GeneClass User Guide

Alexei Nordell Markovits and Dominique Toupin

May 2, 2012

Contents

1 Performing operations on several sites or chromosomes

As part of the library, we offer functions that exist to ease calculations on several sites or chromosomes. Here is a list of those (they will be explained in detail later):

- uGenericNGSChrom: Functions allowing to perform given operations on all sites of a chromosome
  - accumulateSitesInfo: Perform a calculation on all the sites of a chromosome yielding a single result
  - applyOnAllSites: Transform each site in a chromosome
  - computeOnAllSites: Perform a certain operation on every site
  - countSitesWithProperty: Obtain the number of sites in a chromosome with a specific characteristic
  - getSpecificSites: Obtain all the sites in a chromosome with a specific characteristic
  - isSorted: Indicates whether all the sites in the chromosome are sorted according to a specific characteristic
  - sortSites: Sort all the sites in the chromosome according to a specific characteristic
  - maxSite: Obtain the highest site in a chromosome for a specific characteristic
  - minSite: Obtain the lowest site in a chromosome for a specific characteristic
  - minAndMaxSite: Obtain the lowest and highest sites in a chromosome for a specific characteristic
- uGenericNGSExperiment: Functions allowing to perform given operations on all the chromosomes of an experiment
  - accumulateChromsInfo: Perform a calculation on all the chromosomes of an experiment yielding a single result
  - applyOnAllChroms: Transform each chromosome of an experiment
  - computeOnAllChroms: Perform a certain operation on every chromosome
  - countChromsWithProperty: Obtain the number of chromosomes with a specific characteristic in the experiment
  - getSpecificChroms: Obtain all the chromosomes in an experiment with a specific characteristic
  - maxChrom: Obtain the highest chromosome in an experiment for a specific characteristic
  - minChrom: Obtain the lowest chromosome in an experiment for a specific characteristic
  - minAndMaxChroms: Obtain the lowest and highest chromosomes in an experiment for a specific characteristic

1.1 General consideration 1 : What is the action to perform

All methods in this section have one common characteristic: their first parameter refers to an action to perform. This action can be specified in various ways:

- The name of a function
- A functor object
- The method of object
- Lamdba (anonymous) function

The next sections will present you with actual examples for each of these techniques.

1.1.1 Name of a function

This is probably the most straight-forward way of specifing which operation to perform. You simply need to define a function with a signature matching the one required for the operation you wish to perform and then use the name of the function as the first parameter to the "action" call:

```
// Define a function to indicate if the "start" of a site is 120
bool siteStartIs120 (uGenericNGS site)
{
    return site.getStart() == 120;
}

int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    // Use the function name in the call
    std::cout <<
        chrom.countSitesWithProperty(siteStartIs120) <<
        std::endl;
}
```

1.1.2 Functor object

While the previous method is one of the easiest to use, it is sometimes inconvenient; for example, if one wanted to count the number of sites starting at 120, then at 121, 122 and so on, one would need to create a new function for each operation. To circumvent this, it is possible to create functions for which inner parameters can change. Such functions are called functor since they are actual objects being used as functions. A basic functor needs to implement two (2) things: 1) a constructor for which parameters correspond to the inner parameters of the function and 2) the operator () which perform the actual operation. Here is an example of how the previous function could be defined as a functor*:

```
class siteStartIs
{
    private: // Optionnal since the default section of a class is "private"
        int m_start;
    public:
        siteStartIs(int start):m_start(start) {};
        bool operator () (uGenericNGS site)
        {
             return site.getStart() == m_start;
        }
};
```

Then you need to instantiate one object with the correct parameter when performing the "action" call:

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    // Instantiate one object with the correct parameter for each call
    std::cout <<
```

