Comment utiliser Paradiseo-PEO

I. Introduction

Paradiseo-PEO est un module qui permet de paralléliser des méta-heuristiques et des recherches locales.

Afin de faciliter l'utilisation de ce module, de nombreux mécanismes ne sont pas visibles par l'utilisateur (primitives MPI, ...).

Dans un premier temps ce manuel explique l'utilisation du parallélisme au travers différents exemples. Cette première partie permet de prendre rapidement en main le module sur des algorithmes comme des algorithmes génétiques, le PSO, les recherches locales, le multi-objectif ou encore l'hybridation. Bien évidemment les différents niveaux de parallélismes y sont explicités.

Dans un second temps, ce manuel approfondi les mécanismes de PEO. Grâce à ces informations sur le fonctionnement de la plate-forme vous pourrez paralléliser d'autres algorithmes et même créer de nouveaux modèles de parallélismes adaptés spécifique à votre problème.

II. Algorithme génétique

1. Introduction

Afin de comprendre les mécanismes de PEO, nous allons travailler sur un exemple précis d'algorithme génétique.

Code

```
eoGenContinue < Indi > genContPara (MAX_GEN);
eoCombinedContinue <Indi> continuatorPara (genContPara);
eoCheckPoint<Indi> checkpoint(continuatorPara);
eoEvalFuncPtr< Indi > mainEval( f );
eoEvalFuncCounter< Indi > eval(mainEval);
eoUniformGenerator <double> uGen (INIT_POSITION_MIN, INIT_POSITION_MAX);
eoInitFixedLength < Indi > random (VEC_SIZE, uGen);
eoPop < Indi > pop;
pop.append (POP_SIZE, random);
eoRankingSelect<Indi> selectionStrategy;
eoSelectNumber<Indi> select(selectionStrategy,POP_SIZE);
eoSegmentCrossover<Indi> crossover;
eoUniformMutation<Indi>  mutation(EPSILON);
eoSGATransform<Indi> transform(crossover,CROSS_RATE,mutation,MUT_RATE);
eoPlusReplacement<Indi> replace;
eoEasyEA< Indi > eaAlg( checkpoint, eval, select, transform, replace );
eaAlg(pop);
```

Note : Indi est le codage de l'individu et f est la fonction à optimiser.

Cet algorithme va nous permettre de voir les modifications nécessaires à apporter pour paralléliser un algorithme séquentiel (ces changements seront indiqués en rouge).

Bien évidemment **vous pourrez utiliser tous les niveaux de parallélismes en même temps**.

Afin de charger le schéma XML les algorithmes parallèles seront précédés de :

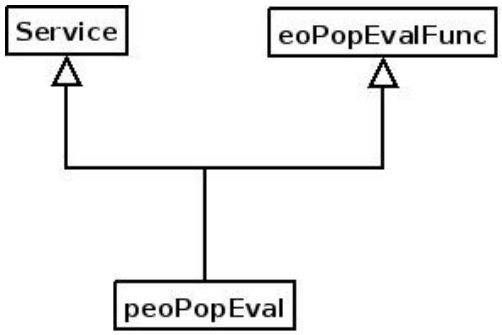
```
peo :: init( __argc, __argv );
```

Grâce à cette primitive, l'environnement de parallélisme sera initialisé selon les instructions donné dans le schéma XML (il est aussi intéressant de regarder le fichier : src/rmc/mpi/node.cpp).

2. Parallélisation de l'évaluation

La parallélisation de l'évaluation nécessite différents composants : le Wrapper, **peoEvalFunc** et **peoPopEval** (le Wrapper sera détaillé dans la partie VI).

La classe peoPopEval hérite des classes **Service** et **eoPopEvalFunc**. Elle permet de gérer automatiquement la mise en paquetage des données à envoyer ainsi que les différents mécanismes essentiels au bon fonctionnement de l'évaluation de la population.



