Parallélisation d'un algorithme à l'aide de *OpenMp* et *Boost*

Travail présenté

à

Monsieur Gabriel Girard

Par

Samuel Boutin

Processus concurrents et parallélisme
IFT-630

Département d'informatique

Université de Sherbrooke

26 avril 2012

Table des matières

1 Introduction

L'objectif de mon projet est de paralléliser un code séquentiel cherchant le plus près de la meilleure solution possible au problème du voyageur de commerce. L'algorithme utilisé est basé sur le principe des colonies de fourmis. Le code séquentiel a été implémenter suite à une suggestion du professeur Jean Goulet dans le cadre d'un travail pratique du cours structure de données (ift-339) visant à trouver une solution au problème du voyageur de commerce.

Puisque la parallèlisation de l'algorithme s'est avéré plus simple que prévu, deux solutions au problème seront présentés. Ce sera ainsi l'occasion pour moi d'explorer deux outils de parallélisation et de prendre le temps pour chacun d'eux de lire la documentation et de faire une synthèse des éléments importants des deux outils. La première solution utilisera l'outil de programmation parallèle en mémoire partagée *OpenMP*, alors que la seconde utilisera la bibliothèque pour la gestion des fils d'éxécutions de *boost*.

Dans un premier temps, les deux outils de programmations seront présentés. Ensuite, l'algorithme séquentiel basé sur les colonies de fourmis sera présenté ainsi que les modifications nécessaire à l'implémentation d'une solution parallèle. Finalement, la performance de la version séquentielle et des deux versions parallèles seront comparées.

2 OpenMP

OpenMP est une série de directives de précompilation (pragma) et de fonctions qui peuvent être utilisées pour générer un programme à multiple fils d'exécutions sans avoir à programmer directement les fils et la synchronisation (le moins possible). Chaque directive de précompilation s'applique sur le bloc de code suivant. Il est utilisé pour la programmation parallèle à mémoire partagée.

2.1 Directives de précompilation

Je regroupe ci-dessous différentes directives de précompilation pour référence future.

1. Parallélisation de code :
   - *#pragma omp parallel{...}* : Tout ce qui ce trouve entre accolade sera exécuté par chacun des fils. Par défaut, toute variable déclarée avant l'entrée du bloc sera partagée, alors que toutes variables déclarées à l'intérieur du bloc seront privés. Des modifications à cette méthode par défaut peuvent être obtenus. Par exemple, dans ce cas : *#pragma omp parallel private(x,y){...}*, x et y seront privés.
   - *#pragma omp single* : un seul thread execute le bloc suivant.
   - *#pragam omp for* : les différentes itérations sont séparés parmi les fils d'éxécutions. Différentes options peuvent controler la méthode de répartition des itérations. Voir la documentation pour plus de détails.
2. Synchronisation :
   - *#pragma omp barrier* : barrière traditionelle, tous les threads s'attendent avant de continuer plus loin.
   - *#pragma omp critical* : section critique, un seul thread peut y entrer à la fois. Afin de créer plusieurs sections critique indépendantes, un nom associé à la section critique peut être donné entre parenthèse. Par exemple, un bloc de code précédé de *#pragma omp critical(nomDuMutex)* sera exécuté par un fil uniquement s'il n'y a aucun autre fil dans une section critique portant le même nom.

Il est à noter qu'à la fin d'un bloc *single*, *parallel*, *for* ou *sections* il y a une barrière implicite. Afin de ne pas avoir de barrière implicite, l'option *nowait* doit être ajouté au pragma précédant le bloc.

2.2 Fonctions

1. Gestion du nombre de *thread* et des numéros d'identification :
   - *omp_get_thread_num()* : cette commande retourne le numéro associé au thread.
   - *omp_get_num_threads()* : cette commande retourne le nombre de threads du programme.
   - *omp_get_num_procs()* : retourne le nombre de processeurs auxquels le programme a accès.
2. Synchronisation
   - Différentes fonctions permettent de créer et d'utiliser des verrous explicitement sous forme de fonctions plutôt que par des directives de précompilation.

2.3 *Includes* et compilation

Afin d'utiliser *OpenMP* il est nécessaire d'inclure le fichier *omp.h* (*#include <omp.h>*) afin d'avoir accès aux fonctions. De plus, la compilation en ligne de commandes avec *g++* ce fait par l'ajout du lien *-fopenmp*. Par exemple, *g++ -o exec main.cpp -fopenmp*. Finalement, la variable d'environnement *OMP_NUM_THREADS* permet de contrôler le nombre de fils créés. Elle peut être modifiée en ligne de commande par *export OMP_NUM_THREADS=N* où N est le nombre de fils souhaité.

3 Fils d'éxécution de *boost*

Boost est un ensemble de bibliothèques C++ pouvant être utilisées dans une variété d'applications. Il contient entre autres une bibliothèque pour le développement de programmes multifils (*multithreading*). Dans cette section, je résume les principales commandes utiles à la création et à la synchronisation de fils. Ce n'est bien entendu pas aussi complet que la documentation mais agit plutôt à titre de synthèse et de référence rapide dans le future.

3.1 Opérations sur les fils d'executions

1. Création d'un nouveau fil :
   - Il suffit de créer un nouvel objet *boost : :thread*.
   - Le premier argument passé à la fonction est le nom de la fonction que le *thread* doit exécuter ou bien une référence à une méthode d'un objet.
   - Les arguments suivants sont les arguments de la fonctions. Dans le cas d'une méthode d'un objet, le premier de la série doit être un pointeur sur l'objet (*this* si l'appel est fait de l'intérieur de l'objet).
   - Note, cette syntaxe n'est pas valide avec les vieilles versions de la librairie.
2. Attendre la fin d'une execution :
   - La méthode *join* de la classe *thread* attends la fin du fil d'exécution avant de poursuivre.

