ParadisEO-PEO - Lesson 3

Insular Model. Synchronous & Asynchronous Migrations

Introduction

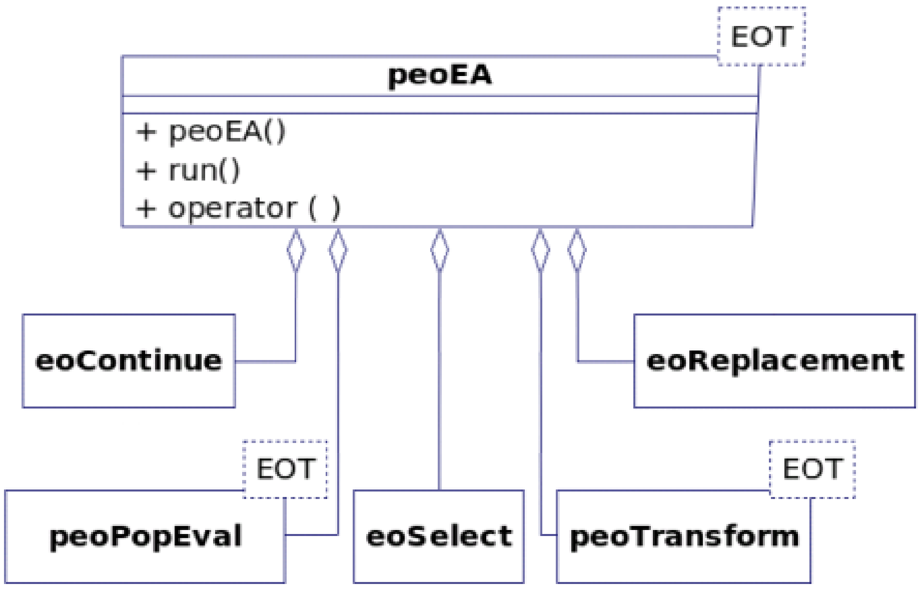
In the previous lessons strategies allowing for parallel evaluation of the population or parallel fitness function were presented. The included source code examples considered only one evolutionary algorithm per application. While this model can be easily extended to include multiple evolutionary algorithms (hence having the possibility to construct multi-start methods), no real interest in this approach exists as there would be no interaction between the concurrently executing algorithms. A more interesting scenario considers co-evolving algorithms disposed according to a given topology and which periodically exchange information in order to coordinate for improved convergence.

While offering the possibility of designing advanced algorithmic architectures, co-evolving algorithms also rise several parametrization issues. The questions one has to ask relate to information exchange flow, parametrization of each algorithm as part of a composite algorithm and the interaction pattern between the algorithms.

The **ParadisEO-PEO**
framework offers the required components for building in a facile manner a ring of evolutionary algorithms which exchange individuals across periodic migrations. Thus, insular models may be envisaged by grouping multiple algorithms as being part of the same topology. The afferent classes are the **RingTopology** class which offers the basis for ring communication models and the **peoSyncIslandMigration/peoAsyncIslandMigration** classes which offer the support for (a)synchronous migrations.

Considering the signature of the **peoEA** constructor, we will focus our attention on the **eoContinue** class - the migration operator is called at the end of each generation of an evolutionary algorithms as a checkpoint object:

```
peoEA(
        eoContinue< EOT >& __cont,

        peoPopEval< EOT >& __pop_eval,

        eoSelect< EOT >& __select,

        peoTransform< EOT >& __trans,

        eoReplacement< EOT >& __replace

);
```