NB : Si l'utilisateur souhaite développer un nouveau type de données il faudra impérativement ajouter de nouvelles méthodes **pack** et **unpack** adaptées à ces données dans le fichier paradiseo-peo/src/core/eoVector_comm.h

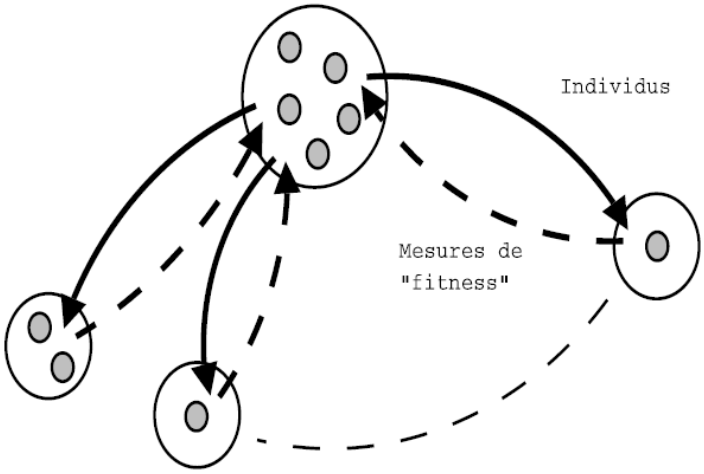
La méthode **setOwner** permet de définir les différents composants à paralléliser à l'intérieur de l'algorithme (l'appel de cette méthode est obligatoire).

Code

```
peo :: init( __argc, __argv );

eoGenContinue < Indi > genContPara (MAX_GEN);
eoCombinedContinue <Indi> continuatorPara (genContPara);
eoCheckPoint<Indi> checkpoint(continuatorPara);

peoEvalFunc<Indi> mainEval( f );
```

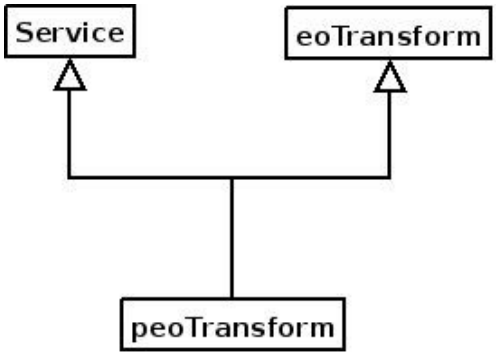
```
peoPopEval <Indi> eval(mainEval);

eoUniformGenerator <double> uGen (INIT_POSITION_MIN, INIT_POSITION_MAX);
eoInitFixedLength < Indi > random (VEC_SIZE, uGen);
eoPop < Indi > pop;
pop.append (POP_SIZE, random);
eoRankingSelect<Indi> selectionStrategy;
eoSelectNumber<Indi> select(selectionStrategy,POP_SIZE);
eoSegmentCrossover<Indi> crossover;
eoUniformMutation<Indi>  mutation(EPSILON);
eoSGATransform<Indi> transform(crossover,CROSS_RATE,mutation,MUT_RATE);
eoPlusReplacement<Indi> replace;
eoEasyEA< Indi > eaAlg( checkpoint, eval, select, transform, replace );

peoParallelAlgorithmWrapper parallelEA( eaAlg, pop);
eval.setOwner(parallelEA);
peo :: run();
peo :: finalize();
```

3. Parallélisation de la transformation



Le principe de la parallélisation du composant de transformation reste le même que celui utilisé pour l'évaluation en parallèle.

Dans un premier temps nous devons déclarer un objet **peoTransform** puis indiquer, grâce à la méthode **setOwner**, que ce composant est un composant est parallèle.

