Services

Namespace: ttgLib::Services

In ttgLib, service is an interface that implements certain functionality. Most of ttgLib features are implemented as services. For example, Logger service is responsible for logging application activity, and Timer service implements time management and time measurement. Every service is a singleton available from static method GetRef() of the service interface. For instance, the call Timer::GetRef()->GetSeconds() returns current time in seconds.

Service starting

Services are initialized with the instantiation of the `ServiceStarter` class that requires one of the preset modes to be passed as an argument. These modes are implemented as static methods of `ServiceConfigPresets` class:

- `BasicConfig()`. Defines the mode that prohibits all incoming connections to ttgLib. It is intended for use when Utility subsystem functionality is not required.
- `LocalUtilityConfig()`. Allows utilities to connect to the executable at runtime, but only if the request is sent from the loopback network interface. This mode is ideal for debugging or to ensure high security level.
- `RemoteUtilityConfig()`. Enables incoming connections from any network accessible computer via utilities or web browser. Port to be used to listen to the connections can be specified as an argument.
- `RemoteWithAuthUtilityConfig()`. This mode is similar to the previous one but it requires HTTP authorization before connection can be established. Login and password are specified as arguments.

Here is the typical code to enable and initialize ttgLib library:

```
#include <ttgLib.h>

using namespace ttgLib::Services;

int main()
{
   ServiceStarter ss(ServiceConfigPresets::BasicConfig());
   //Computing.
}
```

When a specific configuration is required for a particular service, a preset mode can be overridden using the '+' operator and configuration parameters of this service:

```
ServiceStarter ss(ServiceConfigPresets::BasicConfig() + LoggerConfig("MyLog.txt"))
```

The settings that are added as a right operand override the ones on the left. This allows to change any basic mode.

Dynamic parameters

Namespace ttgLib::Parameters

The basic mechanism that allows ttgLib library to integrate into the application being optimized is the dynamic parameters subsystem. These parameters are implemented as template classes and are equivalent to the ordinary variables from the developer's point of view. Dynamic parameters are needed for optimization subsystem to gather statistics and to tune application's performance. Furthermore, the network control subsystem allows to monitor the values of dynamic parameters and even to change their values at runtime from a remote computer using ttgUtils or web browser.

Basic types of parameters

The following basic parameter's types are currently implemented:

– `Parameter<T>` is the basic template class that allows making almost any variable changeable at runtime. It is intended primarily for use when external access to a variable that represents an iterator, an information text, etc. is required. Examples:

```
Parameter<int> imax = 10;
Parameter<double> res = 0.0;
//...
for (int i = 0; i < imax; i++)
   res += arr[i];
Parameter<std::string> out = "Finished";
```

– `BoundedParameter<T>` extends the basic parameter with the boundary support. In other words, the corresponding dynamic variable can hold any value from [lo, hi] interval which is set in the constructor. Use of this parameter for "magic variables" instead of the basic ones speeds up optimization process because hinting the possible values interval reduces the optimization area. Examples:

```
BoundedParameter<int> block_size(16, 32);
block_size = 24;
//...
for (int i = 0; i < size / block_size + 1; i++)
   for (int j = 0; j < block_size; j++)
      //...
```

– `EnumParameter`. This is a version of a dynamic parameter for enumeration. It allows to define the enumeration as a set of elements of the basic type. This set will then be processed in the same way as using the parameter of the `Parameter<T>` type provided that only values from this enumeration can be assigned. This version is intended both for facilitating any manipulations with the software via utilities and for the optimization itself. Examples:

```
enum basicEnum { first, second, third };
//...
EnumParameter en("{ first, second, third }");
en = first;
//...
switch (en)
{
   case first:
      //...
   case second:
      //...
}
```