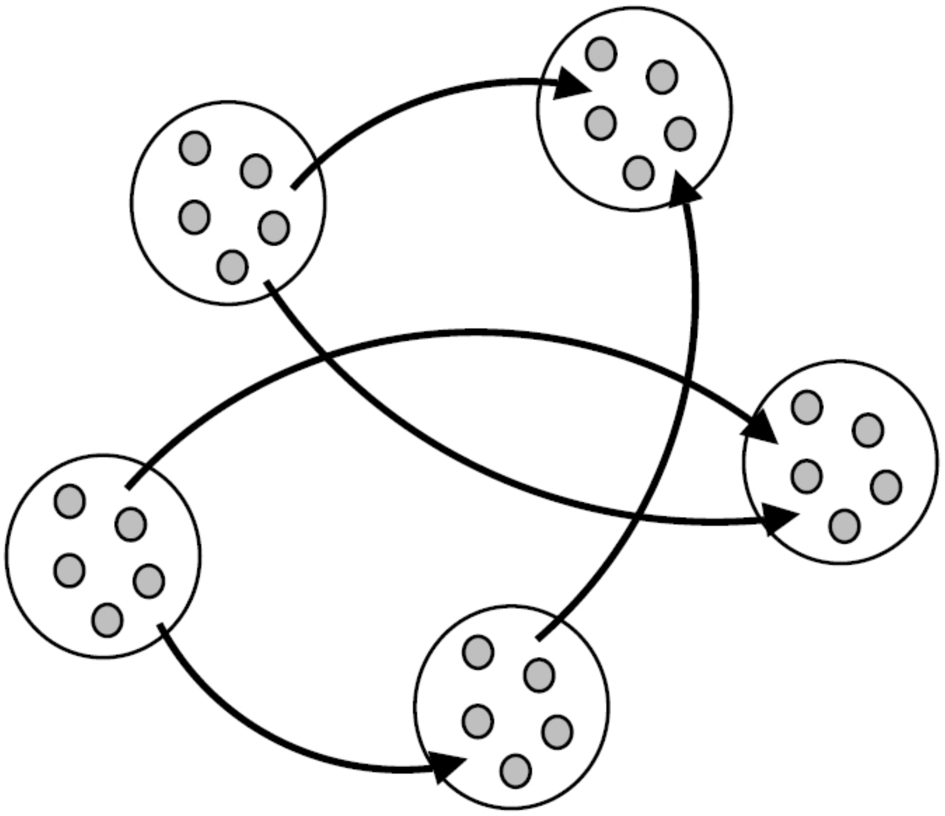


In order to fully understand the insular and the migration model you should completely understand the checkpointing mechanism. A brief pseudo-code description of a ParadisEO-PEO evolutionary algorithm is offered bellow - the entire source code for the ParadisEO-PEO evolutionary algorithm is defined in the **peoEA.h** header file. A more complete presentation of the **peoEA** class and the components of a PEO evolutionary algorithm may be found in **Lesson 1**.

```
do
{
    select( population, offsprings );     // select the offsprings from the current population
    transform( offsprings );               // crossover and mutation operators are applied on the selected
                                           offsprings
    evaluate( offsprings );                // evaluation step of the resulting offsprings
    replace( population, offsprings );     // replace the individuals in the current population whith
                                           individuals from the offspring population, according to a
                                           specified replacement strategy
}   while ( eaCheckpointContinue( population
    ) );
```

The checkpoint objects are called at the end of each iteration - for the above pseudo-code, the **eaCheckpointContinue**
may correspond to multiple checkpoints. For the migration model, the checkpoint may result, depending on the parametrization, in emigration/immigration of the individuals from/towards the current population of the algorithm.

A schematic representation of an arbitrary topological model with migrations between the *islands* is offered bellow:



Let's consider the following scenario - emigrations/immigrations should occur at every ten generations. In addition we would like to have control on selecting the individuals to emigrate as well as on the replacement process of the current individuals with the immigrant ones. In other words, constructing an insular migration model consists in (1) having a ring topological model including several evolutionary algorithms, (2) defining the migration frequency as well as the size of the migration, *i.e.* the number of individuals that emigrate and (3) the selection and replacement strategies for selecting the emigrant individuals and for integrating the immigrant ones, respectively.

NOTE: All the presented examples have as case study the Traveling Salesman Problem (TSP). All the presented tutorials rely on a common shared source code defining transformation operators, evaluation functions, etc. for the TSP problem. For a complete understanding of the presented tutorials please take your time for consulting and for studying the additional underlying defined classes.

