Bachelorarbeit

Extending the PERMORY-Algorithm: Parallelization and Handling of Quantitative Phenotypes

zur Erlangung des akademischen Grades

Bachelor of Science Informatik

vorgelegt dem
Fachbereich Mathematik, Naturwissenschaften und Informatik der
Technischen Hochschule Mittelhessen

Volker Steiß
2011–08–31

Referent: Prof. Dr. Thomas Karl Letschert
Korreferent: Prof. Dr. Helmut Schäfer

Versicherung der Selbständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Marburg, 31. August 2011

Volker Steiß

Danksagung

An dieser Stelle bedanke ich mich bei Herrn Prof. Dr. Thomas Letschert, bei Herrn Prof. Dr. Helmut Schäfer und bei Herrn Dipl. Inf. Roman Pahl, die mir die Teilnahme am Projekt PERMORY ermöglicht und mich während meiner Bachelorarbeit und der vorangegangenen Projektphase betreut haben.

Großer Dank gilt auch XXXXXXXXXXX für das Korrekturlesen dieser Arbeit.

Weiterhin danke ich Dr. Hossain und Prof. Chakraborty für die Überlassung der Sepsis-GWAS-Daten.

Contents

I. Introduction **1**

1. Genome-wide Association Studies **3**

1.1. Permutation Test 4

2. PERMORY **7**

2.1. Permutation Matrix 7

2.2. Boosters 9

2.3. Build System 10

II. Extensions **11**

3. Parallelization **13**

3.1. Environment 13

3.2. Message Passing Interface 14

3.2.1. Implementation 14

3.2.2. Benchmarks 21

3.2.3. Accuracy 25

3.3. OpenMP 26

3.3.1. Implementation 28

3.3.2. Benchmarks 30

4. Quantitative Phenotypes **33**

4.1. Test statistic 33

4.2. Boosters 34

4.3. Implementation 37

4.3.1. Replacement of the Contingency Tables 38

4.3.2. Program Option 39

4.3.3. Test Statistic 40
4.4. Benchmarks 41

III. Conclusion **43**

5. Results **45**
5.1. Parallelization 45
5.2. Quantitative Phenotypes 46

6. Outlook **47**
6.1. OpenMP 47
6.2. Automatic Detection of Phenotype Type 47
6.3. Multiple Test Statistics 47
6.4. Multiple Traits 48
6.5. Flow Based Programming 48

List of Figures

1.1. Schema of the DNA 4

2.1. UML activity diagram of PERMORY 8

3.1. Schema of the PERMORY-algorithm using MPI 15
3.2. Class diagram excerpt of Analyzer with MPI extension 16
3.3. MPI Benchmark Runtimes 23
3.4. OpenMP benchmark runtimes 31

4.1. Dichotomous phenotypes: Class diagram of PERMORY 37
4.2. Quantitative phenotypes: Class diagram of PERMORY 38

6.1. Data-flow-based schema of the PERMORY-algorithm 49

List of Tables

3.1. Accuracy evaluation of adjusted p-values computed by PERMROY-MPI . 26

4.1. Results of the quantitative phenotypes benchmark 41
4.2. Benchmark of quantitative vs. dichotomous traits 42

Listings

3.1. Determining permutations per process 17
3.2. Merge intermediate results 18
3.3. Configuration of analyzer factory 19
3.4. Modification of `jamroot.jam` for MPI 19
3.5. Empty hook `None` 21
3.6. *Argument hook* implementation 21
3.7. *Argument hook* configuration 22
3.8. *Argument hook* behavior for MPI support. 22
3.9. OpenMP optimization location 1 28
3.10. OpenMP optimization location 2 29
3.11. OpenMP BJam changes 30

4.1. Program option `phenotype` 39
4.2. Parsing of the program option `phenotype` 40

Preface

PERMORY is an LD-exploiting permutation test algorithm for powerful genome-wide association testing developed by Pahl and Schäfer. The efforts of this work are based on their research.

Currently PERMORY is processing data sequentially and can handle binary traits. The numbers of markers examined and individuals participating in a genome-wide association study are growing, hence, the computationally costs for analysis increase. In this work we describe the process of parallelization of the algorithm to speed up the analysis. We also discuss the extension of PERMORY to handle quantitative traits.

The first part gives a brief introduction into genome-wide association studies and PERMORY.
In part II we describe how to parallelize the algorithm with two established methods—MPI and OpenMP. The second extension discussed in that chapter is the handling of quantitative phenotypes.
Finally, in the last chapters we summarize the results of the discussed extensions and give an outlook of future projects.

Part I.

Introduction

This part gives an introduction into PERMORY and genome-wide association studies.

1. Genome-wide Association Studies

The PERMORY-algorithm developed by Pahl and Schäfer[1] is used to analyze *genome-wide association studies* (GWA study, GWAS). Thus, we give a brief introduction into this type of study.

In GWAS genes of individuals are analyzed for differences to discover associations between genes and observable traits (phenotypes). The genes are coded in the *deoxyribonucleic acid* (DNA). We use *genome* as a synonym for DNA.
In figure 1.1 a schematic representation of the DNA units is shown. It is built up of two strands of *nucleotides*, and a nucleotide is the connection of the strand's backbone with a base. Four types of *bases* exist: Adenine (A), thymine (T), cytosine (C) and guanine (G). The strands are arranged in a way that two nucleotides of each strand form a pair, also called *base pair*.
In this work we call the position of a base pair in the genome *locus* or *marker*. Different combinations of nucleotides among individuals at a locus are called *alleles*. The variation of the DNA sequence of individuals in one nucleotide is called *single-nucleotide polymorphism* (SNP, pronounced like 'snip'). Today, millions of SNPs per individual are tested for associations in a GWAS.

Single-nucleotide polymorphisms are genotyped. The *genotype* per SNP is defined by the number of minor alleles. Considering diploid cells, which have two double strands—one maternal and one paternal—per chromosome, there can be two alleles per locus.

In most cases a GWAS is conducted as a case-control study, where the individuals are separated into two groups, e. g., diseased and non-diseased.

A quantitative value, subsuming the information of the difference of individuals in respect of their phenotype, is calculated for each locus using the *test statistic* T. Subsequently, the p-value of a marker j is calculated by applying a statistical hypothesis test, e. g. Pearson's chi-square test. As the whole genome is scanned for associations, the *null*

[1] Pahl and Schäfer 2010.

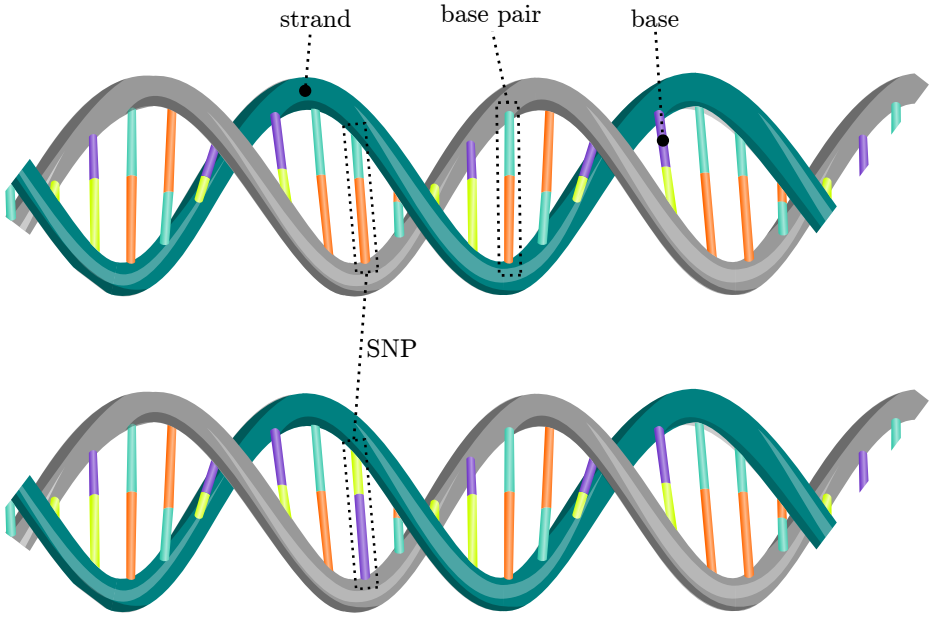

Figure 1.1.: This figure shows a schematic of two DNA segments. Both segments are equal, except in one base pair, where a single-nucleotide polymorphism occurs. The different bases are illustrated with different colors (purple, lime, cyan, orange).

hypothesis H_0 to test in GWAS is: For any locus it holds that it is independent in respect of the genotype.

A random association between phenotypes and genotypes may be observed, which is called *false positive* or *type I error*. Due to this fact, the p-value needs adjustment.

1.1. Permutation Test

In order to control the genome-wide type I error rate, Pahl and Schäfer use *permutation testing* to adjust the p-value. The idea of permutation testing is to simulate the distribution under the null hypothesis by permuting cases and controls several times. This process is called *resampling*. Subsequently, the test statistic T_i for $i = 1, \ldots, N$ of each permutation is calculated, where N is the number of permutations, and T_i of a permutation i is the maximum test statistic of all SNPs in this permutation. Thereafter the p-value is determined by counting how many times the result is greater or equal than

the original result T_{orig} using

$$\sigma_i = \begin{cases} 1 & \text{for } T_i \geq T_{orig} \\ 0 & \text{else} \end{cases} \tag{1.1}$$

and dividing by the number of permutations:

$$p = \frac{\sum_{i=1}^{N} \sigma_i \ + 1}{N + 1}. \tag{1.2}$$

Adding 1 to the numerator and denominator yields in a p-value with $0 < p \leq 1$. The p-value is the probability of the occurrence of a false positive. If it is less or equal the *significance level* α, H_0 is discarded. In most cases, α is chosen to a value exactly to 5 %.

2. PERMORY

The PERMORY-algorithm is written in C++ and uses the Boost library[2]. The name 'PERMORY' is constructed from the words 'permutation' and 'memoization' as it executes permutation tests and uses memoization (described later).

In figure 2.1 on the following page a schematic of PERMORY is shown. The algorithm is working as follows: After the initialization, blocks of permutations of the trait vector are created. For each block PERMORY loops over the list of SNPs and calculates the test statistics with excessive usage of the boosters. After processing the permutations, the p-value is calculated as described in chapter 1.1 using the original test statistic and the permutation's test statistics. At the end the results are stored for further processing.