3.2 Synchronisation

1. Différents types de verrous
   - La classe *boost : :mutex* implante un verrou traditionnel qui peut être verrouillé à l'aide de la méthode *lock()* et déverrouillé à l'aide de la méthode *unlock()*. La méthode *try_lock()* peut également être utilisée afin de tester si le verrou est disponible. Cette méthode entre en section critique et retourne vrai si le verrous est disponible et retourne faux dans le cas contraire.
   - La classe *boost : :shared_mutex* implante un verrous de type multiples lecteurs, un seul écrivain. En plus des mêmes fonctions que ci-dessus qui permettent d'assurer l'exclusion mutuelle, les méthodes *lock_shared* et *unlock_shared* permettent de partager le verrous. Ainsi, si plusieurs processus lecteurs partage le verrous et qu'un processus écrivains tente d'entrer en section critique à l'aide de la méthode *lock()*, ce dernier devra attendre que tous les lecteurs relâche le verrou avant d'entrer. Il aura priorité sur de nouveaux processus lecteurs souhaitant accéder à la ressource.
   - Des objets permettent d'utiliser les verrous dans une philosophie *RAII* (Resource Allocation Is Initialisation). Ces objets permettent d'assurer la libération du verrous à la sortie du bloc de code l'ayant acquis. Un exemple d'utilisation est *boost : :unique_lock<boost : :shared_mutex> verrou(nomDuMutex) ;*. Lorsque le destructeur de *verrou* sera appelé, le mutex sera libéré s'il était encore verrouillé.
2. Variables conditionnelles
   - Des variables conditionnelles peuvent également être utilisées. Voir la documentation pour plus de détails.
3. Barrière
   - Une barrière peut également être utilisée afin de synchroniser plusieurs fils d'exécution. Voir la documentation pour plus de détails.

3.3 Remarques technique

Afin d'utiliser toutes ces outils, la ligne suivante doit être ajouté à l'en-tête du fichier (*#include <boost-/thread.hpp>*). Avec le compilateur g++, il suffit d'ajouter l'option *-lboost_thread* à la compilation, ainsi que l'emplacement des fichiers d'en-têtes et de la librairie *Boost*.

4 Algorithme

4.1 Version séquentielle

L'algorithme utilisé a été présenté en classe (IFT 339) par M. Jean Goulet et proposé comme exercice bonus dans le cadre du TP1 de la présente session [1]. Il s'agit d'une version simple d'un algorithme d'optimisation proposé dans les années 1990. Il consiste à imiter le comportement d'une colonie de fourmi qui cherche à trouver le chemin le plus court entre son nid et sa source de nourriture. L'algorithme se base sur les observations de biologistes qui ont remarqués que les fourmis libèrent des phéromones sur leur passage. Ces phéromones incitent d'autres fourmis à emprunter le même chemin, mais sans les forcer. Ainsi, d'autres chemins continuent à être explorés, mais plus un chemin est bon, plus il risque d'être emprunter souvent et donc plus il y aura de phéromone le long de celui-ci.

D'un point de vue algorithmique, des *fourmis informatique* sont disposés sur les différents noeuds du graphe. Celles-ci choisissent un des noeuds voisins (qui n'a pas préalablement été visité) comme destination. Le choix se fait de façon probabiliste à l'aide d'un nombre aléatoire. La probabilité de choisir un noeud N à partir d'un noeud I est donnée par :

$$Prob(I \to N) = V(I \to N)^{\alpha} P(I \to N)^{\beta} \tag{1}$$

où α et β sont des paramètres ajustables, $V(I \to N)$ l'indice de visibilité du noeuds N à partir du noeuds I. Celui-ci est donné par l'inverse du poids de l'arc reliant les deux noeuds dans cette direction. Finalement, $P(I \to N)$ est la quantité de phéromone présente sur l'arc. Donc, plus un arc est court et plus il contient de la phéromone, plus il a de chance d'être emprunter.

Une notion de temps est également introduite afin de favoriser les chemins les plus courts. Ainsi, si à l'instant t_i une fourmi choisi sa destination, elle choisira son prochain noeud au temps $t_f = t_i + p$ où p est le poids de l'arc parcouru. De plus, le temps avance, par l'évolution des évènements. Ainsi, le temps passera de t_i à une valeur supérieure au moment où il n'y aura plus d'évènements de temps t_i à traiter.

Finalement, lorsque une fourmi trouve une solution (un chemin fermé parcourant tous les noeuds) de la phéromone est déposée le long du chemin. Plus un chemin est court, plus la quantité de phéromone est grande. De plus, au bout d'un certain, un pourcentage de la phéromone est évaporée sur tous les arcs afin d'éviter que l'algorithme s'enlise trop rapidement dans une solution qui pourrait être très loin du minimum global.

5 Implémentation

5.1 Code séquentiel

La version séquentielle du code est présentée à l'annexe A. Le code est composée des types abstraits *graphe*, *colonie* et *fourmi*. Voici les faits saillants des différentes clases.

1. *graphe*
   - Dérive d'un *set* composé de l'ensemble des noeuds.
   - Contient une matrice creuse contenant le poids des arcs (représentée par un *map<string, map<string,double»*).
2. *fourmi*
   - Contient son noeud de départ, l'ensemble des noeuds à parcourir et son chemin parcouru.
   - Possède entre autres des méthodes pour avancer (incluant le choix du prochain noeud à partir des informations fournies par la colonie) à un prochain noeud, retourner au point de départ et indiquer le poids (longueur) du chemin parcouru.
3. *colonie*
   - Contient un vecteur de fourmis et une structure de l'ensemble des paramètres ajustables de l'algorithme.
   - Contient un pointeur vers le graphe que l'on cherche à solutionner.
   - Contient une matrice creuse de la phéromone sur les différents arcs du graphe.
   - Contient un multimap dans lequel les évènements sont indiqués. La notion de temps provient de cette structure qui est triée grâce à un double qui représente le temps associé à au prochain évènement d'une fourmi. Je ferai référence à cette structure sous le nom d'agenda.

1. Le travail pratique avait pour but d'explorer les différents conteneurs de la *STL* par le biais d'un programme trouvant une solution (bonne ou mauvaise) au problème du voyageur de commerce.

5.2 Parallélisation du code

Une analyse du code et des performances permet de réaliser que seul la méthode *trouver_ circuit* de la classe colonie nécessite une parallélisation. En effet, la lecture du graphe et l'initialisation de la colonie de fourmi se font en moins d'une seconde, alors que cette méthode selon les paramètres peut nécessiter plusieurs minutes de calculs. Le coeur de cette méthode est une boucle infinie qui traite un par un les évènements de « l'agenda »dans leur ordre chronologique (à chaque tour de boucle le premier élément du multimap est traité). Ainsi, moyennant quelques éléments de synchronisation, il est facile de paralléliser cette méthode.