Synchronous & Asynchronous Migrations

The source code for this example may be found in the main.cpp file, under the
paradiseo-peo/tutorial/Lesson3
directory. Please make sure you are under the right directory (!!). At this point you have two options: (a) you can just follow the example without touching the main.cpp or, (b) you can start from the **basic sequential program**
presented in Lesson1, following the below exposed steps, in which case you are required to make a backup copy of the Lesson3 main.cpp file (the one located in the **paradiseo-peo/tutorial/Lesson3** directory).

For starting, please make sure you are under the right directory and that your main.cpp file source code is identical to the one presented below. You can copy the main.cpp file from **paradiseo-peo/tutorial/Lesson1** overwriting the main.cpp file located in the **paradiseo-peo/tutorial/Lesson3** directory or you can simply copy and paste the bellow presented code. While the herein example should guide you through all the required steps for building the complete program, for safety reasons you are strongly encouraged to make a backup copy of the already existing main.cpp file.

```
#include "route.h"
#include "route_init.h"
#include "route_eval.h"

#include "order_xover.h"
#include "city_swap.h"

#include "param.h"

#include <paradiseo>


#define POP_SIZE 10
#define NUM_GEN 100
#define CROSS_RATE 1.0
#define MUT_RATE 0.01


int main( int __argc, char** __argv ) {

        // initializing the ParadisEO-PEO environment
        peo :: init( __argc, __argv );

        // processing the command line specified parameters
        loadParameters( __argc, __argv );

        // init, eval operators, EA operators --------------------------------------------

        // random init object - creates random Route objects
        RouteInit route_init;
        // evaluator object - offers a fitness value for a specified Route object
        RouteEval full_eval;

        // crossover operator - creates two offsprings out of two specified parents
        OrderXover crossover;
        // mutation operator - randomly mutates one gene for a specified individual
        CitySwap mutation;
        // ---------------------------------------------

        // evolutionary algorithm components --------------------------------------------

        // initial population for the algorithm having POP_SIZE individuals
        eoPop< Route > population( POP_SIZE, route_init );
        // evaluator object - to be applied at each iteration on the entire population
        peoSeqPopEval< Route > eaPopEval( full_eval );

        // continuation criterion - the algorithm will iterate for NUM_GEN generations
        eoGenContinue< Route > eaCont( NUM_GEN );
        // checkpoint object - verify at each iteration if the continuation criterion is met
        eoCheckPoint< Route > eaCheckpointContinue( eaCont );

        // selection strategy - applied at each iteration for selecting parent individuals
        eoRankingSelect< Route > selectionStrategy;
        // selection object - POP_SIZE individuals are selected at each iteration
        eoSelectNumber< Route > eaSelect( selectionStrategy, POP_SIZE );

        // transform operator - includes the crossover and the
        // mutation operators with a specified associated rate
        eoSGATransform< Route > transform( crossover, CROSS_RATE, mutation, MUT_RATE );
        // ParadisEO transform operator (peo prefix) - wraps an e EO transform object
        peoSeqTransform< Route > eaTransform( transform );

        // replacement strategy - for replacing the initial
        // population with offspring individuals
        eoPlusReplacement< Route > eaReplace;
        // ---------------------------------------------

        // ParadisEO-PEO evolutionary algorithm -------------------------------------------

        peoEA< Route > eaAlg( eaCheckpointContinue, eaPopEval, eaSelect, eaTransform, eaReplace );

        // specifying the initial population for the algorithm, to be iteratively evolved
        eaAlg( population );
        // ---------------------------------------------


        peo :: run( );
        peo :: finalize( );
        // shutting down the ParadisEO-PEO environment

        return 0;
}
```

1. **Necessary header files**

Please notice the **#include <paradiseo>** line in the header inclusion part of the main.cpp file. The **paradiseo** header file is in fact a "super-header" **-** it includes all the essential ParadisEO-PEO header files. It is at at your choice if you want use the paradiseo header file or to explicitly include different header files, like the peoEA.h header file, for example.