The handling of the permutations is described in the next section.

2.1. Permutation Matrix

PERMORY uses a matrix containing permutations of the trait vector, called *permutation matrix*. The maximum number of vectors in the permutation matrix is defined by the permutation block size. Instead of looping over the locus data files for each single permutation, PERMORY processes a block of permutations. This reduces file system traffic and results in a higher throughput, but increases memory usage.
The default block size is 10 000 permutations.

The permutations are created using the GNU Scientific Library[3]. It provides interfaces to permutation of arrays and *pseudo-random number generator*s (PRNG). Pahl and Schäfer use the PRNG called Mersenne twister[4]. It is fast and provides a good level of *pseudo-randomness*, having proper statistical characteristics. To alter the pseudo

[2] `http://boost.org/`
[3] `http://www.gnu.org/software/gsl/`
[4] Matsumoto and Nishimura 1998.

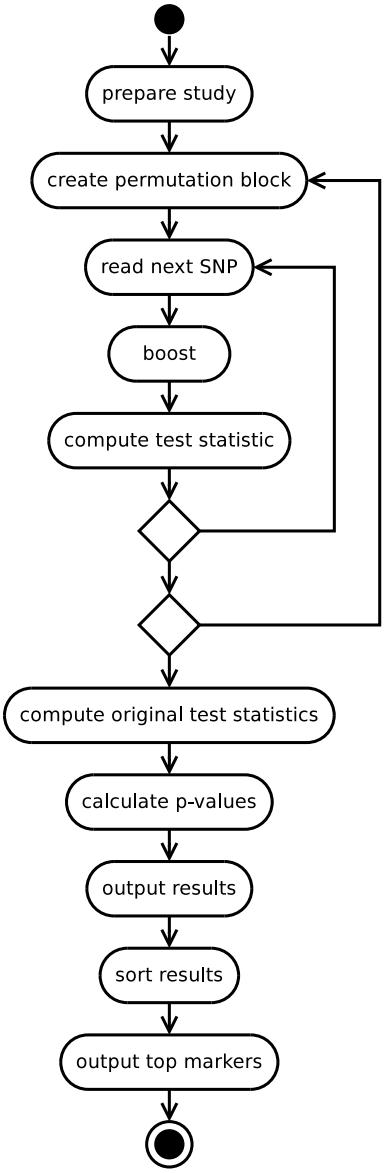

Figure 2.1.: This UML activity diagram shows a schematic of the PERMORY-algorithm.

random number sequence, generated by the PRNG, the function is initialized with a value called *seed*.

2.2. Boosters

Pahl and Schäfer describe the three optimizations *bit arithmetic*, *genotype indexing with transposed permutations* (GIT) and *reconstruction memoization* (REM) used in PERMORY. These optimizations are called *boosters*. Depending on the locus data the optimal booster is selected to build the contingency table.

Bit arithmetic requires boolean representations of genotypes and phenotypes. Thus, the genotype frequency of the affected individuals is counted using boolean operations, which is more efficient in comparison to summing up integers. Pahl and Schäfer call the boolean representation of the traits *dummy code*.

Instead of saving the genotypes in a simple vector containing the genotype values, for each genotype the positions in the locus are stored, except the common genotype. This technique is called *genotype indexing*. In this way, instead of parsing the whole genotype vector, only few positions are inspected. The common genotype is derived from the others. While looping over genotypes and permutation blocks in a nested loop, the genotype frequencies are calculated. Because of low-level optimizations it is faster to iterate the genotypes in the outer-loop and the permutations in the inner-loop than vice versa. Pahl and Schäfer call this technique *transposed permutations*.

Reconstruction memoization utilizes the linkage disequilibrium and uses memoization.
In computer science *memoization* is a method to speedup calculations by remembering previously computed results.[5] For example, it can be used in recursive functions like the factorial function:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \cdot n & \text{if } n > 0 \end{cases} \tag{2.1}$$

where each result for the factorial of a number n is the multiplication of itself with the factorial of $(n-1)$. One can remember—*memoize*—the results of the factorial to speed up calculations on repetitive calls.
PERMORY remembers a tail of previously calculated frequencies because the frequencies at one locus can be calculated from another locus'. Furthermore, if two loci differ barely,

[5] Michie 1968.

this approach allows rapid calculation of the unknown frequencies. Due to the *linkage disequilibrium* (LD) neighbored loci most likely correlate. With rising density of the loci this efficient technique is used more frequently.

2.3. Build System

To compile PERMORY the build system Boost Build[6] (BJam) is employed. In contrast to the build utility 'make' the configuration is less verbose. For example, BJam automatically parses the source code files for dependencies. Configuration files end with `.jam`. BJam supports system-wide configuration files and user configuration files to specify global options. The latter is named `site-config.jam` and located in the system-wide configuration directory of the operating system by default. Users can set global configuration options in a file named `user-config.jam` located in their home directory. When starting BJam to build a program, first it reads the system-wide configuration file and the configuration file of the current user—if existing—before it processes the project's `jamroot.jam` file.

[6] `http://www.boost.org/boost-build2/`

Part II.

Extensions

In this part two extensions of PERMORY are discussed. First we look at the parallelization of the algorithm. Then we change the program to handle quantitative traits.

3. Parallelization

Permutation tests of thousands of individuals with tens of thousands of permutations and millions of markers are computationally intensive and take a lot of time. Furthermore numbers of individuals participating and SNPs examined in GWAS are growing. Thus, we discuss the parallelization of PERMORY to spread the computation process to multiple processors and gain a speedup in runtime.

3.1. Environment

For the parallelization process of PERMORY access to the parallel computing environment *Marburger RechenCluster*[7] (MaRC) is available. MaRC consists of 142 nodes with each two AMD Opteron Dual Core processors. It is managed by the Sun GridEngine Software. The environment provides MPI and OpenMP. More about MPI and OpenMP is said in the next two sections.

The nodes run a 64 bit Debain GNU/Linux 5 with a Linux in version 2.6.26.

AMD's Opteron processors are designed to have each their own dedicated memory. However, processors on the same mainboard can access the memory of other processors, but the latency is higher unlike accessing their own memory.[8] This architecture is called *non-uniform memory access* (NUMA).[9] Since two cores are resided on one processor, they share the processor's memory. Such accesses are called *uniform memory access* (UMA), whereas each core has the same latency when accessing the memory.

[7] `http://www.uni-marburg.de/hrz/infrastruktur/zserv/cluster`
[8] AMD 2006, pp. 20–22.
[9] Rauber and Rünger 2007, p. 28.

3.2. Message Passing Interface

The *Message Passing Interface*[10] (MPI) is a specification for communication of processes in a parallel environment. There are several implementations like MPICH and Open MPI. As we program in C++ we use the C++ library Boost MPI, which acts as an adapter to an MPI implementation. In our environment we use the implementation Open MPI, available on MaRC. But as stated, the implementation is exchangeable because Boost MPI can be used with several implementations.

Open MPI is installed in version 1.2.7rc2 on MaRC.

MPI defines processes as smallest form of execution unit. Processes can run on processors and processors are part of a node. An MPI environment usually consists of many nodes, as those environments are computer clusters. As mentioned on MaRC every node has two processors. It is possible to run multiple processes on one processor, but doing so unlikely results in a speedup as MPI is designed for environments with distributed memory[11]. The speedup of computations in MPI environments results from splitting the computation into multiple independent parts and executing each part on a different processor. Processes can communicate with each other by sending messages. This allows us to split a computation into subcomputations and merge the results.

3.2.1. Implementation

PERMORY was designed with parallelization in mind that means: The location where to parallelize is already identified. The idea is to distribute the permutations and the independent calculation of the test statistic to different processes. This approach is a distributed variation of the data parallel model[12].

To visualize this we memorize the structure of the algorithm described in section 2 and look at figure 3.1. We split the algorithm to execute the steps between *prepare study* and *compute original test statistic* on multiple nodes in parallel.

[10] The Message Passing Interface Forum publishes the official MPI standard and is resided at the website `http://www.mpi-forum.org/`.
[11] Rauber and Rünger 2007, p. 207.
[12] Rauber and Rünger 2007, p. 122 f.

