start = local_markov_chain_offset; // markov_chains_id_offset
00563             std::size_t mv_id_end = mv_id_start + num_markov_chains_in_this_work_group;
00564             for (std::size_t id = mv_id_start; id < mv_id_end; id++) {
00565                 seed_t mc_seed = seed_t(seed_engine, &sgen);
00566                 std::cout « "c markov chain " « id_fwidth « id « " with seed "
00567                     « std::setw(13) « mc_seed.short_rep() « "..."
00568                     « "\t" « "(on workgroup " « workgroup_id « ")" « std::endl;
00569
00570                 add_new_mc_chain(id, mc_seed);
00571             }
00572         }
00573
00574         // if (0 == workgroup.parent_comm_id) {
00575             // std::cout « "continue?" « std::endl;
00576             // std::string s;
00577             // std::cin » s;
00578             // MPI_Barrier(MPI_COMM_WORLD);
00579         // }
00580
00581         const bool use_mc_limit = 0 < markov_chain_target_length;
00582
00583         using priority_id_pair_t = std::pair<std::size_t, std::size_t>;
00584         auto priority_cmp = [use_mc_limit](priority_id_pair_t left, priority_id_pair_t right) {
00585             // in case limit is set: sort by work to do, else sort after work done
00586             return use_mc_limit ? left.first < right.first : left.first > right.first;
00587         };
00588         std::priority_queue<
00589             priority_id_pair_t,
00590             std::vector<priority_id_pair_t>,
00591             decltype(priority_cmp)
00592         > work_queue(priority_cmp);
00593
00594         std::map<std::size_t, metropolis_hastings_algorithm_t> mh_algorithms;
00595
00596
00597
00598
00599         remote_execution_context_t wg_remote_exec_context(wg_rpmanager);
00600         on_demand_scheduler wg_scheduler(workgroup.workgroup_comm, 3, wg_remote_exec_context);
00601
00602         if (workgroup.is_workgroup_head()) {
00603             assert(0 == wg_scheduler.env_comm_id); // otherwise renumbering would be required
00604
00605             wg_leaders = std::make_unique<wg_leader_exec_env>(workgroup.mpi_comm_leaders, 4);
00606
00607             if (1 < workgroup.workgroup_count) {
00608                 if (0 == workgroup.workgroup_id) {
00609                     reschedule_manager = std::make_unique<rescheduling_manager>(
00610                         workgroup.workgroup_count, num_markov_chains, use_mc_limit);
00611                     wg_leaders->rpmanager.add_function("request_rescheduling", request_rescheduling);
00612                     wg_leaders->rpmanager.add_function("pass_progress_info", pass_progress_info);
00613                 }
00614
00615                 wg_leaders->rpmanager.add_function("request_progress_info", request_progress_info);
00616                 wg_leaders->rpmanager.add_function("move_markov_chain", move_markov_chain);
00617                 wg_leaders->rpmanager.add_function("recieve_markov_chain", recieve_markov_chain);
00618             }
00619
00620
00621 /* WG Leaders Test
00622             wg_leaders->rpmanager.add_function("test_rcg", test_rcg);
00623
00624             for (int i = 0; i < workgroup.workgroup_count; i++) {
00625                 auto cmd = wg_leaders->rpmanager.prepare_call("test_rcg", workgroup.workgroup_id, i);
00626
00627                 std::function<void()> result_fct = [i, &workgroup]() {
```

```
00628                     std::cout « "wg " « workgroup.workgroup_id « " got ok for " « i « std::endl;
00629                 };
00630                 cmd = wg_leaders->remote_exec_context.on_result(cmd, result_fct);
00631 
00632                 wg_leaders->rcg.schedule(cmd, i);
00633             }
00634 
00635             while (wg_leaders->rcg.is_work_outstanding()) {
00636                 wg_leaders->rcg.do_work();
00637             }
00638 
00639             MPI_Barrier(workgroup.mpi_comm_leaders);
00640             if (0 == workgroup.workgroup_id) {
00641                 std::cout « "wg leader test finished" « std::endl;
00642             }
00643 */
00644         }
00645 
00646         std::function<void(std::size_t, int, int)> do_rescheduling =
00647             [](std::size_t id, int from, int to) {
00648                 auto cmd = wg_leaders->rpmanager.prepare_call("move_markov_chain", id, to);
00649                 wg_leaders->rcg.schedule(cmd, from);
00650         };
00651 
00652 
00653 /*
00654         for (auto & [id, mc_state] : mc_chains_to_add) {
00655             const bool already_done = use_mc_limit && (mc_state.iteration >=
      markov_chain_target_length);
00656 
00657             if (already_done) {
00658                 std::cout « "c markov chain " « id « " has already reached"
00659                     « " computation limit of length " « markov_chain_target_length « std::endl;
00660             } else {
00661                 std::size_t priority = use_mc_limit
00662                     ? markov_chain_target_length - mc_state.iteration
00663                     : mc_state.iteration;
00664 
00665                 mh_algorithms.emplace(id, metropolis_hastings_algorithm_t(id, mc_state,
      probsat_modell, acc_prob_fct));
00666                 // assert(mc_state == mh_algorithms.at(id).get_last_state());
00667 
00668                 work_queue.push(std::make_pair(priority, id));
00669             }
00670         }*/
00671 
00672 
00673 
00674         std::size_t actual_working_time = 0;
00675         decltype(std::chrono::high_resolution_clock::now()) start_work, end_work;
00676         if (workgroup.is_workgroup_head())
00677         {
00678             // bool restarted[2] = {false};
00679             // bool accepted[8] = {true, true, false, true, true, true, true, true};
00680             // std::size_t skip_from = 1;
00681             // std::size_t skip_amount = 4;
00682             do {
00683                 // std::cout « workgroup.workgroup_id « " XXX " « "main loop start" « std::endl;
00684 
00685                 if (!mc_chains_to_add.empty()) {
00686                     // std::cout « workgroup.workgroup_id « " XXX " « "mc_chains_to_add" « std::endl;
00687 
00688                     for (auto & [id, mc_state] : mc_chains_to_add) {
00689                         const bool already_done = use_mc_limit && (mc_state.iteration >=
      markov_chain_target_length);
00690 
00691                         if (already_done) {
00692                             // std::cout « "c moved markov chain " « id « " has already reached"
00693                                 // « " computation limit of length " « markov_chain_target_length «
      std::endl;
00694                         } else {
00695                             assert(mc_state.cfg_vector.data.size() == cfg_vector_length);
00696                             // std::cout « "c adding mc " « id « " in iteration " « mc_state.iteration
      « std::endl;
00697 
00698                             std::size_t priority = use_mc_limit
00699                                 ? markov_chain_target_length - mc_state.iteration
00700                                 : mc_state.iteration;
00701 
00702                             mh_algorithms.emplace(id, metropolis_hastings_algorithm_t(id, mc_state,
      probsat_modell, acc_prob_fct));
00703 
00704                             work_queue.push(std::make_pair(priority, id));
00705                         }
00706                     }
00707 
00708                     mc_chains_to_add.clear();
```

```
00709                 }
00710
00711                 // std::cout « workgroup.workgroup_id « " XXX " « "main loop p1" « std::endl;
00712
00713                 if (!work_queue.empty()) {
00714                     // std::cout « workgroup.workgroup_id « " XXX " « "!work_queue empty" « std::endl;
00715                     bool was_moved = false;
00716                     do {
00717                         const auto [priority, id] = work_queue.top();
00718                         was_moved = !mh_algorithms.contains(id);
00719                         if (was_moved) { work_queue.pop(); }
00720                     } while ((!work_queue.empty()) && (was_moved));
00721                 }
00722
00723                 std::size_t iteration = 0;
00724                 if (work_queue.empty()) {
00725                     // std::cout « workgroup.workgroup_id « " XXX " « "work_queue empty" « std::endl;
00726                     running = rescheduling_in_progress || wg_leaders->rcg.is_work_outstanding() ||
      !marcov_chains_to_move.empty();
00727                 } else {
00728                     // std::cout « workgroup.workgroup_id « " XXX " « "process work" « std::endl;
00729                     /*
00730                     const auto [priority, id] = work_queue.top();
00731                     work_queue.pop();
00732
00733                     // std::cout « "c processing mc " « id « " with priority " « priority « std::endl;
00734
00735                     bool is_last_iteration = use_mc_limit && (1 == priority);
00736
00737                     assert(mh_algorithms.contains(id));
00738                     metropolis_hastings_algorithm_t &mha = mh_algorithms.at(id);
00739                     */
00740                     std::size_t priority, id;
00741                     std::queue<std::pair<std::size_t, std::size_t» tmp_queue;
00742                     while (true) {
00743                         const auto [priority_, id_] = work_queue.top();
00744                         priority = priority_; id = id_;
00745                         work_queue.pop();
00746                         if (work_queue.empty()) {
00747                             break;
00748                         }
00749
00750                         assert(mh_algorithms.contains(id));
00751                         metropolis_hastings_algorithm_t &mha = mh_algorithms.at(id);
00752                         if (mha.still_waiting_for_computation()) {
00753                             tmp_queue.push(std::make_pair(priority, id));
00754                         } else {
00755                             break;
00756                         }
00757                     }
00758
00759                     while (!tmp_queue.empty()) {
00760                         work_queue.push(tmp_queue.front());
00761                         tmp_queue.pop();
00762                     }
00763
00764                     // if (mha.can_start_next_iteration()) {
00765                         // if (restarted[id]) {
00766                             // if (skip_from == mha.get_iteration()) {
00767                                 // for (std::size_t i = skip_from; i < skip_from + skip_amount; i++) {
00768                                     // std::cout « "skipping " « i « std::endl;
00769                                     // mha.skip_computation(accepted[i]);
00770                                 // }
00771                             // }
00772                         // }
00773                     // }
00774
00775                     //iteration = mha(wg_scheduler, is_last_iteration);
00776                     //wg_scheduler.do_work();
00777
00778                     assert(mh_algorithms.contains(id));
00779                     metropolis_hastings_algorithm_t &mha = mh_algorithms.at(id);
00780
00781                     bool is_last_iteration = use_mc_limit && (1 == priority);
00782                     iteration = mha(wg_scheduler, is_last_iteration);
00783                     wg_scheduler.do_work();
00784
00785                     // if (mha.can_start_next_iteration()) {
00786                         // marcov_chain_state_t mcs = mha.get_last_state();
00787                         // if (1 == mcs.iteration) {
00788                             // if (!restarted[id]) {
00789                                 // std::cout « "restart " « id « std::endl;
00790                                 // restarted[id] = true;
00791                                 // mha.cleanup();
00792                                 // mh_algorithms.erase(id);
00793                                 // mh_algorithms.emplace(id, metropolis_hastings_algorithm_t(id, mcs,
      probsat_modell, acc_prob_fct));
```

```
00794                             // }
00795                         // }
00796                     // }
00797
00798                     std::size_t new_priority = use_mc_limit
00799                         ? markov_chain_target_length - iteration
00800                         : iteration;
00801
00802                     bool is_done = use_mc_limit && (0 == new_priority);
00803                     if (is_done) {
00804                         mha.cleanup();
00805                         mh_algorithms.erase(id);
00806                         marcov_chains_finished.insert(id);
00807                         // TODO: inform wg 0
00808                     } else {
00809                         work_queue.push(std::make_pair(new_priority, id));
00810                     }
00811                 }
00812
00813                 // std::cout « workgroup.workgroup_id « " XXX " « "done with work, communicate" «
      std::endl;
00814
00815                 wg_leaders->rcg.do_work();
00816
00817                 if (!marcov_chains_to_move.empty()) {
00818                     // std::cout « workgroup.workgroup_id « " XXX " « "mc to move" « std::endl;
00819                     for (auto const& [id, new_wg] : marcov_chains_to_move) {
00820                         // std::cout « "MC TO MOVE: " « id « " on " « workgroup.workgroup_id «
      std::endl;
00821                         auto search = marcov_chains_finished.find(id);
00822                         if (search != marcov_chains_finished.end()) {
00823                             std::cout « "moving dummy for id " « id « " id to " « new_wg « std::endl;
00824
00825                             marcov_chain_state_t mcs(cfg_vector_length);
00826                             // buffer_t b = serialize_mcs(mcs);
00827                             auto cmd = wg_leaders->rpmanager.prepare_call("recieve_markov_chain", id,
      mcs);
00828                             wg_leaders->rcg.schedule(cmd, new_wg);
00829                         } else {
00830                             std::cout « "prepare movement for " « id « " on " « workgroup.workgroup_id
      « std::endl;
00831                             assert(mh_algorithms.contains(id));
00832                             metropolis_hastings_algorithm_t &mha = mh_algorithms.at(id);
00833                             while (!mha.can_start_next_iteration()) {
00834                                 bool is_last_iteration = use_mc_limit && (markov_chain_target_length
      == iteration);
00835                                 iteration = mha(wg_scheduler, is_last_iteration);
00836                                 wg_scheduler.do_work();
00837                                 wg_leaders->rcg.do_work();
00838                             }
00839
00840                             // std::cout « "get last state for " « id « " on " «
      workgroup.workgroup_id « std::endl;
00841                             marcov_chain_state_t mcs = mha.get_last_state();
00842
00843                             {
00844                                 // buffer_t b = serialize_mcs(mcs);
00845                                 auto cmd = wg_leaders->rpmanager.prepare_call("recieve_markov_chain",
      id, mcs);
00846                                 wg_leaders->rcg.schedule(cmd, new_wg);
00847                             }
00848
00849                             {
00850                                 mha.cleanup();
00851                                 mh_algorithms.erase(id);
00852                             }
00853                         }
00854                     }
00855                     marcov_chains_to_move.clear();
00856                 }
00857
00858                 // std::cout « workgroup.workgroup_id « " XXX " « "loop p 622" « std::endl;
00859
00860                 if (send_progress_info) {
00861                     // std::cout « workgroup.workgroup_id « " XXX " « "send_progress_info" «
      std::endl;
00862                     for (auto const& [id, mh_alg] : mh_algorithms) {
00863                         std::size_t num_it = mh_alg.get_iteration();
00864                         std::size_t progress = num_it; // use_mc_limit ? markov_chain_target_length -
      num_it : num_it;
00865
00866                         // if (0 == workgroup.workgroup_id) {
00867                             // pass_progress_info(id, progress, workgroup.workgroup_id);
00868                         // } else {
00869                         auto cmd = wg_leaders->rpmanager.prepare_call("pass_progress_info", id,
      progress, workgroup.workgroup_id);
00870                         wg_leaders->rcg.schedule(cmd, 0);
```

```
00871                         // }
00872                     }
00873
00874                     send_progress_info = false;
00875                 }
00876
00877                 // std::cout « workgroup.workgroup_id « " XXX " « "loop p641" « std::endl;
00878
00879                 if ((0 == workgroup.workgroup_id) && (1 < workgroup.workgroup_count)) {
00880                     static bool move_requested = false;
00881                     if ((!move_requested) && (2 == iteration)) {
00882                         request_rescheduling();
00883                         move_requested = true;
00884                     }
00885
00886                     assert(reschedule_manager);
00887                     if (reschedule_manager->is_rescheduling_in_progress()) {
00888                         if (reschedule_manager->information_complete()) {
00889                             std::cout « "rescheduling algorithm start" « std::endl;
00890                             reschedule_manager->reschedule(do_rescheduling);
00891                             std::cout « "rescheduling algorithm end" « std::endl;
00892                         }
00893                     }
00894                 }
00895
00896                 wg_leaders->rcg.do_work();
00897                 // std::cout « workgroup.workgroup_id « " XXX " « "main loop end" « std::endl;
00898
00899             } while (running);
00900
00901             std::cout « "rescheduling_in_progress: " « rescheduling_in_progress « std::endl;
00902             std::cout « "marcov_chains_to_move.size(): " « marcov_chains_to_move.size() « std::endl;
00903
00904             std::cout « "wg " « workgroup.workgroup_id « " is done running" « std::endl;
00905
00906             while (wg_scheduler.is_work_outstanding() || wg_leaders->rcg.is_work_outstanding()) {
00907                 wg_leaders->rcg.do_work();
00908                 wg_scheduler.do_work();
00909             }
00910
00911             auto stop_call = wg_rpmanager.prepare_call("stop_running");
00912             for (int i = 1; i < workgroup.workgroup_comm_size; i++) {
00913                 wg_scheduler.schedule(stop_call, i);
00914             }
00915         } else {
00916             start_work = std::chrono::high_resolution_clock::now();
00917             while (running) {
00918                 actual_working_time += wg_scheduler.do_work(true);
00919             }
00920             end_work = std::chrono::high_resolution_clock::now();
00921         }
00922
00923
00924         auto end = std::chrono::high_resolution_clock::now();
00925         auto overall_duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
00926         std::cout « "c duration: " « overall_duration.count() « " milliseconds" « std::endl;
00927
00928         if (!workgroup.is_workgroup_head()) {
00929             auto work_duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_work -
    start_work);
00930             auto init_duration = overall_duration.count() - work_duration.count();
00931             auto efficiency = 100.0 * (double) actual_working_time / (double) work_duration.count();
00932
00933             std::cout « "c init: " « init_duration « " ms, work: " « work_duration.count() « " ms with
"
00934                 « std::setprecision(2) « std::fixed « efficiency « " % efficiency" « std::endl;
00935         }
00936
00937         MPI_Barrier(MPI_COMM_WORLD);
00938
00939         /*
00940     } catch (const std::exception& ex) {
00941         std::cout « ex.what() « std::endl;
00942     }
00943
00944     try {
00945     */
00946         {
00947             int is_initialized;
00948             MPI_Initialized(&is_initialized);
00949             assert(is_initialized);
00950
00951             MPI_Barrier(MPI_COMM_WORLD);
00952             MPI_Finalize();
00953
00954             int is_finalized;
00955             MPI_Finalized(&is_finalized);
```

```
00956             assert(is_finalized);
00957         }
00958     } catch (const std::exception& ex) {
00959         std::cout « ex.what() « std::endl;
00960     }
00961
00962     return EXIT_SUCCESS;
00963 }
```

7.14 acceptance_probability.hpp

```
00001 #ifndef ACCEPTANCE_PROPABILITY_HPP
00002 #define ACCEPTANCE_PROPABILITY_HPP
00003
00004 #include <string>
00005
00007 template <typename prec_t>
00008 class exponential_acceptance_probability
00009 {
00010     private:
00012         prec_t T;
00013
00014     public:
00015         using prec_type = prec_t;
00016         constexpr static bool is_acceptance_probability_implementation = true;
00017
00018         std::string get_description() {
00019             return std::string("exp-T-") + std::to_string(T);
00020         }
00021
00023         exponential_acceptance_probability(prec_t T_) : T(T_) {}
00024
00026         prec_t operator()(prec_t old_value, prec_t new_value) const {
00027             prec_t result = exp( -(new_value - old_value) / T );
00028             result = std::max(0.0, std::min(result, 1.0));
00029
00030             // std::cout « "new: " « new_value « std::endl;
00031             // std::cout « "old: " « old_value « std::endl;
00032             // std::cout « "x: " « (new_value - old_value) « std::endl;
00033             // std::cout « "result: " « result « std::endl;
00034
00035             return result;
00036         }
00037 };
00038
00039 #endif
```

7.15 simple_acceptance_computation.hpp