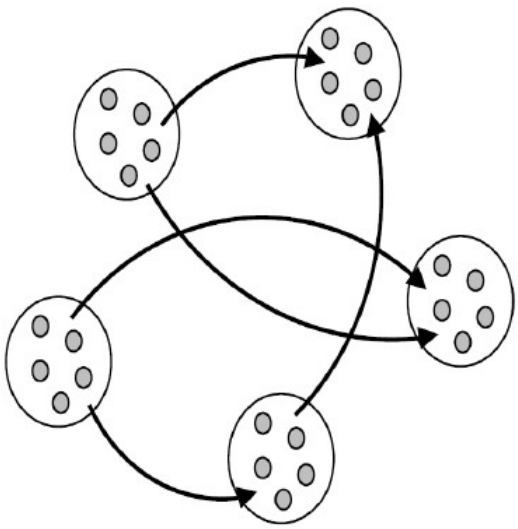


Code

```
peo :: init( __argc, __argv );
eoGenContinue < Indi > genContPara (MAX_GEN);
eoCombinedContinue <Indi> continuatorPara (genContPara);
eoCheckPoint<Indi> checkpoint(continuatorPara);
peoEvalFunc<Indi> mainEval( f );
peoPopEval <Indi> eval(mainEval);
eoUniformGenerator <double> uGen (INIT_POSITION_MIN, INIT_POSITION_MAX);
eoInitFixedLength < Indi > random (VEC_SIZE, uGen);
eoPop < Indi > pop;
pop.append (POP_SIZE, random);
eoRankingSelect<Indi> selectionStrategy;
eoSelectNumber<Indi> select(selectionStrategy,POP_SIZE);
eoSegmentCrossover<Indi> crossover;
eoUniformMutation<Indi>  mutation(EPSILON);

peoTransform<Indi> transform(crossover,CROSS_RATE,mutation,MUT_RATE);

eoPlusReplacement<Indi> replace;
eoEasyEA< Indi > eaAlg( checkpoint, eval, select, transform, replace );
peoParallelAlgorithmWrapper parallelEA( eaAlg, pop);
eval.setOwner(parallelEA);

transform.setOwner(parallelEA);

peo :: run();
peo :: finalize();
```

4. Modèle d'îles

Le modèle d'îles consiste à faire évoluer plusieurs algorithmes en même temps et à faire communiquer ces algorithmes entre eux. Le mode de communication peut être très varié : échange d'individus au bout d'un certain temps, échange d'information concernant la population ...



Différentes topologies de communication sont disponibles. Ces communications peuvent être synchrones ou asynchrones en fonction du besoin de l'utilisateur.

Code

```
peo :: init( __argc, __argv );

RingTopology topo;

eoGenContinue < Indi > genContPara (MAX_GEN);
eoCombinedContinue <Indi> continuatorPara (genContPara);
eoCheckPoint<Indi> checkpoint(continuatorPara);
peoEvalFunc<Indi> mainEval( f );
peoPopEval <Indi> eval(mainEval);
eoUniformGenerator <double> uGen (INIT_POSITION_MIN, INIT_POSITION_MAX);
eoInitFixedLength < Indi > random (VEC_SIZE, uGen);
eoPop < Indi > pop;
pop.append (POP_SIZE, random);
eoRankingSelect<Indi> selectionStrategy;
eoSelectNumber<Indi> select(selectionStrategy,POP_SIZE);
eoSegmentCrossover<Indi> crossover;
eoUniformMutation<Indi>  mutation(EPSILON);
peoTransform<Indi> transform(crossover,CROSS_RATE,mutation,MUT_RATE);
eoPlusReplacement<Indi> replace;

eoPeriodicContinue <Indi> mig_c( MIG_FREQ );
eoRandomSelect<Indi> mig_select_one;
eoSelectNumber<Indi> mig_sel (mig_select_one,MIG_SIZE);
eoPlusReplacement<Indi> mig_repl

eoGenContinue < Indi > genContPara2 (MAX_GEN);
eoCombinedContinue <Indi> continuatorPara2 (genContPara2);
eoCheckPoint<Indi> checkpoint2(continuatorPara2);
peoEvalFunc<Indi> mainEval2( f );
peoPopEval <Indi> eval2(mainEval2);
eoUniformGenerator <double> uGen (INIT_POSITION_MIN, INIT_POSITION_MAX);
eoInitFixedLength < Indi > random2 (VEC_SIZE, uGen);
eoPop < Indi > pop2;
pop2.append (POP_SIZE, random2);
eoRankingSelect<Indi> selectionStrategy2;
eoSelectNumber<Indi> select2(selectionStrategy2,POP_SIZE);
eoSegmentCrossover<Indi> crossover2;
eoUniformMutation<Indi>  mutation2(EPSILON);
peoTransform<Indi> transform2(crossover2,CROSS_RATE,mutation2,MUT_RATE);
eoPlusReplacement<Indi> replace2;

eoPeriodicContinue <Indi> mig_c2( MIG_FREQ );
eoRandomSelect<Indi> mig_select_one2;
eoSelectNumber<Indi> mig_sel2 (mig_select_one2,MIG_SIZE);
eoPlusReplacement<Indi> mig_repl2;

peoAsyncIslandMig<Indi> mig(mig_c,mig_sel,mig_repl,topo,pop,pop);
checkpoint.add(mig);
peoAsyncIslandMig<Indi> mig2(mig_c2,mig_sel2,mig_repl2,topo,pop2,pop2);
```

```
checkpoint2.add(mig2);

eoEasyEA< Indi > eaAlg( checkpoint, eval, select, transform, replace );
peoParallelAlgorithmWrapper parallelEA( eaAlg, pop);
eval.setOwner(parallelEA);
transform.setOwner(parallelEA);
eoEasyEA<Indi>eaAlg2(checkpoint2, eval2, select2, transform2, replace2);
peoParallelAlgorithmWrapper parallelEA2( eaAlg2, pop2);
eval2.setOwner(parallelEA2);
transform2.setOwner(parallelEA2);
peo :: run();
peo :: finalize();
```