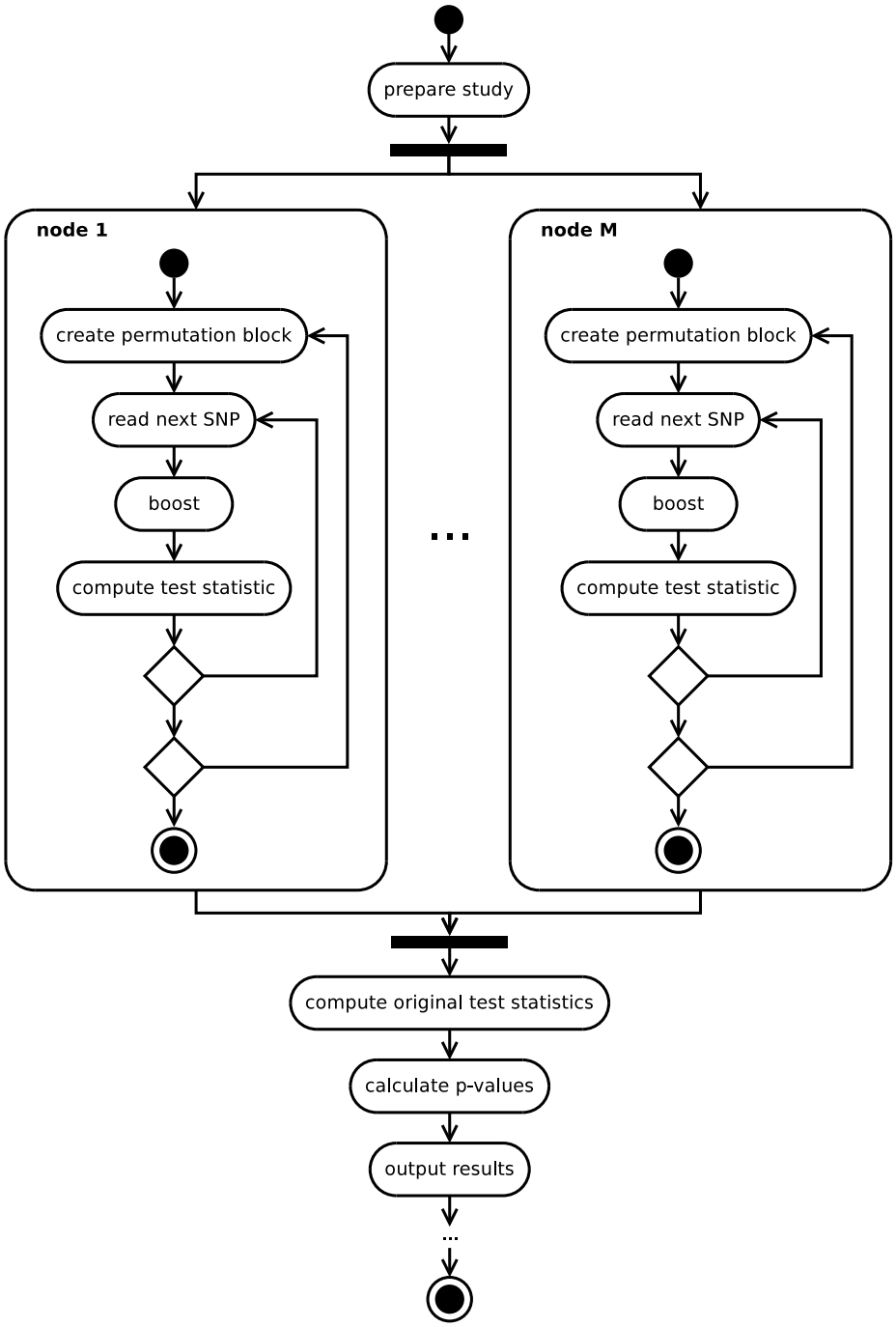

Figure 3.1.: UML activity diagram showing a schematic of the PERMORY-algorithm. This figure is an adjusted version of figure 2.1. Permutations are distributed on different nodes which only compute a subset of the whole permutations.

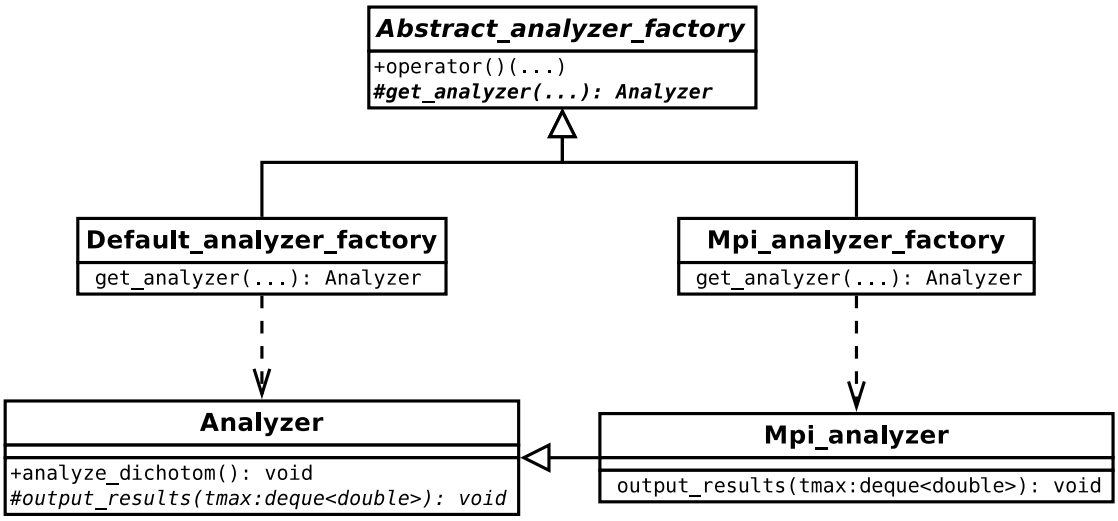

Figure 3.2.: This figure shows an excerpt of the class diagram of `Analyzer`, MPI extension `Mpi_analyzer` and their factory classes.

Quick Implementation

Our first implementation to parallelize PERMORY using MPI was done by adding preprocessor directives directly into the code. It was quickly done and a good way to test if it is feasible to parallelize the algorithm with MPI. But, maintenance of code with several preprocessor directives is a big disadvantage. If we need to change the code, maybe because of fixing a bug, we have to take care of all those preprocessor commands. Also using preprocessor directives in several locations in the code is neither object-oriented nor functional nor procedural. To fix this we need to restructure the existing code.

Refactoring

First we need to introduce a new class which encapsulates the control of the process seen in figure 2.1 on page 8: Looping over the SNPs, calling the permutation of the traits, calculating the test statistics as well as results and finally printing the results. The free function `analyze_dichotom` is controlling this process until now. We move this function to a newly created class called `Analyzer`. As the original function does a lot of things, we split the method to perform the output of the results in a separate method. We call this method `output_results`.

Listing 3.1: Determination of permutations to process in the current process.

```
size_t Mpi_analyzer::nperm_per_process() const
{
  size_t per_process = par_->nperm_total / world_->size();
  return world_->rank() == 0
              ? par_->nperm_total - size_t(per_process *
                 (world_->size() - 1))
              : per_process;
}
```

After refactoring we now have a class which has two methods we can manipulate in an object-oriented way. Their position nearly correspond with fork and join in figure 3.1.

The MPI Analyzer Class

Next we introduce the class `Mpi_analyzer` as subclass of `Analyzer`. The constructor of this class is used to simulate the behavior of fork. And the method `output_results` is overridden to add the join behavior.

To simulate the fork behavior in the constructor the number of permutations are calculated such that each process handles the same amount of permutations. In summary the number of permutations is still the same as if PERMORY was running as a single process. For more details refer to listing 3.1. Also the seed for the PRNG is changed. We use the value given by the user and add the MPI process-ID. Using this method every process computes different permutations.

In `output_results` we collect the sub results of the process in the root process. This method is called *reduce* in MPI. After collecting all sub results all processes but the root process terminate. Now the process behaves like the serial algorithm: It calculates the final result and writes it to file. This is achieved by calling the overridden method of the base class. The overriding method is shown in listing 3.2.

After defining the process we need to make the choice which Analyzer-class shall be used. This decision is put into an *abstract factory*[13] which is responsible for creating instances of a class. We create an abstract base class `Abstract_analyzer_factory`. This class provides an interface to retrieve an object of type `Analyzer`. We use the function operator

[13] Gamma et al. 2007, p. 87 ff.

Listing 3.2: In method `output_results` of class `Mpi_analyzer` the intermediate results of the processes are merged into one result. The MPI method `reduce` is used to collect the data from all nodes on node 0.

```
1  void Mpi_analyzer::output_results(std::deque<double> tmax)
2  {
3    using namespace std;
4    using namespace boost::mpi;
5    using namespace detail;
6
7    sort(tmax.begin(), tmax.end());
8
9    if (world_->rank() == 0) {
10     deque<double> tmax_result;
11     reduce(*world_, tmax, tmax_result, deque_concat<double>(), 0);
12     par_->nperm_total = orig_nperm_total_;  // reset nperm_total
           for correct output calculations
13     Analyzer::output_results(tmax_result);
14   }
15   else {
16     reduce(*world_, tmax, deque_concat<double>(), 0);
17   }
18 }
```

for this and define a virtual method `get_analyzer`. Then we need two specializations of the class: `Default_analyzer_factory` and `Mpi_analyzer_factory`. Those are called *concrete factory*. Both classes implement the protected method `get_analyzer` from the base class. The abstract factory is using this method to request an instance of type `Analyzer`. When calling the method `Default_analyzer_factory` returns an instance of `Analyzer` and `Mpi_analyzer_factory` returns an instance of `Mpi_analyzer`.

If we need an analyzer object we can now refer to the abstract factory. Depending on the passed concrete factory we obtain a single process analyzer or an MPI aware analyzer. The resulting structure can be seen in figure 3.2.

The decision of the factory to use is done with a preprocessor statement in file `config.hpp`. If the compiler flag `USE_MPI` is set, the type `analyzer_factory_t` is set to be of class `Mpi_analyzer_factory`, else the type is `Default_analyzer_factory` (see listing 3.3). The decision at compile-time is necessary because we do not want to force a user to have the MPI libraries installed. We also change the build configuration in file `jamroot.jam` to automatically detect if MPI support should be used. This is achieved by importing the module `mpi` and checking if MPI usage is configured in the BJam system. The changes

Listing 3.3: Selection of analyzer factory to use. This configuration is done in file `detail/config.hpp`.

```
#ifdef USE_MPI
  ...
  namespace gwas { class Mpi_analyzer_factory; }
  typedef gwas::Mpi_analyzer_factory analyzer_factory_t;
#else
  ...
  namespace gwas { class Default_analyzer_factory; }
  typedef gwas::Default_analyzer_factory analyzer_factory_t;
#endif  // USE_MPI
```

Listing 3.4: Modification of `jamroot.jam` for MPI.

```
import mpi ;

flags = ;

if [ mpi.configured ]
{
  echo "Using MPI." ;
  alias /libs : bfs bpop bio bsys zlib gsl gslcblas utf bs bmpi ;
  flags += <define>USE_MPI ;
} else {
  alias /libs : bfs bpop bio bsys zlib gsl gslcblas utf ;
}
```

can be seen in listing 3.4. To configure MPI in BJam we add the following two lines to the system-wide or user configuration of BJam:

```
using mpi ;
using gcc : : mpicxx : ;
```

The first line imports the module `mpi`. The second line chooses the compiler. In this case we are using the GCC[14] C++ environment but instead of using the default compiler executable `g++` we use the MPI wrapper compiler `mpicxx`.

[14] *GNU Compiler Collection* (GCC) is a project of GNU which provides compilers for many languages. The Website of the GCC is located at `http://gcc.gnu.org/`.

MPI Initialization

MPI requires initialization on program start with the arguments passed from the command line. The first approach coming to mind is to use the factory. An issue here is that the default analyzer does not need this information.