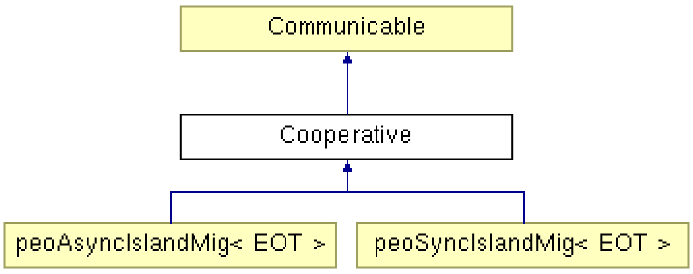
```
...
#include <paradiseo>
...
```

For the herein example, the following header files are required:

- **ring_topo.h -** provides the **RingTopology** class which defines a ring topological model thus facilitating the construction of an insular model where migrations occur in a *ring* (for each island there is only one immigration source and only one emigration destination);

- **peoSyncIslandMig.h** - class providing a *synchronous migration object* (**peoSyncIslandMig**) - allows for the specification of different migration related parameters (frequency, size of the migration, selection and replacement strategies, topological model, source and destination population). Migrations occur in synchronized manner at the end of each generation of each of the involved evolutionary algorithms;

- **peoAsyncIslandMig.h** - class providing an *asynchronous migration object* (**peoAsyncIslandMig**) - allows for the specification of different migration related parameters (frequency, size of the migration, selection and replacement strategies, topological model, source and destination population). As opposed to **peoSyncIslandMig**, no synchronization exists between the evolutionary algorithms, migrations occurring in a decoupled manner.



All the above header files are already provided by including the **paradiseo** file, hence, no modifications are required at this point.

NOTE: The link between multiple evolutionary algorithms is given by the topology that bounds them, all the implemented algorithms sharing the same topology object, as we will see in the following steps. In addition, in order to specify that a migration object is bound to an evolutionary algorithm we need to specify the algorithm as *owner*
of the migration object. We will explore these concepts step by step while constructing the example source code.

2. **Migration Frequency & Size**

We have to define the frequency of the migrations, *i.e.* how often the migrations will occur, as well as the size of the migrations - how many individuals will migrate. In order to facilitate later modifications, you should define two constants as follows:

```
...
#define MUT_RATE 0.01

#define MIG_FREQ 10
#define MIG_SIZE 5
...
```

For the herein example we chose to have five individuals to emigrate at each ten generations. You may experiment by choosing different values with the only constraint of not having more individuals that migrate than the population itself. Also please note that we chose the above values on empirical basis for exemplification purposes only.

2. **Migration Topology**

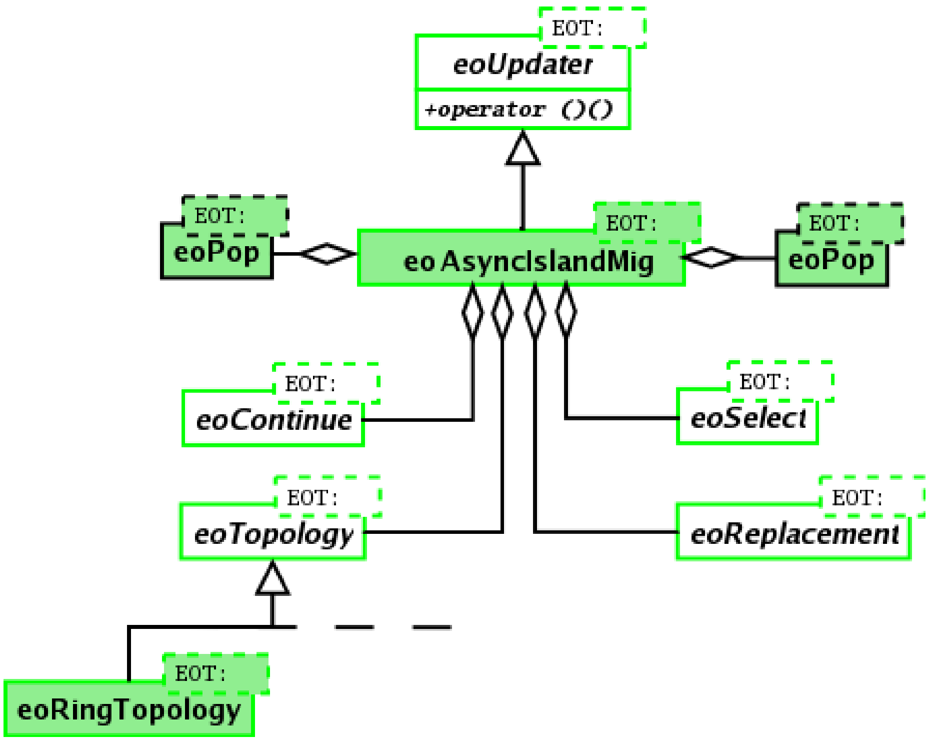
At this time ParadisEO-PEO provides support for ring topologies only - the associated class may stand as base for developing more complex problem-specific topologies. For the moment we will use the already provided class with no further modifications. In your code add the following line:

```
...
RingTopology topology;
...
```