III. PSO

1. Introduction

L'utilisation d'un PSO sous ParadisEO-PEO est très similaire à celle d'un algorithme génétique. Le principe reste le même et seul les noms des composants changent.

Comme précédemment nous allons utiliser un code séquentiel référence pour illustrer nos propos.

Code

```
eoGenContinue < Indi > genContPara (MAX_GEN);
eoCombinedContinue <Indi> continuatorPara (genContPara);
eoCheckPoint<Indi> checkpoint(continuatorPara);
eoEvalFuncPtr< Indi > mainEval( f );
eoEvalFuncCounter< Indi > eval(mainEval);
eoUniformGenerator <double> uGen (INIT_POSITION_MIN, INIT_POSITION_MAX);
eoInitFixedLength <Indi> random (VEC_SIZE, uGen);
eoUniformGenerator <double> sGen (INIT_VELOCITY_MIN, INIT_VELOCITY_MAX);
eoVelocityInitFixedLength <Indi> veloRandom (VEC_SIZE, sGen);
eoFirstIsBestInit <Indi> localInit;
eoRealVectorBounds bnds(VEC_SIZE,INIT_POSITION_MIN,INIT_POSITION_MAX);
eoStandardFlight < Indi > flight(bnds);
eoPop < Indi > pop;
pop.append (POP_SIZE, random);
eoLinearTopology<Indi> topology(NEIGHBORHOOD_SIZE);
eoRealVectorBounds bnds(VEC_SIZE,INIT_VELOCITY_MIN,INIT_VELOCITY_MAX);
eoStandardVelocity < Indi > velocity (topology,C1,C2,bnds);
eoInitializer <Indi> init(eval,veloRandom,localInit,topology,pop);
eoSyncEasyPSO <Indi> psa(init,checkpoint,eval, velocity, flight);
psa(pop);
```

2. Parallélisation de l'évaluation

Bien que la parallélisation de l'évaluation d'un PSO ressemble à celle d'un algorithme génétique, il est important de remarquer :

- L'utilisation de l'algorithme **eoSyncEasyPSO** est plus judicieuse que celle de **eoEasyPSO** pour une raison de conception de l'algorithme.
- L'initialisation du PSO peut se faire en parallèle ou en séquentiel en fonction du composant passé en paramètre dans le constructeur de l'objet **eoInitializer**.