So to fulfill this requirement we introduce the concept of *hooks*. A *hook* is a defined point in an application where custom code can be called. This can be a function or a method. In fact, this is related to the behavioral design pattern *Template Method* and used constantly in object-oriented languages.[15]

The hook definition is separated into implementation and configuration. The implementation is done by using a parameterized struct which only implements the function operator. The template parameter is used for creating different specializations for the hook—it is used as a marker. A hook is executed by creating an instance of the struct and then calling the function operator:

```
hook::Some_hook()(some_parameter);
```

To be able to define a default behavior if no specialized hook implementation is used, we create a default hook named `None` with an empty implementation. Its definition is shown in listing 3.5. The default hook implementations are located in file `detail/hooks.hpp`. The hook itself is a specialization of the hook implementation. By specifying the type the implementation is chosen. An example is shown in listing 3.7. The configuration is done in `detail/config.hpp`.

With this knowledge in hand we now create the hook named *argument hook* to handle the initialization of MPI. The hook takes the command line parameters `int argc` and `char **argv`. The default behavior is *do nothing*. Listing 3.6 and 3.7 show the default implementation and configuration of the argument hook. The hook is called in function `main` directly after program start:

```
hook::Argument_hook()(&ac, &av);
```

After creating the argument hook we now can create the MPI specific behavior. First we define the marker class `MPI` for MPI specific hooks. Then we create a new static method `void init_mpi(int *argc, char ***argv)` at class `Mpi_analyzer_factory`. This method takes care of the initialization of MPI. Finally we can create the actual MPI argument

[15] Gamma et al. 2007, p. 325 ff.

Listing 3.5: `None` is defined as empty implementation. It is used as default when a *do nothing* behavior is needed.

```
namespace hook { // Global definitions for hooks
  // NIL type.
  class None { };
}
```

Listing 3.6: This listing shows the default argument hook `Argument_hook_impl`. It uses the None-type to set a default behavior which is *do nothing* in this case.

```
namespace hook { // Static hooks for namespace "Permory"
  //
  // Argument hook is called at the start of the application.
     This hook
  // can be used to alter the arguments passed from commandline.
  //
  template<class T> struct Argument_hook_impl
  {
    public:
      void operator()(int *argc, char ***argv);
  };
  template<> void Argument_hook_impl<None>::operator()
    (int *argc, char ***argv) { }
}
```

hook. The only step necessary is to call the newly created method (see listing 3.8). The MPI argument hook is selected to be executed in `detail/config.hpp` as seen in listing 3.7.

3.2.2. Benchmarks

After implementing MPI support we want to measure the speedup of the parallelized version in comparison to the serial version. According to Pahl and Schäfer we are using real data from the Wellcome Trust Case Control Consortium Phase II (WTCCC2) study[16]. The data consists of 5 667 individuals and 1 115 428 SNPs. Former is divided randomly into 2 834 cases and 2 833 controls. For measurements single and multi process runs are done with 100 000 permutations.

[16]The Wellcome Trust Case Control Consortium 2007.

Listing 3.7: This listing shows the argument hook `Argument_hook` configuration. If PERMORY is build with MPI support it executes the MPI specific behavior.

```
#ifdef USE_MPI
  namespace hook {
    struct Argument_hook : public Argument_hook_impl<MPI> { };
  }
  ...
#else
  namespace hook {
    struct Argument_hook : public Argument_hook_impl<None> { };
  }
  ...
#endif  // USE_MPI
```

Listing 3.8: *Argument hook* behavior for MPI support. The parameters are passed through to the static initialization method of `Mpi_analyzer_factory`.

```
namespace hook {
  class MPI { };

  template<> void Argument_hook_impl<MPI>::operator()(int *argc,
      char ***argv)
  {
    gwas::Mpi_analyzer_factory::init_mpi(argc, argv);
  }
}
```

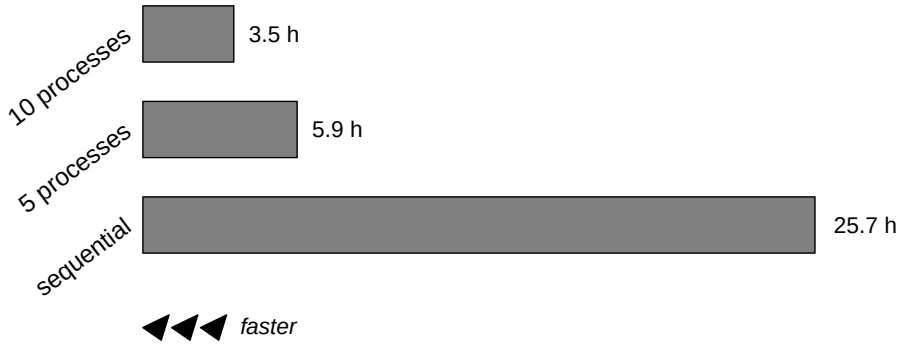

Figure 3.3.: The figure shows the runtimes of the sequential run and the parallel runs using Open MPI.

To calculate the *speedup* S_p we use the following formula[17]:

$$S_p = \frac{T_1}{T_p} \tag{3.1}$$

where T_1 is the time of the serial run and T_p the time of the parallel run using p processes. To compare parallel runs with a different number of processes we also need to calculate the *efficiency*[18]:

$$E_p = \frac{S_p}{p} \tag{3.2}$$

where S_p is the speedup of a parallel run with p processes.

Furthermore, to explore the impact of existing fixed costs we perform two runs using MPI with different numbers of processes: 5 and 10. The *fixed costs* consists of preparation and p-value calculation. They are the part of the algorithm which is not parallelized. We expect a speedup between 4 and 5 using 5 processes, and a speedup between 8 and 10 using 10 process, respectively.

As one can see in figure 3.3, the run with 10 processes has a speedup of 7.4, while the speedup of the run with 5 processes is 4.4. The efficiencies are 74 % and 88 %, respectively. Regarding the fixed costs we conclude: The more permutations per process the less the part of the fixed costs.

[17] Rauber and Rünger 2007, p. 168 ff.
[18] Rauber and Rünger 2007, p. 170.

Fixed Costs Calculation

For the following calculations we need a more precise definition of fixed costs. Therefore we define the formula to calculate the fixed costs T_{fix} as follows:

$$T_{\text{fix}} = T_p - \frac{T_1 - T_p}{p - 1} \tag{3.3}$$

where T_1 is the runtime of the sequential run and T_p the runtime of the parallel run with p processes. We ignore the communication overhead as it is negligible.

Input Data Rate

To find possible bottlenecks we examine the input data rate. The data rate can be estimated by dividing the size of input data by the difference of runtime and the fixed costs. We ignore the data overhead produced by file system and network protocols. Input data size is defined by the number of traits, the number of permutations and the block-wise processing of permutations. We yield the following formula:

$$R_p = C \cdot \frac{N}{K} \cdot \frac{1}{T_p - T_{\text{fix}}} \tag{3.4}$$

where C is the size of the trait data to read, N is the number of permutations, K is the block size, T_{fix} is the fixed costs and T_p is the runtime of the parallel run with p processes.

The SLIDE data format is a line-based ASCII file. Markers are separated by the newline character and individuals by a space character. The genotype of an individual is represented by one character. Using the WTCCC2 study we have 5 667 individuals and 5 667 separation characters per marker. So one marker has a size of 11 334 bytes.

PERMORY can read the data raw or compressed. Having 1 115 428 markers this makes approximately 12 GB of raw data, while compressed the data is approximately 1.2 GB. Therefore, the formula 3.4 leads to a data rate about 48.65 MB/s and 4.87 MB/s, respectively.

We recommend compressing the trait data to reduce the overall network traffic. The Advantage: The number of processes can be increased. Consider a maximum throughput of 500 MB/s to read trait data. If a process has an input data rate of 50 MB/s we can

start 10 processes without effect to the input data rate. Any additional process slows down all processes, because the maximum throughput is exceeded. Contrary, an input rate of 5 MB/s allows 100 processes for the given network limitation.

Network Traffic

The only network traffic related to MPI operations is generated when collecting the results of the test statistics via reduce. Vectors of double are transmitted from the children to the root node. The vector length is equivalent to the number of permutations done by a process. For example, computing one-million permutations on ten nodes leads to the transmission of nine vectors of length 100 000 over the network. In sum this are 900 000 values of type double. Boost MPI uses the Boost Serialization API to communicate C++ data between processes. The size of the data passed is minimal as it uses binary packed archives without headers. In binary representation a double is 64 bits that are 8 bytes. Packed and without headers the data is 900 000 times 8 bytes. This results in less than 7 MB. In a 100 Mbit/s Fast Ethernet network this amount of data is transmitted within one second.

3.2.3. Accuracy

We evaluate the accuracy of the adjusted p-values computed by PERMORY-MPI using a data set consisting of 299 085 SNPs, 60 cases and 77 controls.[19] As reference we use PERMORY in serial mode and the program PLINK[20]. The runs are executed with 10^5 permutations and PERMORY-MPI additionally with 10^6 permutations. The top ten SNPs of the evaluation runs are shown in table 3.1. Comparing the adjusted p-values of PERMORY-MPI with the sequential version and PLINK reveals no significant difference.

[19] This data is a subset of the collective described in Menges et al. 2008 gathered in course of the project 'Aufbau einer Core Facility für die vergleichende statistische Analyse hochdimensionaler molekularer und zellulärer Markerdaten auf der Basis von beschleunigten Algorithmen für Permutationstests, mit Durchführung einer genomweiten Assoziationsanalyse und von Pathwayanalysen zur Trauma-induzierten Sepsis' (Von-Behring-Röntgen-Stiftung No. 57–0048) and was provided by the Institut für Medizinische Mikrobiologie of the Justus-Liebig-Universität Gießen.

[20] Purcell et al. 2007.

		PERMORY		
	PLINK	serial	parallel	
SNP	10^5 perm.	10^5 perm.	10^5 perm.	10^6 perm.
rs10490341	0.1603	0.1580	0.1611	0.1586
rs10490342	0.3258	0.3250	0.3240	0.3254
rs11119170	0.3798	0.3862	0.3866	0.3817
rs4463671	0.5442	0.5471	0.5382	0.5404
rs12029831	0.6233	0.6230	0.6104	0.6169
rs4509197	0.6733	0.6766	0.6606	0.6670
rs4481849	0.7823	0.7806	0.7738	0.7772
rs6670927	0.7823	0.7806	0.7738	0.7772
rs1522987	0.8823	0.8756	0.8739	0.8755
rs1334328	0.9996	0.9991	0.9991	0.9992

Table 3.1.: We evaluate the accuracy of the adjusted p-values of parallelized PERMORY in comparison to the serial version and PLINK. The data set consists of 299 085 SNPs, 60 cases and 77 controls.
The top ten SNPs and their p-value, calculated by the corresponding algorithm, are listed in the table. As one can see the ranking is the same in all four variations.