L'algorithme utilisé est de faire exécutée cette boucle infinie (jusqu'à un évènement d'arrêt) par chacun des fils d'exécutions. Tous les fils partagent les différentes ressources (matrice du poids des arcs, agenda, matrice de phéromone, ...). Trois éléments doivent être synchronisés puisque modifiés à l'occasion. Il s'agit de l'agenda, de la meilleur solution trouvée jusqu'à mainenant (gardée en mémoire par la colonie) et la matrice de phéromone. Les deux premiers éléments sont uniquement utilisés pour y écrire ou bien lire puis écrire directement après. L'exclusion mutuelle est donc assurée par un simple mutex. Par contre, dans le cas de la matrice de phéromone, elle est fréquemment utilisée en lecture et parfois en écriture. Un verrou de type plusieurs lecteurs/un écrivain sera donc approprié.

5.2.1 *OpenMp*

Seul le fichier *Colonie.cpp* a dû être modifié pour la parallélisation du code. Celui-ci est présenté à l'annexe B. Il est a noté que malheureusement *OpenMP* ne propose pas le concept de verrous du type plusieurs lecteurs ou bien un seul écrivain. Or, puisque la structure de la matrice de phéromone n'est pas changée au cours du temps, il a été choisi de permettre des lectures de façon simultanée à l'écriture. En effet, les éléments non nuls de la matrice creuse restent les mêmes tout au long du calcul. Seul la valeur associée aux clés change. Ainsi, le seul problème de cette approche est un risque d'incohérence entre les fils d'exécution. Or, une cohérence strice de la matrice de phéromone n'est pas essentiel au bon déroulement de l'algorithme et il a donc été choisi de permettre une certaine incohérence.

5.2.2 *Boost*

Pour cette implantation, des mutex et un mutex partagé (mutex de type lecteurs/écrivain) ont été ajoutés aux attributs de la classe colonie. Le code modifié est présenté à l'annexe C. Puisque *Boost* rend facilement accessible l'utilisation d'un verrous partagé, il a été utilisé pour protéger l'accès à la matrice de phéromone. Il est à noter que dans le code, j'explore les fonctionnalités de *Boost* et les verrous sont donc parfois vérouillés directement à l'aide de la méthode *lock()* et parfois à l'aide des objets utilisant la philosophie *RAII*. Pour un code à plus grande échelle, il me semble préférable d'utiliser une approche plus uniforme afin de diminuer le risque d'erreurs.

6 Analyse de performance

Afin de comparer la performance des deux solutions implantés, les programmes ont été exécutés à répétition sous différentes conditions. Les tests ont été effectués au laboratoire *D4-1017* du département. L'ordinateur utiilisé possédait un processeur *i7-2600* contenant 8 coeurs cadencés à 3.4 GHz ainsi que 8 Go de RAM. De plus, les tests ont été exécutés sur le système d'exploitation Ubuntu 10.04. Tous les tests ont été effectués en calculant sur un même graphe de 100 noeuds.

La figure 1 présente le temps de calcul en secondes de la solution utilisant *Boost* pour trois charges de calculs en fonction du nombre de fils d'exécution. Dans tout ce qui suit, la notion de charge de calcul réfère à la durée du calcul en unités de longueur d'arcs du graphe. Ainsi, pour une charge de 100, le programme arrêtera lorsqu'il arrivera aux évènements de temps $t = 100$. On voit que le temps de calcul diminue bien avec l'augmentation du nombre de coeurs. Ainsi, le coût de la synchronisation n'est pas supérieur à celui du calcul et le parallélisme permet donc de faire un gain en temps d'horloge murale. Il est intéressant de remarquer que les gains sont très grand en passant de 1 à 2 puis de 2 à 4 fils, mais beaucoup plus faible pour le passage à 6 ou bien à 8 fils. Par exemple, pour une charge de 400, l'accélélation du calcul à deux fils est de 1,86 ($T_{1coeur}/T_{2coeurs}$), alors que pour 4 fils il est de 2.97, pour 6 fils de 4,12 et pour 8 fils de 4,95. Ainsi, selon les besoins et les objectifs, il peut être préférable de lancer deux programmes simultanéement utilisant chacun 4 fils (donc 4 coeurs) par exemple plutôt que de lancer un programme utilisant les 8 coeurs.

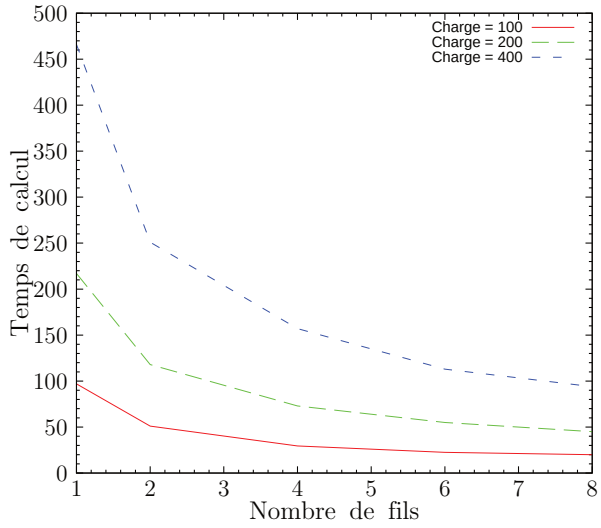

FIGURE 1 – Temps de calcul du programme utilisant *Boost* en fonction du nombre de *threads* pour trois charges de travail différentes.

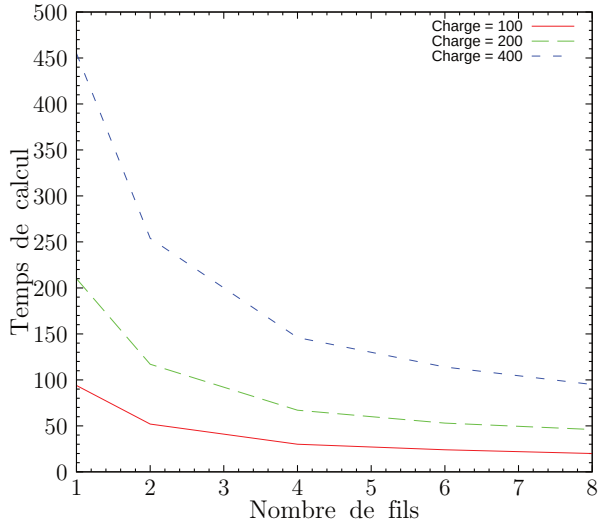

FIGURE 2 – Temps de calcul du programme utilisant *OpenMp* en fonction du nombre de *threads* pour trois charges de travail différentes.