```
        chrom.countSitesWithProperty(siteStartIs(120)) <<
        std::endl;

    std::cout <<
        chrom.countSitesWithProperty(siteStartIs(121)) <<
        std::endl;

    std::cout <<
        chrom.countSitesWithProperty(siteStartIs(122)) <<
        std::endl;
}
```

* For more information on functors, please look up http://www.bogotobogo.com/cplusplus/functors.php for a detailed tutorial.

1.1.3 Object's methods (Advanced technique)

Sometimes, the operation one would like to perform can be defined by the class of the objects being iterated on. To tell the "action" to use these methods, a call to std::mem_fun_ref* must be used. Note that when using this technique, only methods with no or one parameter can be used. To specify which value to use, a call to std::bind_2nd* must be used to indicate the value of the parameter for ALL CALLS.

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    auto result = chrom.computeOnAllSites(std::mem_fun_ref(&uGenericNGS::getStart));

    chrom.applyOnAllSites(std::bind2nd(std::mem_fun_ref(&uGenericNGS::setChr), "X"));
}
```

* For more information on std::mem_fun_ref, std::bind_2nd* and functional programming in C++, please look up http://www.cplusplus.com/reference/std/functional/ .

1.1.4 Lamdba (anonymous) functions

Lamdba (anonymous) functions is a new feature introduced by the latest C++ standard (C++11) which allows users to define a one-time use function where needed. A basic example of this would be this:

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    std::cout <<
        chrom.countSitesWithProperty([] (uGenericNGS site) -> bool
                                     { return site.getStart() == 120; }) <<
        std::endl;
}
```

Let's analyze the syntax of this lambda function:

```
[] (uGenericNGS site) -> bool { return site.getStart() == 120; }
```

- "[]" represent the environmental capture; if no external variables are required in the lambda, "[]" should be used. Otherwise, it is possible to capture the whole environment by using "[&]";
- "(uGenericNGS site)" are the arguments of the lambda functions; in the particular case of the library, those arguments will need to match the ones required by the action to perform;
- "-> bool" is the return type of the lambda function; in this case a boolean value is returned;
- "{ return site.getStart() == 120; }" is the body of the lambda function, i.e. what the function does.

Another good tutorial can be found here: http://www.codeproject.com/Articles/71540/Explicating-the-new-C-standard-C-0x-and-its-implem#LambdaExpressions . Several examples can also be found here: http://msdn.microsoft.com/en-us/library/dd293599.aspx .

1.2 General consideration 2: What is the required function signature

As previously stated, all operations require an action to perform. Furthermore, those actions need to conform to a standard. This standard is the function signature. Depending on the operation you need to perform, you will need to conform to one of these signatures:

- UnaryOperation
  - First argument: A reference a type of the elements in the collection[1]
  - Return type: void
  - Used by: applyOnAll*
- UnaryPredicate
  - First argument: Type of the elements in the collection[1]
  - Return type: bool (represent the value of the predicate for a given element)
  - Used by: count*WithProperty, getSpecific*
- UnaryFunction
  - First argument: Type of the elements in the collection[1]
  - Return type: Anything (preferably not void...)
  - Used by: computeOnAll*
- BinaryOperation
  - First argument: Anything (preferably not void...)
  - Second argument: Type of the elements in the collection[1]
  - Return type: Same thing as the first argument
  - Used by: accumulate*Info
- BinaryComparison
  - First argument: Type of the elements in the collection[1]
  - Second argument: Type of the elements in the collection[1]
  - Return type: bool (Indicates if the first argument/element is lower than the second)
  - Used by: sortSites, isSorted, max*, min*, minAndMax*

[1] The type of the elements in the collection will generally be one of those:

- uGenericNGS
- uGenericNGSChrom<uGenericNGS>
- etc.

1.3 General consideration 3: Using parallel computation

To get the most out of the library, you need to use the GCC 4.7 compiler (required for C++11 features and parallel standard (STL) algorithms) with these options:

- When compiling:
  - `-fopenmp` : This is required to compile using OpenMP
  - `-D_GLIBCXX_PARALLEL` : This is required to indicate to the compiler to use the parallel versions of STL algorithms

1.4 uGenericNGSChrom

Currently, sites of a chromosome are represented using a vector and as such can be sorted. Otherwise, all other "batch" operations are identical as those of an experiment.

1.4.1 uGenericNGSChrom::accumulateSitesInfo*

First argument: BinaryOperation
Second argument: The initial value for the first argument of the BinaryOperation
Return value: The accumulated result for which the type is the first argument of BinaryOperation

This function queries all sites and accumulates information on those. For instance, this function can be used to find the total length of a chromosome (sum of all sites' length) or the number of unique start positions.

Examples

1. Total length of the chromosome (sum of all sites' length):

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    std::cout <<
        chrom.accumulateSitesInfo(
            [] (long long partialSum, uGenericNGS site)
                -> long long
                { return partialSum + site.getLenght(); },
            0) <<
        std::endl;
}
```

