ParadisEO-PEO - Lesson 2

Parallel Evaluation of the Population. Parallel Fitness Function

Introduction

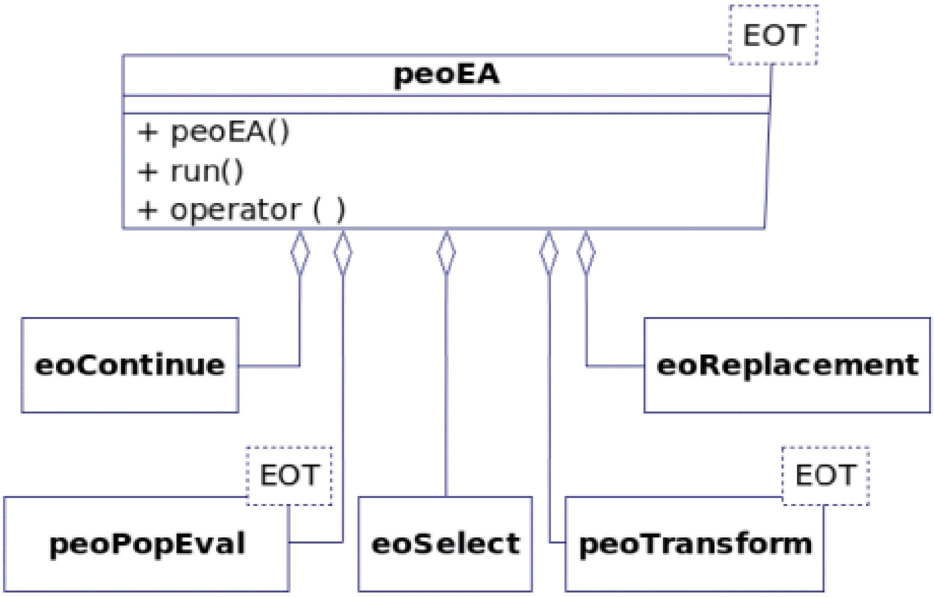
For high computing-intensive evolutionary algorithms where the fitness evaluation step is non-negligible, a parallel evaluation step of the population may and should significantly improve the execution time. As a general remark, the rapport between communication and fitness evaluation times has to be considered. If the fitness evaluation time and the communication time are comparable, evaluating the fitness in parallel brings less or no improvement - it may even increase the execution time. At the opposite extreme, if the fitness evaluation requires a long time as compared to communication, a significant speed-up should be achieved.

While basic notions regarding parallel fitness evaluation were exposed in **Lesson 1 - Introducing Parallelism**
- in this lesson we will explore the requirements and the constraints a program has to meet for achieving parallel fitness evaluation. As we will be constructing our example on the TSP problem, which was already presented in the first lesson, we will not go into further details - refer to **Lesson 1 - Problem Definition and Representation** for the afferent notions.

In order to have a complete and clear overview on the components required for building an evolutionary algorithm with parallel fitness evaluation, you are strongly encouraged to first familiarize with the basic components of the **peoEA**
class. While we will reintroduce a few notions in the current lesson, you should reffer to **Lesson 1** for a more complete presentation.

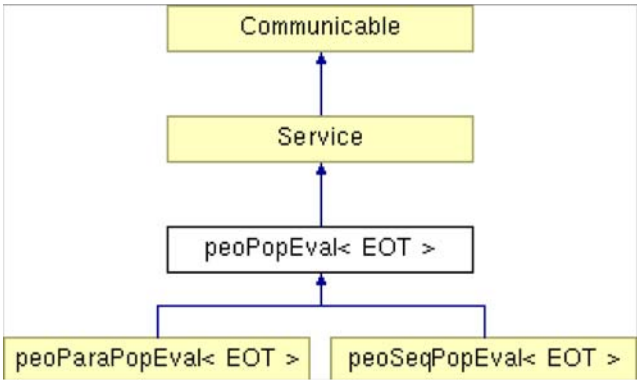
Considering the signature of the **peoEA** constructor, we will focus our attention on the **peoPopEval** class:

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

        peoPopEval< EOT >& __pop_eval,

        eoSelect< EOT >& __select,

        peoTransform< EOT >& __trans,

        eoReplacement< EOT >& __replace

);
```



ParadisEO-PEO Fitness Evaluation - Sequential vs. Parallel

Inside the ParadisEO-PEO package two fitness evaluation classes are availabe, allowing you to employ, out of the box, either simple sequential or parallel fitness evaluation. These classes are **peoSeqPopEval** for sequential evaluation and **peoParaPopEval**
for parallel evaluation, respectively. A schematic class inheritance overview is offered bellow:



Furthermore, the **peoParaPopEval**
comports a specialization allowing for the parallel evaluation of the fitness function as (a) single entity or (b) as an aggregate function composed of several sub-functions. In this later case, the fitness function is defined as a composite function:

$$F(x) = f(f1(x), f2(x), ..., fn(x))$$

where *F(x)* denotes the fitness value of an arbitrary individual *x* and *f(.)* represents the compositing function. In order to have a basic example, we may consider *f(.)* as being a simple sum of the sub-function values:

$$f(f1(x), f2(x), ..., fn(x)) = \sum f_i(x), \text{ with i in } [1,n]$$

For the case of an aggregate fitness function, each of the sub-functions may be evaluated in parallel, for each individual in the population. While the first case does not have special implementation requirements, an agregation function has to be defined for the second one in order to define how the composite final fitness is computed out of the sub-function fitness values. As a general remark, one has to keep in mind that not all problems have or allow for aggregate fitness function but, where applicable, it may offer the means to fully exploit the available computation power.

In the following we will discuss each of two presented cases, starting from the simple **sequential program** which was offered as example in **Lesson 1**.

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.

Parallel Evaluation of the Population

The source code for this example may be found in the main.cpp file, under the
paradiseo-peo/tutorial/Lesson2
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 Lesson2 main.cpp file (the one located in the **paradiseo-peo/tutorial/Lesson2** directory).

NOTE: the main.cpp file under the **paradiseo-peo/tutorial/Lesson2** directory offers you the possibility of testing either parallel population evaluation or parallel fitness evaluation. By default, parallel evaluation of the population is performed - in order to configure the program for parallel fitness evaluation, you should uncomment the **#define PARALLEL_FIT_EVALUATION** line. For parallel fitness evaluation you should have something as follows:

```
...
// by default, parallel evaluation of the population is performed;
// for parallel fitness evaluation, uncomment the following line

#define PARALLEL_FIT_EVALUATION
...
```

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/Lesson2** 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 **peoParaPopEval.h** header file is required - already provided by including the **paradiseo** file, hence, no modifications are required at this point.

2.1 **peoParaPopEval - parallel evaluation of the population**

Constructing an evolutionary algorithm which has as feature the parallel evaluation of the population requires having as basis a full fitness evaluation object. For our case, this corresponds to the **full_eval** object defined at the begining of the main function - it serves at computing the fitness of an arbitrary individual.

The parallel evaluation step is performed by the **peoParaPopEval** object which acts as a wrapper enclosing internally the full fitness evaluation object. The entire parallelization process is transparent for the user requiring only to replace a single line. At this time executing the **tspExample** file (the binary obtained after compiling the **main.cpp**
source file) corresponds to nothing more than a sequential evolutionary algorithm. Transforming the sequential program into a parallel one requires you to change

```
peoSeqPopEval< Route > eaPopEval( full_eval );
```

with

```
peoParaPopEval< Route > eaPopEval( full_eval );
```

At this point you should recompile your program in order to assure everything is in place. In case everything went fine and there are no compilation errors you may proceed to executing the program - this requires a mapping between the available computing resources and the deployed algorithms.

2.2 **XML Resource Mapping File**

For the herein example a single evolutionary algorithm has been defined - as we will later see in the following lessons, multiple inter-communicating algorithms may be defined. In either case, when launching multiple parallel processes, we should specify which process(es) will execute the evolutionary algorithm(s) and which ones will be actually computing the fitness function. 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, for fitness function evaluations. 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. 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>
                </node>

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

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.

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/Lesson2** directory - you should open a console and you should change your current directory to the one of Lesson2.

Compilation: being in the **paradiseo-peo/tutorial/Lesson2 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/Lesson2**
- you saved your modifications in a file called **main.cpp**, in the **paradiseo-peo/tutorial/Lesson2** 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.

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/Lesson2** 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]
```

Parallel Fitness Evaluation

The source code for this example may be found in the main.cpp file, under the
paradiseo-peo/tutorial/Lesson2
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 Lesson2 main.cpp file (the one located in the **paradiseo-peo/tutorial/Lesson2** directory).