La figure 2 présente le même type de résultats mais pour la solution utilisant *OpenMP* les résultats sont pratiquemment identiques à ceux de la figure précédente. J'ai été surpris de ces résultats, puisque tel que discuté précédemment, la solution utilisant *Boost* utilise un verrou de type lecteurs/écrivain pour la matrice de phéromone, alors que la solution *OpenMP* ne synchronise que les écritures à cette matrice et laisse libre les lectures. Les temps de virement sont donc équivalents pour les deux programmes malgré cette distinction.

Finalement, la figure 3 présente le temps de calcul en fonction de la charge pour les deux programmes. Il intéressant d'observer que celui-ci est le même pour les deux solutions. Il ne semble donc pas y avoir de gain inhérent à l'utilisation d'un outil par rapport à l'autre. De plus, la relation semble plutôt linéaire. Doubler la charge de calcul double le temps de calcul.

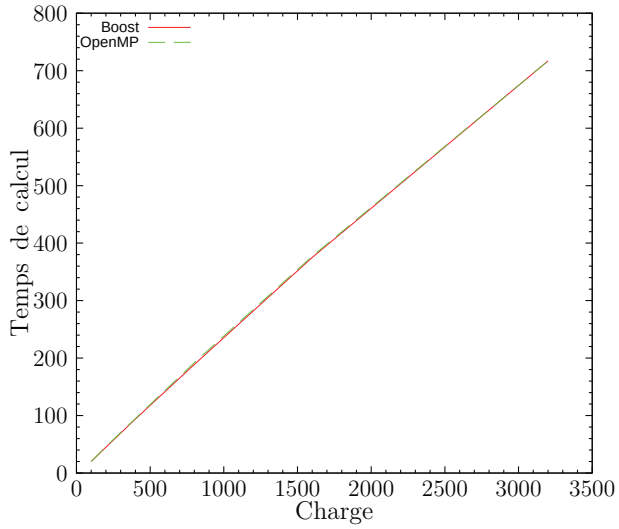

FIGURE 3 – Comparaison du temps de calcul en fonction de la charge pour les deux programmes parallèles. Dans les deux cas, 8 fils d'exécutions ont été utilisés.

7 Conclusion

Ce projet m'aura permis d'explorer deux outils permettant la programmation parallèle selon une approche utilisant les multiples fils d'exécutions. Pour conclure, je discuterai brièvement de mon appréciation de ces deux outils. De mon point de vue, la grande force de *OpenMP* est sa facilité d'utilisation. Ainsi, une fois que tu connais seulement quelques directives de bases, il peut être excessivement rapide de paralléliser un programme simple. Par contre, le contrepoids de cette facilité d'utilisation est la quantité d'outils de synchronisation beaucoup plus limitée que *Boost*. L'exemple flagrant est l'absence d'un verrou de type lecteurs/écrivain. De son côté, *Boost* est beaucoup plus complet en terme de fonctionnalités. Par contre, ceci à le défaut de le rendre un peu plus difficile d'approche.

Je retire donc de ce projet que pour les projets simple nécessitant peu de synchronisation et de communication entre les fils *OpenMP* est un outil idéale. Au contraire, dans le cas d'un projet un peu plus élaboré et nécessitant d'avantage d'échanges et de synchronisation entre les fils, l'utilisation de *boost* peut être préférable.

A Code séquentiel

A.1 *graphe.h*

```
1  /*
2   *  Classe graphe : fichier d'en-tête
3   *  Créé par Jean Goulet et modifier par Samuel Boutin
4   */
5
6  #pragma once
7  #include <iostream>
8  #include <fstream>
9  #include <map>
10 #include <string>
11 #include <set>
12 #include <list>
13
14 using namespace std;
15
16 string lire_ETIQ(istream&);
17 void changer_repertoire(const char * argv);
```

```
18
19 class graphe:public set<string>{
20 public:
21     void lire(istream&);
22     void afficher(ostream&);
23     void afficher_chemin(list<string>,ostream&);
24     bool contient(string);
25     double valeur_arc(string,string)const;
26
27 private:
28     map<string,map<string,double>> lesPoids;
29 };
```

A.2 *graphe.cpp*

```
1  /*
2   *  Code de la classe graphe
3   *  Créé par Jean Goulet et modifier par Samuel Boutin
4   */
5
6  #include "graphe.h"
7
8  #include <iostream>
9  #include <fstream>
10 #include <map>
11 #include <string>
12 #include <set>
13 #include <list>
14 #include <unistd.h>
15
16 using namespace std;
17
18 void graphe::afficher_chemin(list<string> ch,ostream& flot){
19     if(ch.size()<2){
20         flot<<"Aucun chemin"<<endl;
21         return;
22         }
23     string a=ch.back(),de=ch.front();
24     double unP,totP=0.;
25     flot<<endl<<"Chemin de "<<de<<" a "<<a<<endl;
26     for(ch.pop_front();!ch.empty();ch.pop_front()){
27         a=ch.front();
28         unP=lesPoids[de][a];
29         totP+=unP;
30         flot<<de<<" "<<a<<" "<<unP<<endl;
31         de=a;
32         }
33     flot<<"Total= "<<totP<<endl;
34     }
35
36 void graphe::lire(istream& flot){
37     string dep,arr;
38     double poi;
39     clear();
40     lesPoids.clear();
41     while(true){
42         dep=lire_ETIQ(flot);
```

```
43          arr=lire_ETIQ(flot);
44          flot>>poi;
45          if(!flot)break;
46          insert(dep);
47          insert(arr);
48          lesPoids[dep][arr]=poi;
49          }
50      }
51
52  void graphe::afficher(ostream& flot){
53      flot<<"Le graphe du projet..."<<endl;
54      map<string,map<string,double> >::iterator iL=lesPoids.begin();
55      map<string,double>::iterator iA;
56      for(;iL!=lesPoids.end();iL++){
57          map<string,double>& AD=iL->second;
58          flot<<iL->first;
59          for(iA=AD.begin();iA!=AD.end();iA++)
60              flot<<" -> "<<iA->first<<" "<<iA->second;
61          flot<<endl;
62          }
63      }
64
65  bool graphe::contient(string e){
66      return count(e)!=0;
67      }
68
69  double graphe::valeur_arc(string de,string a)const{
70      map<string,map<string,double> >::const_iterator i;
71      map<string,double>::const_iterator j;
72      i=lesPoids.find(de);
73      if(i!=lesPoids.end()){
74          j=i->second.find(a);
75          if(j!=i->second.end())return j->second;
76          }
77      return -1;
78  }
79
80  ////////////////////////////////////////////////////////////
81
82  string lire_ETIQ(istream& entree){
83      string re;
84      entree>>re;
85      for(int i=0;i<re.length();i++)re[i]=tolower(re[i]);
86      return re;
87      }
88
89  ////////////////////////////////////////////////////////////
90
91  void changer_repertoire(const char * argv){
92  #ifdef __APPLE__
93      const char delim='/';     //MAC
94  #else
95      const char delim='\\';    //PC
96  #endif
97
98      cout<<"ATTENTION! Ce programme ouvre les fichiers dans son repertoire d'origine"
```

```
        <<endl<<endl;
99      string repertoire(argv);
100     int i=repertoire.find_last_of(delim);
101     if(i>0){
102         repertoire.resize(i);
103         //cout<<repertoire<<endl;
104         i=chdir(repertoire.c_str());
105     }
106 }
```