Enumeration members can be defined in two ways. The first one implies the usage of a string like "`{ value1, value2, value3, valueN }`" where valueX is a string equivalent of the corresponding member. The second method is to create a container `std::vector<T>`, to fill it

in with necessary enumeration members and then to pass it to the parameter constructor. It should be noted that the first method is applicable only when the basic type is the standard one and can be recognized by the string value while the second method is more universal and as such can be also used for user-defined types. Therefore, the above example can be rewritten in the following way:

```
std::vector<float> en1Vals;
en1Vals.push_back(-1.0);
en1Vals.push_back(0.0);
en1Vals.push_back(1.0);
EnumParameter<float> en1(en1Vals);
en1 = 0.0f;

std::vector<int> en2Vals;
en2Vals.push_back(0);
en2Vals.push_back(1);
en2Vals.push_back(2);
EnumParameter<int> en2(en2Vals);
en2 = (int)(en1 + 1.0);
```

It is also worth mentioning that the string names of enumerated elements under which they will be mapped in external utilities can be also reassigned. To this end, a function like `std::string (*NameConverter)(T)` that for a given element returns its mapped name should be passed to the parameter constructor.

– `ActionParameter`. Allows to handle external events that can be initiated both by the optimization subsystem and from the network control subsystem. Examples:

```
void ProcessEvent(Void)
{ printf("Event has been processed\n"); }
//...
ActionParameter action;
action.OnTriggered() += MakeDelegate(ProcessEvent);
//...
action.Trigger();
```

– `ParameterGroup`. Allows to group parameters or other groups for systematization purposes. In optimization process, groups are used to specify parameters that influence performance of the specific kernel. Network control shows each parameter group as a tree node. Examples:

```
Parameter<int> p1;
BoundedParameter<double> p2(0.0, 1.0);
ActionParameter p3;

ParameterGroup group;
p1.Attach(&group);
p2.Attach(&group);
p3.Attach(&group);
```

Complex parameter types

Complex parameter types are built upon the basic ones and implement the most frequently used patterns. These parameters are recognized by network control subsystem to display a specific visualization UI and by optimization subsystem to enhance the optimization process using heuristics based on the type's nature. Currently the following complex parameter type are implemented:

– `GridParameter1D` allows to specify a decomposition of an arbitrary one-dimensional array into unequal parts using dynamic parameters. It is primarily intended to perform a load balancing between non-uniform computational devices and/or nodes. Therefore, an optimal decomposition can be easily revealed by the optimization subsystem similarly with the following example:

```
GridParameter1D grid(N);          //constructing grid with N cells
```

```
grid.SetBoundaries(0, 1024);
grid.SetAlignment(128);
//...
for (int i = 0; i < grid.GetCount(); i++)
   device[i]->compute(grid[i].GetLowerBoundary(), grid[i].GetUpperBoundary());
```

– `CudaGridParameter1D`. Opposite to the previous type `GridParameter1D`, `CudaGridParameter1D` is intended to define such a parameter of starting CUDA-kernels as the number of threads running on a single multiprocessor since this parameter directly impacts on the performance. It is important that the functionality of this parameter can be implemented, for instance, via `EnumParameter<UInt32>`. However, its allowed values strongly depend on GPU architecture. This type has been introduced to facilitate the programming for GPUs.
To get the current number of threads (or the block size), one should call the `size_t GetBlockSize()` method. The returned value will depend on the thread where this method was called. If the thread is associated with a virtual ttgLib device which is always true for hybrid primitives, the method will return the size supported by the device. In other case, the method will call for the properties of CUDA device that has been chosen in the given thread via the cudaSetDevice() method from CUDA Runtime API. For example, if computations on a graphics accelerator are accomplished with CUDA Compute Capability 1.3, the maximum allowed block size equals 512 threads while for an accelerator with Fermi or Kepler architecture, the block size can reach 1024 threads. Examples:

```
CudaGridParameter1D p;
dim3 threads(p.GetBlockSize());
dim3 grid(N / p.GetBlockSize());

someCudaKernel<<<grid, threads>>>();
```