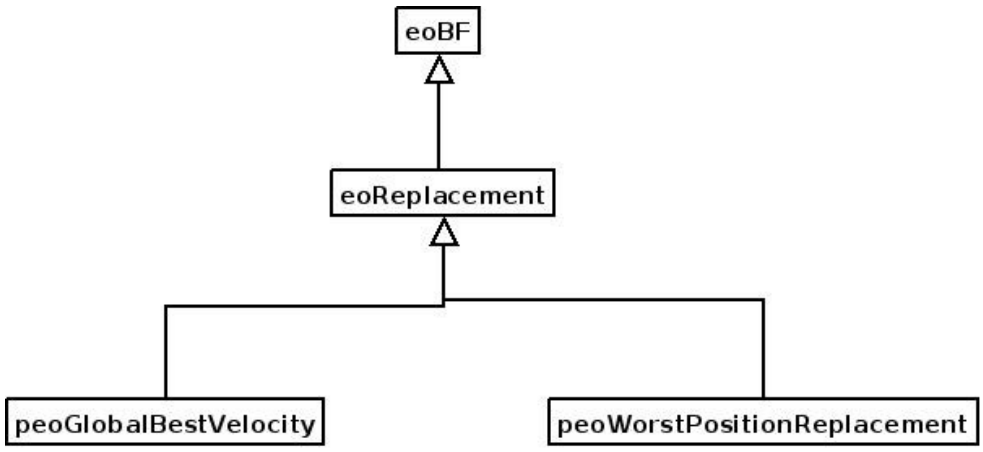
Code

```
peo :: init( __argc, __argv );

eoGenContinue < Indi > genContPara (MAX_GEN);
eoCombinedContinue <Indi> continuatorPara (genContPara);
eoCheckPoint<Indi> checkpoint(continuatorPara);

peoEvalFunc<Indi, double, const Indi& > plainEval(f);
peoPopEval< Indi > eval(plainEval);

eoUniformGenerator <double> uGen (INIT_POSITION_MIN, INIT_POSITION_MAX);
eoInitFixedLength <Indi> random (VEC_SIZE, uGen);
eoUniformGenerator <double> sGen (INIT_VELOCITY_MIN, INIT_VELOCITY_MAX);
eoVelocityInitFixedLength <Indi> veloRandom (VEC_SIZE, sGen);
eoFirstIsBestInit <Indi> localInit;
eoRealVectorBounds bnds(VEC_SIZE,INIT_POSITION_MIN,INIT_POSITION_MAX);
eoStandardFlight < Indi > flight(bnds);
eoPop < Indi > pop;
pop.append (POP_SIZE, random);
eoLinearTopology<Indi> topology(NEIGHBORHOOD_SIZE);
eoRealVectorBounds bnds(VEC_SIZE,INIT_VELOCITY_MIN,INIT_VELOCITY_MAX);
eoStandardVelocity < Indi > velocity (topology,C1,C2,bnds);
eoInitializer <Indi> init(eval,veloRandom,localInit,topology,pop);

eoSyncEasyPSO <Indi> psa(init,checkpoint,eval, velocity, flight);
peoWrapper parallelPSO( psa, pop);
eval.setOwner(parallelPSO);

peo :: run();
peo :: finalize();
```

3. Modèle d'îles

Le modèle d'îles du PSO possède deux modes d'incorporation des individus prédéfinis dans ParadisEO-PEO :

- **peoWorstPositionReplacement** : remplace l'individu d'une population de destination ayant la moins bonne position par l'individu d'une population source ayant la meilleure position.
- **peoGlobalBestVelocity** : ce mode d'incorporation utilise la meilleure velocity d'une population source pour effectuer le calcul de la « best velocity » globale d'une population de destination.