A.3 *fourmi.h*

```
1  /*
2   *  Classe fourmi : fichier d'en-tête
3   *  Créé par Samuel Boutin
4   *  Janvier 2012
5   */
6  #pragma once
7  #include <set>
8  #include <list>
9  #include <map>
10 #include <string>
11
12
13 using namespace std;
14
15 ////////////////////////////////////////////////////////////
16
17 class fourmi{
18 public:
19     //Méthodes public:
20     fourmi(string, set<string>);
21     string noeudActuel();
22     string avancer(map<string,double>&, map<string,double>&);
23     void retourDepart();
24     bool circuitEstComplet();
25     double poidsDuCircuit();
26     list<string> circuitComplet();
27 private:
28     //Attributs de la classe :
29     string noeudDepart;
30     list<string> chemin;
31     set<string> reste;
32     double poidsTotal;
33     //Méthodes privées :
34     double hasard();
35 };
```

A.4 *fourmi.cpp*

```
1  /*
2   *  Code de la classe fourmi
3   *  Créé par Samuel Boutin
4   *  Janvier 2012
5   */
6  #include "fourmi.h"
7
```

```
 8 #include <map>
 9 #include <string>
10 #include <set>
11 #include <list>
12 #include <unistd.h>
13 #include <iostream>
14 #include <cstdlib>
15
16 using namespace std;
17
18 fourmi::fourmi(string depart, set<string> ensembleNoeuds){
19     noeudDepart = depart;
20     reste=ensembleNoeuds;
21     poidsTotal = 0.0;
22
23     // Se place au premier point :
24     chemin.push_back(noeudDepart);
25     reste.erase(noeudDepart);
26 }
27
28 string fourmi::noeudActuel(){
29     return chemin.back();
30 }
31
32 string fourmi::avancer(map<string,double>& proba, map<string,double>& poids){
33     map<string,double>::iterator i;
34     i = proba.begin();
35     double total(0.); // Somme de la probabilité des noeuds disponibles
36     double nbHasard;
37     string noeudChoisi;
38
39     // Rends indisponible tous les noeuds ayant déjà été sélectionné.
40     for(i;i!=proba.end();++i){
41         if (!reste.count(i->first)){
42             i->second = 0;
43         }
44         total+=i->second;
45     }
46
47     // Cas ou aucun noeud n'est disponible :
48     if (total==0){
49         // Si tous les noeuds ont été visité et que le circuit peut être fermé :
50         if (reste.empty() && proba.count(noeudDepart)){
51             // Ferme la boucle
52             chemin.push_back(noeudDepart);
53             poidsTotal +=poids[noeudDepart];
54             return "circuit";
55         }
56         else{
57             retourDepart();
58             return "bloque";
59         }
60     }
61     else{
62         // Génération d'un nombre pseudo-aléatoire :
63         nbHasard=total*hasard();
```

```
64          // Choix du prochain noeud :
65          for(i=proba.begin();i!=proba.end();++i){
66              if((i->second > nbHasard) && (i->second != 0.0)){
67                  noeudChoisi= i->first;
68                  chemin.push_back(noeudChoisi);
69                  reste.erase(noeudChoisi);
70                  poidsTotal +=poids[noeudChoisi];
71                  return noeudChoisi;
72              }
73              else{
74                  nbHasard -= i->second;
75              }
76          }
77      }
78      // Ne devrait pas se produire.
79      return "erreur";
80  }
81
82  void fourmi::retourDepart(){
83      poidsTotal = 0;
84      list<string>::iterator i;
85      for (i=chemin.begin();i!=chemin.end();++i){
86          reste.insert(*i);
87      }
88      chemin.clear();
89      chemin.push_back(noeudDepart);
90      reste.erase(noeudDepart);
91  }
92
93  bool fourmi::circuitEstComplet(){
94      if (chemin.front()==chemin.back()){
95          return true;
96      }
97      else{
98          return false;
99      }
100 }
101
102 double fourmi::poidsDuCircuit(){
103     if (circuitEstComplet()){
104         return poidsTotal;
105     }
106     else{
107         return -1.;
108     }
109 }
110
111 list<string> fourmi::circuitComplet(){
112     return chemin;
113 }
114
115 double fourmi::hasard(){
116     double resultat(0);
117     double valeur(0.5);
118     for(int i=0;i<53;i++){
119         resultat+=rand()%2*valeur;
```

```
120         valeur/=2;
121     }
122     return resultat;
123 }
```