2. Number of unique start positions:

```
#include <set>

int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    std::set<long long> temporarySet;
    std::cout <<
        chrom.accumulateSitesInfo(
            [] (std::set<long long>& uniqueSet, uGenericNGS site)
                -> std::set<long long>&
                { uniqueSet.insert(site.getStart()); return uniqueSet; },
            temporarySet).size() <<
        std::endl;
}
```

Note that this version is about 1000 times faster than the following due to the multiple copies required in the latter. The difference reside in the use of the "&" sign which indicates that the variable is a reference to an element of the given type rather than a completely different variable (achieved by copying the elements).

```
#include <set>

int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    std::cout <<
        chrom.accumulateSitesInfo(
            [] (std::set<long long> uniqueSet, uGenericNGS site)
                -> std::set<long long>
                { uniqueSet.insert(site.getStart()); return uniqueSet; },
            std::set<long long>()).size() <<
        std::endl;
}
```

*Note: currently this wrapper cannot use parallel computation because of technical considerations.

1.4.2 uGenericNGSChrom::applyOnAllSites

First argument: UnaryOperation
Return value: None

This function can be used to modify all sites.

Examples

1. Changing the "Chr" identifier to "X" for all sites (lamdba):

```
int main(int argc, char* argv[])
```

```
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    chrom.applyOnAllSites(
        [] (uGenericNGS& site)
            { site.setChr("X"); });
}
```

2. Changing the "Chr" identifier to "X" for all sites (member function):

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    chrom.applyOnAllSites(
        std::bind2nd(std::mem_fun_ref(&uGenericNGS::setChr), "X"));
}
```

1.4.3 uGenericNGSChrom::computeOnAllSites

First argument: UnaryFunction
Return value: A vector of value for which the type is the return value type of UnaryFunction

This function can be used to compute a certain value for each site and getting the resulting values in the form of a vector. It can also be used to get a transformed copy of the collection of sites.

Examples

1. Getting the list of all start positions (not unique, member function):

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    auto result = chrom.computeOnAllSites(
                      std::mem_fun_ref(&uGenericNGS::getStart));
}
```

2. Getting the list of all sites with their "Chr" set to "X" (does not modify original list):

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    std::vector<uGenericNGS> result =
        chrom.computeOnAllSites(
            [] (uGenericNGS site) -> uGenericNGS
                { site.setChr("X"); return site; });
}
```

1.4.4 uGenericNGSChrom::countSitesWithProperty

First argument: UnaryPredicate
Return value: An integer representing the number of sites for which the predicate is true

This function can be used to get the number of sites for which a certain criteria (predicate) is present.

Example

1. Count the number of sites for which the start position is 120:

```
// Define a function to indicate if the "start" of a site is 120
bool siteStartIs120 (uGenericNGS site)
{
    return site.getStart() == 120;
}

int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    std::cout <<
        chrom.countSitesWithProperty(siteStartIs120) <<
        std::endl;
}
```

1.4.5 uGenericNGSChrom::getSpecificSites

First argument: UnaryPredicate
Return value: A vector containing all the sites for which the predicate is true

This function can be used to get all the sites for which a certain criteria (predicate) is present.

Example

1. Get all the sites for which the start position is 120::

```
// Define a function to indicate if the "start" of a site is 120
bool siteStartIs120 (uGenericNGS site)
{
    return site.getStart() == 120;
}

int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    // auto == std::vector<uGenericNGS>
    auto result = chrom.getSpecificSites(siteStartIs120);
}
```

1.4.6 uGenericNGSChrom::isSorted

First argument: BinaryComparison
Return value: Boolean value indicating if the sites are sorted or not

This function indicates if all the sites within the chromosome are sorted according to some criteria. Default criteria: start position.

Examples

1. Check is the sites are sorted according to their start position:

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    bool result = chrom.isSorted();
}
```