```
00001 #ifndef SIMPLE_ACCEPTANCE_COMPUTATION_HPP
00002 #define SIMPLE_ACCEPTANCE_COMPUTATION_HPP
00003
00004
00005 #include <random>
00006 #include <cassert>
00007 #include <functional>
00008
00009 #include "acceptance_probability.hpp"
00010
00011 #include "../../util/seed.hpp"
00012 #include "../../config.hpp"
00013
00015 template <
00016     typename prng_t,
00017     typename cfg_vector_t,
00018     typename prec_t,
00019     typename value_comp_t,
00020     typename acc_prob_comp_t
00021         = exponential_acceptance_probability<prec_t>>
00022 requires (
00023     value_comp_t::is_value_computation_implementation
00024     && acc_prob_comp_t::is_acceptance_probability_implementation
00025 )
00026 class simple_acceptance_computation
00027 {
00028     private:
00030         std::uniform_real_distribution<prec_t> prob_distribution;
00031
00033         typedef struct {
```

```
00035             value_comp_t::prepared_computation pcom;
00036
00038             prng_t prng;
00039
00041             prec_t value;
00042         } state_data_t;
00043
00045         state_data_t state_data[2];
00046
00048         bool is_first_cfg_vector;
00049
00051         bool accept;
00052
00054         uint8_t new_index;
00055
00057         bool computation_scheduled_state;
00058
00060         std::size_t remaining_computations;
00061
00062
00064         auto derive_prng(prng_t &main_prng) {
00065             seed_t seed = seed_t(main_prng);
00066             auto sseq = seed.get_seed_seq();
00067             return prng_t(sseq);
00068         }
00069
00070     public:
00071         constexpr static bool is_acceptance_computation = true;
00072
00073         std::string get_description() {
00074             return std::string("sac");
00075         }
00076
00078         simple_acceptance_computation() :
00079             prob_distribution(0.0, 1.0),
00080             state_data(),
00081             is_first_cfg_vector(true),
00082             accept(false),
00083             new_index(0),
00084             computation_scheduled_state(false),
00085             remaining_computations(0)
00086         {}
00087
00088
00090         void skip_computation(prng_t &main_prng) {
00091             derive_prng(main_prng);
00092         }
00093
00094
00096         template<typename scheduler_t,
00097             typename result_projection_t = std::identity,
00098             typename result_function_t = std::function<void(prec_t)>>
00099         void start_computation(
00100             prng_t &main_prng,
00101             const cfg_vector_t &v,
00102             scheduler_t &scheduler,
00103             value_comp_t &value_comp,
00104             result_projection_t &result_projection = {})
00105         {
00106             assert(!computation_scheduled_state);
00107             computation_scheduled_state = true;
00108
00109             state_data[new_index].prng = derive_prng(main_prng);
00110
00111             result_function_t result_fct = [&](auto... results){
00112                 remaining_computations--;
00113                 auto result = result_projection(results...);
00114                 static_assert(
00115                     std::is_same<prec_t, decltype(result)>::value
00116                     && "result_projection has wrong result type!");
00117                 state_data[new_index].value = result;
00118             };
00119
00120             value_comp.prepare_computation_with_cfg_vector(
00121                 v, state_data[new_index].pcom);
00122
00123             auto cmd = value_comp(
00124                 state_data[new_index].prng,
00125                 scheduler.rec.rpmanager,
00126                 state_data[new_index].pcom);
00127             cmd = scheduler.rec.on_result(cmd, result_fct);
00128
00129             remaining_computations++;
00130             scheduler.schedule(cmd);
00131         }
00132
00133
```

```
00135         bool still_waiting_for_computation() {
00136             return (0 < remaining_computations);
00137         }
00138
00139
00141         template<typename scheduler_t>
00142         bool continue_computation(
00143             scheduler_t &scheduler,
00144             value_comp_t &value_comp,
00145             acc_prob_comp_t &acc_prob_fct)
00146         {
00147             assert(computation_scheduled_state);
00148             ignore(&scheduler, &value_comp, &acc_prob_fct);
00149             return 0 == remaining_computations;
00150         }
00151
00152
00154         bool finish_computation(
00155             value_comp_t &value_comp, acc_prob_comp_t &acc_prob_fct)
00156         {
00157             assert(0 == remaining_computations);
00158             assert(computation_scheduled_state);
00159             computation_scheduled_state = false;
00160
00161             value_comp.finish_computation(state_data[new_index].pcom);
00162
00163             uint8_t old_index = 1 - new_index;
00164             if (is_first_cfg_vector) {
00165                 is_first_cfg_vector = false;
00166                 new_index = old_index;
00167                 accept = true;
00168                 return true;
00169             } else {
00170                 prec_t z = prob_distribution(state_data[new_index].prng);
00171
00172                 prec_t p = acc_prob_fct(
00173                     state_data[old_index].value,
00174                     state_data[new_index].value);
00175
00176                 bool accept = (z <= p);
00177                 if (accept) {
00178                     new_index = old_index;
00179                 }
00180
00181                 return accept;
00182             }
00183         }
00184
00185
00187         void cleanup(value_comp_t &value_comp) {
00188             ignore(value_comp);
00189         }
00190
00191
00193         prec_t get_current_value() {
00194             assert(!computation_scheduled_state);
00195             uint8_t index = 1 - new_index;
00196             return state_data[index].value;
00197         }
00198
00200         prec_t get_last_computed_value() {
00201             assert(!computation_scheduled_state);
00202             uint8_t index = accept ? 1 - new_index : new_index;
00203             return state_data[index].value;
00204         }
00205
00207         prec_t get_previous_computed_value() {
00208             assert(!computation_scheduled_state);
00209             uint8_t index = new_index;
00210             return state_data[index].value;
00211         }
00212
00213 /*
00214         prec_t get_value() {
00215             assert(!computation_scheduled_state);
00216             return state_data[1 - new_index].value;
00217         }
00218 */
00219 };
00220
00221 #endif
```

7.16 statistical_acceptance_computation.hpp

```
00001 #ifndef STATISTICAL_ACCEPTANCE_COMPUTATION_HPP
00002 #define STATISTICAL_ACCEPTANCE_COMPUTATION_HPP
00003
00004 #include <random>
00005 #include <cassert>
00006 #include <functional>
00007
00008 #include "acceptance_probability.hpp"
00009
00010 #include "../../util/statistics.hpp"
00011 #include "../../util/seed.hpp"
00012 #include "../../config.hpp"
00013
00015 template<typename prec_t, class acc_prob_comp_t>
00016 auto compute_p_range(
00017     const prec_t cif,
00018     acc_prob_comp_t &acc_prob_fct,
00019     const basic_statistical_metrics<prec_t> &new_statistics,
00020     const basic_statistical_metrics<prec_t> &old_statistics)
00021 {
00022     prec_t new_value = new_statistics.average;
00023     prec_t old_value = old_statistics.average;
00024
00025     prec_t new_sigma = new_statistics.stddev;
00026     prec_t old_sigma = old_statistics.stddev;
00027
00028     prec_t new_iv[2] = {
00029         new_value + cif*new_sigma, new_value - cif*new_sigma};
00030     prec_t old_iv[2] = {
00031         old_value + cif*old_sigma, old_value - cif*old_sigma};
00032
00033     // std::cout « "\t" « "statistics:" « std::endl;
00034     // std::cout « "\t" « "new_iv: [" « new_iv[0] « ", "
00035         // « new_value « ", " « new_iv[1] « "]" « std::endl;
00036     // std::cout « "\t" « "old_iv: [" « old_iv[0] « ", "
00037         // « old_value « ", " « old_iv[1] « "]" « std::endl;
00038
00039     prec_t max_x[2] = {
00040         std::max(new_iv[0], new_iv[1]),
00041         std::min(-old_iv[0], -old_iv[1])};
00042     prec_t min_x[2] = {
00043         std::min(new_iv[0], new_iv[1]),
00044         std::max(-old_iv[0], -old_iv[1])};
00045
00046     prec_t p[2] = {
00047         acc_prob_fct(-max_x[1], max_x[0]),
00048         acc_prob_fct(-min_x[1], min_x[0])};
00049
00050     prec_t p_max = std::max(p[0], p[1]);
00051     prec_t p_min = std::min(p[0], p[1]);
00052     prec_t p_avg = acc_prob_fct(old_value, new_value);
00053
00054     /* debug output
00055     std::cout « "\t" « "stddev: [" « new_statistics.stddev « ", "
00056         « old_statistics.stddev « "]" « std::endl;
00057     std::cout « "\t" « "new range: [" « new_statistics.min « ", "
00058         « new_statistics.max « "]" « std::endl;
00059     std::cout « "\t" « "old range: [" « old_statistics.min « ", "
00060         « old_statistics.max « "]" « std::endl;
00061     std::cout « "\t" « "n: [" « new_statistics.n « ", "
00062         « old_statistics.n « "]" « std::endl;
00063     std::cout « "\t" « "new_iv: [" « new_iv[0] « ", "
00064         « new_value « ", " « new_iv[1] « "]" « std::endl;
00065     std::cout « "\t" « "old_iv: [" « old_iv[0] « ", "
00066         « old_value « ", " « old_iv[1] « "]" « std::endl;
00067     std::cout « "\t" « "max_x: [" « max_x[0] « ", "
00068         « max_x[1] « "]" « std::endl;
00069     std::cout « "\t" « "min_x: [" « min_x[0] « ", "
00070         « min_x[1] « "]" « std::endl;
00071     std::cout « "\t" « "x: [" « max_x[0] + max_x[1] « ", "
00072         « new_value - old_value « ", "
00073         « min_x[0] + min_x[1] « "]" « std::endl;
00074     std::cout « "\t" « "r: [" « p[0] « ", "
00075         « p[1] « "]" « std::endl;
00076     std::cout « "\t" « "new: [" « new_value « ", "
00077         « new_sigma « "]" « std::endl;
00078     std::cout « "\t" « "old: [" « old_value « ", "
00079         « old_sigma « "]" « std::endl;
00080     std::cout « "\t" « "p: [" « p_min « ", "
00081         « p_avg « ", " « p_max « "]" « std::endl;
00082     std::cout « std::endl;
00083     */
00084
00085     assert(p_min <= p_avg);
00086     assert(p_avg <= p_max);
```

```
00087
00088     return std::make_tuple(p_avg, p_min, p_max);
00089 }
00090
00091
00093 template <
00094     typename prng_t,
00095     typename cfg_vector_t,
00096     typename prec_t,
00097     typename value_comp_t,
00098     typename acc_prob_comp_t
00099         = exponential_acceptance_probability<prec_t>>
00100 requires (
00101     value_comp_t::is_value_computation_implementation
00102     && acc_prob_comp_t::is_acceptance_probability_implementation
00103 )
00104 class statistical_acceptance_computation
00105 {
00106     private:
00108         std::uniform_real_distribution<prec_t> prob_distribution;
00109
00111         typedef struct {
00113             value_comp_t::prepared_computation pcom;
00114
00116             prng_t prng;
00117
00119             std::vector<prec_t> values;
00120
00122             std::size_t iteration;
00123
00125             basic_statistical_metrics<prec_t> statistics;
00126         } state_data_t;
00127
00129         prec_t nextz;
00130
00132         prec_t z;
00133
00134
00136         state_data_t state_data[2];
00137
00139         bool is_first_cfg_vector;
00140
00142         bool accept;
00143
00145         uint8_t new_index;
00146
00147
00149         bool computation_scheduled_state;
00150
00152         std::size_t remaining_computations;
00153
00154
00156         double coincidence_interval_factor;
00157
00159         double testniveau;
00160
00162         std::size_t values_per_iteration;
00163
00165         std::size_t max_iterations;
00166
00168         std::size_t values_required;
00169
00171         std::function<
00172             void(void *scheduler_ptr, std::size_t i, uint8_t index)
00173         > schedule_computation;
00174
00175
00177         auto derive_prng(prng_t &main_prng) {
00178             // std::cout « "derive prng: " « main_prng « std::endl;
00179             seed_t seed = seed_t(main_prng);
00180             // std::cout « "seed: " « seed « std::endl;
00181             auto sseq = seed.get_seed_seq();
00182             // std::cout « "sseq: ";
00183             // sseq.param(std::ostream_iterator<int>(std::cout, ", "));
00184             // std::cout « std::endl;
00185             return prng_t(sseq);
00186         }
00187
00188     public:
00189         constexpr static bool is_acceptance_computation = true;
00190
00191         std::string get_description() {
00192             return std::string("cac");
00193         }
00194
00196         statistical_acceptance_computation(
00197                 // prng_t &prng,
```

```
00198                 // double coincidence_interval_factor = 1.0,
00199                 // double testniveau = 0.005,
00200                 // std::size_t values_per_iteration = 2400,
00201                 // std::size_t max_iterations = 100,
00202                 // std::size_t values_required = 10000
00203
00204                 // testing
00205                 // double coincidence_interval_factor = 1.0,
00206                 // double testniveau = 0.01,
00207                 // std::size_t values_per_iteration = 240*2,
00208                 // std::size_t max_iterations = 12,
00209                 // std::size_t values_required = 2400
00210
00211                 // configurable
00212                 double coincidence_interval_factor = cfg_sac_confidence_interval_scaling_factor,
00213                 double testniveau = cfg_sac_testniveau,
00214                 std::size_t values_per_iteration = cfg_sac_num_additional_values_per_iteration,
00215                 std::size_t max_iterations = cfg_sac_max_iterations,
00216                 std::size_t values_required = cfg_sac_num_initial_values
00217
00218                 // works
00219                 // double coincidence_interval_factor = 1.0,
00220                 // double testniveau = 0.01,
00221                 // std::size_t values_per_iteration = 240*2,
00222                 // std::size_t max_iterations = 100,
00223                 // std::size_t values_required = 10000
00224         ) :
00225             prob_distribution(0.0, 1.0),
00226             nextz(0),
00227             z(0),
00228             state_data(),
00229             is_first_cfg_vector(true),
00230             accept(false),
00231             new_index(0),
00232             computation_scheduled_state(false),
00233             remaining_computations(0),
00234             // main_prng(prng),
00235             coincidence_interval_factor(coincidence_interval_factor),
00236             testniveau(testniveau),
00237             values_per_iteration(values_per_iteration),
00238             max_iterations(max_iterations),
00239             values_required(values_required),
00240             schedule_computation(nullptr)
00241         {}
00242
00243
00244         // statistical_acceptance_computation(
00245             // statistical_acceptance_computation &&other) = default;
00246         // statistical_acceptance_computation(
00247             // const statistical_acceptance_computation &other) = delete;
00248
00249         // ~statistical_acceptance_computation() {}
00250
00251
00253         void skip_computation(prng_t &main_prng) {
00254             derive_prng(main_prng);
00255         }
00256
00257
00259         template<typename scheduler_t,
00260             typename result_projection_t = std::identity,
00261             typename result_function_t = std::function<void(prec_t)>>
00262         void start_computation(
00263             prng_t &main_prng,
00264             const cfg_vector_t &v,
00265             scheduler_t &scheduler,
00266             value_comp_t &value_comp,
00267             result_projection_t &result_projection = {})
00268         {
00269             assert(!computation_scheduled_state);
00270             computation_scheduled_state = true;
00271
00272             ignore(&scheduler);
00273
00274
00275             // std::cout << "main_prng: " << main_prng << std::endl;
00276             state_data[new_index].prng = derive_prng(main_prng);
00277             // std::cout << "derived prng: "
00278                 // << state_data[new_index].prng() << ", "
00279                 // << state_data[new_index].prng() << ", "
00280                 // << state_data[new_index].prng() << ", "
00281                 // << state_data[new_index].prng << std::endl;
00282
00283             prob_distribution.reset();
00284             z = nextz;
00285             nextz = prob_distribution(state_data[new_index].prng);
00286             // std::cout << "start_computation: z: "
```

```
00287                 // « z « ", v: " « v « std::endl;
00288
00289             value_comp.prepare_computation_with_cfg_vector(
00290                 v, state_data[new_index].pcom);
00291
00292             state_data[new_index].iteration = 0;
00293             state_data[new_index].statistics =
00294                 basic_statistical_metrics<prec_t>();
00295
00296             // reset statistics but not already computed values
00297             uint8_t old_index = 1 - new_index;
00298             state_data[old_index].statistics =
00299                 basic_statistical_metrics<prec_t>();
00300             accept = false;
00301
00302             schedule_computation =
00303                 [&](void *scheduler_ptr, std::size_t i, uint8_t index)
00304             {
00305                 scheduler_t &scheduler =
00306                     *(static_cast<scheduler_t *>(scheduler_ptr));
00307
00308                 result_function_t result_fct
00309                     = [&, i, index](auto... results)
00310                 {
00311                     remaining_computations--;
00312                     auto result = result_projection(results...);
00313                     static_assert(
00314                         std::is_same<prec_t, decltype(result)>::value
00315                         && "result_projection has wrong result type!");
00316                     // std::cout « "c R state_data[" « (int) index
00317                         // « "].values[" « i « "] = "
00318                         // « result « std::endl;
00319                     state_data[index].values[i] = result;
00320                 };
00321
00322                 auto cmd = value_comp(
00323                     state_data[index].prng,
00324                     scheduler.rec.rpmanager,
00325                     state_data[index].pcom);
00326                 cmd = scheduler.rec.on_result(cmd, result_fct);
00327
00328                 remaining_computations++;
00329                 scheduler.schedule(cmd);
00330             };
00331         }
00332
00333
00335         bool still_waiting_for_computation() {
00336             return (0 < remaining_computations);
00337         }
00338
00339
00341         template<typename scheduler_t>
00342         bool continue_computation(
00343             scheduler_t &scheduler,
00344             value_comp_t &value_comp,
00345             acc_prob_comp_t &acc_prob_fct)
00346         {
00347             assert(computation_scheduled_state);
00348             ignore(&value_comp);
00349
00350             if (is_first_cfg_vector) {
00351                 return true; // nothing to do
00352             }
00353
00354             if (0 < remaining_computations) {
00355                 return false; // not yet done
00356             }
00357
00358             uint8_t old_index = 1 - new_index;
00359             uint8_t indices[2] = {new_index, old_index};
00360
00361             // update statistics
00362             std::size_t num_values =
00363                 state_data[new_index].iteration * values_per_iteration;
00364             if (0 < num_values) {
00365                 for (auto j = 0; j < 2; j++) {
00366                     uint8_t index = indices[j];
00367
00368                     auto it_start =
00369                         state_data[index].values.begin()
00370                         + state_data[index].statistics.counted();
00371                     auto it_end =
00372                         state_data[index].values.begin() + num_values;
00373                     state_data[index].statistics(it_start, it_end);
00374
00375                     #if DEBUG_ASSERTIONS
```