This line should precede the instantiation of any of the evolutionary algorithms you would like to link in the ring as it has to be later specified as topology (passed as parameter) for each the algorithms.

3. **Migration Strategy**

Defining the migration strategy for either the synchronous or the asynchronous model relates to defining each of the components required by the **peoSyncIslandMig/peoAsyncIslandMig** classes - please refer to the bellow schematic representation for an overview:



At this point you already have the migration topology. Also the migration frequency has been specified as a constant - to be wrapped as a continuation criterion object (an **eoContinue** derived object). As we would like to have control on the individuals that migrate we can specify a selection strategy and, in addition, a replacement strategy specifying how to integrate the immigrant individuals into the target population.

Two populations have to be specified - the source population where from emigrants are selected and the destination one in which the immigrants are integrated. In most cases, for the algorithms you develop, the source and the destination population will be the same. This is equivalent to selecting the individuals to emigrate from the current population of an evolutionary algorithm, sending them to another island, while integrating the incoming individuals in this same population. This mechanism may assure for fine diversity control.

In order to specify the entire migration strategy defined, add the following lines between the line were you defined the topology model and the line were you instantiate your evolutionary algorithm:

```
...
eoPeriodicContinue< Route > mig_cont( MIG_FREQ );       // migration occurs periodically

eoRandomSelect< Route > mig_select_one;                 // emigrants are randomly selected
eoSelectNumber< Route > mig_select( mig_select_one, MIG_SIZE );

eoPlusReplacement< Route > mig_replace;                 // immigrants replace the worse individuals

peoSyncIslandMig< Route > mig( MIG_FREQ, mig_select, mig_replace, topology, population, population );
//peoAsyncIslandMig< Route > mig( mig_cont, mig_select, mig_replace, topology, population, population );

eaCheckpointContinue.add( mig );
...
```

NOTE: in the above code you have the options to opt between a synchronous or an asynchronous migration model - you only have to comment/uncomment the corresponding lines (peoSyncIslandMig/peoAsyncIslandMig, respectively). Only one of the migration policies should be defined at a time.

4. **Owner of the Migration Strategy**

Now that we defined all the required components we only have to create a link between the migration policy we defined and an evolutionary algorithm - in other words, we will specify the evolutionary algorithm as owner for the migration object:

```
...
peoEA< Route > eaAlg( eaCheckpointContinue, eaPopEval, eaSelect, eaTransform, eaReplace );

mig.setOwner( eaAlg );

eaAlg( population );
...
```