2. Check is the sites are sorted according to their start and end positions:

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    bool result =
        chrom.isSorted(
            [] (const uGenericNGS &site1, const uGenericNGS &site2) -> bool
                { return site1.getStart() < site2.getStart() ||
                         (site1.getStart() == site2.getStart() &&
                          site1.getEnd() < site2.getEnd()); });
}
```

1.4.7 uGenericNGSChrom::sortSites

First argument: BinaryComparison
Return value: None

This function sort all the sites within the chromosome sited on some criteria. Default criteria: start position.

Examples

1. Sort the sites according to their start position:

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    chrom.sortSites();
}
```

2. Sort the sites according to their start and end positions:

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    chrom.sortSites(
        [] (const uGenericNGS &site1, const uGenericNGS &site2) -> bool
            { return site1.getStart() < site2.getStart() ||
                     (site1.getStart() == site2.getStart() &&
                      site1.getEnd() < site2.getEnd()); });
}
```

1.4.8 uGenericNGSChrom::maxSite

First argument: BinaryComparison
Return value: An iterator on the "maximum" site

This function finds the site having the maximum value according to some criteria.

Examples

1. Find the site with the highest start position:

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    uGenericNGS result =
        *chrom.maxSite(
            [] (const uGenericNGS &site1, const uGenericNGS &site2) -> bool
                { return site1.getStart() < site2.getStart(); });
}
```

2. Find the site with the highest start and end positions:

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    uGenericNGS result =
        *chrom.maxSite(
            [] (const uGenericNGS &site1, const uGenericNGS &site2) -> bool
                { return site1.getStart() < site2.getStart() ||
                         (site1.getStart() == site2.getStart() &&
                          site1.getEnd() < site2.getEnd()); });
}
```

1.4.9 uGenericNGSChrom::minSite

First argument: BinaryComparison
Return value: An iterator on the "minimum" site

This function finds the site having the minimum value according to some criteria.

Examples

1. Find the site with the lowest start position:

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    uGenericNGS result =
        *chrom.minSite(
            [] (const uGenericNGS &site1, const uGenericNGS &site2) -> bool
                { return site1.getStart() < site2.getStart(); });
}
```

2. Find the site with the lowest start and end positions:

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    uGenericNGS result =
        *chrom.minSite(
            [] (const uGenericNGS &site1, const uGenericNGS &site2) -> bool
                { return site1.getStart() < site2.getStart() ||
                         (site1.getStart() == site2.getStart() &&
                          site1.getEnd() < site2.getEnd()); });
}
```

1.4.10 uGenericNGSChrom::minAndMaxSite

First argument: BinaryComparison
Return value: A pair of iterators on the "minimum" and maximum sites

This function finds the sites having the minimum and maximum values according to some criteria.

Examples

1. Find the sites with the lowest and highest start positions:

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]
```

```
    //auto == std::pair<std::vector<uGenericNGS>::const_iterator,
    //                  std::vector<uGenericNGS>::const_iterator>
    auto result =
        chrom.minAndMaxSites(
            [] (const uGenericNGS &site1, const uGenericNGS &site2) -> bool
                { return site1.getStart() < site2.getStart(); });
}
```

2. Find the sites with the lowest and highest start and end positions:

```
int main(int argc, char* argv[])
{
    uGenericNGSChrom<uGenericNGS> chrom;
    [Perform some data initialisation...]

    //auto == std::pair<std::vector<uGenericNGS>::const_iterator,
    //                  std::vector<uGenericNGS>::const_iterator>
    auto result =
        chrom.minAndMaxSites(
            [] (const uGenericNGS &site1, constuGenericNGS &site2) -> bool
                { return site1.getStart() < site2.getStart() ||
                         (site1.getStart() == site2.getStart() &&
                          site1.getEnd() < site2.getEnd()); });
}
```

1.5 uGenericNGSExperiment

Currently, chromosomes in an experiment are represented using a map and as such cannot be sorted. Otherwise, all other "batch" operations are the sames as for a chromosome. Also, take note that even though this is a different container type, the way the arguments are passed to the wrappers stays the same.