3.3. OpenMP

Until now we employ MPI to distribute the computation among several nodes. On each node one process is running. That means there are three free cores per node. One method to use an additional core is to populate all nodes with two processes. In newer versions, Open MPI can be advised to do so using the command line option `-npernode`. In our architecture employing one additional process per node can speedup the computation. MPI is used in distributed environments where memory is non-shared[21]. Hence, more than one additional process could result in lower performance as both cores compete for the memory shared by the processor's cores. Furthermore PERMORY has no long blocking periods, i. e., the process computes all permutation blocks successively.

So to take advantage of all cores we can use threads. Threads work together with shared memory but competitive accesses must be synchronized. A simple and elegant way to introduce threads into an existing C/C++ algorithm is using *OpenMP*[22]. OpenMP stands for *Open Multi Processing* and provides a mechanism to parallelize algorithms in *symmetric multiprocessor* (SMP) environments, where multiple processors share the same

[21] Rauber and Rünger 2007, p. 207.
[22] `http://openmp.org/`

memory. Preprocessor instructions are used to control the parallelization, in contrast to MPI where native language constructs are used. The preprocessor instructions are used like annotations[23].

In OpenMP, code which shall be executed in parallel, is marked by *parallel regions*. Those are introduced by the directive `#pragma omp parallel` followed by a block of statements. Inside a parallel region several parallelization constructs can be used.

Here we give only a brief explanation of the *parallel for* loop, which is activated by the preprocessor directive `#pragma omp for`. The parallel for is restricted to a simple integer increment loop. Hence, pointers and other types are not possible. The execution of the loop is distributed to the threads available to OpenMP. Also different methods for distribution can be chosen. One can specify the number of threads via the environment variable `OMP_NUM_THREADS`. Furthermore, the parallel for is a method to implement the data parallel model, thus, the calculations in the for body need to be independent.

There also exists a combined method to put the parallel region and parallel for into a single statement, which is called *parallel for region*:

```
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    ...
}
```

which is the same as

```
#pragma omp parallel
{
  #pragma omp for
  for (int i = 0; i < 10; ++i) {
    ...
  }
}
```

More detailed information about the directives is available in the OpenMP manual.

[23] An annotation is a type of meta information for the compiler. The compiler handles the annotated code in a special and predefined way.

Listing 3.9: Parallelizing `permutation_test` with OpenMP.

```
template<uint K, uint L, class T> template<class D> inline void
    Dichotom<K, L, T>::permutation_test(const
       gwas::Locus_data<D>& data)
{
  ...
  // Fill contingency tables
  uint j = 0; //row index of matrix with case frequency results
  uint c = 0; //column index of contingency table

  // the unique_iterator is defined in discretedata.hpp:
  // std::map<elem_type, count_type> unique_;//unique elements
     with counts
  typename gwas::Locus_data<D>::unique_iterator uniques =
     data.unique_begin();
  for (; uniques!=data.unique_end(); uniques++) {
    bool ok = not (uniques->first == data.get_undef());
    if (ok) {
      uint n = uniques->second; //frequency of both (cases +
         controls)
      #pragma omp parallel for default(shared)
      for (uint t=0; t<con_tabs_.size(); ++t) {
        con_tabs_[t][0][c] = caseFreqs_[j][t];     //cases r[j]
        con_tabs_[t][1][c] = n - caseFreqs_[j][t]; //controls s[j]
      }
      c++;
    }
    j++;
  }
  ...
}
```

3.3.1. Implementation

If we want to add parallelism by OpenMP to the nodes, we need to put it in one or more of the steps executed on the nodes. These can be seen in figure 3.1 on page 15: *Read next SNP*, *boost* and *compute test statistic*. Because the permutation matrix is using a fast PRNG we exclude the permutation process from our investigations. As described in section 3.2.1 the parallel execution starts in method `analyze_dichotom` of class `Analyzer`. All code called from there can be considered for parallelization using OpenMP.

The first location to add parallelism is in method `permutation_test` of class `Dichotom`. In this method the contingency tables are filled just before they are used to calculate

Listing 3.10: Parallelizing `do_permutation` with OpenMP.

```
1  template<uint K, uint L, class T> template<class D> inline void
2      Dichotom<K, L, T>::do_permutation(const gwas::Locus_data<D>&
           data)
3  {
4    ...
5    caseFreqs_[worst_idx] = nCases_; //init with marginal sum
6    #pragma omp parallel for default(shared)
7    for (uint i=0; i<L+1; i++) {
8      if (i != worst_idx) {
9        boosters_[i].permute(
10             index_[i],          //index coded data
11             dummy_[i],          //dummy coded data
12             boost_index[i],     //index into booster's buffer
13             &caseFreqs_[i]);    //resulting case frequenices
14       #pragma omp critical
15       {
16         caseFreqs_[worst_idx] -= caseFreqs_[i];
17       }
18     }
19   }
20   ...
21 }
```

the test statistic. The computation is done using the intermediate result calculated by the boosters. We parallelize the for loop which enumerates over the contingency tables. Synchronization is not needed because any position in the array is accessed only once and therefore exclusively. For parallelization we use the parallel for region. The change is shown in line 16 of listing 3.9.

The second location identified to use with OpenMP is located in method `do_permutation` of class `Dichotom`. In this method the boosters are called to calculate the intermediate result which is used to fill the contingency tables. The method is called from `permutation_test` before the contingency tables are calculated. We parallelize the for-statement which calls the boosters for each genotype separately. Again we use a parallel for region. The change is shown in line 6 of listing 3.10. In line 14 we see an OpenMP `critical` statement. This statement ensures that only one thread at a time accesses the code in the following block; it synchronizes the threads. Synchronization in the loop is needed because the intermediate result of the genotype with the least significant speedup is calculated from the other genotypes.

Listing 3.11: This listing is showing an excerpt of the BJam configuration file `jamroot.jam` changed to instruct the compiler and linker to use OpenMP.

```
1  ...
2  project permory
3      : requirements
4        <include>src
5        <link>static
6        <define>BOOST_FILESYSTEM_VERSION=2
7        <linkflags>-fopenmp
8        <cflags>-fopenmp
9      : usage-requirements
10       <include>.
11       $(flags)
12     ;
13 ...
```

These two locations are more fine-grained than the MPI approach but still more coarse-grained to execute at least 10 000 instructions, depending on the permutation block size. Before getting into more detailed parallelization a we perform a benchmark. This requires adjustment of the build process to use OpenMP support.

To activate OpenMP we need to pass a flag to the compiler. This flag is compiler dependent and can be gathered from the OpenMP documentation. For the GCC C and C++ compilers the argument is `-fopenmp`. Using BJam we need to change the file `jamroot.jam`. The flag needs to be passed to the compiler via `<cflags>` and the linker via `<linkflags>`. In listing 3.11 the necessary changes are in lines 7 and 8.

3.3.2. Benchmarks

We now run benchmarks using the 22nd chromosome of the WTCCC2 study. Again we divide the individuals in 2 834 cases and 2 833 controls. Each run is done using 100 000 permutations. The parallelized locations are affected by the permutation block size. Thus, we do runs with different block sizes to show if the runtime is affected. We choose block sizes of 10 000, 25 000, 50 000 and 100 000. As reference we also measure the runtime in sequential mode using the same block sizes. So we compare the runtimes of the parallelized run and sequential run with the same block sizes. The parallel runs are done using two threads. To achieve this we set the environment variable `OMP_NUM_THREADS` to 2.

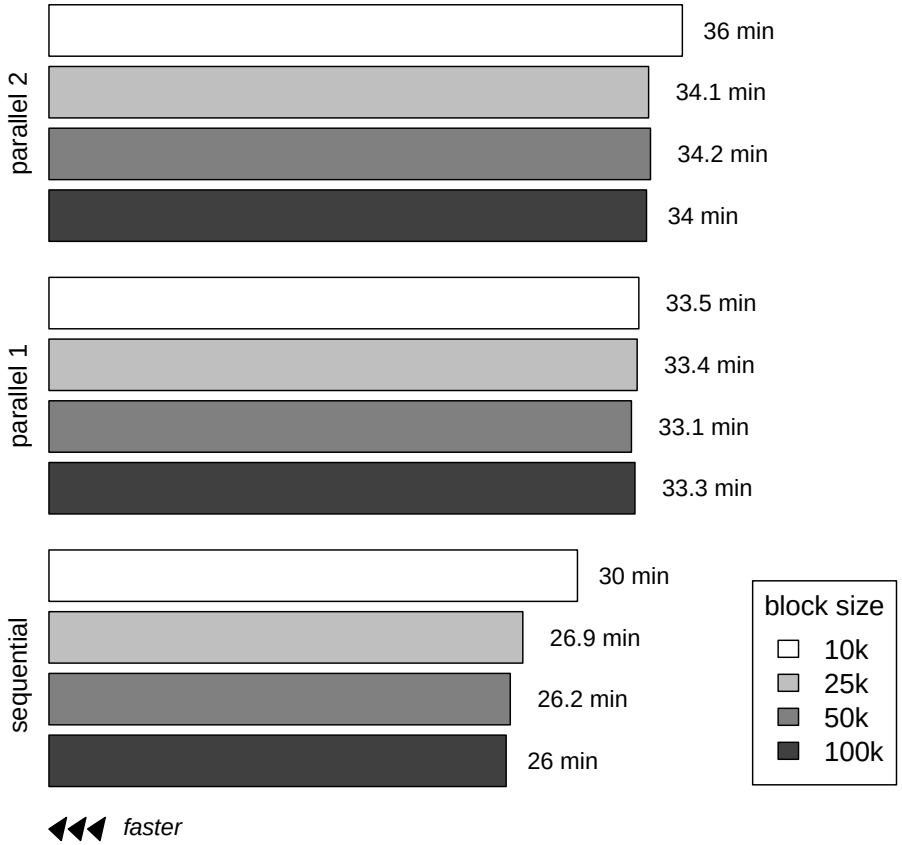

Figure 3.4.: The figure shows the runtimes of the sequential run and the parallel runs using OpenMP. Where *parallel 1* is the first location OpenMP support was implemented, and *parallel 2* the second location, respectively. In the sequential run the runtime decreases as the block size increases, whereas the parallel runtime seems to be random and never beats its sequential counterpart.