5. **Creating additional evolutionary algorithms**

The code of the example till this point defines only one evolutionary algorithm with its associated migration object. In order to have multiple co-evolving interacting algorithms, you have to define more than one algorithm each of them with its own strategies and its own migration object. The only common point between all the defined algorithms should be the topological model defined by the **topology** variable. You can simply duplicate the code for constructing more than one algorithm while taking care not to have variable name conflicts. For our example we will limit to only two evolutionary algorithms - you may extend the example to include more algorithms. After adding/duplicating the required code, your source code should look as follows:

```
#include "route.h"
#include "route_init.h"
#include "route_eval.h"

#include "order_xover.h"
#include "city_swap.h"

#include "param.h"


#include <paradiseo>


#define POP_SIZE 10
#define NUM_GEN 100
#define CROSS_RATE 1.0
#define MUT_RATE 0.01

#define MIG_FREQ 10
#define MIG_SIZE 5


int main( int __argc, char** __argv ) {

        // initializing the ParadisEO-PEO environment
        peo :: init( __argc, __argv );


        // processing the command line specified parameters
        loadParameters( __argc, __argv );


        // init, eval operators, EA operators ----------------------------

        // random init object - creates random Route objects
        RouteInit route_init;
        // evaluator object - offers a fitness value for a specified Route object
        RouteEval full_eval;

        // crossover operator - creates two offsprings out of two specified parents
        OrderXover crossover;
        // mutation operator - randomly mutates one gene for a specified individual
        CitySwap mutation;
        // --------------------------------------------------------------


        // evolutionary algorithm components ----------------------------

        // initial population for the algorithm having POP_SIZE individuals
        eoPop< Route > population( POP_SIZE, route_init );
        // evaluator object - to be applied at each iteration on the entire population
        peoParaPopEval< Route > eaPopEval( full_eval );

        // continuation criterion - the algorithm will iterate for NUM_GEN generations
        eoGenContinue< Route > eaCont( NUM_GEN );
        // checkpoint object - verify at each iteration if the continuation criterion is met
        eoCheckPoint< Route > eaCheckpointContinue( eaCont );
        // selection strategy - applied at each iteration for selecting parent individuals
        eoRankingSelect< Route > selectionStrategy;
        // selection object - POP_SIZE individuals are selected at each iteration
        eoSelectNumber< Route > eaSelect( selectionStrategy, POP_SIZE );

        // transform operator - includes the crossover and the mutation operators with a specified associated rate
        eoSGATransform< Route > transform( crossover, CROSS_RATE, mutation, MUT_RATE );
        // ParadisEO transform operator (please remark the peo prefix) - wraps an e EO transform object
        peoSeqTransform< Route > eaTransform( transform );

        // replacement strategy - for replacing the initial population with offspring individuals
        eoPlusReplacement< Route > eaReplace;
        // --------------------------------------------------------------
        RingTopology topology;

        // migration policy and components -------------------------------
        // migration occurs periodically
        eoPeriodicContinue< Route > mig_cont( MIG_FREQ );

        // emigrants are randomly selected
        eoRandomSelect< Route > mig_select_one;
        eoSelectNumber< Route > mig_select( mig_select_one, MIG_SIZE );

        // immigrants replace the worse individuals
        eoPlusReplacement< Route > mig_replace;

        peoSyncIslandMig< Route > mig( MIG_FREQ, mig_select, mig_replace, topology, population, population );
        //peoAsyncIslandMig< Route > mig( mig_cont, mig_select, mig_replace, topology, population, population );

        eaCheckpointContinue.add( mig );
        // ---------------------------------------------------------------


        // ParadisEO-PEO evolutionary algorithm --------------------------

        peoEA< Route > eaAlg( eaCheckpointContinue, eaPopEval, eaSelect, eaTransform, eaReplace );

        mig.setOwner( eaAlg );

        // specifying the initial population for the algorithm, to be iteratively evolved
        eaAlg( population );
        // ---------------------------------------------------------------


        // evolutionary algorithm components -----------------------------

        // initial population for the algorithm having POP_SIZE individuals
        eoPop< Route > population2( POP_SIZE, route_init );
        // evaluator object - to be applied at each iteration on the entire population
        peoParaPopEval< Route > eaPopEval2( full_eval );

        // continuation criterion - the algorithm will iterate for NUM_GEN generations
        eoGenContinue< Route > eaCont2( NUM_GEN );
        // checkpoint object - verify at each iteration if the continuation criterion is met
        eoCheckPoint< Route > eaCheckpointContinue2( eaCont2 );

        // selection strategy - applied at each iteration for selecting parent individuals
        eoRankingSelect< Route > selectionStrategy2;
        // selection object - POP_SIZE individuals are selected at each iteration
        eoSelectNumber< Route > eaSelect2( selectionStrategy2, POP_SIZE );

        // transform operator - includes the crossover and the mutation operators with a specified associated rate
        eoSGATransform< Route > transform2( crossover, CROSS_RATE, mutation, MUT_RATE );
        // ParadisEO transform operator (please remark the peo prefix) - wraps an e EO transform object
        peoSeqTransform< Route > eaTransform2( transform2 );

        // replacement strategy - for replacing the initial population with offspring individuals
        eoPlusReplacement< Route > eaReplace2;
        // ---------------------------------------------------------------


        // migration policy and components -------------------------------

        // migration occurs periodically
        eoPeriodicContinue< Route > mig_cont2( MIG_FREQ );

        // emigrants are randomly selected
        eoRandomSelect< Route > mig_select_one2;
        eoSelectNumber< Route > mig_select2( mig_select_one2, MIG_SIZE );

        // immigrants replace the worse individuals
        eoPlusReplacement< Route > mig_replace2;

        peoSyncIslandMig< Route > mig2( MIG_FREQ, mig_select2, mig_replace2, topology, population2, population2 );
        //peoAsyncIslandMig< Route > mig2( mig_cont2, mig_select2, mig_replace2, topology, population2, population2 );

        eaCheckpointContinue2.add( mig2 );
        // ---------------------------------------------------------------


        // ParadisEO-PEO evolutionary algorithm --------------------------

        peoEA< Route > eaAlg2( eaCheckpointContinue2, eaPopEval2, eaSelect2, eaTransform2, eaReplace2 );

        mig2.setOwner( eaAlg2 );

        eaAlg2( population2 );  // specifying the initial population for the algorithm, to be iteratively evolved
        // ---------------------------------------------------------------

        peo :: run( );
        peo :: finalize( );
        // shutting down the ParadisEO-PEO environment

        return 0;
}
```