1.5.1 uGenericNGSExperiment::accumulateChromsInfo*

First argument: BinaryOperation
Second argument: The initial value for the first argument of the BinaryOperation
Return value: The accumulated result for which the type is the first argument of BinaryOperation

This function allow to query all chromosomes and accumulate information on those. For example, this function can be used to find the total length of the experiment (sum of all chromosomes' size) or the number of unique chromosomes' sizes.

Examples

1. Total length of the experiment (sum of all chromosomes' size):

```
int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
    [Perform some data initialisation...]

    std::cout <<
        experiment.accumulateChromsInfo(
            [] (long long partialSum, uGenericNGSChrom<uGenericNGS> chrom)
```

```
                -> long long
                { return partialSum + chrom.getChromSize(); },
            0) <<
        std::endl;
}
```

2. Number of unique chromosomes' sizes:

```
#include <set>

int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
    [Perform some data initialisation...]

    std::set<long long> temporarySet;
    std::cout <<
        experiment.accumulateChromsInfo(
            [] (std::set<long long>& uniqueSet, uGenericNGSChrom<uGenericNGS> chrom)
                -> std::set<long long>&
                { uniqueSet.insert(chrom.getChromSize()); return uniqueSet; },
            temporarySet).size() <<
        std::endl;
}
```

*Note: currently this wrapper cannot use parallel computation because of technical considerations.

1.5.2 uGenericNGSExperiment::applyOnAllChroms

First argument: UnaryOperation
Return value: None

This function can be used to modify all chromosomes.

Examples

1. Sort all chromosomes (lamdba):

```
int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
    [Perform some data initialisation...]

    experiment.applyOnAllChroms(
        [] (uGenericNGSChrom<uGenericNGS>& chrom)
            { chrom.sortSites(); });
}
```

2. Sort all chromosomes (member function, same thing as "uGenericNGSExperiment::sortData()"):

```
int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
```

```
    [Perform some data initialisation...]

    // The static_cast is used to indicate that we want to use the sortSites
    // function taking no argument (we specify the function signature)
    experiment.applyOnAllChroms(
        std::mem_fun_ref(static_cast<void (uGenericNGSChrom<uGenericNGS>::*)()>
                         (&uGenericNGSChrom<uGenericNGS>::sortSites)));
}
```

1.5.3 uGenericNGSExperiment::computeOnAllChroms

First argument: UnaryFunction
Return value: A map for which the keys are the keys of the corresponding chromosomes and the values are the return values of UnaryFunction

This function can be used to compute a certain value for each chromosome and getting the resulting values in the form of a map. It can also be used to get a transformed copy of the collection of chromosomes.

Examples

1. Get the size of all chromosomes:

```
int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
    [Perform some data initialisation...]

    // auto == std::map<std::string, long long>
    auto result =
        experiment.computeOnAllChroms(
            [] (uGenericNGSChrom<uGenericNGS> chrom)
                -> long long
                { return chrom.getChromSize(); });
}
```

2. Get a copy of all the chromosome for which their last site was removed:

```
int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
    [Perform some data initialisation...]

    // auto == std::map<std::string, uGenericNGSChrom<uGenericNGS>>
    auto result =
        experiment.computeOnAllChroms(
            [] (uGenericNGSChrom<uGenericNGS> chrom)
                -> uGenericNGSChrom<uGenericNGS>
                { if (chrom.count() > 0)
                      chrom.removeSite(chrom.count()-1);
                  return chrom; });
}
```

1.5.4 uGenericNGSExperiment::countChromsWithProperty

First argument: UnaryPredicate
Return value: An integer representing the number of chromosomes for which the predicate is true

This function can be used to get the number of chromosomes for which a certain criteria (predicate) is present.

Examples

1. Count the number of chromosomes having 22 sites:

```
bool chromSizeIs22 (const uGenericNGSChrom<uGenericNGS>& chrom)
{
    return chrom.getChromSize() == 22;
}

int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
    [Perform some data initialisation...]

    std::cout <<
        experiment.countChromsWithProperty(chromSizeIs22) <<
        std::endl;
}
```

1.5.5 uGenericNGSExperiment::getSpecificChroms

First argument: UnaryPredicate
Return value: A map containing all the chromosomes for which the predicate is true

This function can be used to get all the chromosomes for which a certain criteria (predicate) is present.