As we can see in figure 3.4 the runtimes of the parallelized runs are longer than the runtime of the sequential run. The single process run took 30 min using a block size of 10 000 permutations. It dropped by increasing the block size. The parallel code never was faster than 33 minutes. Furthermore, we do not see any relation between block size and runtime in the parallel runs.

Receiving these benchmark results we discard the intent to parallelize even more fine-grained with OpenMP.

4. Quantitative Phenotypes

The existing version of PERMORY can handle binary phenotypes only. In this chapter we describe how to extend the algorithm to handle quantitative phenotypes.

Dichotomous phenotypes indicate the disease status of an individual—case or control—whereas quantitative phenotypes are described by a continuous value.
Often, the disease status is derived from a continuous value by specifying a margin. Examples are hypertension and overweight with quantitative phenotypes blood pressure and body mass index, respectively. The margin may be hard to define and vary in different studies. Using quantitative values, this disadvantage is prevented. The recommendation of Ziegler and König is to prefer quantitative phenotypes over dichotomous when both are available.[24]

4.1. Test statistic

The test statistic used for dichotomous phenotypes cannot be used with quantitative values, because there is no distinction between case and control anymore. Thus we need another test statistic[25]:

$$T_j^2 = \frac{\left[\sum_{i=1}^{N} U_{ji}\right]^2}{\sum_{i=1}^{N} {U_{ji}}^2} \tag{4.1}$$

with

$$U_{ji} = \left(Y_i - \overline{Y}\right)\left(X_{ji} - \overline{X_j}\right);\ \overline{Y} = \frac{1}{N}\sum_{i=1}^{N} Y_i;\ \overline{X_j} = \frac{1}{N}\sum_{i=1}^{N} X_{ji};\ X_{ji} \in \{0,1,2\}$$

where Y_i is the phenotype of the i-th individual, X_{ji} is the genotype of the ith individual in the j-th locus, N is the number of individuals, $\overline{Y}$ is the number mean of the phenotypes

[24] Ziegler and König 2010, p. 222 f.
[25] Lin 2006.

and $\overline{X_j}$ is the number mean of the genotypes in the j-th locus. By replacing the variables into formula 4.1 we get:

$$T_j^2 = \frac{\left[\sum_{i=1}^{N}\left(Y_i - \overline{Y}\right)\left(X_{ji} - \overline{X_j}\right)\right]^2}{\sum_{i=1}^{N}\left[\left(Y_i - \overline{Y}\right)\left(X_{ji} - \overline{X_j}\right)\right]^2} \tag{4.2}$$

The test statistic gives an indication of the correlation between the phenotypes and genotypes of a locus. The higher the value the higher the correlation.

4.2. Boosters

As described in section 2.2 on page 9 the boosters efficiently calculate the frequencies of cases and controls separately for each genotype value before the contingency tables are filled with the intermediate result. Using quantitative phenotypes and a floating point data type instead of binary, we need to prove, that the boosters can still be used.

Bit arithmetics cannot be used as we do not have binary representation anymore. If summing up the phenotypes is possible, grouped by the corresponding genotype, the two methods GIT and REM are suitable.

$$T_j^2 = \frac{\left[\sum_{i=1}^{N}\left(Y_i - \overline{Y}\right)\left(X_{ji} - \frac{1}{N}\sum_{i=1}^{N} X_{ji}\right)\right]^2}{\sum_{i=1}^{N}\left[\left(Y_i - \overline{Y}\right)\left(X_{ji} - \frac{1}{N}\sum_{i=1}^{N} X_{ji}\right)\right]^2} \tag{4.3}$$

Looking at formula 4.3 we see that a permutation of Y is a reassignment of phenotypes to genotypes. To simplify the following calculations, we can decide which of the terms $\left(Y_i - \overline{Y}\right)$ and $\left(X_{ji} - \overline{X_j}\right)$ we want to look at as variable. For the next sections we choose $\left(X_{ji} - \overline{X_j}\right)$ to be the variable and $\left(Y_i - \overline{Y}\right)$ to be the constant.

In the following we separately analyze the numerator and denominator. For this we introduce the substitudes ${S_1}^2$ for the numerator and S_2 for the denominator:

$$T_j^2 = \frac{{S_1}^2}{S_2} \tag{4.4}$$

Examining the Numerator

First we transform S_1 to simplify the term:

$$\begin{aligned} S_1 &= \sum_{i=1}^{N} \left(Y_i - \overline{Y}\right)\left(X_{ji} - \overline{X_j}\right) \\ &= \sum_{i=1}^{N} \left[Y_i X_{ji} - Y_i \overline{X_j} - \overline{Y} X_{ji} + \overline{Y}\,\overline{X_j}\right] \\ &= \sum_{i=1}^{N} \left[X_{ji}\left(Y_i - \overline{Y}\right) - \overline{X_j}\left(Y_i - \overline{Y}\right)\right] \\ &= \sum_{i=1}^{N} X_{ji}\left(Y_i - \overline{Y}\right) - \sum_{i=1}^{N} \overline{X_j}\left(Y_i - \overline{Y}\right) \\ &= \sum_{i=1}^{N} X_{ji}\left(Y_i - \overline{Y}\right) - \overline{X_j}\sum_{i=1}^{N}\left(Y_i - \overline{Y}\right) \\ &= \sum_{i=1}^{N} X_{ji}\left(Y_i - \overline{Y}\right) \end{aligned} \tag{4.5}$$

The term $\overline{X_j}\sum_{i=1}^{N}\left(Y_i - \overline{Y}\right)$ can be omitted, because:

$$\begin{aligned} & \sum_{i=1}^{N}\left(Y_i - \overline{Y}\right) \\ = \; & \sum_{i=1}^{N} Y_i - \sum_{i=1}^{N} \overline{Y} \\ = \; & \sum_{i=1}^{N} Y_i - N\,\overline{Y} \\ = \; & \sum_{i=1}^{N} Y_i - N\left(\frac{1}{N}\sum_{i=1}^{N} Y_i\right) \\ = \; & \sum_{i=1}^{N} Y_i - \sum_{i=1}^{N} Y_i \\ = \; & 0 \end{aligned} \tag{4.6}$$

So we only need to look at $\sum_{i=1}^{N} X_{ji}\left(Y_i - \overline{Y}\right)$. This is exactly the form the boosters need, as we can handle the term separated by genotype value:

$$X_{ji}\left(Y_i - \overline{Y}\right) = \begin{cases} 0 \cdot \left(Y_i - \overline{Y}\right) & \text{if } X_{ji} = 0 \\ 1 \cdot \left(Y_i - \overline{Y}\right) & \text{if } X_{ji} = 1 \\ 2 \cdot \left(Y_i - \overline{Y}\right) & \text{if } X_{ji} = 2 \end{cases} \tag{4.7}$$

Consider two markers G_j and G_{j-1} which differ in only one position z, we can compute the numerator part S_{1j} of G_j from the numerator part S_{1j-1} of G_{j-1} by simply adding the difference described in position z. Yielding the computation of S_{1j} from S_{1j-1} for $X_{(j-1)1} = X_{j1}, X_{(j-1)2} = X_{j2}, \ldots, X_{(j-1)(z-1)} = X_{j(z-1)}, X_{(j-1)z} \neq X_{jz}, X_{(j-1)(z+1)} =$

$X_{j(z+1)}, \ldots, X_{(j-1)N} = X_{jN}$:

$$\begin{aligned} S_{1j} &= \sum_{i=1}^{N} X_{ji} \left(Y_i - \overline{Y}\right) \\ &= \sum_{i=1}^{N} X_{(j-1)i} \left(Y_i - \overline{Y}\right) \quad + X_{jz} \left(Y_i - \overline{Y}\right) - X_{(j-1)z} \left(Y_i - \overline{Y}\right) \\ &= \sum_{i=1}^{N} X_{(j-1)i} \left(Y_i - \overline{Y}\right) \quad + \left(X_{jz} - X_{(j-1)z}\right) \left(Y_i - \overline{Y}\right) \\ &= S_{1j-1} + \left(X_{jz} - X_{(j-1)z}\right) \left(Y_i - \overline{Y}\right) \end{aligned} \tag{4.8}$$

It follows by induction that we can compute any S_{1j} from any S_{1j-1} for $G_j \neq G_{j-1}$.