```
00376                     assert(num_values
00377                         == state_data[index].statistics.counted());
00378                     #endif
00379                 }
00380
00381                 // test for evidence of result
00382                 std::size_t num_values_available =
00383                     std::min(state_data[new_index].statistics.n,
00384                     state_data[old_index].statistics.n);
00385
00386                 if (num_values_available >= values_required) {
00387                     auto [p_avg, p_min, p_max] = compute_p_range(
00388                         coincidence_interval_factor,
00389                         acc_prob_fct,
00390                         state_data[new_index].statistics,
00391                         state_data[old_index].statistics);
00392
00393                     // std::cout « "\t" « "p: [" « p_min « ", "
00394                         // « p_avg « ", " « p_max « "]" « std::endl;
00395
00396                     // test for statistical evidence of accept/decline
00397                     if (z <= p_min) {
00398                         // std::cout « "clear accept ("
00399                             // « z « ", " « p_min « ")" « std::endl;
00400                         accept = true;
00401                         return true;
00402                     }
00403
00404                     if (z > p_max) {
00405                         // std::cout « "clear decline ("
00406                             // « z « ", " « p_max « ")" « std::endl;
00407                         accept = false;
00408                         return true;
00409                     }
00410
00411                     const bool max_iterations_reached =
00412                         max_iterations <= state_data[new_index].iteration;
00413                     if (max_iterations_reached) {
00414                         // std::cout « "max iterations reached ("
00415                             // « z « ", " « p_avg « ")" « std::endl;
00416                         accept = (z <= p_avg); // false
00417                         return true; // done with best guess
00418                     }
00419
00420                     const bool testniveau_unterschritten =
00421                         std::abs(p_max - p_min) < testniveau;
00422                     if (testniveau_unterschritten) {
00423                         // std::cout « "testniveau_unterschritten ("
00424                             // « z « ", " « p_avg « ")" « std::endl;
00425                         accept = (z <= p_avg); // false
00426                         return true; // done with best guess
00427                     }
00428                 }
00429             }
00430
00431             // not yet done => schedule next iteration(s) for more data
00432             const bool new_values_for_old_index_are_required =
00433                 state_data[new_index].iteration
00434                 >= state_data[old_index].iteration;
00435
00436             assert(state_data[new_index].iteration
00437                 <= state_data[old_index].iteration);
00438
00439             for (auto j = 0;
00440                 j < (new_values_for_old_index_are_required ? 2 : 1); j++)
00441             {
00442                 uint8_t index = indices[j];
00443                 state_data[index].iteration++;
00444
00445                 std::size_t num_values =
00446                     state_data[index].iteration*values_per_iteration;
00447                 if (num_values > state_data[index].values.size()) {
00448                     state_data[index].values.resize(num_values);
00449                 }
00450
00451                 for (std::size_t i = num_values - values_per_iteration;
00452                     i < num_values; i++)
00453                 {
00454                     schedule_computation(
00455                         static_cast<void *>(&scheduler), i, index);
00456                 }
00457             }
00458
00459             return false;
00460         }
00461
00462
```

```
00468         bool finish_computation(
00469             value_comp_t &value_comp, acc_prob_comp_t &acc_prob_fct)
00470         {
00471             assert(0 == remaining_computations);
00472             assert(computation_scheduled_state);
00473             computation_scheduled_state = false;
00474 
00475             ignore(&acc_prob_fct);
00476 
00477             uint8_t old_index = 1 - new_index;
00478             if (is_first_cfg_vector) {
00479                 is_first_cfg_vector = false;
00480                 new_index = old_index;
00481                 return true;
00482             } else {
00483                 if (accept) {
00484                     // std::cout « "accept "
00485                         // « state_data[new_index].statistics.average
00486                         // « std::endl;
00487                     value_comp.finish_computation(
00488                         state_data[old_index].pcom);
00489                     new_index = old_index;
00490                 } else {
00491                     // std::cout « "decline "
00492                         // « state_data[new_index].statistics.average
00493                         // « std::endl;
00494                     // std::cout « "keep value "
00495                         // « state_data[old_index].statistics.average
00496                         // « std::endl;
00497                     value_comp.finish_computation(
00498                         state_data[new_index].pcom);
00499                 }
00500                 return accept;
00501             }
00502         }
00503 
00504 
00506         void cleanup(value_comp_t &value_comp) {
00507             if (!is_first_cfg_vector) {
00508                 uint8_t old_index = 1 - new_index;
00509                 value_comp.finish_computation(state_data[old_index].pcom);
00510             }
00511         }
00512 
00513 
00515         prec_t get_current_value() {
00516             assert(!computation_scheduled_state);
00517             uint8_t index = 1 - new_index;
00518             return (0 == state_data[index].statistics.n) ?
00519                 NAN : state_data[index].statistics.average;
00520         }
00521 
00523         prec_t get_last_computed_value() {
00524             assert(!computation_scheduled_state);
00525             uint8_t index = accept ? 1 - new_index : new_index;
00526             return (0 == state_data[index].statistics.n) ?
00527                 NAN : state_data[index].statistics.average;
00528         }
00529 
00531         prec_t get_previous_computed_value() {
00532             assert(!computation_scheduled_state);
00533             uint8_t index = new_index;
00534             return (0 == state_data[index].statistics.n) ?
00535                 NAN : state_data[index].statistics.average;
00536         }
00537 
00538 /*
00539         prec_t get_value() {
00540             assert(!computation_scheduled_state);
00541             uint8_t index = accept ? 1 - new_index : new_index;
00542             return (0 == state_data[index].statistics.n) ?
00543                 NAN : state_data[index].statistics.average;
00544         }
00545 */
00546 };
00547 
00548 
00549 #endif
```

7.17 algorithm.hpp

```
00001 #ifndef ALGORITHM_HPP
00002 #define ALGORITHM_HPP
```

```
00003
00004
00005 #include <set>
00006 #include <iostream>
00007 #include <cassert>
00008
00009 #include "../config.hpp"
00010
00011
00013 template<
00014     typename marcov_chain_state_t,
00015     typename change_generator_t,
00016     typename acceptance_computation_t,
00017     typename computation_modell_t,
00018     typename acc_prob_comp_t,
00019     typename result_statistics_t,
00020     typename result_function_t,
00021     const bool enforce_change,
00022     const bool skip_unchanged_vectors
00023     >
00024 class metropolis_hastings_algorithm
00025 {
00026     private:
00027         using cfg_vector_t = marcov_chain_state_t::cfg_vector_type;
00028
00030         std::size_t id;
00031
00033         marcov_chain_state_t mc_state;
00034
00036         decltype(mc_state.prn_engine) last_prng;
00037
00039         change_generator_t change_gen;
00040
00042         cfg_vector_t next_cfg_vector;
00043
00045         acceptance_computation_t acc_computation;
00046
00048         bool is_processing;
00049
00051         computation_modell_t &computation_modell;
00052
00054         acc_prob_comp_t &acc_prob_fct;
00055
00057         result_statistics_t execution_statistics;
00058
00059
00061         void generate_next_cfg_vector()
00062         {
00063             next_cfg_vector = mc_state.cfg_vector;
00064
00065             bool change_happened = enforce_change;
00066             do {
00067                 using index_t = decltype(
00068                     change_gen(mc_state.prn_engine,
00069                     next_cfg_vector).index);
00070
00071                 // generate changes
00072                 std::set<index_t> changed_positions;
00073
00074                 for (decltype(mcmc_num_changes) i = 0;
00075                     i < mcmc_num_changes; i++)
00076                 {
00077                     auto cfg_change =
00078                         change_gen(mc_state.prn_engine, next_cfg_vector);
00079                     next_cfg_vector.data[cfg_change.index]
00080                         = cfg_change.new_value;
00081
00082                     if ((!change_happened) && (skip_unchanged_vectors))
00083                     {
00084                         changed_positions.insert(cfg_change.index);
00085                     }
00086                 }
00087
00088                 // check if config vector is changed
00089                 if ((!change_happened) && (skip_unchanged_vectors)) {
00090                     for(auto p : changed_positions) {
00091                         if (next_cfg_vector.data[p]
00092                             != mc_state.cfg_vector.data[p])
00093                         {
00094                             change_happened = true;
00095                             break;
00096                         }
00097                     }
00098                 }
00099             } while ((!change_happened) && (skip_unchanged_vectors));
00100         }
00101
```

```
00102     public:
00103
00105         metropolis_hastings_algorithm(
00106             std::size_t id,
00107             auto mc_state_,
00108             computation_modell_t &computation_modell,
00109             acc_prob_comp_t &acc_prob_fct
00110         ) :
00111             id(id),
00112             mc_state(mc_state_),
00113             last_prng(mc_state.prn_engine),
00114             change_gen(mc_state.cfg_vector),
00115             next_cfg_vector(mc_state.cfg_vector),
00116             acc_computation(/*mc_state.prn_engine*/),
00117             is_processing(false),
00118             computation_modell(computation_modell),
00119             acc_prob_fct(acc_prob_fct),
00120             execution_statistics()
00121         {}
00122
00123         // metropolis_hastings_algorithm(
00124             // metropolis_hastings_algorithm &&other) = default;
00125         // metropolis_hastings_algorithm(
00126             // const metropolis_hastings_algorithm &other) = delete;
00127
00128         // ~metropolis_hastings_algorithm() {}
00129
00130
00132         marcov_chain_state_t get_last_state() {
00133             assert(!is_processing);
00134
00135             return marcov_chain_state_t(
00136                 mc_state.iteration - 1, mc_state.cfg_vector, last_prng);
00137         }
00138
00139
00141         template<typename scheduler_t>
00142         void prepare_iteration(scheduler_t &scheduler) {
00143             // std::cout « "prepare" « std::endl;
00144             assert(!is_processing);
00145             is_processing = true;
00146
00147             last_prng = mc_state.prn_engine;
00148             // std::cout « "test prepare_iteration "
00149                 // « mc_state.iteration « ": " « last_prng « std::endl;
00150
00151
00152             acc_computation.template start_computation
00153                 <scheduler_t, result_statistics_t, result_function_t>
00154                 (mc_state.prn_engine,
00155                 next_cfg_vector,
00156                 scheduler,
00157                 computation_modell,
00158                 execution_statistics);
00159         }
00160
00161
00163         bool still_waiting_for_computation() {
00164             return acc_computation.still_waiting_for_computation();
00165         }
00166
00167
00169         template<typename scheduler_t>
00170         bool continue_iteration(scheduler_t &scheduler) {
00171             // std::cout « "continue" « std::endl;
00172             assert(is_processing);
00173             bool is_finished = acc_computation.continue_computation(
00174                 scheduler, computation_modell, acc_prob_fct);
00175             // scheduler.do_work();
00176             return is_finished;
00177         }
00178
00179
00181         void finish_iteration(bool last_iteration = false) {
00182             // std::cout « "finish" « std::endl;
00183             assert(is_processing);
00184             is_processing = false;
00185
00186             bool accept = acc_computation.finish_computation(
00187                 computation_modell, acc_prob_fct);
00188             if (accept) {
00189                 mc_state.cfg_vector = next_cfg_vector;
00190             }
00191
00192             if (1 == mc_state.iteration) {
00193                 auto initial_value = accept ?
00194                     acc_computation.get_previous_computed_value() :
```

```
00195                     acc_computation.get_current_value();
00196                 std::cout « "mc " « id
00197                     « " iteration " « mc_state.iteration
00198                     « " value " « initial_value
00199                     « " accepted " « initial_value « std::endl;
00200             }
00201
00202             if (0 < mc_state.iteration) {
00203                 std::cout « "mc " « id
00204                     « " iteration " « (mc_state.iteration + 1)
00205                     « " value " « acc_computation.get_current_value()
00206                     « (accept ? " accepted " : " declined ")
00207                     « acc_computation.get_last_computed_value()
00208                     « std::endl;
00209             }
00210
00211             #if DEBUG_ASSERTIONS
00212             if (accept) {
00213                 assert(mc_state.cfg_vector == next_cfg_vector);
00214             } else {
00215                 assert((mc_state.cfg_vector != next_cfg_vector)
00216                     || !(skip_unchanged_vectors || enforce_change));
00217             }
00218             #endif
00219
00220             if (last_iteration) {
00221                 is_processing = true; // prevent further execution
00222                 cleanup();
00223             } else {
00224                 generate_next_cfg_vector();
00225             }
00226
00227             mc_state.iteration++;
00228         }
00229
00230
00232         void skip_computation(bool accept) {
00233             acc_computation.skip_computation(mc_state.prn_engine);
00234             if (accept) {
00235                 mc_state.cfg_vector = next_cfg_vector;
00236             }
00237             generate_next_cfg_vector();
00238             mc_state.iteration++;
00239         }
00240
00241
00243         void cleanup() {
00244             acc_computation.cleanup(computation_modell);
00245         }
00246
00247
00249         bool can_start_next_iteration() const {
00250             return !is_processing;
00251         }
00252
00253
00255         std::size_t get_iteration() const {
00256             return mc_state.iteration;
00257         }
00258
00259
00261         void print_execution_statistic() {
00262             std::cout « "mc " « id « " execution_statistics: "
00263                 « execution_statistics « std::endl;
00264         }
00265
00266
00268         template<typename scheduler_t>
00269         std::size_t operator()(
00270             scheduler_t &scheduler, bool last_iteration = false)
00271         {
00272             if (false == is_processing) {
00273                 prepare_iteration(scheduler);
00274             } else {
00275                 bool is_finished = continue_iteration(scheduler);
00276                 if (is_finished) {
00277                     finish_iteration(last_iteration);
00278                 }
00279             }
00280
00281             return mc_state.iteration;
00282         }
00283 };
00284
00285 #endif
```

7.18 change_generator.hpp

```
00001 #ifndef CHANGE_GENERATOR_HPP
00002 #define CHANGE_GENERATOR_HPP
00003
00004
00005 #include <random>
00006 #include <cassert>
00007 #include <sstream>
00008
00009 #include "../../util/basic_serialization.hpp"
00010
00011
00013 template<typename size_type, typename value_t>
00014 struct change {
00016     size_type index;
00017
00019     value_t new_value;
00020
00022     auto operator==(const change<size_type, value_t> &rhs) const {
00023         return (index == rhs.index) && (new_value == rhs.new_value);
00024     }
00025
00027     auto operator!=(const change<size_type, value_t> &rhs) const {
00028         return (*this == rhs);
00029     }
00030 };
00031
00032
00034 template <
00035     typename cfg_vector_t,
00036     bool enforce_change,
00037
00038     typename value_t = std::conditional<
00039         std::is_same<bool, typename cfg_vector_t::value_type>::value,
00040         short,
00041         typename cfg_vector_t::value_type
00042     >::type,
00043
00044     typename value_distr_type = std::uniform_int_distribution<value_t>,
00045
00046     decltype(value_distr_type().min()) value_distr_minv
00047         = cfg_vector_t::min_value,
00048
00049     decltype(value_distr_type().max()) value_distr_maxv
00050         = enforce_change ?
00051             cfg_vector_t::max_value - 1 :
00052             cfg_vector_t::max_value
00053 >
00054 class simple_change_generator
00055 {
00056     public:
00057         constexpr static bool is_change_generator = true;
00058         using index_t = cfg_vector_t::size_type;
00059
00060     private:
00062         std::uniform_int_distribution<index_t> index_distr;
00063
00065         value_distr_type value_distr;
00066
00067     public:
00068         std::string get_description() {
00069             return enforce_change ? "scg-ec" : "scg-ac";
00070         }
00071
00073         simple_change_generator(const cfg_vector_t &v) :
00074             index_distr(0, v.data.size() - 1),
00075             value_distr(value_distr_minv, value_distr_maxv)
00076         {
00077             assert(0 < v.data.size());
00078             static_assert(
00079                 cfg_vector_t::min_value < cfg_vector_t::max_value);
00080         }
00081
00083         auto operator()(auto &prng, const cfg_vector_t &v)
00084         {
00085             index_t index = index_distr(prng);
00086             value_t value = value_distr(prng);
00087
00088             if constexpr(enforce_change) {
00089                 if (value >= v.data[index]) {
00090                     value++;
00091                 }
00092             }
00093
00094             return change<index_t, typename cfg_vector_t::value_type>{
00095                 index,
```

```
00096                 static_cast<cfg_vector_t::value_type>(value)
00097             };
00098         }
00099 };
00100
00101
00103 template <typename S, typename simple_change_generator_t>
00104     requires (simple_change_generator_t::is_change_generator)
00105 void serialize(S& s, simple_change_generator_t &change_gen) {
00106     std::string descr = change_gen.get_description();
00107     s.text1b(descr, 7);
00108     assert(6 == descr.length());
00109     assert(change_gen.get_description() == descr);
00110
00111     std::stringstream sstr;
00112     sstr « change_gen.index_distr;
00113     sstr « change_gen.value_distr;
00114     std::string distributions = sstr.str();
00115     assert(!sstr.fail());
00116
00117     generic_serialize(s, distributions);
00118
00119     sstr.str(distributions);
00120     sstr » change_gen.index_distr;
00121     sstr » change_gen.value_distr;
00122     assert(!sstr.fail());
00123     assert(sstr.eof());
00124 }
00125
00126
00127 #endif
```

7.19 configuration_generator.hpp

```
00001 #ifndef CONFIGURATION_MANAGER_HPP
00002 #define CONFIGURATION_MANAGER_HPP
00003
00004
00005 #include <string>
00006 #include <random>
00007 #include <cassert>
00008
00009
00011 template <typename cfg_vector_t>
00012 class constant_configuration_generator
00013 {
00014     public:
00015         constexpr static bool is_configuration_generator = true;
00016
00017     private:
00018         using value_t = cfg_vector_t::value_type;
00019
00021         value_t cvalue;
00022
00023     public:
00024         std::string get_description() {
00025             return std::string("ci-") + std::to_string(cvalue);
00026         }
00027
00029         constant_configuration_generator(value_t constant_value)
00030             : cvalue(constant_value)
00031         {
00032             assert(cfg_vector_t::min_value <= constant_value);
00033             assert(cfg_vector_t::max_value >= constant_value);
00034         }
00035
00037         void operator()(auto &prng, cfg_vector_t &v) const {
00038             ignore(prng);
00039
00040             for (typename cfg_vector_t::size_type i = 0;
00041                 i < v.data.size(); i++)
00042             {
00043                 v.data[i] = cvalue;
00044             }
00045         }
00046 };
00047
00048
00050 template <typename cfg_vector_t>
00051 class uniform_configuration_generator
00052 {
00053     private:
00054         using value_t = std::conditional<
```

```
00055             std::is_same<bool, typename cfg_vector_t::value_type>::value,
00056             short,
00057             typename cfg_vector_t::value_type>::type;
00058
00059     public:
00060         constexpr static bool is_configuration_generator = true;
00061
00062         std::string get_description() {
00063             return std::string("ui");
00064         }
00065
00067         uniform_configuration_generator() {}
00068
00070         template <typename rng_t>
00071         void operator()(rng_t &prng, cfg_vector_t &v) const {
00072             std::uniform_int_distribution<value_t>
00073                 distr(cfg_vector_t::min_value, cfg_vector_t::max_value);
00074             for (typename cfg_vector_t::size_type i = 0;
00075                 i < v.data.size(); i++)
00076             {
00077                 value_t value = distr(prng);
00078                 v.data[i] = static_cast<cfg_vector_t::value_type>(value);
00079             }
00080         }
00081 };
00082
00083
00084 #endif
```

7.20 configuration_vector.hpp