To simplify the usage of this parameter, an auxiliary method `std::pair<size_t, size_t> GetGrid(size_t threadCount)` has been introduced. It allows one to create such a number of blocks of a given size that the total number of their threads will be definitely not less than `threadCount`. Therefore, the previous example can be rewritten in the following way:

```
CudaGridParameter1D p;
dim3 threads(p.GetGrid(N).second);
dim3 grid(p.GetGrid(N).first);

someCudaKernel<<<grid, threads>>>();
```

To change the value of this parameter, one can use either the optimization subsystem that will calculate the best value of the block size or the void `SetNearestValue(size_t blockSize)` method that will assign a correct value which is closest to the required one.

Parameters' functionality

Most of parameters have the following common methods which can be divided into three groups:

– Parameter tree management:
  – `void Attach(const char *name, ParameterGroup *group)`. Adds the parameter to the specified parameter group with the specified name. Parameter can be a member of one group only. If `NULL` is passed as the parameter group pointer the parameter will be added to the root user parameter group.
  – `void Detach()`. Detaches the parameter from the group. Can lead to unpredictable results if the parameter is used in the optimization session.
  – `bool IsAttached()`. Returns `true` if the parameter is attached to any parameter group, or `false` otherwise.

```
ParameterGroup group;
Parameter<int> p1;
Parameter<float> p2;
```

```
p1.Attach(&group);                          //Attaching "p1" to "group".
group.Attach("My group");                   //Creating root-level group.
p2.Attach("My parameter");                  //Creating root-level parameter.
//...
if (p1.IsAttached())
{
   p1.Detach();
   p1.Attach("Another my parameter");
}
```

- Parameters' event handling:
  - `Event<ReleasableObject *> OnRelease()`. The event that is raised when the parameter has been released.
  - `Event<ValueChangedEventArgs<T> > &OnValueChanged()`. The event that is raised whenever the value of the parameter has changed.

```
void OnReleaseHandler(ReleasableObject *)
{ printf("Parameter was destroyed\n"); }

void OnValueChangeHandler(ValueChangedEventArgs<T>)
{ printf("Parameter's value was changed\n"); }

Parameter<int> p;
p.OnRelease() += MakeDelegate(OnReleaseHandler);
p.OnValueChanged() += MakeDelegate(OnValueChangeHandler);
p = 10;
p = 9;
p.OnValueChanged() -= MakeDelegate(OnValueChangeHandler);
p = 8;
```

- Changing parameters' value and behavior
  - `const T &GetValue()`. Returns current parameter value. Equivalent to the implicit cast operator. Intended for the cases when compiler fails to perform an implicit cast. Examples:

```
Parameter<int> p = 3;

printf("%d != 3", p);
printf("%d == 3", p.GetValue());
printf("%d == 3", (int)p);
```

  - `SetValue(const T &value)`. Sets current parameter value. Equivalent to the assignment operator.
  - `ReadOnlyMode GetReadOnlyMode()`. Returns current read/write policy of the parameter:
    - `None`. Parameter value can be changed both by the application itself and by the network control subsystem.
    - `Utilities`. Restricts the network control subsystem from changing the parameter value. Can be useful for the internal variables that need to be visible from the network, but an attempt to change their value will lead to an error.
    - `Core`. Restricts application from changing the parameter value.
    - `All`. Restricts both the application and the network control subsystem from changingthe parameter value.
  - `SetReadOnlyMode(ReadOnlyMode mode)`. Sets current read/write policy of the parameter.

```
Parameter<std::string> currentAction = "None";
currentAction.SetReadOnlyMode(Utilities);
currentAction.Attach("Current action");
```

```
EnumParameter requiredAction("{ Reading, Writing, Processing }");
requiredAction = 0;
requiredAction.SetReadOnlyMode(Core);
requiredAction.Attach("Required action");

while (true)
{
   switch (requiredAction)
   {
      case 0:
         //...
         currentAction = "Reading data";
         break;
      case 1:
         //...
         currentAction = "Writing data";
         break;

      case 2:
         //...
         currentAction = "Processing data";
         break;
   }
}
```