Examining the Denominator

After examination of the numerator we examine the denominator. Again, we transform the equation:

$$\begin{aligned} S_2 &= \sum_{i=1}^{N} {U_{ji}}^2 \\ &= \sum_{i=1}^{N} \left[\left(Y_i - \overline{Y}\right) \left(X_{ji} - \overline{X_j}\right)\right]^2 \\ &= \sum_{i=1}^{N} \left[\left(Y_i - \overline{Y}\right)^2 \left(X_{ji} - \overline{X_j}\right)^2\right] \end{aligned} \tag{4.9}$$

We separately look at the genotypes another time:

$$\left(Y_i - \overline{Y}\right)^2 \left(X_{ji} - \overline{X_j}\right)^2 = \begin{cases} \left(Y_i - \overline{Y}\right)^2 \left(0 - \overline{X_j}\right)^2 & \text{if } X_{ji} = 0 \\ \left(Y_i - \overline{Y}\right)^2 \left(1 - \overline{X_j}\right)^2 & \text{if } X_{ji} = 1 \\ \left(Y_i - \overline{Y}\right)^2 \left(2 - \overline{X_j}\right)^2 & \text{if } X_{ji} = 2 \end{cases} \tag{4.10}$$

Given two markers G_j and G_{j-1} which differ in only one position z, we can compute the denominator part S_{2j} of G_j from the denominator part S_{2j-1} of G_{j-1} by simply adding the difference described in position z. We yield the computation of S_{2j} from S_{2j-1} for $X_{(j-1)1} = X_{j1}, X_{(j-1)2} = X_{j2}, \ldots, X_{(j-1)(z-1)} = X_{j(z-1)}, X_{(j-1)z} \neq X_{jz}, X_{(j-1)(z+1)} = X_{j(z+1)}, \ldots, X_{(j-1)N} = X_{jN}$:

$$\begin{aligned} S_{2j} &= \sum_{i=1}^{N} \left[\left(Y_i - \overline{Y}\right)^2 \left(X_{ji} - \overline{X_j}\right)^2\right] \\ &= \sum_{i=1}^{N} \left[\left(Y_i - \overline{Y}\right)^2 \left(X_{(j-1)i} - \overline{X_{j-1}}\right)^2\right] \\ &\quad + \left(Y_i - \overline{Y}\right)^2 \left[\left(X_{ji} - \overline{X_j}\right)^2 - \left(X_{(j-1)i} - \overline{X_{j-1}}\right)^2\right] \\ &= S_{2j-1} + \left(Y_i - \overline{Y}\right)^2 \left[\left(X_{ji} - \overline{X_j}\right)^2 - \left(X_{(j-1)i} - \overline{X_{j-1}}\right)^2\right] \end{aligned} \tag{4.11}$$

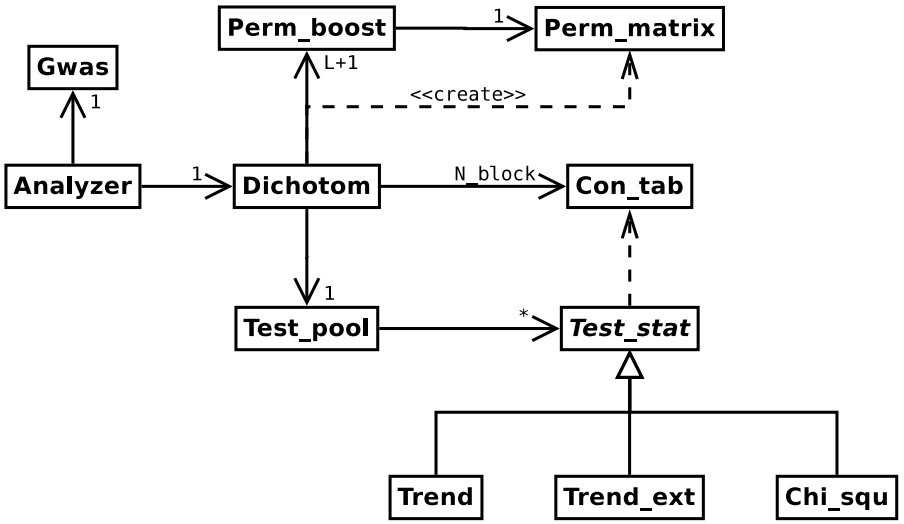

Figure 4.1.: Simplified class diagram of PERMORY using binary traits. The constant L is the number of different trait values.

It follows by induction that we can compute any S_{2j} from any S_{2j-1} for $G_j \neq G_{j-1}$.

We see, derivation of the numerator and denominator of a marker G_j from another marker G_{j-1} is possible. Thus we can use REM as well as GIT with the boosters and quantitative phenotypes.

4.3. Implementation

After proving that the usage of GIT and REM is possible in combination with the new test statistic, we can start with the implementation.

In figure 4.1 a simplified class diagram of PERMORY's structure is shown. The diagram shows the state after our refactoring described on page 16. Class template parameters are omitted in the figure to keep it simple. An instance of `Analyzer` initiates and calls an instance of `Dichotom`. The newly created instance takes control of the computational process. It uses an instance of the class `Test_pool` which holds different test statistics. The actual calculation of the test statistic is encapsulated in classes of type `Test_stat`. PERMORY currently has the three specializations `Trend`, `Trend_ext` and `Chi_squ` of `Test_stat`. Also the preparation of the contingency tables is handled by `Dichotom` by calling the boosters.

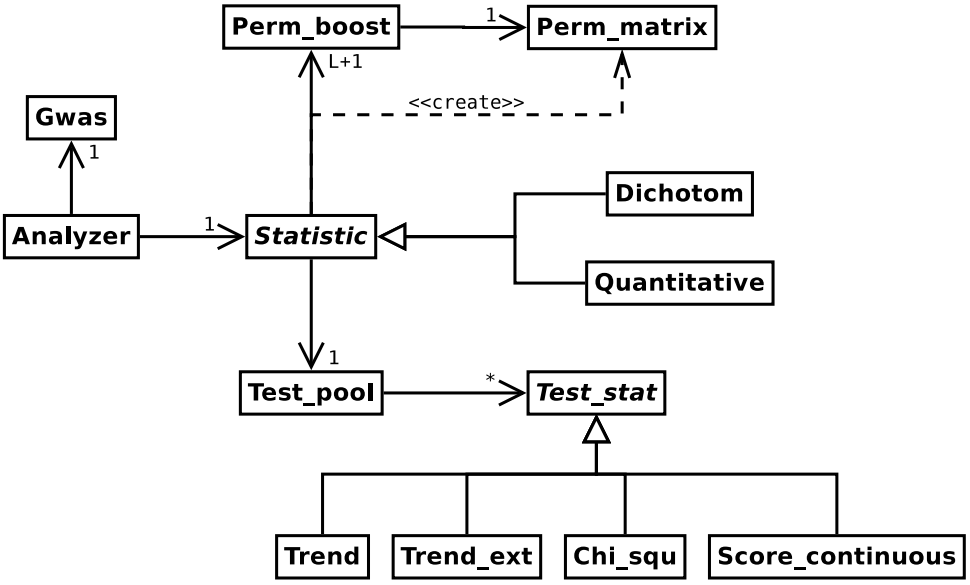

Figure 4.2.: Simplified class diagram of PERMORY using binary and quantitative traits. This diagram shows the conceptual design of the class hierarchy for using binary and quantitative traits. The constant L is the number of different trait values.

In order to handle quantitative phenotypes we introduce a new class `Quantitative`. An instance of this new class is used in place of an instance of `Dichotom` if processing quantitative phenotypes. So the new class has the same responsibilities as `Dichotom`. To make the instance of `Analyzer` aware of the class to use, we introduce a common base class named `Statistic` for `Dichotom` and `Quantitative`. Also we use this base class to reduce redundancy in the source code. It processes any operation that is related to dichotomous and quantitative traits. The specific behavior is placed in the two subclasses. An additional specialization of `Test_stat` is required to compute the test statistic described above. This class is named `Score_continuous`. A simplified class diagram without class template parameters of the redesigned structure is shown in figure 4.2.

4.3.1. Replacement of the Contingency Tables

As described in section 2.2 'Boosters' PERMORY employs dummy code to compute an intermediate result using the boosters. This intermediate result is further processed to fill the contingency tables. These tables are required by the test statistics for dichotomous traits.

Listing 4.1: The new program option `phenotype` is added to the `options_description` object.

```
1 options_description analysis("Analysis");
2 analysis.add_options()
3   ...
4   ("phenotype",
5     value<Record::Value_type>(&par.val_type)->
6       default_value(Record::dichotomous),
7    "type␣of␣phenotype␣data:␣1=dichotom,␣2=continuous")
8   ;
```

The test statistic for quantitative phenotypes cannot be calculated using contingency tables. Instead we can use vectors of doubles. Because of caching purposes we use another type which is summable. As mentioned in the beginning of section 'Boosters' we consider $\left(Y_i - \overline{Y}\right)$ being invariant. To save computing power we use $\left(Y_i - \overline{Y}\right)$ instead of just the phenotype Y_i in the permutation matrix. Because of the separate calculation of the numerator and denominator we utilise the invariant for the numerator and the squared invariant for the denominator. We buffer both values for each trait on creation of an instance of `Quantitative`.

The buffer is a vector of pairs. To store the data in a pair we create a structure `Pair` which inherits from `boost::pair`. This is necessary to add the arithmetical operations required by the permutation matrix. The employment of a structure or class is essential as we need the assignment operator, which is not available in free functions.

4.3.2. Program Option

We introduce a new program option `phenotype` to be able to instruct PERMORY to use quantitative phenotypes. If the option is set to `1` or `dichotom` then PERMORY processes dichotomous data. Quantitative phenotypes are processed if we pass `2` or `continuous`. By default PERMORY processes dichotomous phenotypes.

PERMORY uses the Boost Program Options library[26] to handle program options. We add the new option to the `boost::operation_description` object. The attribute `val_type` of class `Parameter` is used to store the information of the type of the traits (see listing 4.1). The Boost Program Option library employs the input stream operator to parse values.

[26] `http://www.boost.org/doc/libs/release/doc/html/program_options.html`

Listing 4.2: This listing shows the helper function to parse command line arguments for type `Record::Value_type`.

```
std::istream& operator>>(std::istream& in, Record::Value_type&
    out)
{
  using std::string;

  string token;
  in >> token;
  if (token == "dichotomous"
    or token == boost::lexical_cast<string>(Record::dichotomous))
       {
    out = Record::dichotomous;
  }
  else if (token == "continuous"
    or token == boost::lexical_cast<string>(Record::continuous)) {
    out = Record::continuous;
  }
  else {
    out = Record::undefined;
  }
  return in;
}
```

With the help of a templated stream-operator function, the library can parse custom types. All it needs is a type hint, which is passed to the options description instance. In our case we pass `Record::Value_type`. We create an `operator>>`-function which handles this type. The code for parsing the argument passed from the command line is shown in listing 4.2.

In case of continuous traits we deactivate the usage of bit arithmetics:

```
par.useBar = par.val_type == Record::dichotomous;
```

4.3.3. Test Statistic

Instances of the abstract class `Test_stat` calculate the test statistic. The class is implemented using contingency tables. To enable the class to handle any type we change the template parameters from `<uint K, uint L>` to `<class T>`. This enforces us to change the three existent subclasses, which is a straight forward approach. Furthermore we need

Permutations	PERMORY	PLINK
100	—	0.52 h
1 000	2.15 min	5.22 h
10 000	16.10 min	52.22 h[a]

Table 4.1.: Results of the quantitative phenotypes benchmark using the 22nd chromosome of the WTCCC2 study, which provides 18 649 SNPs. Each datum is the mean of the runtime of five runs rounded to the second position after decimal point. Because of the big difference the times of PERMORY are in minutes and the times of PLINK are in hours.