```
00001 #ifndef CONFIGURATION_VECTOR_HPP
00002 #define CONFIGURATION_VECTOR_HPP
00003
00004
00005 #include <vector>
00006 #include <type_traits>
00007
00008 #include "../../util/bitfield.hpp"
00009 #include "../../util/basic_serialization.hpp"
00010
00011
00013 template<bool use_bitfield = false>
00014 struct binary_configuration_vector
00015 {
00016     public:
00018         using value_type = bool;
00019
00021         static constexpr value_type min_value = false;
00022
00024         static constexpr value_type max_value = true;
00025
00027         using vector_type =
00028             std::conditional< use_bitfield,
00029                 bitfield<std::size_t>,
00030                 std::vector<value_type>
00031             >::type;
00032         // using vector_type = std::vector<value_type>;
00033
00035         using size_type = decltype(vector_type().size());
00036
00038         vector_type data;
00039
00040
00041
00043         binary_configuration_vector() : data(0) {}
00044
00046         binary_configuration_vector(size_type length) : data(length) {}
00047
00048
00050         binary_configuration_vector(
00051             const binary_configuration_vector &other)
00052         {
00053             data = other.data;
00054         }
00055
00056
00058         friend void swap(
00059             binary_configuration_vector<use_bitfield> &l,
00060             binary_configuration_vector<use_bitfield> &r)
00061         {
00062             std::swap(l.data, r.data);
00063         }
```

```
00064
00065
00067         binary_configuration_vector &operator=(
00068             const binary_configuration_vector<use_bitfield> &other)
00069         {
00070             data = other.data;
00071             return *this;
00072         }
00073
00074         /*
00076         binary_configuration_vector &operator=(
00077             const binary_configuration_vector<use_bitfield> &other)
00078         {
00079             swap(*this, other);
00080             return *this;
00081         }
00082         */
00083
00084
00086         binary_configuration_vector(binary_configuration_vector &&other)
00087         {
00088             swap(*this, other);
00089         }
00090
00091
00093         friend std::ostream &operator«(
00094             std::ostream &os,
00095             const binary_configuration_vector<use_bitfield> &bf)
00096         {
00097             os « "binary_configuration_vector:";
00098             for (size_type i = 0; i < bf.data.size(); i++) {
00099                 os « " " « (bf.data[i] ? "1" : "0");
00100             }
00101             return os;
00102         }
00103
00104
00106         std::size_t count_ones() const
00107         {
00108             std::size_t ones = 0;
00109
00110             for (size_type i = 0; i < data.size(); i++) {
00111                 if (data[i]) { ones++; }
00112             }
00113
00114             return ones;
00115         }
00116
00118         bool operator==(
00119             const binary_configuration_vector<use_bitfield> &rhs) const
00120         {
00121             if (data.size() == rhs.data.size()) {
00122                 for (size_type i = 0; i < data.size(); i++) {
00123                     if (data[i] != rhs.data[i]) { return false; }
00124                 }
00125                 return true;
00126             }
00127             return false;
00128         }
00129
00131         bool operator!=(
00132             const binary_configuration_vector<use_bitfield> &rhs) const
00133         {
00134             return !(*this == rhs);
00135         }
00136 };
00137
00138
00140 template<typename S, bool use_bitfield>
00141 void serialize(
00142     S &s, binary_configuration_vector<use_bitfield> &cfg_vector)
00143 {
00144     serialize_marker(s, "bcv", 4);
00145
00146     bool ub = use_bitfield;
00147     s.value1b(ub);
00148     assert(use_bitfield == ub);
00149
00150     serialize(s, cfg_vector.data);
00151 }
00152
00153
00154 #endif
```

7.21 mcmc.hpp

```
00001 #ifndef MCMC_HPP
00002 #define MCMC_HPP
00003
00004
00005 #include <cstddef>
00006 #include <random>
00007 #include <sstream>
00008
00009 #include "../config.hpp"
00010 #include "../util/basic_serialization.hpp"
00011
00012
00014 template<typename cfg_vector_t>
00015 class marcov_chain_state
00016 {
00017     public:
00019         using cfg_vector_type = cfg_vector_t;
00020
00022         std::size_t iteration;
00023
00025         cfg_vector_t cfg_vector;
00026
00028         prn_engine_t prn_engine;
00029
00030
00032         marcov_chain_state(std::size_t cfg_vector_length = 0) :
00033             iteration(std::numeric_limits<std::size_t>::max()),
00034             cfg_vector(cfg_vector_length),
00035             prn_engine() {}
00036
00038         marcov_chain_state(
00039             std::size_t cfg_vector_length,
00040             auto prn_engine
00041         ) :
00042             iteration(0),
00043             cfg_vector(cfg_vector_length),
00044             prn_engine(prn_engine)
00045         {}
00046
00048         marcov_chain_state(
00049             std::size_t iteration,
00050             cfg_vector_t &cfg_vector,
00051             prn_engine_t &prn_engine
00052         ) :
00053             iteration(iteration),
00054             cfg_vector(cfg_vector),
00055             prn_engine(prn_engine) {}
00056
00058         marcov_chain_state(const marcov_chain_state &other) :
00059             iteration(other.iteration),
00060             cfg_vector(other.cfg_vector),
00061             prn_engine(other.prn_engine) {}
00062
00064         marcov_chain_state(marcov_chain_state &&other) {
00065             swap(*this, other);
00066         }
00067
00069         marcov_chain_state &operator=(
00070             const marcov_chain_state<cfg_vector_t> &other)
00071         {
00072             iteration = other.iteration;
00073             cfg_vector = other.cfg_vector;
00074             prn_engine = other.prn_engine;
00075             return *this;
00076         }
00077
00079         friend void swap(
00080             marcov_chain_state<cfg_vector_t> &l,
00081             marcov_chain_state<cfg_vector_t> &r)
00082         {
00083             std::swap(l.iteration, r.iteration);
00084             std::swap(l.cfg_vector, r.cfg_vector);
00085             std::swap(l.prn_engine, r.prn_engine);
00086         }
00087
00089         bool operator==(
00090             const marcov_chain_state<cfg_vector_t> &rhs) const
00091         {
00092             return (iteration == rhs.iteration)
00093                 && (cfg_vector == rhs.cfg_vector)
00094                 && (prn_engine == rhs.prn_engine);
00095         }
00096
00098         bool operator!=(
00099             const marcov_chain_state<cfg_vector_t> &rhs) const
```

```
00100         {
00101             return !(*this == rhs);
00102         }
00103 
00105         friend std::ostream &operator«(
00106             std::ostream &os,
00107             const marcov_chain_state<cfg_vector_t> &mcs)
00108         {
00109             os  « "state of iteration " « mcs.iteration « ":\n"
00110                 « mcs.cfg_vector « "\n" « mcs.prn_engine « std::endl;
00111             return os;
00112         }
00113 
00115         std::string get_prn_engine_as_string() const {
00116             std::stringstream isstr;
00117             isstr « prn_engine;
00118             assert(!isstr.fail());
00119             return isstr.str();
00120         }
00121 
00123         void set_prn_engine_from_string(auto prn_state) {
00124             std::stringstream osstr(prn_state);
00125             osstr » prn_engine;
00126             assert(!osstr.fail());
00127             assert(osstr.eof());
00128         }
00129 };
00130 
00131 
00133 template<typename S, typename cfg_vector_t>
00134 void serialize(S &s, marcov_chain_state<cfg_vector_t> &mc)
00135 {
00136     std::string prn_state = mc.get_prn_engine_as_string();
00137 
00138     // serialize_marker(s, "ms1", 4);
00139     serialize_basic_type<S, decltype(mc.iteration)>(s, mc.iteration);
00140 
00141     // serialize_marker(s, "ms2", 4);
00142     serialize(s, mc.cfg_vector);
00143 
00144     // serialize_marker(s, "ms3", 4);
00145     generic_serialize(s, prn_state);
00146     mc.set_prn_engine_from_string(prn_state);
00147 
00148     // serialize_marker(s, "ms4", 4);
00149 }
00150 
00151 /*
00152 template<typename S, typename cfg_vector_t>
00153 void serialize(S &s, marcov_chain_state<cfg_vector_t> &mc)
00154 {
00155     std::cout « "start" « std::endl;
00156     std::string prn_state;
00157 
00158     std::stringstream isstr;
00159     isstr « mc.prn_engine;
00160     prn_state = isstr.str();
00161 
00162     // generic_serialize(s, mc.iteration, mc.cfg_vector, prn_state);
00163     // generic_serialize(s, mc.iteration);
00164     s.value8b(mc.iteration);
00165 
00166     // if (mc.cfg_vector.data.length()
00167     // cfg_vector(cfg_vector_length)
00168 
00169     // std::cout « "mc.cfg_vector size a: " « mc.cfg_vector.data.size() « std::endl;
00170 
00171     serialize(s, mc.cfg_vector);
00172 
00173     // std::cout « "mc.cfg_vector size b: " « mc.cfg_vector.data.size() « std::endl;
00174 
00175     std::string marker1 = "m42";
00176     s.text1b(marker1, 4);
00177     assert(std::string("m42") == marker1);
00178 
00179     std::size_t strlength = prn_state.size();
00180     generic_serialize(s, strlength);
00181     prn_state.reserve(strlength);
00182     s.text1b(prn_state, strlength);
00183     assert((prn_state.size() == strlength) || print_trace());
00184 
00185     std::string marker2 = "fm!";
00186     s.text1b(marker2, 4);
00187     assert(std::string("fm!") == marker2);
00188 
00189     std::stringstream osstr(prn_state);
00190     osstr » mc.prn_engine;
```

```
00191
00192     std::cout « "ok" « std::endl;
00193 }
00194 */
00195
00196 /*
00197 // output statistic
00198 class mcmc_statistic
00199 {
00200     private:
00201         std::size_t mc_id;
00202
00203     public:
00204         mcmc_statistic(std::size_t mc_id) : mc_id(mc_id) {}
00205 };
00206
00207 // able to fast reload application state
00208 class mcmc_history
00209 {
00210     private:
00211         std::size_t mc_id;
00212         std::fstream hfile;
00213         uint8_t no_counter = 0;
00214
00215 Problem: do changes need consistent prn generator?
00216
00217     full save:
00218         store "F"
00219         size of marcov_chain_state
00220         marcov_chain_state
00221             iteration
00222             cfg_vector
00223             prn_engine
00224
00225         go to the start of the file and save last position of "F"
00226
00227     incremental save:
00228         accepted?
00229             if yes:
00230                 store "n" save no counter
00231                 store "y"
00232                 save changes in cfg vector (size + changes)
00233             if no:
00234                 increment no counter
00235                 prevent overflow (i.e. store "n" and save no counter)
00236
00237     restore:
00238         load last full save
00239         restore current iteration cfg_vector and prn_engine state by fast forwarding
00240         continue as from beginning
00241
00242
00243         void save_no() {
00244             hfile.put("n");
00245             hfile.put(no_counter);
00246             no_counter = 0;
00247         }
00248
00249         void save_full(marcov_chain_state &mc_state)
00250         {
00251             const std::size_t offset = hfile.tellp();
00252
00253             // write "F", size, mc_state
00254             hfile.put("F");
00255
00256             type_abstraction_adapter_t bitsery_taa;
00257             buffer_t data = bitsery_taa.serialize(mc_state);
00258
00259             make_generic_serializeable<std::size_t> gssize = data.size();
00260             buffer_t size = bitsery_taa.serialize(gssize);
00261             assert(size.size() == sizeof(std::size_t)); // otherwise load won't work!
00262
00263             assert(!hfile.write(size.data(), size.size()).fail());
00264             assert(!hfile.write(data.data(), data.size()).fail());
00265             hfile.flush();
00266             assert(!hfile.fail());
00267
00268             // write offset at start of file
00269             const std::size_t end = hfile.tellp();
00270             hfile.seekp(0, hfile.beg);
00271             make_generic_serializeable<std::size_t> gsoffset = offset;
00272             buffer_t offs = bitsery_taa.serialize(gsoffset);
00273             assert(offs.size() == sizeof(std::size_t));
00274             assert(!hfile.write(offs.data(), offs.size()).fail());
00275             hfile.seekp(end, hfile.beg);
00276
00277             hfile.flush();
```

```
00278             assert(!hfile.fail());
00279         }
00280
00281         void load_full(marcov_chain_state &mc_state)
00282         {
00283             char c = 0;
00284             assert(hfile.get(c) && c == 'F');
00285             type_abstraction_adapter_t bitsery_taa;
00286
00287             // read size
00288             make_generic_serializeable<std::size_t> gssize = 0;
00289             std::vector inbuf(sizeof(std::size_t));
00290             assert(!hfile.read(inbuf.data(), inbuf.size()).fail());
00291             assert(bitsery_taa.deserialize(inbuf.data(), gssize));
00292
00293             // read mc_state
00294             inbuf.resize(gssize.data);
00295             assert(!hfile.read(inbuf.data(), inbuf.size()).fail());
00296             assert(bitsery_taa.deserialize(inbuf.data(), mc_state));
00297         }
00298
00299     public:
00300         void add_changes(bool accepted, auto &changes) {
00301             if (accepted) {
00302                 if (0 < no_counter) { save_no(); }
00303                 hfile.put("y");
00304                 // type_abstraction_adapter_t bitsery_taa;
00305                 // buffer_t data = bitsery_taa.serialize(changes);
00306                 // assert(!hfile.write(data.data(), data.size()).fail());
00307                 changes.save_to(hfile);
00308                 hfile.flush();
00309             } else {
00310                 no_counter++;
00311                 if (std::numeric_limits<decltype(no_counter)>::max() == no_counter) {
00312                     save_no();
00313                     hfile.flush();
00314                 }
00315             }
00316
00317             assert(!hfile.fail());
00318         }
00319
00320         mcmc_history(std::size_t mc_id, marcov_chain_state &mc_state) :
00321             mc_id(mc_id)
00322             hfile(data_directory + "/" + std::to_string(mc_id) + ".mch",
00323                 std::ios::in | std::ios::out | ios::binary | std::ios::app),
00324             no_counter(0)
00325         {
00326             assert(hfile.is_open());
00327
00328             hfile.seekp(0, hfile.end);
00329             const auto fsize = hfile.tellp();
00330
00331             if (0 == fsize) {
00332                 // write offset and first full save
00333                 std::size_t s = 0;
00334                 assert(!hfile.write(&s, sizeof(std::size_t)).fail());
00335                 save_full(mc_state);
00336             } else {
00337                 // read offset of last full save
00338                 hfile.seekg(0, hfile.beg);
00339                 type_abstraction_adapter_t bitsery_taa;
00340                 make_generic_serializeable<std::size_t> gsoffset = 0;
00341                 std::vector inbuf(sizeof(std::size_t));
00342                 assert(hfile.read(inbuf.data(), inbuf.size()).good());
00343                 assert(bitsery_taa.deserialize(inbuf.data(), gsoffset));
00344
00345                 // read first save
00346                 hfile.seekg(sizeof(std::size_t), hfile.beg);
00347                 marcov_chain_state_t mc_state_(mc_state.cfg_vector.data.size(), prn_engine_t());
00348                 load_full(mc_state_);
00349                 assert(mc_state_ == mc_state && "initial state must match!");
00350
00351                 // read last full save
00352                 hfile.seekg(gsoffset.data, seed_file.beg);
00353                 load_full(mc_state);
00354
00355                 // fast forward
00356                 while () {
00357
00358                 }
00359             }
00360
00361             assert(!hfile.fail());
00362         }
00363 };
00364 */
```

```
00365
00366 #endif
```

7.22 modell.hpp

```
00001 #ifndef MODELL_HPP
00002 #define MODELL_HPP
00003
00004 #include <random>
00005 #include <cassert>
00006
00007
00009 template<typename prec_t>
00010 prec_t modell_identity(prec_t p) { return p; }
00011
00012
00014 template<typename prec_t>
00015 prec_t modell_multiply(prec_t a, prec_t b) { return a*b; }
00016
00017
00019 template<typename cfg_vector_t, typename prec_t>
00020 static prec_t bitvector_to_num(const cfg_vector_t &v)
00021 {
00022     prec_t r = 0.0;
00023     using size_type = cfg_vector_t::size_type;
00024     for (size_type i = 0; i < v.data.size(); i++) {
00025         r *= 2.0;
00026         r += v[i];
00027     }
00028
00029     return r / powl(2.0, v.data.size());
00030 }
00031
00032
00034 template<typename modell_t>
00035 requires (modell_t::is_value_computation_implementation)
00036 class inaccurate_modell
00037 {
00038     public:
00039         using cfg_vector_t = modell_t::cfg_vector_type;
00040         using prec_type = modell_t::prec_type;
00041         using prec_t = prec_type;
00042         constexpr static bool is_value_computation_implementation
00043             = modell_t::is_value_computation_implementation;
00044
00046         typedef struct {
00048             modell_t::prepared_computation pcom;
00049
00051             std::lognormal_distribution<prec_t> error_distr;
00052         } prepared_computation;
00053
00054     private:
00056         modell_t modell_;
00057
00058         // lognormal distributed errors
00059         // std::lognormal_distribution<prec_t> error_distr; // (mu, sigma)
00060
00064         const prec_t error_strength;
00065
00067         const prec_t error_offset;
00068
00069     public:
00070
00072         inaccurate_modell(modell_t &modell_, prec_t error_strength_) :
00073             modell_(modell_),
00074             error_strength(error_strength_),
00075             // error_distr(0, error_strength_),
00076             error_offset(exp(error_strength_*error_strength_/2) - 1)
00077         {}
00078
00080         void prepare_computation_with_cfg_vector(
00081             const cfg_vector_t &v, prepared_computation &pcom)
00082         {
00083             modell_.prepare_computation_with_cfg_vector(v, pcom.pcom);
00084             pcom.error_distr =
00085                 std::lognormal_distribution<prec_t>(0, error_strength);
00086         }
00087
00089         template<typename prng_t, typename rpmanager_t>
00090         auto operator()(
00091             prng_t &prng,
00092             rpmanager_t &rpmanager,
00093             prepared_computation &pcom) const
```

```
00094         {
00095             // rpmanager.execute_rpc(modell_(prng, rpmanager, pcom.pcom))
00096
00097             prec_t f = pcom.pcom.f;
00098             prec_t errf = pcom.error_distr(prng) - error_offset;
00099
00100             return rpmanager.prepare_call("modell_multiply", f, errf);
00101         }
00102
00104         void finish_computation(prepared_computation &pcom) {
00105             modell_.finish_computation(pcom.pcom);
00106         }
00107 };
00108
00109
00111 template<typename cfg_vector_t, typename prec_t>
00112 class modell
00113 {
00114     public:
00115         using prec_type = prec_t;
00116         using cfg_vector_type = cfg_vector_t;
00117         constexpr static bool is_value_computation_implementation = true;
00118
00120         typedef struct {
00122             prec_t f;
00123         } prepared_computation;
00124
00125     private:
00127         const prec_t s;
00128
00129         friend inaccurate_modell<modell<cfg_vector_t, prec_t>>;
00130
00131     public:
00133         modell(prec_t s_) : s(s_) {}
00134
00135
00137         void prepare_computation_with_cfg_vector(
00138             const cfg_vector_t &v, prepared_computation &pcom)
00139         {
00140             prec_t x = bitvector_to_num<cfg_vector_t, prec_t>(v);
00141             assert(0 <= x);
00142             assert(x <= 1);
00143
00144             static const prec_t nf = (cosh(s) - 3.*sinh(s) - 1);
00145             pcom.f = 1./2. + sinh(s - 2.*s*x) / nf;
00146         }
00147
00149         template<typename prng_t, typename rpmanager_t>
00150         auto operator()(
00151             prng_t &prng,
00152             rpmanager_t &rpmanager,
00153             prepared_computation &pcom) const
00154         {
00155             ignore(prng);
00156
00157             return rpmanager.prepare_call("modell_identity", pcom.f);
00158         }
00159
00161         void finish_computation(prepared_computation &pcom) {
00162             ignore(pcom);
00163         }
00164 };
00165
00166
00167
00168
00169 #endif
```

7.23 probsat-with-resolvents.hpp