Examples

1. Get all the chromosomes having 22 sites:

```
bool chromSizeIs22 (const uGenericNGSChrom<uGenericNGS>& chrom)
{
    return chrom.getChromSize() == 22;
}

int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
    [Perform some data initialisation...]

    // auto == std::map<std::string, uGenericNGSChrom<uGenericNGS>>
    auto result = experiment.getSpecificChroms(chromSizeIs22);
}
```

1.5.6 uGenericNGSExperiment::maxChrom

First argument: BinaryComparison
Return value: An iterator on the "maximum" chromosome

This function finds the chromosome having the maximum value according to some criteria.

Examples

1. Find the chromosome with the highest size:

```
int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
    [Perform some data initialisation...]

    uGenericNGSChrom<uGenericNGS> result =
        experiment.maxChrom(
            [] (const uGenericNGSChrom<uGenericNGS>& chrom1,
                const uGenericNGSChrom<uGenericNGS>& chrom2) -> bool
                { return chrom1.getChromSize() < chrom2.getChromSize(); })->second;
}
```

2. : Find the chromosome with the most sites starting at 120:

```
bool siteStartIs120 (const uGenericNGS& site)
{
    return site.getStart() == 120;
}

int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
    [Perform some data initialisation...]

    uGenericNGSChrom<uGenericNGS> result =
        experiment.maxChrom(
            [] (const uGenericNGSChrom<uGenericNGS>& chrom1,
                const uGenericNGSChrom<uGenericNGS>& chrom2) -> bool
                { return chrom1.countSitesWithProperty(siteStartIs120) <
                         chrom2.countSitesWithProperty(siteStartIs120); })->second;
}
```

1.5.7 uGenericNGSExperiment::minChrom

First argument: BinaryComparison
Return value: An iterator on the "minimum" chromosome

This function finds the chromosome having the minimum value according to some criteria.

Examples

1. Find the chromosome with the lowest size:

```
int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
    [Perform some data initialisation...]

    uGenericNGSChrom<uGenericNGS> result =
        experiment.minChrom(
            [] (const uGenericNGSChrom<uGenericNGS>& chrom1,
                const uGenericNGSChrom<uGenericNGS>& chrom2) -> bool
                { return chrom1.getChromSize() < chrom2.getChromSize(); })->second;
}
```

2. : Find the chromosome with the least sites starting at 120:

```
bool siteStartIs120 (const uGenericNGS& site)
{
    return site.getStart() == 120;
}

int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
    [Perform some data initialisation...]

    uGenericNGSChrom<uGenericNGS> result =
        experiment.minChrom(
            [] (const uGenericNGSChrom<uGenericNGS>& chrom1,
                const uGenericNGSChrom<uGenericNGS>& chrom2) -> bool
                { return chrom1.countSitesWithProperty(siteStartIs120) <
                         chrom2.countSitesWithProperty(siteStartIs120); })->second;
}
```

1.5.8 uGenericNGSExperiment::minAndMaxChroms

First argument: BinaryComparison
Return value: A pair of iterators on the "minimum" and maximum chromosomes

This function finds the chromosomes having the minimum and maximum values according to some criteria.

Examples

1. Find the chromosomes with the lowest and highest sizes:

```
int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
    [Perform some data initialisation...]

    // auto == std::pair<
    //             std::map<
    //                 std::string,
    //                 uGenericNGSChrom<uGenericNGS>
    //                 >::const_iterator,
```

```
    //             std::map<
    //                 std::string,
    //                 uGenericNGSChrom<uGenericNGS>
    //                 >::const_iterator>
    auto result =
        experiment.minAndMaxChroms(
            [] (const uGenericNGSChrom<uGenericNGS>& chrom1,
                const uGenericNGSChrom<uGenericNGS>& chrom2) -> bool
                { return chrom1.getChromSize() < chrom2.getChromSize(); });
}
```