Code

```
peo :: init( __argc, __argv );

RingTopology topologyMig;

eoGenContinue < Indi > genContPara (MAX_GEN);
eoCombinedContinue <Indi> continuatorPara (genContPara);
eoCheckPoint<Indi> checkpoint(continuatorPara);
peoEvalFunc<Indi, double, const Indi& > plainEval(f);
peoPopEval< Indi > eval(plainEval);
eoUniformGenerator <double>uGen (INIT_POSITION_MIN, INIT_POSITION_MAX);
eoInitFixedLength < Indi > random (VEC_SIZE, uGen);
eoUniformGenerator <double>sGen (INIT_VELOCITY_MIN, INIT_VELOCITY_MAX);
eoVelocityInitFixedLength < Indi > veloRandom (VEC_SIZE, sGen);
eoFirstIsBestInit < Indi > localInit;
eoRealVectorBounds
bndsFlight(VEC_SIZE,INIT_POSITION_MIN,INIT_POSITION_MAX);
eoStandardFlight < Indi > flight(bndsFlight);
eoPop < Indi > pop;
pop.append (POP_SIZE, random);
eoLinearTopology<Indi> topology(NEIGHBORHOOD_SIZE);
eoRealVectorBounds bnds(VEC_SIZE,INIT_VELOCITY_MIN,INIT_VELOCITY_MAX);
eoStandardVelocity < Indi > velocity (topology,C1,C2,bnds);
eoInitializer <Indi> init(eval,veloRandom,localInit,topology,pop);

eoPeriodicContinue< Indi > mig_cont( MIG_FREQ );
peoPSOSelect<Indi> mig_selec(topology);
eoSelectNumber< Indi > mig_select(mig_selec);
peoWorstPositionReplacement<Indi> mig_replace;

eoGenContinue < Indi > genContPara2 (MAX_GEN);
eoCombinedContinue <Indi> continuatorPara2 (genContPara2);
eoCheckPoint<Indi> checkpoint2(continuatorPara2);
peoEvalFunc<Indi, double, const Indi& > plainEval2(f);
peoPopEval< Indi > eval2(plainEval2);
eoUniformGenerator <double>uGen2 (INIT_POSITION_MIN, INIT_POSITION_MAX);
eoInitFixedLength < Indi > random2 (VEC_SIZE, uGen2);
eoUniformGenerator <double>sGen2 (INIT_VELOCITY_MIN, INIT_VELOCITY_MAX);
```

```
eoVelocityInitFixedLength < Indi > veloRandom2 (VEC_SIZE, sGen2);
eoFirstIsBestInit < Indi > localInit2;
eoRealVectorBounds
bndsFlight2(VEC_SIZE,INIT_POSITION_MIN,INIT_POSITION_MAX);
eoStandardFlight < Indi > flight2(bndsFlight2);
eoPop < Indi > pop2;
pop2.append (POP_SIZE, random2);
eoLinearTopology<Indi> topology2(NEIGHBORHOOD_SIZE);
eoRealVectorBounds bnds2(VEC_SIZE,INIT_VELOCITY_MIN,INIT_VELOCITY_MAX);
eoStandardVelocity < Indi > velocity2 (topology2,C1,C2,bnds2);
eoInitializer <Indi> init2(eval2,veloRandom2,localInit2,topology2,pop2);

eoPeriodicContinue< Indi > mig_cont2( MIG_FREQ );
peoPSOSelect<Indi> mig_selec2(topology2);
eoSelectNumber< Indi > mig_select2(mig_selec2);
peoWorstPositionReplacement<Indi> mig_replace2;

peoAsyncIslandMig<  Indi  >  mig(  mig_cont,  mig_select,  mig_replace,
topologyMig, pop, pop);
checkpoint.add( mig );
peoAsyncIslandMig<  Indi  >  mig2(  mig_cont2,  mig_select2,  mig_replace2,
topologyMig, pop2, pop2);
checkpoint2.add( mig2 );

eoSyncEasyPSO <Indi> psa(init,checkpoint,eval, velocity, flight);
peoWrapper parallelPSO( psa, pop);
eval.setOwner(parallelPSO);
mig.setOwner(parallelPSO);
eoSyncEasyPSO <Indi> psa2(init2,checkpoint2,eval2, velocity2, flight2);
peoWrapper parallelPSO2( psa2, pop2);
eval2.setOwner(parallelPSO2);
mig2.setOwner(parallelPSO2);
peo :: run();
peo :: finalize();
```

IV. Multi-objectif

1. Introduction

Afin d'optimiser l'efficacité du module ParadisEO-MOEO, il est possible de paralléliser la fonction d'évaluation des algorithmes multi-objectif.

NB : L'utilisation des fichiers do_make (make_continue, make_checkpoint, make_ea et make_eval) demande cependant une attention particulière de l'utilisateur. En effet, les objets d'évaluation déclarés dans ces fichiers doivent être du même type que celui déclaré dans le main.

2. Exemple d'utilisation

L'exemple proposé utilise le « Flow shop ». Il s'agit de la version parallèle de la leçon 1 du

module MOEO.

Code

```
peo :: init( argc,argv );

eoParser parser(argc, argv);
eoState state;
peoMoeoPopEval<FlowShop>& eval = do_make_para_eval(parser, state);
eoInit<FlowShop>& init = do_make_genotype(parser, state);
eoGenOp<FlowShop>& op = do_make_op(parser, state);
eoPop<FlowShop>& pop = do_make_pop(parser, state, init);
moeoArchive<FlowShop> arch;
eoContinue<FlowShop>& term = do_make_continue_moeo(parser, state, eval);
eoCheckPoint<FlowShop>& checkpoint = do_make_checkpoint_moeo(parser,
state, eval, term, pop, arch);
eoAlgo<FlowShop>& algo = do_make_ea_moeo(parser, state, eval,
checkpoint, op, arch);

peoWrapper parallelMOEO( algo, pop);
eval.setOwner(parallelMOEO);
peo :: run();
peo :: finalize();
```

V. Hybridation

1. Introduction

Dans de nombreux problèmes les recherches locales sont utilisées pour pré conditionner les problèmes ou accélérer la convergence.

Dans ce cadre, il est possible avec ParadisEO-PEO de traiter une population avec des recherche locales parallèles. Ces recherches locales peuvent être appliquées sur la population avant le lancement d'un algorithme génétique (ou PSO) et/ou après le l'exécution de cet algorithme.