Hybrid primitives

ttgLib::Primitives namespace

To simplify the implementation of various parallel programming patterns, ttgLib library introduces a number of hybrid primitives that allow load balancing between all available computing devices. With these primitives built upon dynamic parameters it is possible to group several alternative kernels implemented using NVIDIA CUDA, OpenCL and/or SSE/AVX extensions. As a result, an optimal kernel will be chosen for every computational device, and the load will be balanced between all devices chosen.

HybridFor

HybridFor primitive implements the parallel_for pattern that allows to replace a single loop with several independent loops each processing its own portion of data on its target computing device. To use the primitive, an instance of the HybridFor class is required which is then used to register all available kernels and to initiate the computation process. Examples:

```
void cudaKernel(void *data, size_t lo, size_t hi)
{ /*Processing data array sub-range using current CUDA-device*/ }

void sseKernel(void *data, size_t lo, size_t hi)
{ /*Processing data array sub-range using SSE extensions*/ }

void sseOmpKernel(void *data, size_t lo, size_t hi)
{ /*Processing data array sub-range using SSE extensions and OpenMP directives*/ }

//...

HybridFor hFor;
hFor.CUDA() += cudaKernel;
hFor.Serial() += sseKernel;
hFor.Parallel() += sseOmpKernel;
std::vector<int> data;
//...
hFor.Process(data);
```

Kernel syntax is described in details in another section. Except for this, the primitive has the following non-general members:

- `SetAlignment(size_t alignment)`. Allows to set the alignment of the intervals passed to each kernel. Intended for use when the technology enforces specific alignment (for example, to enable SSE).
- `SetMaxRange(size_t maxSize)`. *Under development*.

HybridTask **(not available in current version)**

This primitive is intended for launching a set of stand-alone non-uniform tasks when each task can run on one of available units probably with the use of alternative processor cores. For instance, this allows to load all computing devices uniformly when running batch tasks and for each of them not only an optimal kernel will be assigned but tasks for which the device demonstrates the highest performance will be also passed to it. Examples:

```
void cudaKernel(void *data)
{ /*processing data using current CUDA-device*/ }

void cudaKernel_Textures(void *data)
{ /*processing data using current CUDA-device using texture memory*/ }

void cpuKernel(void *data)
{ /*processing data using current CUDA-device*/ }

//...
```

```
HybridTask hTask;
hTask.CUDA() += cudaKernel;
hTask.CUDA() += cudaKernel_Textures;
hTask.Serial() += cpuKernel;
std::vector<void *> data;
//...
hTask.Process(data);
```

A detailed description of the supported kernel is provided in the next Section. However, one particular method should be mentioned for this primitive:

– `SetMetrics(double (*f)(void *))`. Defines a metrics for an abstract task complexity (in relative units). The processing time on the same accelerator is assumed to linearly depend on the tasks complexity thus enabling to more efficiently distribute the computational load between all computing devices and to choose tasks of optimal size for each type of computing devices.

Kernel syntax

To specify a kernel, the developer is required to implement a function or a method that takes a pointer to the data being processed and additionally some primitive-specific data. Arguments of these functions should match one of the following signatures that the primitive supports:

Primitive type	*Supported kernel signatures*

`HybridFor`	`void kernel(void *data, size_t lo, size_t hi);`
	`void kernel(size_t lo, size_t hi);`
	`void kernel(HybridForKernelData data)`

`HybridTask`	`void kernel(void *data);`
	`void kernel();`
	`void kernel(HybridTaskKernelData data)`