Compilation and execution

Your file should be called **main.cpp**
- please make sure you do not rename the file (we will be using a pre-built makefile, thus you are required not to change the file names). Please make sure you are in the **paradiseo-peo/tutorial/Lesson3** directory - you should open a console and you should change your current directory to the one of Lesson3.

Compilation: being in the **paradiseo-peo/tutorial/Lesson3 directory**, you have to type *make*. As a result the **main.cpp** file will be compiled and you should obtain an executable file called **tspExample**. If you have errors, please verify any of the followings:

- you are under the right directory - you can verify by typing the *pwd* command - you should have something like **.../paradiseo-peo/tutorial/Lesson3**
- you saved your modifications in a file called **main.cpp**, in the **paradiseo-peo/tutorial/Lesson3** directory
- there are no differences between the above presented source code and your file.

NOTE: in order to successfully compile your program you should already have installed an MPI distribution in your system.

XML Resource Mapping File

When launching multiple parallel processes, we should specify which process(es) will execute the evolutionary algorithm(s). In addition, one of the processes should execute as a scheduler, having the role of receiving tasks from the *master* processes and (re)distributing them to the *slave* processes.

Considering the case of the **MPI**
standard, which we will not discuss as being out of scope, each computing resource is identified by a *rank*. Thus, assuming you have, for example two bi-processor machines, in order to fully exploit the available computing capacity, you should be launching four parallel processes. Each of the four processes will be ranked by MPI from **0** to **3**
- for the general case, one process per processor. In this scenario, one might imagine having a scheduler process assigned to the first processor, an evolutionary algorithm executing on the second processor while the last two processors may be used for executing *slave* processes. This corresponds to specifying that the *rank 0* process will be executed as scheduler, the *rank 1* as master and the *rank 2* and *rank 3* as slave processes. Furthermore, more than one evolutionary algorithm may be specified, to be executed in parallel on the same node or on different nodes. The following XML file exemplifies this mapping structure:

```
<?xml version="1.0"?>

<schema>
        <group scheduler="0">
                <node name="0" num_workers="0">
                </node>

                <node name="1" num_workers="0">
                <runner>1</runner>
                <runner>2</runner>
                </node>

                <node name="2" num_workers="1">
                </node>
                <node name="3" num_workers="1">
                </node>
        </group>
</schema>
```