A.5 *colonie.h*

```
1  /*
2   *  Classe colonie : fichier d'en-tête
3   *  Créé par Samuel Boutin
4   *  Janvier 2012
5   */
6  #pragma once
7  #include <map>
8  #include <list>
9  #include <vector>
10 #include <string>
11
12 #include "fourmi.h"
13 #include "graphe.h"
14
15 using namespace std;
16 class graphe;
17
18 // Création d'une structure pour contenir l'ensemble des paramètres de gestion de
       la colonie.
19 struct paramsColonie
20 {
21     int nb_fourmis; // Nombre de fourmis par noeud.
22     double minPh;   // Quantité minimale de phéromone par arc.
23     double maxPh;   // Quantité maximale de phéromone par arc.
24     double delai;   // Intervalle de temps auquel la phéromone est évaporée.
25     double rho;     // Coefficient d'évaporation de la phéromone.
26     double coeffPh; // Quantité de phéromone a distribué.
27     double alpha;   // Exposant de l'indice de visibilité.
28     double beta;    // Exposant de la phéromone.
29 };
30
31 class colonie{
32 public:
33     //Méthodes public:
34     colonie(const graphe&);
35     list<string> trouverCircuit(double);
36
37 private:
38     //Attributs de la classe :
39     vector<fourmi> lesFourmis;
40     map<string,map<string,double>> pheromone;
41     list<string> meilleurCircuit;
42     double meilleurPoids;
43     multimap<double,int> agenda;
44     // Structure contenant les paramètres de la colonie :
45     paramsColonie parametres;
46     //Pointeur vers le graphe que l'on cherche à solutionner.
47     const graphe *G;
48
49     //Méthodes privées :
```

```
50
51      // Chaque fourmi demande à la colonie le poids de chacun des arcs voisins.
52      map<string,double> poidsDesArcsVoisins(string);
53      // Chaque fourmi demande à la colonie la probabilité de chacun des arcs voisins.
54      map<string,double> calculProba(string,const map<string,double>&);
55      // Évaporation de la phéromone :
56      void evapPh();
57      // Chaque fourmi demande à la colonie de distribuer de la phéromone sur son
            circuit lorsqu'il est complété.
58      void ajouterPh(const list<string>&,double);
59  };
```

A.6 *colonie.cpp*

```
1   /*
2    *  Code de la classe colonie
3    *  Créé par Samuel Boutin
4    *  Janvier 2012
5    */
6
7   #include "colonie.h"
8   #include "graphe.h"
9
10  #include <map>
11  #include <string>
12  #include <set>
13  #include <list>
14  #include <iostream>
15  #include <cmath>
16  #include <unistd.h>
17  #include <cstdlib>
18
19  using namespace std;
20
21  colonie::colonie(const graphe& _G){
22      // 1) Conserve un pointeur vers le graphe afin d'avoir accès à la matrice des
            poids via la méthode valeur_arc.
23      G = &_G;
24
25      // 2) Initialisation des paramètres :
26      // (Il serait également possible de construire un autre constructeur acceptant
            directement la structure en paramètre).
27      parametres.nb_fourmis = 20;  // Nombre de fourmis par noeud
28      parametres.minPh = 1;        // Quantité minimale de phéromone sur chaque arc
29      parametres.maxPh = 200;      // Quantité maximale de phéromone sur chaque arc
30      parametres.rho   = 0.5;      // Taux d'évaporation de la phéromone
31      parametres.coeffPh = 100;    // Lors de la distribution de la phéromone, une
            quantité coeffPh/lopngueur_arc sera distribuée sur chaque arc
32      parametres.alpha = 1;        // Exposant de l'inverse de la longueur de l'arc
            dans le calcul des probabilités
33      parametres.beta =1;          // Exposant de la phéromone dans le calcul des
            probabilités.
34
35
36      // 3) Initialisation du vecteur de fourmis :
37      int totalFourmi = parametres.nb_fourmis*G->size();
38      lesFourmis.reserve(totalFourmi);
```

```
39
40
41      // 4) Initialisation de chaque fourmi :
42      // Afin d'initialiser les fourmis, récupère l'ensemble des noeuds du graphe.
43      set<string> tousLesNoeuds;
44      set<string>::iterator it;
45      for (it=G->begin();it!=G->end();++it){
46          tousLesNoeuds.insert(*it);
47      }
48
49      // Boucle sur les noeuds :
50      for(it=G->begin();it!=G->end();++it){
51          // Boucle sur le nombre de fourmis
52          for(int j=0;j<parametres.nb_fourmis;++j){
53              lesFourmis.push_back(fourmi(*it,tousLesNoeuds));
54          }
55      }
56
57
58      // 5) Initialisation de la matrice creuse de phéromone :
59      set<string>::iterator it1;
60      set<string>::iterator it2;
61      // Boucle sur l'ensemble des noeuds
62      for (it1=G->begin();it1!=G->end();++it1){
63          // Boucle sur l'ensemble des noeuds
64          for (it2=G->begin();it2!=G->end();++it2){
65              if (G->valeur_arc(*it1,*it2) != -1){
66                  pheromone[*it1][*it2] = parametres.minPh;
67              }
68          }
69      }
70
71
72      // 6) Initialisation de la fréquence d'évaporation de la phéromone :
73      map<string, map<string,double> >::iterator itmap1;
74      map<string, double>::iterator itmap2;
75      itmap1 = pheromone.begin();
76      itmap2 = (itmap1->second).begin();
77      parametres.delai = G->size()*G->valeur_arc(itmap1->first,itmap2->first)*5.;
78
79
80      // 7) Initialisation de l'agenda :
81      // Un événement pour chaque fourmi
82      for (int j=0;j<totalFourmi;++j){
83          agenda.insert(pair<double,int>(0.,j) );
84      }
85      // Événement pour l'évaporation de la phéromone :
86      agenda.insert(pair<double,int>(parametres.delai,-1));
87  }
88
89  list<string> colonie::trouverCircuit(double tempsEvol){
90      // Trouve le prochain événement dans l'agenda :
91      multimap<double,int>::iterator i;
92      i=agenda.begin();
93      // Insère une condition d'arrêt :
94      agenda.insert(pair<double,int>(i->first+tempsEvol,-2));
```

```
95
96      double t;
97      string noeud1,noeud2;
98      map<string,double> prob;
99      map<string,double> arcs;
100     pair<double,int> ev; // Événement à traiter.
101
102     while(true){
103         i=agenda.begin();
104         ev = *i;
105         agenda.erase(i);
106         if (ev.second ==-2){
107             // Fin du temps de recherche
108             return meilleurCircuit;
109         }
110         if (ev.second == -1){
111             // Évaporation de la phéromone
112             evapPh();
113             ev.first+=parametres.delai;
114         }
115         else{
116             // Fait avancer une fourmi :
117             noeud1 = lesFourmis[ev.second].noeudActuel();
118             arcs = poidsDesArcsVoisins(noeud1);
119             prob = calculProba(noeud1,arcs);
120             // La colonie fournie à la fourmi la probabilité et la longueur des arcs
                    voisins.
121             noeud2 = lesFourmis[ev.second].avancer(prob,arcs);
122             if (noeud2=="circuit"){
123                 // Vérifie si le circuit trouver est le meilleur :
124                 if((lesFourmis[ev.second].poidsDuCircuit() < meilleurPoids) ||
                        meilleurCircuit.empty()){
125                     meilleurPoids = lesFourmis[ev.second].poidsDuCircuit();
126                     meilleurCircuit = lesFourmis[ev.second].circuitComplet();
127                 }
128                 ajouterPh(lesFourmis[ev.second].circuitComplet(),lesFourmis[ev.second].
                        poidsDuCircuit());
129             //    cout << "Poids du circuit touvé : " << lesFourmis[ev.second].
                    poidsDuCircuit() << "   au temps t = " << ev.first << endl;
130                 lesFourmis[ev.second].retourDepart();
131             }
132             else if (noeud2 == "erreur"){
133                 exit(-1);
134             }
135             else if (noeud2 !="bloque"){
136                 ev.first += G->valeur_arc(noeud1,noeud2);
137             }
138             // Remet la fourmi à l'agenda.
139             // Dans le cas où la fourmi est dans un cul-de-sac, elle retourne au dé
                    part par elle-même.
140             // Elle est remis à l'agenda au même temps.
141             agenda.insert(ev);
142         }
143     }
144 }
145
```

```
146 // Demande au graphe le poids des arcs voisins d'un noeud donné.
147 map<string,double> colonie::poidsDesArcsVoisins(string noeudActuel){
148     // Utilise la matrice de phéromone afin de connaître les noeuds voisins.
149     map<string,double> poids = pheromone[noeudActuel];
150     map<string,double>::iterator it;
151     for (it=poids.begin();it!=poids.end();++it){
152         poids[it->first]=G->valeur_arc(noeudActuel,it->first);
153     }
154 
155     return poids;
156 }
157 
158 map<string,double> colonie::calculProba(string noeud,const map<string,double>&
        voisins){
159     map<string,double> probabilites;
160     map<string,double>::iterator it;
161     probabilites = voisins;
162     double visibilite;
163     for(it=pheromone[noeud].begin();it!=pheromone[noeud].end();++it){
164         visibilite =1/probabilites[it->first];
165         probabilites[it->first] =pow(visibilite,parametres.alpha)*pow(it->second,
              parametres.beta);
166     }
167 
168     return probabilites;
169 }
170 
171 void colonie::evapPh(){
172     map<string,map<string,double> >::iterator it1;
173     map<string,double>::iterator it2;
174     for (it1=pheromone.begin();it1!=pheromone.end();++it1){
175         for (it2=pheromone[it1->first].begin();it2!=pheromone[it1->first].end();++it2
              ){
176             it2->second *=parametres.rho;
177             if (it2->second < parametres.minPh){
178                 it2->second = parametres.minPh;
179             }
180         }
181     }
182 }
183 
184 // Ajout de la phéromone. Total : poids total du circuit.
185 void colonie::ajouterPh(const list<string>& chemin,double total){
186     // Utilisation de deux itérateurs décalés d'une position.
187     list<string>::const_iterator it1;
188     list<string>::const_iterator it2;
189     // Plus le circuit est court, plus il y a de phéromone.
190     double ratio = parametres.coeffPh/total;
191     it2 =chemin.begin();
192     ++it2;
193     for(it1=chemin.begin();it2!=chemin.end();++it1){
194         pheromone[*it1][*it2] +=ratio ;
195         if (pheromone[*it1][*it2] > parametres.maxPh)
196         {
197                 pheromone[*it1][*it2] = parametres.maxPh;
198         }
```

```
199         ++it2;
200     }
201 }
```