If the kernel is implemented as a function, it can be simply added to the primitive with the '+=' operator. If the kernel is a member of a class, it must be first cast to the delegate with `MakeDelegate()` function, and then added with the '+=' operator. Examples:

```
void functionKernel(HybridTaskKernelData data);

class MyClass
{
    public:
        void methodKernel(HybridTaskKernelData data);
        static void staticMethodKernel(HybridTaskKernelData data);
};

//...

HybridTask hTask;
hTask.Serial() += functionKernel;
hTask.Serial() += MakeDelegate(functionKernel);
hTask.Serial() += MakeDelegate(MyClass::staticMethodKernel);
MyClass mc;
hTask.Serial() += MakeDelegate(&mc, &MyClass::methodKernel);
```

Sometimes, there are several dynamic parameters that influence the kernel's performance and that can be optimized. They can be added to a single parameter group that in turn is then passed to an auxiliary class upon registering the kernel. Examples:

```
BoundedParameter<int> blockSize(10, 32);
ParameterGroup kernelParameters;

void kernel(void *data);

//...
```

```
HybridTask hTask;
blockSize.Attach(&kernelParameters);
hTask.Cuda() += HybridTaskKernel(kernel, &kernelParameters);
```

The parameters passed to the primitive will be independently optimized to achieve maximum performance of the current kernel. Due to the fact that one kernel can be executed by several computing devices independently and simultaneously, there will be created a separate instance of each parameter value for each device. Therefore, in the previous example, blockSize will evaluate to 16 in the CPU kernel and to 32 in the GPU kernel.

At runtime, every kernel is executed in a separate thread associated with the target computing device. In case of error, the kernel must throw an exception that looks like or is derived from `std::runtime_error`. This exception is caught in the device thread and re-thrown in the calling thread.

To detect the parameters of computing device the kernel is running on, the following methods are implemented in the `DeviceManager` service:

- `CudaDevice *GetCurrentCudaDevice()`. Returns a pointer to an object that corresponds to the current CUDA device. If this method was called outside the primitive or the `CudaDevice::Execute()` method, an exception will be thrown.
- `X86Device *GetCurrentX86Device()`. Returns a pointer to an object that corresponds to the current x86 compatible device. If this method was called outside the primitive or the `X86Device::Execute()` method, an exception will be thrown.

Examples:

```
void cudaKernel(void *data)
{
   CudaDevice *dev = DeviceManager::GetRef()->GetCurrentCudaDevice();
   double gflops;
   //...
   printf("%s performance: %lf Gflops\n", dev->GetStaticInfo()->GetName(),gflops);
}
```

Choosing the devices

Primitive registration of kernels is based on the APIs they support. That is, if there is a CPU and NVIDIA GPU, ttgLib will discover four computational devices: 1 x86 compatible, 2 OpenCL compatible and 1 CUDA compatible. If several kernels suit to one physical device, the one demonstrating the best performance will be chosen.

The following logical devices are supported when kernels are added to the primitives:

- `CUDA()`. Stands for GPU that supports NVIDIA CUDA technology.
- `Parallel()`. Enables to set kernel that is capable of using all the cores of CPU (for example, with OpenMP technology).
- `Serial()`. Addressed to the kernels that use only one CPU core and support multiple instances to be run.

It should be noted that in most cases the use of serial kernels is more efficient than that of the parallel ones. The reason is that in the first case, ttgLib library can sacrifice some CPU resources to increase the efficiency of CUDA driver. Furthermore, serial kernels improve the performance on processor units that support HyperThreading technology due to giving up virtual resources. In other words, if not using all existing CPU cores is optimal, ttgLib library will surely use this during optimization.