2. : Find the chromosomes with the least and with the most sites starting at 120:

```
bool siteStartIs120 (const uGenericNGS& site)
{
    return site.getStart() == 120;
}

int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
    [Perform some data initialisation...]

    // auto == std::pair<
    //             std::map<
    //                 std::string,
    //                 uGenericNGSChrom<uGenericNGS>
    //                 >::const_iterator,
    //             std::map<
    //                 std::string,
    //                 uGenericNGSChrom<uGenericNGS>
    //                 >::const_iterator>
    auto result =
        experiment.minAndMaxChroms(
            [] (const uGenericNGSChrom<uGenericNGS>& chrom1,
                const uGenericNGSChrom<uGenericNGS>& chrom2) -> bool
                { return chrom1.countSitesWithProperty(siteStartIs120) <
                         chrom2.countSitesWithProperty(siteStartIs120); });
}
```

1.6 Mixing operations

By combining methods from uGenericNGSChrom, uGenericNGSExperiment and the standard library (STL), it is possible to perform operations on large data sets.

For example, here is how one could find the lowest site in an experiment:

```
int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
    [Perform some data initialisation...]

    // Use auto here; you really don't want to try figure out the type of this...
```

```
    // We could have created a function here as well
    auto compareSites = [] (const uGenericNGS &site1, const uGenericNGS &site2) -> bool
                            { return site1.getStart() < site2.getStart() ||
                                (site1.getStart() == site2.getStart() &&
                                  site1.getEnd() < site2.getEnd()); };

    // Get iterators on the smallest sites of each chromosomes
    auto result =
        experiment.computeOnAllChroms(
            [&compareSites]
                (const uGenericNGSChrom<uGenericNGS>& chrom)
                    -> std::vector<uGenericNGS>::const_iterator
                {
                    return chrom.minSite(compareSites);
                });

    // Use the min_element algorithm to find the smallest amongst the smallest...
    min_element(
        begin(result), end(result),
        [&compareSites]
            (const std::map<std::string,
                            std::vector<uGenericNGS>::const_iterator>::value_type& site1,
            const std::map<std::string,
                            std::vector<uGenericNGS>::const_iterator>::value_type& site2)
                -> bool
            {
                return compareSites(*(site1.second), *(site2.second));
            }
        )->second->writeBedToOuput(std::cout, false);
}
```

1.6.1 Sharing variables

However, when combining features from these various sources and using parallel computation, it is very important to make sure none of the variables used in the code is shared and overwritten in the parallelized section.

The following code is an example of bad parallel code:

```
int main(int argc, char* argv[])
{
    uGenericNGSExperiment<uGenericNGSChrom<uGenericNGS>, uGenericNGS> experiment;
    [Perform some data initialisation...]

    // Use auto here; you really don't want to try figure out the type of this...
    // We could have created a function here as well
    auto compareSites = [] (const uGenericNGS &site1, const uGenericNGS &site2)
                                -> bool
                            { return site1.getStart() < site2.getStart() ||
                                (site1.getStart() == site2.getStart() &&
                                  site1.getEnd() < site2.getEnd()); };

    std::vector<uGenericNGS>::const_iterator smallestSite;
```

```
    experiment.minChrom(
        [&smallestSite, &compareSites] (const uGenericNGSChrom<uGenericNGS>& chrom1,
                                        const uGenericNGSChrom<uGenericNGS>& chrom2)
                -> bool
            {
                auto site1 = chrom1.minSite(compareSites);
                auto site2 = chrom2.minSite(compareSites);
                if (compareSites(*site1, *site2)) {
                    smallestSite = site1;
                    return true;
                }
                else
                    return false;

            });

    smallestSite->writeBedToOuput(std::cout, false);

}
```

In this code, the `smallestSite` variable is shared by the various invocations of the lambda function. Therefore, whenever a calculation changes the value of this variable, another parallel calculation could be changing the same variable to another value **at the same time**. This would mean that if the previous lowest site was A and a calculation is performed on B and C at the same time and that B is lower than C, there is no guarantee that `smallestSite` would equals B. Hence, you must make sure that all the variables you use in your parallel computation are either local to the function or, if they come from an higher level, that they are read-only.

A good tutorial on parallel computing can be found here:
https://computing.llnl.gov/tutorials/parallel_comp/