NOTE: the main.cpp file under the **paradiseo-peo/tutorial/Lesson2** directory offers you the possibility of testing either parallel population evaluation or parallel fitness evaluation. By default, parallel evaluation of the population is performed - in order to configure the program for parallel fitness evaluation, you should uncomment the **#define PARALLEL_FIT_EVALUATION** line. For parallel fitness evaluation you should have something as follows:

```
...
// by default, parallel evaluation of the population is performed;
// for parallel fitness evaluation, uncomment the following line

#define PARALLEL_FIT_EVALUATION
...
```

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/Lesson2** 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 **peoParaPopEval.h** header file is required - already provided by including the **paradiseo** file, hence, no modifications are required at this point.

In addition, as we will be using partial fitness functions (which compute different parts of a composite fitness function) and an aggregation function for assembling the partial results into a single value, you have to add also the following files:

```
...
#include "part_route_eval.h"
#include "merge_route_eval.h"
...
```

The **part_route_eval.h**
file defines a partial fitness function evaluating only limited segments of a TSP route while the
merge_route_eval.h
file defines the aggregation function which, in this case, is defined as a sum of the partial fitness evaluations.

2.1 **peoParaPopEval - parallel fitness evaluation**

Constructing an evolutionary algorithm which has as feature parallel fitness evaluation requires having as basis a partial fitness evaluation object and an aggregation function. In addition we also have to consider how many "splits" we would like to have for the fitness function. For the herein example, to a given extent, we may consider an arbitrary number of "splits", each sub-function acting on a limited part of a TSP route.

In order to add the required components, you should create a **MergeRouteEval** object as well as several **PartRouteEval**
objects for the aggregation function and for the partial evaluation functions, respectively. For simplicity reasons we considered a vector of partial fitness evaluation objects as follows:

```
MergeRouteEval merge_eval;

std :: vector< eoEvalFunc< Route >* > part_eval;
for ( unsigned index = 1; index <= NUM_PART_EVALS; index++ )
        part_eval.push_back( new PartRouteEval( ( float )( index - 1 ) / NUM_PART_EVALS, ( float )index / NUM_PART_EVALS ) );
```

In addition, at the begining of the **main.cpp** file you should add a new define entry for specifying the **NUM_PART_EVALS**
constant we used in the for loop (the number of partial evaluation functions) - we chose to have only two *splits*:

```
...
#define MUT_RATE 0.01

#define NUM_PART_EVALS 2
...
```

The parallel evaluation step is performed by the **peoParaPopEval** object which acts as a wrapper enclosing internally the partial fitness evaluation functions and the aggregation function. The entire parallelization process is transparent for the user requiring only to replace a single line. At this time executing the **tspExample** file (the binary obtained after compiling the **main.cpp** source file) corresponds to nothing more than a sequential evolutionary algorithm. Transforming the sequential program into a parallel one requires you to change

```
peoSeqPopEval< Route > eaPopEval( full_eval );
```

with

```
peoParaPopEval< Route > ox_pop_eval( part_eval, merge_eval );
```

At this point you should recompile your program in order to assure everything is in place. In case everything went fine and there are no compilation errors you may proceed to executing the program - this requires a mapping between the available computing resources and the deployed algorithms.

2.2 **XML Resource Mapping File**

For the herein example a single evolutionary algorithm has been defined - as we will later see in the following lessons, multiple inter-communicating algorithms may be defined. In either case, when launching multiple parallel processes, we should specify which process(es) will execute the evolutionary algorithm(s) and which ones will be actually computing the partial fitness functions. 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.

NOTE: The aggregation fitness function is computed on the *master* node(s) - there are no associated attributes or specifications in the XML file.

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, for partial fitness function evaluations. 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. 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>
		</node>

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

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.

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/Lesson2** directory - you should open a console and you should change your current directory to the one of Lesson2.

Compilation: being in the **paradiseo-peo/tutorial/Lesson2 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/Lesson2**
- you saved your modifications in a file called **main.cpp**, in the **paradiseo-peo/tutorial/Lesson2** 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.

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/Lesson2** 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]
```