Another option is defining constraints for computing devices supported by computational kernel. To this end, a delegate-selector that returns `true` if the kernel can be processed on a particular device and `false` otherwise should be passed to a corresponding primitive method. Examples:

```
bool cudaSelector(CudaStaticInfo *info)       //we want only Fermi architecture
{ return info->GetMajor() >= 2; }
```

```
void cudaKernel(void *data);

//...

HybridTask hTask;
hTask.CUDA(MakeDelegate(cudaSelector)) += cudaKernel;
```

The following predefined selectors are provided:

- `X86Memory(size_t megabytes)`. Defines the (common) memory size on a x86 compatible device that is required by the kernel.
- `CudaCaps(size_t major, size_t minor)`. Defines minimal requirements to NVIDIA CUDA architecture from the kernel.
- `CudaMemory(size_t megabytes)`. Defines the (common) memory size on a CUDA accelerator that is required by the kernel.

Therefore, the abovementioned example could be rewritten as:

```
void cudaKernel(void *data);

//...

HybridTask hTask;
hTask.CUDA(CudaCaps(2, 0)) += cudaKernel;
```

Optimization subsystem

Namespace ttgLib::Optimization

A key subsystem of ttgLib library is the optimization engine that allows to choose optimal values for an arbitrary group of dynamic parameters and/or hybrid primitives thus delivering maximum performance. It is assumed that computations are iterative and at each step, the computational complexity depends linearly on the processing data size.

Implementation scheme

To activate the optimization engine, an instantiation of the `OptimizationSession` class should be created and a group of optimized parameters should be passed to this class. Then, a method `OptimizationSession::StartIteration()` should be called at the beginning of each iteration, and a method `OptimizationSession::FinishIteration()` should be called at its end. Examples:

```
BoundedParameter<int> blockSize(16, 128);
ParameterGroup group;
blockSize.Attach(group);

OptimizationSession session(&group);
while (true)
{
    session.StartIteration();
    //Computing using "blockSize".
    session.FinishIteration();
}
```

If hybrid primitives are used, the method `GetOptimizationParameters()` implemented in them should be called. This method returns all parameters used in the primitive. Examples:

```
HybridFor hFor;
hFor.Serial() += serialKernel;
hFor.CUDA() += cudaKernel;

OptimizationSession session(hFor.GetOptimizationParameters());
while (true)
{
    session.StartIteration();
    hFor.Process(data);
    session.FinishIteration();
}
```

It should be noted that during the optimization, all primitives, groups of parameters and separate parameters should not be deleted or assigned to other groups. Otherwise the optimization session will be 'broken' and any attempt to call its methods will lead to an exception of the ttgLib::FatalError type.

Optimization sessions

In the `OptimizationSession` class, the following methods designed to control the optimization process are implemented:

- `void StartIteration()`. Informs about the start of a new iteration.
- `void FinishIteration()`. Informs about the iteration completion. The time between the start and the end of an iteration will be used as a measure for optimization of common parameters.
- `void FinishIteration(double time)`. Informs about the iteration completion. The passed time value will be used for optimization of common parameters.
- `void Suspend()`. 'Suspends' the optimization process thus enforcing the parameters to remain constant while the statistics can still be gathering.

- `void Resume()`. Resumes the optimization process. Current parameters' values will not be taken into account during further optimization.
- `bool IsSuspended()`. Returns `true` if the optimization process is suspended and `false` otherwise.
- `void Restart()`. Restarts the optimization process and deletes all gathered statistics. It makes sense to call this method when the type of computations should be changed or when optimization parameters should be added or deleted manually.
- `void Append(ParameterBase *p)`. Appends one optimization parameter to the current session. Causes a reload of optimization process.
- `void Append(ParameterGroup *g)`. Appends all parameters from a given group to the current session. Causes a reload of optimization process.