```
00001 #ifndef PROBSAT_WITH_RESOLVENTS_HPP
00002 #define PROBSAT_WITH_RESOLVENTS_HPP
00003
00004 #include <filesystem>
00005 #include <string>
00006 #include <cstddef>
00007
00008 #include "../configuration/configuration_vector.hpp"
00009
00010 #include "utility/resolvents_manager.hpp"
00011 #include "utility/probsat-execution.hpp"
00012
00013 #include "../../util/util.hpp"
```

```
00014 #include "../../util/shared_tmpfile.hpp"
00015
00016
00018 template<typename cfg_vector_t, typename prec_t, bool use_bitfield>
00019 requires (
00020     std::is_same<binary_configuration_vector<use_bitfield>,
00021     cfg_vector_t>::value
00022 )
00023 class probsat_resolvents
00024 {
00025     public:
00026         using prec_type = prec_t;
00027         using cfg_vector_type = cfg_vector_t;
00028         constexpr static bool is_value_computation_implementation = true;
00029
00030     private:
00032         std::size_t cnf_num_vars;
00033
00035         std::size_t cnf_num_clauses;
00036
00038         std::string cnf_file;
00039
00041         std::string cnf_file_header;
00042
00044         using seed_distr_t = std::uniform_int_distribution<probsat_seed_t>;
00045
00046     public:
00048         std::string workgroup_description;
00049
00050
00052         typedef struct {
00054             shared_tmpfile stmpfile;
00055
00057             seed_distr_t seed_distr;
00058         } prepared_computation;
00059
00060
00062         resolvents_manager resolvents;
00063
00064
00066         probsat_resolvents(
00067             std::string cnf_filename,
00068             std::string resolvents_fn
00069         ) :
00070             cnf_num_vars(0),
00071             cnf_num_clauses(0),
00072             cnf_file(),
00073             cnf_file_header(),
00074             resolvents()
00075         {
00076             std::ptrdiff_t fsize = slurp_file(cnf_file, cnf_filename);
00077             if ((0 > fsize) || ((std::size_t) fsize != cnf_file.size()))
00078             {
00079                 throw std::runtime_error(
00080                     "cant read cnf file " + cnf_filename);
00081             }
00082
00083             std::string cfg_marker = "p cnf ";
00084             std::size_t pos = cnf_file.find(cfg_marker);
00085             std::size_t eol = cnf_file.find("\n", pos+1, 1);
00086             if ((std::string::npos == pos) || (std::string::npos == eol))
00087             {
00088                 throw std::runtime_error(
00089                     "can't find config line starting with '"
00090                     + cfg_marker + "' in cnf file " + cnf_filename);
00091             }
00092             cnf_file[pos] = 'c'; // comment out the cnf line
00093             std::string buf = cnf_file.substr(
00094                 pos+cfg_marker.length(), eol-pos-cfg_marker.length());
00095             std::istringstream iss(buf);
00096             iss » cnf_num_vars;
00097             iss » cnf_num_clauses;
00098
00099             cnf_file_header = cnf_file.substr(0, eol+1);
00100             cnf_file = cnf_file.substr(eol+1);
00101
00102             if (!resolvents.load(resolvents_fn)) {
00103                 throw std::runtime_error(
00104                     "cant read resolvents file " + resolvents_fn);
00105             }
00106         }
00107
00108         // ~probsat_resolvents() {
00109             // std::cout « "~probsat_resolvents" « std::endl;
00110         // }
00111
00113         probsat_resolvents(const probsat_resolvents &other) = delete;
```

```
00114
00115
00117         void prepare_computation_with_cfg_vector(
00118             const cfg_vector_t &v, prepared_computation &pcom)
00119         {
00120             pcom.seed_distr.reset();
00121
00122             pcom.stmpfile = std::move(shared_tmpfile(
00123                 "sat-with-resolvents-" + workgroup_description));
00124             create_cnf_file_from_cfg_vector(v, pcom.stmpfile.fptr);
00125         }
00126
00127
00129         template<typename prng_t, typename rpmanager_t>
00130         auto operator()(
00131             prng_t &prng,
00132             rpmanager_t &rpmanager,
00133             prepared_computation &pcom) const
00134         {
00135             std::string satfile = pcom.stmpfile.fname;
00136             probsat_seed_t seed = pcom.seed_distr(prng);
00137
00138             // std::cout « "c PWR preparing "
00139                 // « satfile « " with seed " « seed « std::endl;
00140
00141             return rpmanager.prepare_call(
00142                 "execute_probsat", satfile, seed);
00143         }
00144
00145         void finish_computation(prepared_computation &pcom) {
00146             pcom.stmpfile.remove();
00147         }
00148
00149     private:
00150
00152         FILE *create_cnf_file_from_cfg_vector(
00153             const cfg_vector_t &v, FILE *fptr) const
00154         {
00155             // TODO
00156             assert(v.data.size() <= resolvents.get_num_resolvents());
00157             // assert(v.data.size() == resolvents.get_num_resolvents());
00158
00159             // FILE *fptr = tmpfile();
00160             assert(nullptr != fptr);
00161
00162             std::size_t num_clauses = cnf_num_clauses;
00163             for (typename cfg_vector_t::size_type i = 0;
00164                 i < v.data.size(); i++)
00165             {
00166                 if (0 != v.data[i]) {
00167                     num_clauses++;
00168                 }
00169             }
00170
00171             assert(cnf_file_header.size() ==
00172                 fwrite(
00173                     cnf_file_header.data(),
00174                     1,
00175                     cnf_file_header.size(),
00176                     fptr)
00177                 || perror_("fwrite failed"));
00178             std::string cfg_line = std::string("p cnf ")
00179                 + std::to_string(cnf_num_vars) + std::string(" ")
00180                 + std::to_string(num_clauses) + std::string("\n");
00181
00182             cfg_line = "c after adding resolvents:\n" + cfg_line;
00183
00184             assert(cfg_line.size() ==
00185                 fwrite(cfg_line.data(), 1, cfg_line.size(), fptr)
00186                 || perror_("fwrite failed"));
00187
00188             assert(cnf_file.size() ==
00189                 fwrite(cnf_file.data(), 1, cnf_file.size(), fptr)
00190                 || perror_("fwrite failed"));
00191
00192             typename cfg_vector_t::size_type num_clauses_to_write
00193                 = num_clauses - cnf_num_clauses;
00194             for (typename cfg_vector_t::size_type i = 0;
00195                 i < v.data.size(); i++)
00196             {
00197                 if (0 != v.data[i]) {
00198                     std::string rv = resolvents.get_resolvent(i);
00199                     if (1 < num_clauses_to_write) {
00200                         rv += "\n";
00201                     }
00202
00203                     assert(rv.size() ==
```

```
00204                         fwrite(rv.data(), 1, rv.size(), fptr));
00205                     num_clauses_to_write--;
00206                 }
00207             }
00208
00209             assert(0 == num_clauses_to_write);
00210
00211             assert(0 == fflush(fptr) || perror_("fflush failed"));
00212
00213             return std::move(fptr);
00214         }
00215 };
00216
00217 #endif
00218
```

7.24 probsat-execution.cpp

```
00001
00002 #include <chrono>
00003 #include <string>
00004 #include <iostream>
00005 #include <cstdio>
00006 #include <cstddef>
00007 #include <sys/select.h>
00008 #include <stdio.h>
00009
00010 #include "probsat-execution.hpp"
00011
00012 #include "../../../config.hpp"
00013 #include "../../../util/util.hpp"
00014
00015
00016 uint8_t execute_probsat_(
00017     uint64_t &num_flips,
00018     const std::string &filename,
00019     const probsat_seed_t seed,
00020     const uint64_t max_flips,
00021     const std::chrono::seconds &max_exec_time,
00022     std::string *debug_probsat_output)
00023 {
00024     uint8_t reason = 0;
00025
00026     const std::string cmd =
00027         probsat_cmd +
00028         ((0 < max_flips) ?
00029             " --maxflips " + std::to_string(max_flips) : "")
00030         + " " + filename + " " + std::to_string(seed) + " 2>&1";
00031
00032     num_flips = std::numeric_limits<uint64_t>::max();
00033
00034     FILE *pipe_fh = nullptr;
00035     pipe_fh = popen(cmd.c_str(), "r");
00036     if (!pipe_fh) {
00037         perror("popen failed");
00038         return reason;
00039     }
00040
00041     try {
00042         auto start = std::chrono::high_resolution_clock::now();
00043
00044         int piped = fileno(pipe_fh);
00045         if (-1 == piped) {
00046             perror("fileno(pipe_fh)");
00047             throw std::runtime_error("fileno failed");
00048         }
00049
00050         fcntl(piped, F_SETFL, O_NONBLOCK);
00051         if (-1 == piped) {
00052             perror("fcntl(piped, F_SETFL, O_NONBLOCK);");
00053             throw std::runtime_error("fcntl failed");
00054         }
00055
00056         const std::string marker = "c numFlips";
00057         size_t bsize = 4096;
00058         assert(bsize <= std::numeric_limits<ssize_t>::max());
00059         char buffer[bsize];
00060         std::string bstr = "";
00061
00062         fd_set readfds;
00063         FD_ZERO(&readfds);
00064         FD_SET(piped, &readfds);
00065
```

```
00066         timespec timeout;
00067         timespec *timeout_ptr =
00068             (std::chrono::seconds::zero() == max_exec_time) ?
00069                 nullptr : &timeout;
00070         assert(std::chrono::ceil<std::chrono::seconds>(
00071             max_exec_time - std::chrono::seconds::zero()).count()
00072             <= std::numeric_limits<decltype(timeout.tv_sec)>::max());
00073
00074         bool running = fd_is_valid(piped);
00075         while (running)
00076         {
00077             auto now = std::chrono::high_resolution_clock::now();
00078             auto current_execution_time = now - start;
00079
00080             if (nullptr != timeout_ptr) {
00081                 timeout.tv_sec = (decltype(timeout.tv_sec)) std::max(0l,
00082                     std::chrono::ceil<std::chrono::seconds>(
00083                         max_exec_time - current_execution_time).count());
00084                 timeout.tv_nsec = 1;
00085             }
00086
00087             int rval = pselect(
00088                 piped+1, &readfds, nullptr, nullptr, timeout_ptr, nullptr);
00089             switch (rval) {
00090                 case 0: // timeout
00091                     now = std::chrono::high_resolution_clock::now();
00092                     current_execution_time =
00093                         std::chrono::duration_cast<std::chrono::seconds>
00094                         (now - start);
00095                     if (max_exec_time <= current_execution_time) {
00096                         std::cerr « "timeout after "
00097                             « current_execution_time.count()
00098                             « " seconds" « std::endl;
00099                         running = false;
00100                         reason = 1;
00101                     }
00102                     break;
00103
00104                 case 1: // data available
00105                     while (running)
00106                     {
00107                         ssize_t num_chars_read =
00108                             read(piped, &buffer[0], bsize);
00109                         buffer[num_chars_read] = 0; // null terminate
00110
00111                         if (0 > num_chars_read)
00112                         {
00113                             if ((errno != EAGAIN)
00114                                 and (errno != EWOULDBLOCK))
00115                             {
00116                                 perror("read");
00117                                 throw std::runtime_error("read failed");
00118                                 running = false;
00119                             } else {
00120                                 break;
00121                             }
00122                         } else {
00123                             if (nullptr != debug_probsat_output) {
00124                                 *debug_probsat_output +=
00125                                     std::string(&buffer[0]) + "|BLOCK|";
00126                             }
00127                         }
00128
00129                         if (0 < num_chars_read)
00130                         {
00131                             std::size_t offset = bstr.length();
00132                             std::string bfs = std::string(&buffer[0]);
00133                             bstr += bfs;
00134                             assert(bfs.length()
00135                                 == (std::size_t) num_chars_read);
00136
00137                             while (true) {
00138                                 size_t pos = bstr.find("\n", offset);
00139                                 if (pos == std::string::npos) {
00140                                     break;
00141                                 }
00142
00143                                 std::string line = bstr.substr(0, pos);
00144                                 bstr =
00145                                     bstr.substr(pos+1, std::string::npos);
00146                                 offset = 0;
00147
00148                                 // std::cout « "c probsat: "
00149                                     // « line « std::endl;
00150
00151                                 if (line.starts_with(marker)) {
00152                                     size_t pos =
```

```
00153                                       line.find(": ", marker.length());
00154                                   assert(pos != std::string::npos);
00155
00156                                   std::string num_flips_str =
00157                                       line.substr(pos + 2);
00158                                   assert(!num_flips_str.empty());
00159
00160                                   num_flips = from_string<std::size_t>
00161                                       (num_flips_str);
00162                                   running = false;
00163                                   reason = 255;
00164                                   break;
00165                               }
00166                           }
00167                       }
00168                   }
00169
00170                   if (running) {
00171                       running = fd_is_valid(piped);
00172                   }
00173                   break;
00174
00175               default:
00176                   perror("pselect");
00177                   throw std::runtime_error("pselect failed");
00178                   running = false;
00179                   break;
00180           };
00181       }
00182   } catch(std::exception &ex) {
00183       std::cerr « "exception executing probsat:\n";
00184       if (!cmd.empty()) { std::cerr « cmd « "\n"; }
00185       std::cerr « ex.what() « std::endl;
00186   }
00187
00188   if (pipe_fh) {
00189       if (nullptr != debug_probsat_output) {
00190           *debug_probsat_output += "|CLOSE|";
00191       }
00192
00193       auto rval = pclose(pipe_fh);
00194       if (-1 == rval) {
00195           perror("pclose(pipe_fh)");
00196       }
00197       pipe_fh = nullptr;
00198   }
00199
00200   return reason;
00201 }
00202
00203
00204 std::string execute_cmd(std::string cmd)
00205 {
00206   FILE *pipe_fh = nullptr;
00207   pipe_fh = popen(cmd.c_str(), "r");
00208   if (!pipe_fh) {
00209       perror("popen failed");
00210       return "popen failed";
00211   }
00212
00213   std::string content;
00214   try {
00215       char c;
00216       while ((c = fgetc(pipe_fh)) != EOF) {
00217           content += c;
00218       }
00219   } catch(std::exception &ex) {
00220       std::cerr « "exception executing probsat:\n";
00221       if (!cmd.empty()) { std::cerr « cmd « "\n"; }
00222       std::cerr « ex.what() « std::endl;
00223       return content + ex.what();
00224   }
00225
00226   if (pipe_fh) {
00227       auto rval = pclose(pipe_fh);
00228       if (-1 == rval) {
00229           perror("pclose(pipe_fh)");
00230       }
00231       pipe_fh = nullptr;
00232   }
00233
00234   return content;
00235 }
00236
00237 std::pair<uint64_t, probsat_return_cause::reason> execute_probsat
00238   (std::string filename, probsat_seed_t seed)
00239 {
```

```
00240     using reason_t = probsat_return_cause::reason;
00241     try {
00242         uint64_t num_flips = std::numeric_limits<uint64_t>::max();
00243 
00244         uint8_t reason = execute_probsat_(
00245             num_flips,
00246             filename,
00247             seed,
00248             probsat_max_flips,
00249             probsat_max_exec_time);
00250 
00251         if (1 == reason) {
00252             return std::make_pair<uint64_t, reason_t>(
00253                 std::move(num_flips), probsat_return_cause::TIMEOUT);
00254         } else if (255 == reason) {
00255             if ((0 < num_flips) && (num_flips == probsat_max_flips)) {
00256                 return std::make_pair<uint64_t, reason_t>(
00257                     std::move(num_flips), probsat_return_cause::MAX_FLIPS);
00258             } else {
00259                 return std::make_pair<uint64_t, reason_t>(
00260                     std::move(num_flips), probsat_return_cause::SUCCESS);
00261             }
00262         }
00263     } catch(std::exception &ex) {
00264         std::cerr « "exception executing probsat:\n";
00265         std::cerr « ex.what() « std::endl;
00266     }
00267 
00268     return std::make_pair<uint64_t, reason_t>(
00269         std::numeric_limits<uint64_t>::max(), probsat_return_cause::ERROR);
00270 }
00271 
00272 
00273 
```

7.25 probsat-execution.hpp

```
00001 #ifndef PROBSAT_EXECUTION_HPP
00002 #define PROBSAT_EXECUTION_HPP
00003 
00004 
00005 #include <chrono>
00006 #include <string>
00007 #include <iostream>
00008 #include <cstdio>
00009 #include <cstddef>
00010 #include <sys/select.h>
00011 
00012 #include "../../../config.hpp"
00013 #include "../../../util/util.hpp"
00014 
00015 #include "../../../util/basic_serialization.hpp"
00016 
00017 
00019 namespace probsat_return_cause {
00020 
00022     enum reason : uint8_t {
00023         ERROR = 0,
00024         TIMEOUT = 1,
00025         MAX_FLIPS= 2,
00026         SUCCESS = 3,
00027         NUM_REASONS = 4
00028     };
00029 
00031     const static std::string as_string[reason::NUM_REASONS] = {
00032         "ERROR",
00033         "TIMEOUT",
00034         "MAX_FLIPS",
00035         "SUCCESS"
00036     };
00037 
00039     template<typename S>
00040     void serialize(S &s, reason &r) {
00041         uint8_t v = r;
00042         generic_serialize(s, v);
00043         assert(v <= probsat_return_cause::NUM_REASONS);
00044         r = static_cast<reason>(v);
00045     }
00046 };
00047 
00048 
00050 uint8_t execute_probsat_(
00051     uint64_t &num_flips,
```