[a] approximated

to change the `Test_pool` in the same way as we changed `Test_stat`. The instantiation of `Test_pool` for dichotomous traits looks like:

```
Test_pool<Con_tab<K, L> > testPool_;
```

And for quantitative traits:

```
Test_pool<std::vector<Pair<T> > testPool_;
```

We create a new class `Score_continuous` as specialization of `Test_stat`. As described above the new class takes `std::vector< Pair<T> >` instead of contingency tables.

4.4. Benchmarks

Because of missing real data with quantitative traits we simulate them. As genotype data we use the existing data of the WTCCC2 study and a simulated HapMap[27] data set. Because the actual result is irrelevant in the benchmark we create the phenotypes randomly. We choose a mean of 25 and a standard deviation of 5 as parameters for the creation. To generate `M` phenotypes of this kind we use the following R statement:

```
round(rnorm(M, 25, 5))
```

For example, to create the phenotypes for the WTCCC2 study with 5 667 individuals we need to set `M` to 5 667.
After creating the quantitative traits we start the benchmarks.

[27] The International HapMap Consortium 2007.

	WTCCC2 Chr22	HapMap 1M full	HapMap 2.4M Chr22
Dichotomous	2.15 min	2.39 h	4.77 min
Continuous	16.10 min	16.30 h	25.64 min
Factor	7.48	6.83	5.38

Table 4.2.: The table shows the runtimes of the quantitative and dichotomous runs. We used the mean of five runs with 10 000 permutations. The values are rounded to the second position after decimal point.

First we compare the runtimes of five runs of PERMORY with PLINK as reference using quantitative traits. We employ the 22nd chromosome with 18 649 SNPs of the WTCCC2 study for this benchmark. PLINK's mean runtime doing 1 000 permutations is 5 hours and 13 minutes. Doing 100 permutations takes 31 minutes. We see that PLINK's runtime grows linear with the number of permutations. With this in mind we can approximate the runtime of 10 000 permutations by multiplying the runtime of the 1 000 permutations run by 10. This results in 2 days and 10 minutes. Hence, the speedup of PERMORY is 145.62 for 1 000 permutations and 194.60 for 10 000 permutations in comparison to PLINK. Even for a smaller count of permutations, e. g. 100, PLINK cannot compete with PERMORY. Again we see that the speedup of Pahl and Schäfer's algorithm pays off if more permutations are done in batch. In table 4.1 the results of the runs are shown.

Second we set the runtime of PERMORY with dichotomous and quantitative traits in relation. We do runs with the 22nd chromosome of the WTCCC2 study, the 22nd chromosome of the HapMap data with 2.4 million markers and the full data of the HapMap with 1 million markers. Chromosome 22 of the WTCCC2 study has 18 649 markers and the 22nd chromosome of the 2.4 million data set has 33 815 markers. As we can see in table 4.2 the quantitative runs are between 5 and 8 times slower than the runs with dichotomous data.

Part III.

Conclusion

In this last part we summarize the results of extending PERMORY and give an outlook of possible future projects.

5. Results

In part II we discussed the parallelization of PERMORY and we introduced the handling of quantitative phenotypes. In this chapter we give a summary of the results we have derived from this discussion.

5.1. Parallelization

We have parallelized PERMORY with a nearly linear speedup using Open MPI. The ease of extending the algorithm is owed to the good quality of the source code. This also includes the usage of the Boost Library.

It turns out that OpenMP is not suitable to implement more fine-grained parallelization. Responsible for the longer runtimes is the optimized code structure used in the boosters. Processing a block of 10 000 permutations takes less than one second. However, the synchronization of processes required in parallel architectures slows down the execution significantly.

Furthermore todays scalar and super-scalar processor architectures may have a big impact here. Both techniques use pipelining to execute multiple instructions in one processor cycle. On processing arrays sequentially we expect the compiler to optimize the code to take advantage of the processor pipelining. If we need to synchronize data between the threads one must wait for the other—a data dependency occurs—and the pipeline is interrupted.[28] Further research has to show if this assumption holds.

[28] Rauber and Rünger 2007, p. 11 ff.

5.2. Quantitative Phenotypes

In chapter 4 we have shown a way to treat quantitative phenotypes in GWAS using the optimizations REM and GIT. The measured runtimes compared to the serial version of PERMORY are a factor of 5 to 8 slower. In comparison to PLINK the quantitative phenotype processing has a speedup of several magnitudes.

6. Outlook

In this chapter we provide a small overview of possible future projects.

6.1. OpenMP

OpenMP could be used to parallelize PERMORY in the same location as Open MPI is used for. At the moment a part of the initialization of the study object is done during the first permutation. If this link is resolved the calculation of the permutations can be distributed to several cores. This would result in a longer runtime if run sequentially but could be outweighed by the use of OpenMP. In summary an independent and additionally parallelized initialization process could result in a speedup.

6.2. Automatic Detection of Phenotype Type

In the current version the type of the phenotypes is specified by the user. An automatic detection of the phenotype data is conceivable. For example PLINK automatically detects the type to use.

6.3. Multiple Test Statistics

Instead of using only one test statistic per permutation test it is possible to do multiple tests in one run.

6.4. Multiple Traits

Analogous to calculating multiple test statistics in one permutation test it may be suitable to compute multiple traits in one run.

6.5. Flow Based Programming

The algorithm can be described with design methods of flow-based programming. This may have an impact on the runtime of the parallelization as the components may be distributed on a cluster automatically. A possible division of PERMORY into components can be seen in figure 6.1 on the next page. Refactoring the algorithm to fit the flow-based programming paradigm may be a future project.

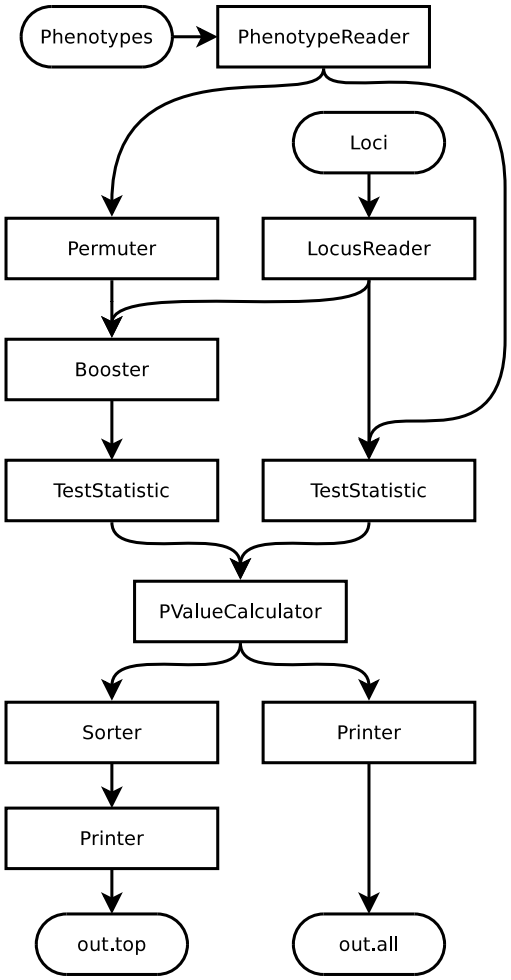

Figure 6.1.: This figure shows a schematic of the data-flow in PERMORY. Rectangles represent active components which operate on the data. The round shapes are data storages. Arrows visualize the data-flow.

Bibliography

Advanced Micro Devices, Inc. (2006). *Performance Guidelines for AMD Athlon^{TM} 64 and AMD Opteron^{TM} ccNUMA Multiprocessor Systems*.

Gamma, Erich et al. (2007). *Design patterns: elements of reusable object-oriented software*. 35th Printing. Addison-Wesley professional computing series. München: Addison-Wesley. ISBN: 0-201-63361-2.

Lin, D Y (2006). 'Evaluating statistical significance in two-stage genomewide association studies'. In: *American Journal Human Genetics* 78.3 (Mar. 2006), pp. 505–509. URL: http://view.ncbi.nlm.nih.gov/pubmed/16408254.

Matsumoto, Makoto and Takuji Nishimura (1998). 'Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator'. In: *ACM Trans. Model. Comput. Simul.* 8 (1 1998), pp. 3–30. ISSN: 1049-3301. DOI: http://doi.acm.org/10.1145/272991.272995. URL: http://doi.acm.org/10.1145/272991.272995.

Menges, T. et al. (2008). 'Sepsis syndrome and death in trauma patients are associated with variation in the gene encoding tumor necrosis factor'. In: *Critical Care Medicine* 36.5 (May 2008), pp. 1456–1462.

Michie, Donald (1968). '"Memo" Functions and Machine Learning'. In: *Nature* 218 (Apr. 1968), pp. 19–22.

Pahl, Roman and Helmut Schäfer (2010). 'PERMORY: an LD-exploiting permutation test algorithm for powerful genome-wide association testing'. In: *Bioinformatics* 26.17 (Sept. 2010), pp. 2093–2100. URL: http://view.ncbi.nlm.nih.gov/pubmed/20605926.

Purcell, Shaun et al. (2007). 'PLINK: A Tool Set for Whole-Genome Association and Population-Based Linkage Analyses'. In: *American Journal Human Genetics* 81 (Sept. 2007), pp. 559–575. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.112.9920;http://pngu.mgh.harvard.edu/~purcell/plink/plink_preprint.pdf.

Rauber, Thomas and Gudula Rünger (2007). *Parallele Programmierung*. 2nd ed. Berlin Heidelberg: Springer. ISBN: 978-3-540-46549-2.

The International HapMap Consortium (2007). 'A second generation human haplotype map of over 3.1 million SNPs'. In: *Nature* 449 (Oct. 2007), pp. 851–861.

The Wellcome Trust Case Control Consortium (2007). 'Genome-wide association study of 14,000 cases of seven common diseases and 3,000 shared controls'. In: *Nature* 447 (June 2007), pp. 661–678.

Ziegler, Andreas and Inke R. König (2010). *A Statistical Approach to Genetic Epidemiology*. 2nd ed. Weinheim: WILEY-VCH Verlag GmbH & Co. KGaA.