Examples:

```
OptimizationSession os;

EnumParameter ep("Left, Right, Center");
os.Append(&ep);

BoundedParameter<int> bp(10, 20);
os.Append(&bp)

HybridTask hTask;
hTask.Serial() += serialKernel;
os.Append(hTask.GetOptimizationParameters());

//...

int iterCount = 0;

while (true)
{
    os.StartIteration();

    //Computing.
    hTask.Process(data);
    ComputeSomethingUsingParameters(ep, bp);

    //Finishing statistics gathering and stopping optimization.
    if (iterCount == 100)
        os.Suspend();

    //Restarting optimization in order to ensure, that all is good.
    if (iterCount == 5000)
    {
        os.Restart();
        os.Resume();
        iterCount = 0;
    }

    os.FinishIteration();
    iterCount++;
}
```

Optimization strategies

In most cases, there are some *a priori* information about the application to be optimized. It can be the expected number of iterations, an approximate time of a single iteration, or the time required the application performance to reach its stable level. Such an information is useful in choosing the exact optimization strategy and, as a consequence, in improving the quality and the rate of algorithm learning.

To formalize the optimization strategy, a class `OptimizationStrategy` should be instantiated with proper configuration parameters and algorithms. By default, the most versatile strategy is used, i.e. the one designed for at least 200 iterations with minimum duration of 5 ms for each.

General parameters of optimization strategies are defined with the following methods:

- `Parameters::Parameter<UInt32> &AggressiveIterationCount()`. Returns a parameter that defines the number of 'aggressive' iterations when the algorithm is allowed to substantially decrease application performance to search for a global optimum. Its recommended values range from 20 to 200 depending on the number of iterations and the optimized parameters.
- `Parameters::Parameter<UInt32> &StabilizationIterationCount()`. Determines the number of iterations that should be omitted before beginning the optimization itself to stabilize the application performance. This method is extremely helpful for applications that use CPU and multiple GPUs simultaneously. Its recommended values range from 0 to 10 depending on the number of computing devices.
- `Parameters::Parameter<Double> &IterationAggregation()`. Returns a parameter that defines the time of shortest iteration. If iteration duration appears to be less than required, actual iterations will be grouped into logical ones with duration higher than the preset threshold. This method is designed for optimization of applications with 1 to 5 ms iterations since it allows to mitigate the influence of overheads and performance abrupt changes.
  This parameter is recommended to be set less than 10 ms. It should be also taken into account that optimization will be based on logical iterations. Therefore, larger groups of iterations can result in a longer algorithm learning curve.
- `Parameters::EnumParameter &BasicAlgorithm()`. Returns a list that defines an algorithm used for optimization of an arbitrary group of parameters for which there is no additional information. Supporting values are `BasicAlgorithmType::Default`, `BasicAlgorithmType::Clicking` and `BasicAlgorithmType::Genetic`.
- `Parameters::EnumParameter &BalancingAlgorithm()`. Defines the type of the algorithm to be used for optimizing the group of parameters that allow load balancing (in current version of ttgLib it is the HybridFor primitive). Supporting values are `BalancingAlgorithmType::Default`, `BalancingAlgorithmType::Sticky` and `BalancingAlgorithmType::LUT`.

To implement the required optimization strategy, it should be sent to the optimization session constructor. Examples:

```
OptimizationStrategy strategy;
strategy.AggressiveIterationCount() = 100;
strategy.IterationAggregation() = 0.01;
strategy.BasicAlgorithm() = BasicAlgorithmType::Clicking;
strategy.BalancingAlgorithm() = BalancingAlgorithmType::LUT;

OptimizationSession os(strategy);

//...
```

Basic optimization algorithms

These algorithms are intended for optimization of arbitrary parameters with no *a priori* information. For instance, algorithms of this group will be used for any parameters passed for optimization outside the domain of hybrid primitives.

Up to date, the following two algorithms of this type have been implemented:

- `Clicking`. This is a modified version of the alternating-variable descent method. In 'aggressive' mode, it searches through all combinations of parameters sorted by descending performance. In basic mode, this algorithm tries to remain in the revealed optimum.
  This algorithm is useful when the number of optimized parameters is small (from one to four) or the number of 'aggressive' iterations is less than 500 since in these cases, it provides the fastest learning curve and requires a minimal number of measurements.
  In optimization strategy, the following settings can be defined for this algorithm (through the `ClickingAlgorithmSettings()` method):
  - `Parameters::BoundedParameter<double> &Quality()`. This is a normalized

parameter with values from 0.0 to 1.0 that determines the optimization quality. The higher the optimization quality the lower the learning rate and vice versa.