A.7 *main.cpp*

```
1  /*
2   * main.cpp
3   * Par : Jean Goulet, modifié par Samuel Boutin
4   * TP1 ift339 : Structure de données
5   * 30 janvier 2012
6   */
7
8  #include <iostream>
9  #include <fstream>
10 #include <map>
11 #include <string>
12 #include <set>
13 #include <list>
14 #include <ctime>
15
16 #include "graphe.h"
17 #include "colonie.h"
18
19 using namespace std;
20
21 int main(int argc, char * const argv[]){
22     cout<<"Debut du programme"<<endl;
23     graphe G;
24     list<string> ch;
25     string nomFichier;
26     changer_repertoire(argv[0]);
27
28     cout<<"Nom du fichier: ";cin>>nomFichier;
29     ifstream flot(nomFichier.c_str());
30     if(!flot){cout<<"Erreur de fichier..."<<endl;return 1;}
31     G.lire(flot);
32     flot.close();
33     G.afficher(cout);
34
35     // Création d'une colonie de fourmi réparti sur tous les noeuds du graphe.
36     colonie c(G);
37     // Recherche du meilleur circuit disponible en un temps donné :
38     double tempsRecherche;
39     cout << "Combien de temps (en unité de longueur d'arcs) souhaitez-vous chercher
           une solution ?" << endl;
40     cin >> tempsRecherche;
41
42     long avant,apres;
43     avant=time(0L);
44     ch = c.trouverCircuit(tempsRecherche);
45
46     apres=time(0L);
47     G.afficher_chemin(ch,cout);
48     cout<<(apres-avant)<<" secondes"<<endl;
49     cout<<endl<<"Fin"<<endl;
50
```

```
51      return 0;
52  }
```