```
00052     const std::string &filename,
00053     const probsat_seed_t seed,
00054     const uint64_t max_flips = 0,
00055     const std::chrono::seconds &max_exec_time
00056         = std::chrono::seconds::zero(),
00057     std::string *debug_probsat_output = nullptr);
00058
00059
00061 std::pair<uint64_t, probsat_return_cause::reason>
00062     execute_probsat(std::string filename, probsat_seed_t seed);
00063
00064
00066 template<typename prec_t = prec_t>
00067 struct result_statistics
00068 {
00070     std::size_t probsat_executions = 0;
00071
00073     std::size_t reasons[probsat_return_cause::NUM_REASONS] = {0};
00074
00076     double total_flips_executed = 0;
00077
00078
00080     prec_t operator()(
00081         std::pair<uint64_t, probsat_return_cause::reason> result)
00082     {
00083         prec_t num_flips = static_cast<prec_t>(result.first);
00084         probsat_executions++;
00085         assert(result.second < probsat_return_cause::NUM_REASONS);
00086         reasons[result.second]++;
00087
00088         if (std::numeric_limits<uint64_t>::max() == num_flips) {
00089             num_flips = std::numeric_limits<prec_t>::quiet_NaN();
00090         } else {
00091             total_flips_executed += num_flips;
00092         }
00093
00094         if ((interpret_timeout_as_max_flips_reached)
00095             && (0 < probsat_max_flips))
00096         {
00097             if (result.second == probsat_return_cause::TIMEOUT) {
00098                 num_flips = probsat_max_flips;
00099             }
00100         }
00101
00102         if (result.second != probsat_return_cause::SUCCESS) {
00103             std::cerr << "probsat executed with " <<
00104                 probsat_return_cause::as_string[result.second]
00105                 << " and " << num_flips << " flips" << std::endl;
00106         }
00107
00108         // return std::log(num_flips);
00109         return num_flips;
00110     };
00111
00113     friend std::ostream &operator<<(
00114         std::ostream &os, const result_statistics &rs)
00115     {
00116         namespace prc = probsat_return_cause;
00117         // new version is easier parseable from the command line
00118         os << rs.probsat_executions << " executions, "
00119             << rs.reasons[prc::SUCCESS] << " successfull, "
00120             << rs.reasons[prc::MAX_FLIPS] << " max_flips_reached, "
00121             << rs.reasons[prc::TIMEOUT] << " terminated_by_timeout, "
00122             << rs.reasons[prc::ERROR] << " with_errors, "
00123             << rs.total_flips_executed << " flips_in_total_(at_least)";
00124 /* old version
00125         os << rs.probsat_executions << " executions ("
00126             << rs.reasons[prc::SUCCESS] << " successfull, "
00127             << rs.reasons[prc::MAX_FLIPS] << " max flips reached, "
00128             << rs.reasons[prc::TIMEOUT] << " terminated by timeout, "
00129             << rs.reasons[prc::ERROR] << " with errors, "
00130             << "and at least " <<
00131                 rs.total_flips_executed << " flips in total)";
00132 */
00133         return os;
00134     }
00135
00136 };
00137
00138
00139 #endif
```

7.26 resolvents_manager.cpp

```
00001
00002 #include <string>
00003 #include <vector>
00004 #include <fstream>
00005 #include <iostream>
00006 #include <cassert>
00007
00008 #include "resolvents_manager.hpp"
00009
00010 bool resolvents_manager::load(std::string filename) {
00011     std::ifstream file(filename, std::ifstream::in);
00012
00013     if (false == file.is_open() ) {
00014         std::string errmsg = std::string("can't open ") + filename;
00015         perror(errmsg.c_str());
00016         return false;
00017     }
00018
00019     while (file.good()) {
00020         std::string line;
00021         std::getline(file, line);
00022         if ((false == line.starts_with("c ")) && (!line.empty())) {
00023             assert(line.ends_with(" 0"));
00024             resolvent.push_back(line);
00025         }
00026     }
00027
00028     file.close();
00029
00030     return true;
00031 }
00032
00033 std::size_t resolvents_manager::get_num_resolvents() const {
00034     return resolvent.size();
00035 }
00036
00037 std::string resolvents_manager::get_resolvent(std::size_t i) const {
00038     return resolvent[i];
00039 }
00040
```

7.27 resolvents_manager.hpp

```
00001 #ifndef RESOLVENTS_MANAGER_HPP
00002 #define RESOLVENTS_MANAGER_HPP
00003
00004 #include <string>
00005 #include <vector>
00006
00008 class resolvents_manager {
00009     private:
00011         std::vector<std::string> resolvent;
00012
00013     public:
00015         resolvents_manager() = default;
00016
00018         bool load(std::string filename);
00019
00021         std::size_t get_num_resolvents() const;
00022
00024         std::string get_resolvent(std::size_t i) const;
00025 };
00026
00027 #endif
```

7.28 basic_serialization.hpp

```
00001 #ifndef BASIC_SERIALIZATION_HPP
00002 #define BASIC_SERIALIZATION_HPP
00003
00004 #include <cassert>
00005 #include <string>
00006 #include <tuple>
00007 #include <vector>
00008
00010 template <typename S, typename basic_t>
00011 void serialize_basic_type(S& s, basic_t &v) {
```

```
00012     if constexpr (1 == sizeof(basic_t)) {
00013         s.value1b(v);
00014     } else if constexpr (2 == sizeof(basic_t)) {
00015         s.value2b(v);
00016     } else if constexpr (4 == sizeof(basic_t)) {
00017         s.value4b(v);
00018     } else if constexpr (8 == sizeof(basic_t)) {
00019         s.value8b(v);
00020     }
00021     #if DEBUG_ASSERTIONS
00022     else {
00023         assert(false);
00024     }
00025     #endif
00026
00027     static_assert(
00028         (1 == sizeof(basic_t)) ||
00029         (2 == sizeof(basic_t)) ||
00030         (4 == sizeof(basic_t)) ||
00031         (8 == sizeof(basic_t)) );
00032
00033     // std::cout « "basic type of size " « sizeof(basic_t) « ":" « v « std::endl;
00034 }
00035
00036
00038 template<typename S>
00039 void serialize_marker(
00040     S &s, const std::string marker, const std::size_t length)
00041 {
00042     // length must contain null byte!
00043     std::string marker_ = marker;
00044     s.text1b(marker_, length);
00045     assert(marker == marker_);
00046 }
00047
00048
00050 template<typename S, typename data_t> requires
00051 ((1 == sizeof(data_t)) && (false == std::is_same<bool, data_t>::value))
00052 void serialize(S &s, std::vector<data_t> &v)
00053 {
00054     std::size_t length = v.size();
00055     serialize_basic_type<S, std::size_t>(s, length);
00056     v.resize(length);
00057     s.container1b(v, length);
00058     assert(v.size() == length);
00059 }
00060
00061
00063 template<typename S>
00064 void generic_serialize(S &s) { ignore(s); }
00065
00067 template<typename S, typename first_t, typename... arg_types>
00068 void generic_serialize(S &s, first_t &a, arg_types &... args);
00069
00071 template<typename S, typename first_t, typename second_t>
00072 void serialize(S &s, std::pair<first_t, second_t> &v) {
00073     generic_serialize(s, v.first, v.second);
00074 }
00075
00077 template<typename S, typename first_t, typename... arg_types>
00078 void generic_serialize(S &s, first_t &a, arg_types &... args)
00079 {
00080     // serialize first argument
00081     if constexpr (std::is_fundamental<first_t>::value) {
00082         serialize_basic_type<S, first_t>(s, a);
00083     } else if constexpr (std::is_same<std::string, first_t>::value) {
00084         auto length = a.size();
00085         generic_serialize(s, length);
00086         a.reserve(length);
00087         s.text1b(a, length);
00088         assert(a.size() == length);
00089     } else if constexpr (std::is_same<const char *, first_t>::value) {
00090         // one way trap for serialization of const char *
00091         // note: when used with remote procedure calls
00092         // the function parameter must be std::string,
00093         // but the argument passed may const char *
00094         std::string tmp_str = a;
00095         generic_serialize(s, tmp_str);
00096     } else {
00097         serialize(s, a);
00098     }
00099
00100     // recursively pass further arguments
00101     generic_serialize(s, args...);
00102 }
00103
00104
```

```
00106 template<typename S>
00107 void serialize(S &s, std::tuple<> &v) {
00108     ignore(s, v);
00109 }
00110
00111 /*
00113 template<typename S, typename head_arg_type, typename... tail_arg_types>
00114 void serialize(S &s, std::tuple<head_arg_type, tail_arg_types...> &v) {
00115     head_arg_type tmp = std::get<0>(v);
00116     generic_serialize<S, head_arg_type>(s, tmp);
00117     if constexpr (0 < std::tuple_size<std::tuple<tail_arg_types...»::value)
00118     {
00119         auto t = tail(v);
00120         serialize<S, tail_arg_types...>(s, t);
00121         v = std::tuple_cat(std::make_tuple(tmp), t);
00122     } else {
00123         v = std::make_tuple(tmp);
00124     }
00125 }
00126 */
00127
00129 template<typename S, typename... arg_types>
00130 void serialize(S &s, std::tuple<arg_types...> &t) {
00131     apply([&s](auto &... args) { generic_serialize<S, arg_types...>(s, args...); }, t);
00132 }
00133
00134
00136 template<typename t>
00137 struct make_generic_serializeable {
00138     t data;
00139 };
00140
00142 template<typename S, typename t>
00143 void serialize(S &s, make_generic_serializeable<t> &o) {
00144     generic_serialize(s, o.data);
00145 }
00146
00147
00148 /*
00149 #include "util.hpp"
00150
00152 template<typename S>
00153 void serialize(S &s, std::vector<bool> &v) {
00154     auto content = vector_to_string(v);
00155     std::cout « "content length before: " « content.length() « std::endl;
00156     // ignore(content);
00157     generic_serialize<S, std::string>(s, content);
00158     std::cout « "content length after: " « content.length() « std::endl;
00159     v = vector_from_string(content);
00160 }
00161 */
00162
00164 template<typename S>
00165 void serialize(S &s, std::vector<bool> &v) {
00166     std::size_t length = v.size();
00167     serialize_basic_type<S, std::size_t>(s, length);
00168     v.resize(length);
00169
00170     for (std::size_t i = 0; i < length; i++) {
00171         uint8_t value = v[i] ? 1 : 0;
00172         s.value1b(value);
00173         v[i] = (0 != value);
00174     }
00175
00176     // s.container1b(v, length);
00177     assert(v.size() == length);
00178 }
00179
00180 #endif
```

7.29 bitfield.hpp

```
00001 #ifndef BITFIELD_HPP
00002 #define BITFIELD_HPP
00003
00004 #include <cstdint>
00005 #include <cassert>
00006 #include <type_traits>
00007 #include <ostream>
00008
00013 template<typename length_t>
00014 class bitfield {
00015     private:
```

```
00017         length_t length;
00018
00019         // helper values to allow returning references to memory
00020
00022         length_t last_mod_index;
00023
00025         bool last_mod_value;
00026
00028         uint8_t *array;
00029
00030
00031         // helper functions
00032
00034         static inline length_t to_bytelength(const length_t l) {
00035             return (l+7) » 3;
00036         }
00037
00039         inline bool get(const length_t index) const
00040         {
00041             #if DEBUG_ASSERTIONS
00042             assert(index < length);
00043             #endif
00044
00045             return 1 & (array[index » 3] » (index & 0b111));
00046         }
00047
00049         inline void set(const length_t index, const bool value)
00050         {
00051             update();
00052
00053             #if DEBUG_ASSERTIONS
00054             assert(index < length);
00055             #endif
00056
00057             const uint8_t mask = 1 « (index & 0b111);
00058             if (value) {
00059                 array[index » 3] |= mask;
00060             } else {
00061                 array[index » 3] &= ~mask;
00062             }
00063         }
00064
00065         // apply changes by returned reference to array
00066         inline void update() {
00067             if (last_mod_index < length) {
00068                 // std::cout « "a["«(int)last_mod_index«"] = "
00069                     // « (int)last_mod_value « std::endl;
00070                 set(last_mod_index, last_mod_value);
00071                 last_mod_index = length;
00072             }
00073         }
00074
00075     public:
00076         // constructors
00077
00079         bitfield() :
00080             length(0),
00081             last_mod_index(length),
00082             last_mod_value(false),
00083             array(nullptr) {}
00084
00086         bitfield(length_t n) :
00087             length(n), last_mod_index(length), last_mod_value(false),
00088             array(length > 0 ? new uint8_t[to_bytelength(length)] : nullptr) {}
00089
00090         // bitfield(const bitfield &other) = delete;
00091
00093         bitfield(const bitfield &other) :
00094             length(other.length),
00095             last_mod_index(other.last_mod_index),
00096             last_mod_value(other.last_mod_value),
00097             array(length > 0 ? new uint8_t[to_bytelength(length)] : nullptr)
00098         {
00099             for (length_t i = 0; i < to_bytelength(length); i++) {
00100                 array[i] = other.array[i];
00101             }
00102         }
00103
00105         ~bitfield() {
00106             if (nullptr != array) {
00107                 #if DEBUG_ASSERTIONS
00108                     assert(0 < length);
00109                 #endif
00110                 delete[] array;
00111                 array = nullptr;
00112             } else {
00113                 #if DEBUG_ASSERTIONS
```

```
00114                   assert(0 == length);
00115               #endif
00116           }
00117       }
00118
00119       // copy & swap idiom
00120
00122       friend void swap(bitfield &a, bitfield &b)
00123           noexcept(std::is_nothrow_swappable_v<length_t>
00124               &&   std::is_nothrow_swappable_v<bool>
00125               &&   std::is_nothrow_swappable_v<uint8_t*>)
00126       {
00127           #if DEBUG_ASSERTIONS
00128           assert(a.length == b.length);
00129           #endif
00130           // or remove const from length
00131           // std::swap(a.length, b.length);
00132           std::swap(a.last_mod_index, b.last_mod_index);
00133           std::swap(a.last_mod_value, b.last_mod_value);
00134           std::swap(a.array, b.array);
00135       }
00136
00138       bitfield(bitfield &&other) : bitfield() {
00139           swap(*this, other);
00140       }
00141
00143       bitfield& operator=(bitfield other) {
00144           swap(*this, other);
00145           return *this;
00146       }
00147
00153       void resize(length_t new_length)
00154       {
00155           update();
00156
00157           if (to_bytelength(new_length) > to_bytelength(length)) {
00158               assert(0 < to_bytelength(new_length));
00159
00160               uint8_t *old_array = array;
00161
00162               array = new uint8_t[to_bytelength(new_length)];
00163
00164               for (length_t i = 0; i < to_bytelength(length); i++) {
00165                   array[i] = old_array[i];
00166               }
00167
00168               if (old_array) {
00169                   delete[] old_array;
00170               }
00171           }
00172
00173           length = new_length;
00174           last_mod_index = length;
00175       }
00176
00178       length_t get_bytelength() const {
00179           return to_bytelength(length);
00180       }
00181
00183       uint8_t *data() {
00184           update();
00185           return array;
00186       }
00187
00188       // access operators
00189
00197       inline bool& operator[](const length_t index)
00198       {
00199           update();
00200
00201           #if DEBUG_ASSERTIONS
00202           assert(index < length);
00203           #endif
00204
00205           last_mod_index = index;
00206           last_mod_value = get(index);
00207           return last_mod_value;
00208       }
00209
00211       inline bool operator[](const length_t index) const {
00212           // dont apply changes, so this function can be const
00213           if (index == last_mod_index) {
00214               return last_mod_value;
00215           }
00216
00217           return get(index);
00218       }
```

```
00219
00221         length_t size() const {
00222             return length;
00223         }
00224
00225         // output
00226
00228         std::ostream &print(std::ostream &os, const char *separator = " ")
00229         {
00230             update();
00231
00232             const length_t bl = to_bytelength(length);
00233
00234             auto sep = "";
00235             for (length_t i = 0; i < bl-1; i++) {
00236                 os « sep;
00237
00238                 uint8_t d = array[i];
00239                 for (uint8_t j = 0; j < 8; j++) {
00240                     os « ((d & 1) ? "1" : "0");
00241                     d »= 1;
00242                 }
00243
00244                 sep = separator;
00245             }
00246
00247             if (0 < bl) {
00248                 os « sep;
00249                 for (length_t i = (bl-1)*8; i < length; i++) {
00250                     os « (get(i) ? "1" : "0");
00251                 }
00252             }
00253
00254             return os;
00255         }
00256 };
00257
00259 template<typename length_t>
00260 std::ostream &operator«(std::ostream& os, bitfield<length_t> bf) {
00261     return bf.print(os);
00262 }
00263
00265 template<typename S, typename length_t>
00266 void serialize(S &s, bitfield<length_t> &bf) {
00267     bf.update();
00268
00269     length_t size = bf.length;
00270     serialize_basic_type<S, length_t>(s, size);
00271
00272     bf.resize(size);
00273
00274     uint8_t *dataptr = bf.data();
00275     for (std::size_t i = 0; i < bf.get_bytelength(); i++) {
00276         s.value1b(dataptr[i]);
00277     }
00278
00279     assert(bf.size() == size);
00280 }
00281
00282 #endif
```

7.30 debug.hpp

```
00001 #ifndef DEBUG_HPP
00002 #define DEBUG_HPP
00003
00004 /*
00005     attach gdb in realtime to current process and print stack trace
00006     extremely useful for multi-process environments like mpi
00007
00008     **complete code in this file is from**
00009
       https://stackoverflow.com/questions/4636456/how-to-get-a-stack-trace-for-c-using-gcc-with-line-number-information
00010 */
00011
00012 #include <stdio.h>
00013 #include <stdlib.h>
00014 #include <sys/wait.h>
00015 #include <unistd.h>
00016 #include <sys/prctl.h>
00017
00027 bool print_trace() {
00028     char pid_buf[30];
```

```
00029     sprintf(pid_buf, "%d", getpid());
00030     char name_buf[512];
00031     name_buf[readlink("/proc/self/exe", name_buf, 511)]=0;
00032     prctl(PR_SET_PTRACER, PR_SET_PTRACER_ANY, 0, 0, 0);
00033     int child_pid = fork();
00034     if (!child_pid) {
00035         dup2(2,1); // redirect output to stderr - edit: unnecessary?
00036         execl("/usr/bin/gdb", "gdb", "--batch", "-n", "-ex", "thread", "-ex", "bt", name_buf, pid_buf,
      NULL);
00037         abort(); /* If gdb failed to start */
00038     } else {
00039         waitpid(child_pid,NULL,0);
00040     }
00041
00042     return false;
00043 }
00044
00045 #endif
```

7.31 include_bitsery.hpp

```
00001 #ifndef INCLUDE_BITSERY_HPP
00002 #define INCLUDE_BITSERY_HPP
00003
00008 #include <bitsery/bitsery.h>
00009 #include <bitsery/adapter/buffer.h>
00010 #include <bitsery/traits/vector.h>
00011 #include <bitsery/traits/string.h>
00012 #include <bitsery/ext/std_tuple.h>
00013
00014 #endif
```

7.32 mpi_async_communication.cpp

```
00001
00002 #include <cassert>
00003 #include <algorithm>
00004
00005 #include "mpi_async_communication.hpp"
00006
00007
00008 void async_comm::wait_for_one_message()
00009 {
00010     std::size_t size = outstanding_requests.size();
00011     if (0 >= size) { return; }
00012
00013     int index = MPI_UNDEFINED;
00014     assert(MPI_SUCCESS == MPI_Waitany(
00015         size, outstanding_requests.data(), &index, MPI_STATUSES_IGNORE));
00016
00017     if (MPI_UNDEFINED != index) {
00018         assert((0 <= index) && ((std::size_t) index < size));
00019         outstanding_requests.erase(outstanding_requests.begin() + index);
00020         outstanding_data.erase(outstanding_data.begin() + index);
00021     }
00022 }
00023
00024
00025 void async_comm::wait_for_all_messages()
00026 {
00027     std::size_t size = outstanding_requests.size();
00028     if (0 >= size) { return; }
00029
00030     assert(MPI_SUCCESS == MPI_Waitall(
00031         size, outstanding_requests.data(), MPI_STATUSES_IGNORE));
00032 }
00033
00034
00035 void async_comm::wait_for_some_messages()
00036 {
00037     std::size_t size = outstanding_requests.size();
00038     if (0 >= size) { return; }
00039
00040     int count_finished = 0;
00041     std::vector<int> indices_finished(size);
00042
00043     assert(MPI_SUCCESS == MPI_Waitsome(size, outstanding_requests.data(),
00044         &count_finished, indices_finished.data(), MPI_STATUS_IGNORE));
00045
```