Toutes les recherches locales s'effectuent en parallèles grâce à l'utilisation du MultiStart (cf section VII)

2. Exemple d'utilisation

L'exemple proposé effectue un algorithme génétique puis un « Hill climbing » sur les individus de la population finale. Le problème traité est dans cet exemple est un TSP.

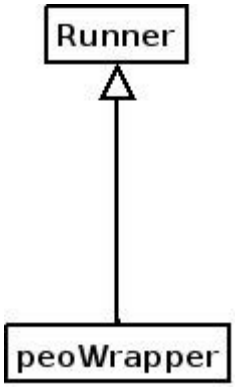
Code

```
peo :: init( argc,argv );
```

```
loadParameters (__argc, __argv);
RouteInit route_init;
RouteEval full_eval;
OrderXover order_cross;
PartialMappedXover pm_cross;
EdgeXover edge_cross;
CitySwap city_swap_mut;
TwoOptInit pmx_two_opt_init;
TwoOptNext pmx_two_opt_next;
TwoOptIncrEval pmx_two_opt_incr_eval;
moBestImprSelect <TwoOpt> pmx_two_opt_move_select;
moHC<TwoOpt>  hc(pmx_two_opt_init,pmx_two_opt_next,pmx_two_opt_incr_eval,
pmx_two_opt_move_select, full_eval);
eoPop <Route> pop (POP_SIZE, route_init);
eoGenContinue <Route> cont (NUM_GEN);
eoCheckPoint <Route> checkpoint (cont);
eoEvalFuncCounter< Route > eval(full_eval);
eoStochTournamentSelect <Route> select_one;
eoSelectNumber <Route> select (select_one, POP_SIZE);
eoSGATransform     <Route>     transform     (order_cross,     CROSS_RATE,
city_swap_mut, MUT_RATE);
eoEPReplacement <Route> replace (2);

eoEasyEA< Route > eaAlg( checkpoint, eval, select, transform, replace );
peoWrapper parallelEA( eaAlg, pop);
peo :: run ();
peo :: finalize ();

peo :: init (__argc, __argv);
peoMultiStart <Route> initParallelHC (hc);
peoWrapper parallelHC (initParallelHC, pop);
initParallelHC.setOwner(parallelHC);
peo :: run( );
peo :: finalize( );
```

VI. Fonctionnement du Wrapper

La classe **peoWrapper** hérite directement de la classe **Runner**.



Afin de réaliser un passage du séquentiel au parallèle quatre options s'offrent à nous :

- Transformer un algorithme séquentiel, qui s'applique sur une donnée passée en argument, en algorithme parallèle.
- Transformer un algorithme séquentiel, sans argument ,en algorithme parallèle.
- Transformer une fonction séquentielle, qui s'applique sur une donnée passée en argument, en fonction parallèle.
- Transformer une fonction séquentielle, sans argument, en algorithme parallèle.

Le Wrapper exécutera l'algorithme (ou la fonction) sur les arguments (s'il y en a). Grâce à son héritage de la classe **Runner**, le Wrapper est capable de gérer l'exécution de composants parallèles (comme par exemple **peoPopEval**).

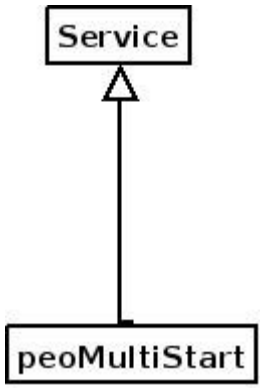
NB : L'utilisation du Wrapper nécessite cependant d'expliciter (avant le lancement de l'algorithme par **peo :: run()**) les composants parallèles. Cette définition se fait à l'aide de la méthode **setOwner**.

Plusieurs Wrapper peuvent être lancés en même temps ou successivement. Cette possibilité permet de réaliser une hybridation méta-heuristique / recherches locales (cf section V).

VII. Fonctionnement du MultiStart

La classe **peoMultiStart** hérite de la classe **Service**. Elle a pour rôle principal de gérer le lancement de plusieurs algorithmes en même temps.

Le constructeur d'un objet MultiStart perd en paramètre un algorithme séquentiel. Lorsque que l'on utilisera l'opérateur ()(Data) de cet objet, plusieurs algorithmes du même type que l'algorithme séquentiel de base seront lancés en parallèle sur tous les éléments de la structure Data.



Pour plus d'informations : clive.canape@inria.fr