NOTE: As we defined two evolutionary algorithms, there are two **<runner>...</runner>** entries, each of them corresponding to one of the evolutionary algorithms. This stands for nothing more than saying that we want to execute both algorithms on the first node. For more than two algorithms you may extend the above file accordingly by adding more **<runner>...</runner>** entries, the ID's of the algorithms ranging from 1 to n, where n is the total number of algorithms.

Each of the mentioned roles - scheduler, master, slave - may be specified by employing different attributes. The *scheduler* node, in the above example, is specified by the **<group scheduler="0">** tag, where **0** indicates the rank of the process to act as scheduler. For each of the processes to be launched, a **<node ...></node>** entry has to be specified. The **name** attribute specifies the rank of the process while the **num_workers** attribute specifies whether the process should act as *master* or *slave*, for **num_workers="0"** and **num_workers="1"**, respectively.

For this specific example, including an additional computing resource (and hence scaling up the application by launching more processes) might require you to add new **<node ...></node>** entries. You may try to experiment with different configurations by adding or removing entries or by simply changing roles. The only constraint is launching as many processes as many *nodes* specified in the XML file.

NOTE: You are not required to have at your disposition two bi-processor machines in order to execute the example. You should be able to run the executable even on only one machine, with no other modifications.

Execution:

The execution of a ParadisEO-PEO program requires having already created an environment for launching MPI programs. For *MPICH-2*, for example, this requires starting a ring of daemons.

At this point you should have successfully compiled the source code you should check you are in the **.../paradiseo-peo/tutorial/Lesson3** directory and you should have the following files: **tspExample** - the executable file, **schema.xml** - the XML resource mapping file and **lesson.param** - a file specifying different parameters for your program (whether to use logging or not, the mapping file to be used - the **schema.xml** file in this case, etc).

Launching the program may be different, depending on your MPI distribution - for MPICH-2, in a console, in the **paradiseo-peo/tutorial/Lesson2** directory you have to type the following command:

mpiexec -n 4 ./tspExample @lesson.param

NOTE: the "-n 4" indicates the number of processes to be launched. Please note that the number of processes has to be equal to the number of nodes specified in the **schema.xml** file.

The result of your execution should be similar to the following:

```
Loading '../data/eil101.tsp'.
NAME: eil101.
COMMENT: 101-city problem (Christofides/Eilon).
TYPE: TSP.
DIMENSION: 101.
EDGE_WEIGHT_TYPE: EUC_2D.
Loading '../data/eil101.tsp'.
NAME: eil101.
COMMENT: 101-city problem (Christofides/Eilon).
EOF.
TYPE: TSP.
DIMENSION: 101.
EDGE_WEIGHT_TYPE: EUC_2D.
EOF.
Loading '../data/eil101.tsp'.
NAME: eil101.
COMMENT: 101-city problem (Christofides/Eilon).
TYPE: TSP.
DIMENSION: 101.
EDGE_WEIGHT_TYPE: EUC_2D.
EOF.
Loading '../data/eil101.tsp'.
NAME: eil101.
COMMENT: 101-city problem (Christofides/Eilon).
TYPE: TSP.
DIMENSION: 101.
EDGE_WEIGHT_TYPE: EUC_2D.
EOF.
STOP in eoGenContinue: Reached maximum number of generations [100/100]
STOP in eoGenContinue: Reached maximum number of generations [100/100]
```

Please note the two lines at the end of the output - as we are running two evolutionary algorithms, there are two termination signals at the end of the 100 generations.