```
00046     indices_finished.resize(count_finished);
00047     sort(indices_finished.begin(), indices_finished.end(), std::greater<>());
00048
00049     for (int i = 0; i < count_finished; i++) {
00050         auto index = indices_finished[i];
00051         assert(index != MPI_UNDEFINED);
00052         outstanding_requests.erase(outstanding_requests.begin() + index);
00053         outstanding_data.erase(outstanding_data.begin() + index);
00054     }
00055 }
00056
00057
00058 void async_comm::test_for_one_message()
00059 {
00060     std::size_t size = outstanding_requests.size();
00061     if (0 >= size) { return; }
00062
00063     int index = MPI_UNDEFINED;
00064     int completed = 0;
00065
00066     assert(MPI_SUCCESS == MPI_Testany(
00067         size,
00068         outstanding_requests.data(),
00069         &index,
00070         &completed,
00071         MPI_STATUS_IGNORE));
00072
00073     if (completed) {
00074         assert(index != MPI_UNDEFINED);
00075         assert(MPI_REQUEST_NULL == outstanding_requests[index]);
00076         // assert(MPI_SUCCESS == MPI_Wait(&outstanding_requests[index], MPI_STATUS_IGNORE));
00077         outstanding_requests.erase(outstanding_requests.begin() + index);
00078         outstanding_data.erase(outstanding_data.begin() + index);
00079     }
00080 }
00081
00082
00083 void async_comm::test_for_messages()
00084 {
00085     std::size_t size = outstanding_requests.size();
00086
00087     if (0 >= size) { return; }
00088     if (1 == size) { return test_for_one_message(); }
00089
00090     int count_finished = 0;
00091     std::vector<int> indices_finished(size);
00092     assert(MPI_SUCCESS == MPI_Testsome(size, outstanding_requests.data(),
00093         &count_finished, indices_finished.data(), MPI_STATUS_IGNORE));
00094
00095     indices_finished.resize(count_finished);
00096     sort(indices_finished.begin(), indices_finished.end(), std::greater<>());
00097
00098     for (int i = 0; i < count_finished; i++) {
00099         auto index = indices_finished[i];
00100         assert(index != MPI_UNDEFINED);
00101         assert(MPI_REQUEST_NULL == outstanding_requests[index]);
00102         // assert(MPI_SUCCESS == MPI_Wait(
00103             // &outstanding_requests[index], MPI_STATUS_IGNORE));
00104         outstanding_requests.erase(outstanding_requests.begin() + index);
00105         outstanding_data.erase(outstanding_data.begin() + index);
00106     }
00107 }
00108
00109
00110 auto async_comm::communications_in_queue() const
00111 {
00112     auto num_waiting = outstanding_requests.size();
00113     #if DEBUG_ASSERTIONS
00114     assert(num_waiting == outstanding_data.size());
00115     #endif
00116     return num_waiting;
00117 }
00118
00119
00120 bool async_comm::communication_is_done() const
00121 {
00122     bool is_empty = outstanding_requests.empty();
00123     #if DEBUG_ASSERTIONS
00124     assert(is_empty == outstanding_data.empty());
00125     #endif
00126     return is_empty;
00127 }
00128
00129
00130 async_comm::~async_comm()
00131 {
00132     // wait_for_all_messages();
```

```
00133     #if DEBUG_ASSERTIONS
00134     assert(outstanding_requests.empty() == outstanding_data.empty());
00135     assert(communication_is_done());
00136     #endif
00137 }
00138
00139
00140
00141 void async_comm_out::send_message(
00142     const MPI_Comm &comm, int dest, int tag, data_t &&content)
00143 {
00144     MPI_Request request;
00145     assert(MPI_SUCCESS == MPI_Isend(
00146         content.data(),
00147         content.size(),
00148         MPI_BYTE,
00149         dest,
00150         tag,
00151         comm,
00152         &request));
00153
00154     int completed = 0;
00155     assert(MPI_SUCCESS == MPI_Test(
00156         &request, &completed, MPI_STATUS_IGNORE));
00157
00158     if (!completed) {
00159         outstanding_requests.emplace_back(std::move(request));
00160         outstanding_data.emplace_back(std::move(content));
00161     } else {
00162         assert(MPI_REQUEST_NULL == request);
00163         // assert(MPI_SUCCESS == MPI_Wait(&request, MPI_STATUS_IGNORE));
00164     }
00165 }
00166
00167
00168
00169 void fake_async_comm_out::send_message(
00170     const MPI_Comm &comm, int dest, int tag, auto content)
00171 {
00172     assert(MPI_SUCCESS == MPI_Send(
00173         content.data(), content.size(), MPI_BYTE, dest, tag, comm));
00174 }
00175
```

7.33 mpi_async_communication.hpp

```
00001 #ifndef MPI_ASYNC_COMMUNICATION_HPP
00002 #define MPI_ASYNC_COMMUNICATION_HPP
00003
00004 #include <vector>
00005 #include <mpi.h>
00006
00008 class async_comm
00009 {
00010     protected:
00012         using data_t = std::vector<uint8_t>;
00013
00015         std::vector<MPI_Request> outstanding_requests;
00016
00018         std::vector<data_t> outstanding_data;
00019
00020     public:
00022         void wait_for_one_message();
00023
00025         void wait_for_all_messages();
00026
00028         void wait_for_some_messages();
00029
00031         void test_for_one_message();
00032
00034         void test_for_messages();
00035
00037         auto communications_in_queue() const;
00038
00040         bool communication_is_done() const;
00041
00043         ~async_comm();
00044 };
00045
00047 class async_comm_out : public async_comm
00048 {
00049     public:
00051         void send_message(
```

```
00052             const MPI_Comm &comm, int dest, int tag, data_t &&content);
00053 };
00054
00056 class fake_async_comm_out : public async_comm
00057 {
00058     public:
00059         void send_message(
00060             const MPI_Comm &comm, int dest, int tag, auto content);
00061 };
00062
00063
00064 #endif
```

7.34 mpi_shared_tmp_workgroup.cpp

```
00001
00002 #include <random>
00003 #include <cassert>
00004 #include <filesystem>
00005
00006 #include "mpi_shared_tmp_workgroup.hpp"
00007
00008 #include "mpi_types.hpp"
00009 #include "mpi_util.hpp"
00010 #include "../util.hpp"
00011
00012
00013 std::string mpi_shared_tmp_dir_workgroup::
00014     build_tmp_dir_path(std::string app_name)
00015 {
00016     std::string tmp_path = std::filesystem::temp_directory_path();
00017     tmp_path += "/mpi_split_by_shared_tmp/" + app_name + "-";
00018
00019     // add shared random number to path
00020     uint32_t random = 0;
00021
00022     if (0 == parent_comm_id) {
00023         std::random_device r;
00024         random = r();
00025     }
00026
00027     assert(MPI_SUCCESS == MPI_Bcast(
00028         &random, 1, get_mpi<decltype(random)>::type(), 0, parent_comm));
00029
00030     tmp_path += std::to_string(random);
00031
00032     return tmp_path;
00033 }
00034
00035
00036 int mpi_shared_tmp_dir_workgroup::identify_workgroup_head()
00037 {
00038     namespace fs = std::filesystem;
00039
00040     // create random tmp path
00041     fs::create_directories(tmp_path);
00042
00043     // ensure its empty
00044     assert(fs::is_empty(tmp_path));
00045     assert(MPI_SUCCESS == MPI_Barrier(parent_comm));
00046
00047     // create dir with id inside
00048     assert(fs::create_directory(
00049         tmp_path + "/" + std::to_string(parent_comm_id)));
00050     assert(MPI_SUCCESS == MPI_Barrier(parent_comm));
00051
00052     // find minimum id in local tmp dir
00053     int min_id = std::numeric_limits<int>::max();
00054     for (const auto & entry : fs::directory_iterator(tmp_path))
00055     {
00056         min_id = std::min(
00057             min_id, from_string<int>(entry.path().filename()));
00058     }
00059
00060     // sanity checks
00061     assert(0 <= min_id);
00062     assert(min_id < parent_comm_size);
00063
00064     // remove tmp directories if all are finished
00065     // note: /tmp/mpi_split_by_shared_tmp/ will persist
00066     MPI_Barrier(parent_comm);
00067     if (parent_comm_id == min_id) {
00068         std::filesystem::remove_all(tmp_path);
```

```
00069     }
00070
00071     return min_id;
00072 }
00073
00074
00075 bool mpi_shared_tmp_dir_workgroup::is_workgroup_head() const {
00076     return parent_comm_id == parent_comm_workgroup_head_id;
00077 }
00078
00079
00080 mpi_shared_tmp_dir_workgroup::mpi_shared_tmp_dir_workgroup(
00081     MPI_Comm parent_comm,
00082     std::string app_name
00083 ) :
00084     parent_comm(parent_comm)
00085 {
00086     parent_comm_id = mpi_get_comm_rank(parent_comm);
00087     parent_comm_size = mpi_get_comm_size(parent_comm);
00088
00089     if (1 == parent_comm_size)
00090     { // shortcut in case of a single process
00091         assert(0 == parent_comm_id);
00092         parent_comm_workgroup_head_id = 0;
00093         workgroup_comm = parent_comm;
00094         workgroup_comm_id = 0;
00095         workgroup_comm_size = 1;
00096         mpi_comm_leaders = parent_comm;
00097         workgroup_count = 1;
00098         workgroup_id = 0;
00099         return;
00100     }
00101
00102     tmp_path = build_tmp_dir_path(app_name);
00103     parent_comm_workgroup_head_id = identify_workgroup_head();
00104
00105     assert(MPI_SUCCESS == MPI_Comm_split(
00106         parent_comm,
00107         parent_comm_workgroup_head_id,
00108         parent_comm_id,
00109         &workgroup_comm));
00110
00111     workgroup_comm_id = mpi_get_comm_rank(workgroup_comm);
00112     workgroup_comm_size = mpi_get_comm_size(workgroup_comm);
00113
00114     const bool is_head = is_workgroup_head();
00115     assert(MPI_SUCCESS == MPI_Comm_split(
00116         parent_comm,
00117         is_head ? 0 : 1,
00118         parent_comm_id,
00119         &mpi_comm_leaders));
00120
00121     if (!is_head) {
00122         assert(MPI_SUCCESS == MPI_Comm_disconnect(&mpi_comm_leaders));
00123         mpi_comm_leaders = MPI_COMM_NULL;
00124     }
00125     assert(MPI_SUCCESS == MPI_Barrier(parent_comm));
00126
00127     int bcast[2];
00128     if (is_head) {
00129         bcast[0] = mpi_get_comm_size(mpi_comm_leaders);
00130         bcast[1] = mpi_get_comm_rank(mpi_comm_leaders);
00131     }
00132
00133     assert(MPI_SUCCESS == MPI_Bcast(
00134         &bcast, 2, get_mpi<int>::type(), 0, workgroup_comm));
00135
00136     workgroup_count = bcast[0];
00137     workgroup_id = bcast[1];
00138
00139     if (0 == parent_comm_id) { assert(is_head); }
00140 }
00141
00142
00143 mpi_shared_tmp_dir_workgroup::~mpi_shared_tmp_dir_workgroup()
00144 {
00145     int was_finalized = 0;
00146     assert(MPI_SUCCESS == MPI_Finalized( &was_finalized ));
00147     if (!was_finalized) {
00148         assert(MPI_SUCCESS == MPI_Barrier(parent_comm));
00149
00150         if (is_workgroup_head()) {
00151             assert(MPI_SUCCESS ==
00152                 MPI_Comm_disconnect(&mpi_comm_leaders));
00153             mpi_comm_leaders = MPI_COMM_NULL;
00154         }
00155
```

```
00156         assert(MPI_SUCCESS == MPI_Barrier(parent_comm));
00157         assert(MPI_SUCCESS == MPI_Comm_disconnect(&workgroup_comm));
00158         workgroup_comm = MPI_COMM_NULL;
00159     }
00160 }
00161
```

7.35 mpi_shared_tmp_workgroup.hpp

```
00001 #ifndef MPI_SHARED_WORKGROUP_HPP
00002 #define MPI_SHARED_WORKGROUP_HPP
00003
00004 #include <string>
00005 #include <mpi.h>
00006
00012 class mpi_shared_tmp_dir_workgroup
00013 {
00014     public:
00016         MPI_Comm parent_comm;
00017
00019         int parent_comm_id;
00020
00022         int parent_comm_size;
00023
00025         int parent_comm_workgroup_head_id;
00026
00027
00029         MPI_Comm workgroup_comm;
00030
00032         int workgroup_comm_id;
00033
00035         int workgroup_comm_size;
00036
00037
00039         MPI_Comm mpi_comm_leaders;
00040
00042         int workgroup_count;
00043
00045         int workgroup_id;
00046
00047     private:
00049         std::string tmp_path;
00050
00052         std::string build_tmp_dir_path(std::string app_name);
00053
00055         int identify_workgroup_head();
00056
00057     public:
00059         bool is_workgroup_head() const;
00060
00062         mpi_shared_tmp_dir_workgroup(
00063             MPI_Comm parent_comm = MPI_COMM_WORLD,
00064             std::string app_name = "main");
00065
00067         ~mpi_shared_tmp_dir_workgroup();
00068 };
00069
00070 #endif
```

7.36 mpi_types.hpp

```
00001 #ifndef MPI_TYPES_HPP
00002 #define MPI_TYPES_HPP
00003
00004 #include <mpi.h>
00005
00007 template<typename t>
00008 struct get_mpi {
00009 };
00010
00011 // exclude template specialications
00013
00014 // with definitions like the following all was working fine in the past... :/
00015 // template<>
00016 // struct get_mpi<std::byte> {
00017     // static constexpr auto type = MPI_BYTE;
00018 // };
00019
00020 // but now hacks are required to make it work again, see:
```

```
00021 // https://github.com/open-mpi/ompi/issues/10017
00022 // therefore it was switched to runtime evaluation...
00023
00024
00025 template<>
00026 struct get_mpi<std::byte> {
00027     static auto type() { return MPI_BYTE; }
00028 };
00029
00030 template<>
00031 struct get_mpi<char> {
00032     static auto type() { return MPI_SIGNED_CHAR; }
00033 };
00034
00035 template<>
00036 struct get_mpi<unsigned char> {
00037     static auto type() { return MPI_UNSIGNED_CHAR; }
00038 };
00039
00040 template<>
00041 struct get_mpi<short> {
00042     static auto type() { return MPI_SHORT; }
00043 };
00044
00045 template<>
00046 struct get_mpi<unsigned short> {
00047     static auto type() { return MPI_UNSIGNED_SHORT; }
00048 };
00049
00050 template<>
00051 struct get_mpi<int> {
00052     static auto type() { return MPI_INT; }
00053 };
00054
00055 template<>
00056 struct get_mpi<unsigned int> {
00057     static auto type() { return MPI_UNSIGNED; }
00058 };
00059
00060 template<>
00061 struct get_mpi<long int> {
00062     static auto type() { return MPI_LONG; }
00063 };
00064
00065 template<>
00066 struct get_mpi<unsigned long int> {
00067     static auto type() { return MPI_UNSIGNED_LONG; }
00068 };
00069
00070 template<>
00071 struct get_mpi<long long int> {
00072     static auto type() { return MPI_LONG_LONG_INT; }
00073 };
00074
00075 template<>
00076 struct get_mpi<float> {
00077     static auto type() { return MPI_FLOAT; }
00078 };
00079
00080 template<>
00081 struct get_mpi<double> {
00082     static auto type() { return MPI_DOUBLE; }
00083 };
00084
00085 template<>
00086 struct get_mpi<long double> {
00087     static auto type() { return MPI_LONG_DOUBLE; }
00088 };
00089
00090 // old code:
00091
00092
00093 // template<>
00094 // struct get_mpi<std::byte> {
00095     // static constexpr auto type = MPI_BYTE;
00096 // };
00097
00098 // template<>
00099 // struct get_mpi<char> {
00100     // static constexpr auto type = MPI_SIGNED_CHAR;
00101 // };
00102
00103 // template<>
00104 // struct get_mpi<unsigned char> {
00105     // static constexpr auto type = MPI_UNSIGNED_CHAR;
00106 // };
00107
```

```
00108 // template<>
00109 // struct get_mpi<short> {
00110     // static constexpr auto type = MPI_SHORT;
00111 // };
00112
00113 // template<>
00114 // struct get_mpi<unsigned short> {
00115     // static constexpr auto type = MPI_UNSIGNED_SHORT;
00116 // };
00117
00118 // template<>
00119 // struct get_mpi<int> {
00120     // static constexpr auto type = MPI_INT;
00121 // };
00122
00123 // template<>
00124 // struct get_mpi<unsigned int> {
00125     // static constexpr auto type = MPI_UNSIGNED;
00126 // };
00127
00128 // template<>
00129 // struct get_mpi<long int> {
00130     // static constexpr auto type = MPI_LONG;
00131 // };
00132
00133 // template<>
00134 // struct get_mpi<unsigned long int> {
00135     // static constexpr auto type = MPI_UNSIGNED_LONG;
00136 // };
00137
00138 // template<>
00139 // struct get_mpi<long long int> {
00140     // static constexpr auto type = MPI_LONG_LONG_INT;
00141 // };
00142
00143 // template<>
00144 // struct get_mpi<float> {
00145     // static constexpr auto type = MPI_FLOAT;
00146 // };
00147
00148 // template<>
00149 // struct get_mpi<double> {
00150     // static constexpr auto type = MPI_DOUBLE;
00151 // };
00152
00153 // template<>
00154 // struct get_mpi<long double> {
00155     // static constexpr auto type = MPI_LONG_DOUBLE;
00156 // };
00157
00159
00160 #endif
```

7.37 mpi_util.cpp

```
00001
00002 #include "mpi_util.hpp"
00003 #include <cassert>
00004
00005 int mpi_get_comm_rank(const MPI_Comm &comm)
00006 {
00007     int mpi_id = -1;
00008     assert(MPI_SUCCESS == MPI_Comm_rank(comm, &mpi_id));
00009     assert(0 <= mpi_id);
00010     return mpi_id;
00011 }
00012
00013
00014 int mpi_get_comm_size(const MPI_Comm &comm)
00015 {
00016     int wg_size = -1;
00017     assert(MPI_SUCCESS == MPI_Comm_size(comm, &wg_size));
00018     assert(0 <= wg_size);
00019     return wg_size;
00020 }
00021
00022 bool is_message_available(
00023     const MPI_Comm &comm, MPI_Status *status, int tag)
00024 {
00025     int received_flag = 0;
00026     assert(MPI_SUCCESS ==
00027         MPI_Iprobe(MPI_ANY_SOURCE, tag, comm, &received_flag, status));
00028     return 0 != received_flag;
```