- `Genetic`. This is an implementation of genetic algorithm. In 'aggressive' mode, it generates active mutations and crossing-overs of the best sets of parameters. In basic mode, this algorithm selects the best sets of parameters from remaining 'population'.
  This algorithm is intended for time-consuming computations that have more than 1,000 aggressive iterations but relatively smooth performance dynamics. It is also useful for optimization of large sets of parameters (more than 10).
  In optimization strategy, the following settings can be defined for this algorithm (through the `GeneticAlgorithmSettings()` method):
  - `Parameters::BoundedParameter<double> &Quality()`. This is a normalized parameter with values from 0.0 to 1.0 that determines the optimization quality (by varying the size of population and the number of mutations). The higher the optimization quality the lower the learning rate and vice versa.

Examples:

```
OptimizationStrategy strategy;
strategy.BasicAlgorithm() = BasicAlgorithmType::Genetic;
strategy.GeneticAlgorithmSettings().Quality() = 0.42;

OptimizationSession os(strategy);

//...
```

Load balancing algorithms

These algorithms are intended for balancing the computational load between various processor units. In particular, the primitive HybridFor is typically optimized based on these algorithms.

Up to date, the following two algorithms of this type have been implemented:

- `Sticky`. The main algorithm that schedules computational load and the set of computing kernels according to the current performance of each unit. It has two threshold values the first of which determines a minimal data set that could be sent to each computing device while the second value determines the largest data set that causes the task to be processed entirely by a single device.
  This algorithm is especially useful when there are multiple computing devices of similar architecture and/or there is no time for additional learning.

  In optimization strategy, the following settings can be defined for this algorithm (through the `StickyAlgorithmSettings()` method):
  - `Parameters::BoundedParameter<double> &ExcludeThreshold()`. This is a normalized parameter with values from 0.0 to 1.0 that determines the low 'sticky' threshold. For instance, if the threshold equals 0.2 and 15% of all data should be sent to a single device, the destination device will not be in use. The recommended value of this parameter is 0.1.
  - `Parameters::BoundedParameter<double> &ExclusiveThreshold()`. This is a normalized parameter with values from 0.0 to 1.0 that determines the high 'sticky' threshold. For instance, if the threshold equals 0.8 and 90% of all data should be sent to a single device, all processing will be implemented on the target device. The recommended value of this parameter is 0.9.

- `LUT`. This algorithm builds the performance functions for each computing device depending on the data size. This enables to take into account the effects of cache as well as the abrupt changes in performance.
  This algorithm is useful when computing devices have different architecture and/or the number of alternative computing devices is high enough (more than two threads per device).

  In optimization strategy, the following settings can be defined for this algorithm (through the `LutAlgorithmSettings()` method):
  - `Parameters::BoundedParameter<double> &ExcludeThreshold()`. This is a normalized

parameter with values from 0.0 to 1.0 that determines the low 'sticky' threshold. For instance, if the threshold equals 0.2 and 15% of all data should be sent to a single device,the target device will not be in use. The recommended value of this parameter is 0.1.

– `Parameters::BoundedParameter<UInt32> &Granularity()`. This parameter with values from 1 to 100 determines the number of points used to build the performance function. The recommended value of this parameter is 10. The more the number of points the more the required number of aggressive iterations.

Examples:

```
OptimizationStrategy strategy;
strategy.GetBalancingAlgorithm() = BalancingAlgorithmType::LUT;
strategy.GetLutAlgorithmSettings().ExcludeThreshold() = 0.1;
strategy.GetLutAlgorithmSettings().Granularity() =  5;

OptimizationSession os(strategy);

//...
```