```
00029 }
00030
00031 void wait_for_message(const MPI_Comm &comm, MPI_Status *status, int tag)
00032 {
00033     assert(MPI_SUCCESS == MPI_Probe(MPI_ANY_SOURCE, tag, comm, status));
00034 }
00035
00036 int get_message_size(MPI_Status *status)
00037 {
00038     int length = -1;
00039     assert(MPI_SUCCESS == MPI_Get_count(status, MPI_BYTE, &length));
00040     assert(0 <= length);
00041     return length;
00042 }
00043
00044 void get_message(
00045     const MPI_Comm &comm,
00046     MPI_Status *status,
00047     std::vector<uint8_t> &content)
00048 {
00049     int length = get_message_size(status);
00050     content.resize(length);
00051
00052     assert(MPI_SUCCESS == MPI_Recv(
00053         content.data(),
00054         length,
00055         MPI_BYTE,
00056         status->MPI_SOURCE,
00057         status->MPI_TAG,
00058         comm,
00059         status));
00060
00061     // assert(get_message_size(status) == length);
00062 }
00063
00064 void send_message(
00065     const MPI_Comm &comm,
00066     int dest,
00067     int tag,
00068     const std::vector<uint8_t> &content)
00069 {
00070     assert(MPI_SUCCESS == MPI_Send(
00071         content.data(), content.size(), MPI_BYTE, dest, tag, comm));
00072 }
00073
00074 void ping(const MPI_Comm &comm, int dest, int tag)
00075 {
00076     assert(MPI_SUCCESS ==
00077         MPI_Send(nullptr, 0, MPI_BYTE, dest, tag, comm));
00078 }
00079
00080 void accept_ping(const MPI_Comm &comm, MPI_Status *status)
00081 {
00082     assert(MPI_SUCCESS == MPI_Recv(
00083         nullptr,
00084         0,
00085         MPI_BYTE,
00086         status->MPI_SOURCE,
00087         status->MPI_TAG,
00088         comm,
00089         status));
00090 }
00091
```

7.38 mpi_util.hpp

```
00001 #ifndef MPI_UTIL_HPP
00002 #define MPI_UTIL_HPP
00003
00004 #include <mpi.h>
00005 #include <vector>
00006
00008 int mpi_get_comm_rank(const MPI_Comm &comm);
00009
00011 int mpi_get_comm_size(const MPI_Comm &comm);
00012
00014 bool is_message_available(
00015     const MPI_Comm &comm,
00016     MPI_Status *status,
00017     int tag = MPI_ANY_TAG);
00018
00020 void wait_for_message(
00021     const MPI_Comm &comm,
```

```
00022     MPI_Status *status,
00023     int tag = MPI_ANY_TAG);
00024
00026 int get_message_size(MPI_Status *status);
00027
00029 void get_message(
00030     const MPI_Comm &comm,
00031     MPI_Status *status,
00032     std::vector<uint8_t> &content);
00033
00035 void send_message(
00036     const MPI_Comm &comm,
00037     int dest,
00038     int tag,
00039     const std::vector<uint8_t> &content);
00040
00042 void ping(const MPI_Comm &comm, int dest, int tag);
00043
00045 void accept_ping(const MPI_Comm &comm, MPI_Status *status);
00046
00047 #endif
```

7.39 seed.hpp

```
00001 #ifndef SEED_HPP
00002 #define SEED_HPP
00003
00004
00005 #include <iostream>
00006 #include <random>
00007 #include <array>
00008 #include <iterator>
00009 #include <cassert>
00010 #include <fstream>
00011
00012
00014 template<typename base_type>
00015 struct seed_type_generator
00016 {
00018     std::uniform_int_distribution<base_type> dist;
00019
00021     seed_type_generator() :
00022         dist(
00023             std::numeric_limits<base_type>::min(),
00024             std::numeric_limits<base_type>::max()
00025         )
00026     {}
00027 };
00028
00029
00031 template<std::size_t size, typename base_type>
00032 class seed_type
00033 {
00034     private:
00036         template<typename rdev_t>
00037         void init(
00038             rdev_t &rng_dev,
00039             seed_type_generator<base_type> *sgen_ptr = nullptr)
00040         {
00041             seed_type_generator<base_type> sgen;
00042             sgen_ptr = sgen_ptr ? sgen_ptr : &sgen;
00043             for (std::size_t i = 0; i < seed_data.size(); i++) {
00044                 seed_data[i] = sgen.dist(rng_dev);
00045             }
00046         }
00047
00048     public:
00050         using base_t = base_type;
00051
00053         std::array<base_type, size> seed_data;
00054
00056         inline auto get_seed_seq() {
00057             return std::seed_seq(
00058                 std::begin(seed_data), std::end(seed_data));
00059         }
00060
00062         seed_type(bool generate_random_seed = false) : seed_data()
00063         {
00064             if (generate_random_seed) {
00065                 std::random_device rdev;
00066                 init(rdev);
00067             }
00068         }
```

```
00069
00071         seed_type(std::string seed_str) : seed_data() {
00072             std::stringstream sstr(seed_str);
00073             sstr » *this;
00074         }
00075
00077         template<typename rdev_t> requires // hack to check for RNG:
00078             (!std::is_same<void, typename rdev_t::result_type>::value)
00079         seed_type(
00080             rdev_t &rng_dev,
00081             seed_type_generator<base_type> *sgen = nullptr
00082         ) :
00083             seed_data()
00084         {
00085             init(rng_dev, sgen);
00086         }
00087
00089         base_type short_rep() {
00090             return seed_data[0];
00091         }
00092
00094         bool operator==(const seed_type<size, base_type> &rhs) const {
00095             return seed_data == rhs.seed_data;
00096         }
00097
00099         bool operator!=(const seed_type<size, base_type> &rhs) const {
00100             return seed_data != rhs.seed_data;
00101         }
00102
00104         friend std::ostream &operator«(
00105             std::ostream &os, const seed_type &st)
00106         {
00107             std::copy(
00108                 st.seed_data.begin(),
00109                 st.seed_data.end(),
00110                 std::ostream_iterator<base_type>(os, " "));
00111             return os;
00112         }
00113
00115         friend std::istream &operator»(std::istream &is, seed_type &st)
00116         {
00117             for (std::size_t i = 0; i < st.seed_data.size(); i++) {
00118                 base_type v;
00119                 is » v;
00120                 st.seed_data[i] = v;
00121             }
00122
00123             return is;
00124         }
00125 };
00126
00127
00129 template <typename S, std::size_t size, typename base_type>
00130 void serialize(S& s, seed_type<size, base_type> &seed)
00131 {
00132     std::size_t sz = size;
00133     std::size_t bts = sizeof(base_type);
00134
00135     static_assert(8 == sizeof(std::size_t));
00136
00137     s.value8b(sz);
00138     s.value8b(bts);
00139     s.container(seed.seed_data, size);
00140
00141     assert(size == sz);
00142     assert(sizeof(base_type) == bts);
00143 }
00144
00145
00147 template<typename seed_t>
00148 auto get_persistent_global_seed(std::string seed_filename)
00149 {
00150     std::fstream seed_file(
00151         seed_filename, std::ios::in | std::ios::out | std::ios::app);
00152     assert(seed_file.is_open());
00153
00154     seed_file.seekp(0, seed_file.end);
00155     const auto fsize = seed_file.tellp();
00156
00157     // generate a random seed
00158     seed_t global_seed = seed_t(true);
00159
00160     if (0 == fsize) {
00161         // and save it if none was available
00162         std::cout « "c writing seed" « std::endl;
00163         seed_file « global_seed;
00164         seed_file.flush();
```

```
00165         assert(seed_file.good());
00166     } else {
00167         // or load if one had been saved before
00168         seed_file.seekg(0, seed_file.beg);
00169         std::cout « "c reading seed" « std::endl;
00170         seed_file » global_seed;
00171         // assert(seed_file.eof());
00172         assert(seed_file.tellg() + 1 == fsize);
00173     }
00174
00175     assert(!seed_file.bad());
00176     seed_file.close();
00177
00178     return global_seed;
00179 }
00180
00181
00182 #endif
```

7.40 shared_tmpfile.cpp

```
00001
00002 #include "shared_tmpfile.hpp"
00003 #include "uuid.hpp"
00004
00005 #include <filesystem>
00006 #include <cassert>
00007 #include <iostream>
00008
00009 shared_tmpfile::shared_tmpfile() : id(0), fname(), fptr(nullptr) {}
00010
00011 shared_tmpfile::shared_tmpfile(std::string purpose) :
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 }
00036
00037 void swap(shared_tmpfile &a, shared_tmpfile &b)
00038 {
00039     std::swap(a.id, b.id);
00040     std::swap(a.fname, b.fname);
00041     std::swap(a.fptr, b.fptr);
00042 }
00043
00044 shared_tmpfile::shared_tmpfile(shared_tmpfile &&other) noexcept :
00045     shared_tmpfile()
00046 {
00047     swap(*this, other);
00048 }
00049
00050 shared_tmpfile &shared_tmpfile::operator=(shared_tmpfile other)
00051 {
00052     swap(*this, other);
00053     return *this;
00054 }
00055
00056 void shared_tmpfile::remove()
00057 {
00058     if (0 < id) {
00059         // std::cout « "removing stmpfile " « id
00060             // « ": " « fname « std::endl;
00061         uuid<shared_tmpfile>::free(id-1);
00062         id = 0;
```

```
00063     }
00064
00065     if (nullptr != fptr) {
00066         assert(0 == fclose(fptr));
00067         fptr = nullptr;
00068         assert(std::filesystem::remove(fname));
00069     }
00070
00071     fname = "";
00072 }
00073
00074 shared_tmpfile::~shared_tmpfile() {
00075     remove();
00076 }
00077
```

7.41 shared_tmpfile.hpp

```
00001 #ifndef SHARED_TMPFILE
00002 #define SHARED_TMPFILE
00003
00004 #include <cstddef>
00005 #include <string>
00006 #include <stdio.h>
00007
00013 class shared_tmpfile
00014 {
00015     private:
00017         std::size_t id;
00018
00019     public:
00021         std::string fname;
00022
00024         FILE *fptr;
00025
00027         shared_tmpfile();
00028
00034         shared_tmpfile(std::string purpose);
00035
00037         shared_tmpfile(const shared_tmpfile &other) = delete;
00038
00039         // copy & swap idiom
00040
00042         friend void swap(shared_tmpfile &a, shared_tmpfile &b);
00043
00045         shared_tmpfile(shared_tmpfile &&other) noexcept;
00046
00048         shared_tmpfile &operator=(shared_tmpfile other);
00049
00051         void remove();
00052
00054         ~shared_tmpfile();
00055 };
00056
00057 #endif
```

7.42 statistics.hpp

```
00001 #ifndef STATISTICS_HPP
00002 #define STATISTICS_HPP
00003
00004 #include <cmath>
00005
00016 template<typename prec_t>
00017 class basic_statistical_metrics
00018 {
00019     public:
00021         std::size_t n;
00022
00024         prec_t sum_xi;
00025
00027         prec_t sum_xi_sq;
00028
00030         prec_t varianz;
00031
00033         prec_t stddev;
00034
00036         prec_t average;
00037
```

```
00039         prec_t min;
00040
00042         prec_t max;
00043
00045         std::size_t not_finite;
00046
00048         basic_statistical_metrics() :
00049             n(0),
00050             sum_xi(0),
00051             sum_xi_sq(0),
00052             varianz(0),
00053             stddev(0),
00054             average(0),
00055             min(std::numeric_limits<prec_t>::max()),
00056             max(std::numeric_limits<prec_t>::min()),
00057             not_finite(0) {}
00058
00060         std::size_t counted() {
00061             return n + not_finite;
00062         }
00063
00065         template <typename it_t>
00066         void operator ()(it_t begin, it_t end)
00067         {
00068             for (auto i = begin; i != end; i++)
00069             {
00070                 if (std::isfinite(*i)) {
00071                     min = std::min(min, *i);
00072                     max = std::max(max, *i);
00073                     n++;
00074                     prec_t x = *i;
00075                     sum_xi += x;
00076                     sum_xi_sq += x*x;
00077                 } else {
00078                     not_finite++;
00079                 }
00080             }
00081
00082             average = sum_xi / n;
00083             varianz = (sum_xi_sq - sum_xi*sum_xi / n) / (n - 1);
00084             stddev = sqrt(varianz);
00085         }
00086 };
00087
00088 // prec_t mean = std::accumulate(list.begin(), list.end(), (prec_t) 0.0) / list.size();
00089
00090 #endif
```

7.43 util.cpp

```
00001
00002 #include <fcntl.h>
00003 #include <sys/stat.h>
00004 #include <unistd.h>
00005
00006 #include "util.hpp"
00007
00008 std::ptrdiff_t slurp_file(std::string &content_str, const std::string filename) {
00009     auto fd = open(filename.c_str(), O_RDONLY);
00010     if (0 > fd) {
00011         std::string errmsg = std::string("can't get file descriptor for ") + filename;
00012         perror(errmsg.c_str());
00013         return -1;
00014     }
00015
00016     struct stat stat_buf;
00017     auto r = fstat(fd, &stat_buf);
00018     if (0 != r) {
00019         std::string errmsg = std::string("can't get file size with fstat on ") + filename;
00020         perror(errmsg.c_str());
00021         return -1;
00022     }
00023
00024     content_str.resize(stat_buf.st_size);
00025     if (stat_buf.st_size != read(fd, content_str.data(), stat_buf.st_size)) {
00026         std::string errmsg = std::string("error reading ") + std::to_string(stat_buf.st_size) +
      std::string(" bytes from ") + filename;
00027         perror(errmsg.c_str());
00028         return -1;
00029     }
00030
00031     close(fd);
00032     return stat_buf.st_size;
```

```
00033 }
00034
00035
00036 void print_vector(std::string msg, const std::vector<uint8_t> &v)
00037 {
00038     std::cout « msg;
00039     for (uint8_t i : v) {
00040         std::cout « " " « (int) i;
00041     }
00042     std::cout « std::endl;
00043 }
00044
00045
00046 uint8_t crc8(const uint8_t *data, const uint32_t length, uint8_t crc)
00047 {
00048     constexpr const uint16_t polynom = 0b100110001;
00049     constexpr const uint8_t p_mask = polynom & 0xFF;
00050
00051     for (uint32_t i = 0; i < length; i++) {
00052         const uint8_t d = data[i];
00053         for (int8_t j = 7; j >= 0; j--) {
00054             uint8_t e = ((crc » 7)^(d » j)) & 1;
00055             crc = (crc « 1) ^ p_mask*e;
00056         }
00057     }
00058
00059     return crc;
00060 }
00061
00062
00064 auto vector_to_string(const std::vector<bool> &v)
00065 {
00066     std::stringstream isstr;
00067     isstr « v.size();
00068     for (auto b : v) {
00069         isstr « " " « b;
00070     }
00071     assert(!isstr.fail());
00072     return isstr.str();
00073 }
00074
00075
00077 auto vector_from_string(auto content)
00078 {
00079     std::stringstream osstr(content);
00080     std::size_t size;
00081     osstr » size;
00082
00083     std::vector<bool> v(size);
00084
00085     for (std::size_t i = 0; i < size; i++) {
00086         bool value;
00087         osstr » value;
00088         v[i] = value;
00089     }
00090
00091     assert(!osstr.fail());
00092     assert(osstr.eof());
00093     return v;
00094 }
00095
00096
00097 bool fd_is_valid(int fd)
00098 {
00099     return fcntl(fd, F_GETFD) != -1 || errno != EBADF;
00100 }
00101
```

7.44 util.hpp

```
00001 #ifndef UTIL_HPP
00002 #define UTIL_HPP
00003
00004
00005 #include <string>
00006 #include <cstddef>
00007 #include <sstream>
00008 #include <tuple>
00009 #include <iostream>
00010 #include <vector>
00011 #include <cassert>
00012
00013 #include <unistd.h>
```

```
00014 #include <fcntl.h>
00015
00016
00018 template<typename t, typename... args>
00019 void ignore(const t&, args...) { }
00020
00021
00023 std::ptrdiff_t slurp_file(
00024     std::string &content_str, const std::string filename);
00025
00026
00028 template<typename t>
00029 t from_string(std::string s) {
00030     t value;
00031     std::istringstream iss(s);
00032     iss » value;
00033
00034     if (!iss) {
00035         std::string errmsg = "can't parse '" + s + "'";
00036         throw std::runtime_error(errmsg);
00037     }
00038
00039     return value;
00040 }
00041
00042
00044 void print_vector(std::string msg, const std::vector<uint8_t> &v);
00045
00046
00053 template <typename head_t, typename... tail_t>
00054 auto tail(const std::tuple<head_t, tail_t...> &tuple) {
00055     return apply([](auto &head, auto &... tail) { ignore(head); return std::make_tuple(tail...); },
      tuple);
00056 }
00057
00058
00059 bool perror_(auto msg) {
00060     perror(msg);
00061     return false;
00062 }
00063
00064
00076 uint8_t crc8(const uint8_t *data, const uint32_t length, uint8_t crc = 0);
00077
00078
00080 auto vector_to_string(const std::vector<bool> &v);
00081
00082
00084 auto vector_from_string(auto content);
00085
00091 bool fd_is_valid(int fd);
00092
00093
00094 /*
00095 // https://stackoverflow.com/questions/41301536/get-function-return-type-in-template
00096 template<typename R, typename... A>
00097 R get_return_type(R(*)(A...));
00098
00099 template<typename C, typename R, typename... A>
00100 R get_return_type(R(C::*)(A...));
00101 */
00102
00103
00104 #endif
```

7.45 uuid.hpp

```
00001 #ifndef UUID_HPP
00002 #define UUID_HPP
00003
00004 #include <cstddef>
00005 #include <vector>
00006 #include <cassert>
00007
00014 template<class crtp, typename uuid_type = std::size_t, uuid_type reserved_ids = 1>
00015 class uuid
00016 {
00017     private:
00019         static uuid_type new_uuid;
00020
00022         static std::vector<uuid_type>
00023             unused;
00024
```

```
00025     public:
00026         uuid() = delete;
00027 
00029         static uuid_type get()
00030         {
00031             uuid_type id;
00032 
00033             if (unused.empty()) {
00034                 id = new_uuid;
00035                 new_uuid++;
00036                 assert(0 != new_uuid + reserved_ids);
00037             } else {
00038                 id = unused.back();
00039                 unused.pop_back();
00040             }
00041 
00042             // std::cout << "get " << id << std::endl;
00043 
00044             return id;
00045         }
00046 
00048         static void free(const uuid_type id) {
00049             // std::cout << "free " << id << std::endl;
00050             unused.push_back(id);
00051         }
00052 };
00053 
00054 // static parameter initialization
00055 
00056 template<class crtp, typename uuid_type, uuid_type reserved_ids>
00057 uuid_type uuid<crtp, uuid_type, reserved_ids>::new_uuid = 0;
00058 
00059 template<class crtp, typename uuid_type, uuid_type reserved_ids>
00060 std::vector<uuid_type> uuid<crtp, uuid_type, reserved_ids>::unused;
00061 
00062 #endif
```

Index