B Code *OpenMP*

B.1 *colonie.cpp*

```
1   /*
2    *  Code de la classe colonie
3    *  Créé par Samuel Boutin
4    *  Janvier 2012
5    */
6
7   #include "colonie.h"
8   #include "graphe.h"
9
10  #include <map>
11  #include <string>
12  #include <set>
13  #include <list>
14  #include <iostream>
15  #include <cmath>
16  #include <unistd.h>
17  #include <cstdlib>
18
19  #include <boost/thread.hpp>
20
21
22  #define nbThread 8
23
24  using namespace std;
25
26  colonie::colonie(const graphe& _G){
27      // 1) Conserve un pointeur vers le graphe afin d'avoir accès à la matrice des
            poids via la méthode valeur_arc.
28      G = &_G;
29
30      // 2) Initialisation des paramètres :
31      // (Il serait également possible de construire un autre constructeur acceptant
            directement la structure en paramètre).
32      parametres.nb_fourmis = 20;   // Nombre de fourmis par noeud
33      parametres.minPh = 1;         // Quantité minimale de phéromone sur chaque arc
34      parametres.maxPh = 200;       // Quantité maximale de phéromone sur chaque arc
35      parametres.rho    = 0.5;      // Taux d'évaporation de la phéromone
36      parametres.coeffPh = 100;     // Lors de la distribution de la phéromone, une
            quantité coeffPh/lopngueur_arc sera distribuée sur chaque arc
37      parametres.alpha = 1;         // Exposant de l'inverse de la longueur de l'arc
            dans le calcul des probabilités
38      parametres.beta =1;           // Exposant de la phéromone dans le calcul des
            probabilités.
39
40
41      // 3) Initialisation du vecteur de fourmis :
42      int totalFourmi = parametres.nb_fourmis*G->size();
43      lesFourmis.reserve(totalFourmi);
44
45
```

```
46      // 4) Initialisation de chaque fourmi :
47      // Afin d'initialiser les fourmis, récupère l'ensemble des noeuds du graphe.
48      set<string> tousLesNoeuds;
49      set<string>::iterator it;
50      for (it=G->begin();it!=G->end();++it){
51          tousLesNoeuds.insert(*it);
52      }
53
54      // Boucle sur les noeuds :
55      for(it=G->begin();it!=G->end();++it){
56          // Boucle sur le nombre de fourmis
57          for(int j=0;j<parametres.nb_fourmis;++j){
58              lesFourmis.push_back(fourmi(*it,tousLesNoeuds));
59          }
60      }
61
62
63      // 5) Initialisation de la matrice creuse de phéromone :
64      set<string>::iterator it1;
65      set<string>::iterator it2;
66      // Boucle sur l'ensemble des noeuds
67      for (it1=G->begin();it1!=G->end();++it1){
68          // Boucle sur l'ensemble des noeuds
69          for (it2=G->begin();it2!=G->end();++it2){
70              if (G->valeur_arc(*it1,*it2) != -1){
71                  pheromone[*it1][*it2] = parametres.minPh;
72              }
73          }
74      }
75
76
77      // 6) Initialisation de la fréquence d'évaporation de la phéromone :
78      map<string, map<string,double> >::iterator itmap1;
79      map<string, double>::iterator itmap2;
80      itmap1 = pheromone.begin();
81      itmap2 = (itmap1->second).begin();
82      parametres.delai = G->size()*G->valeur_arc(itmap1->first,itmap2->first)*5.;
83
84
85      // 7) Initialisation de l'agenda :
86      // Un événement pour chaque fourmi
87      for (int j=0;j<totalFourmi;++j){
88          agenda.insert(pair<double,int>(0.,j) );
89      }
90      // Événement pour l'évaporation de la phéromone :
91      agenda.insert(pair<double,int>(parametres.delai,-1));
92  }
93
94  list<string> colonie::trouverCircuit(double tempsEvol){
95          // Création des fils travailleurs.
96          vector<boost::thread*> lesTravailleurs(nbThread);
97          for (int i=0;i<nbThread;++i)
98          {
99                  lesTravailleurs[i] = new boost::thread(&colonie::trouver,this,
                        tempsEvol);
100         }
```

```
101
102
103         // Attend la fin de chaque fil.
104         for (int i=0;i<nbThread;++i)
105         {
106                 lesTravailleurs[i]->join();
107                 delete lesTravailleurs[i];
108         }
109
110         return meilleurCircuit;
111 }
112
113
114 void colonie::trouver(double tempsEvol)
115 {
116
117         boost::thread::id id =  boost::this_thread::get_id();
118         ecran.lock();
119                 cout << "Thread " << id << endl;
120         ecran.unlock();
121
122   // Trouve le prochain événement dans l'agenda :
123    multimap<double,int>::iterator i;
124   /*
125    modifAgenda.lock();
126             i=agenda.begin();
127             // Insère une condition d'arrêt :
128             agenda.insert(pair<double,int>(i->first+tempsEvol,-2));
129    modifAgenda.unlock();
130 */
131 {
132         boost::unique_lock<boost::mutex> lock(modifAgenda);
133            i=agenda.begin();
134            // Insère une condition d'arrêt :
135            agenda.insert(pair<double,int>(i->first+tempsEvol,-2));
136 }
137    double t;
138    string noeud1,noeud2;
139    map<string,double> prob;
140    map<string,double> arcs;
141    pair<double,int> ev; // Événement à traiter.
142
143    while(true){
144       modifAgenda.lock();
145                i=agenda.begin();
146                ev = *i;
147                agenda.erase(i);
148       modifAgenda.unlock();
149       if (ev.second ==-2){
150          // Fin du temps de recherche
151          return;
152       }
153       if (ev.second == -1){
154          // Évaporation de la phéromone
155          evapPh();
156          ev.first+=parametres.delai;
```

```
157         }
158         else{
159             // Fait avancer une fourmi :
160             noeud1 = lesFourmis[ev.second].noeudActuel();
161             arcs = poidsDesArcsVoisins(noeud1);
162             prob = calculProba(noeud1,arcs);
163             // La colonie fournie à la fourmi la probabilité et la longueur des arcs
                    voisins.
164             noeud2 = lesFourmis[ev.second].avancer(prob,arcs);
165 
166             if (noeud2=="circuit"){
167                 // Vérifie si le circuit trouver est le meilleur :
168                 modifMeilleur.lock();
169                 if((lesFourmis[ev.second].poidsDuCircuit() < meilleurPoids) ||
                        meilleurCircuit.empty()){
170                     meilleurPoids = lesFourmis[ev.second].poidsDuCircuit();
171                     meilleurCircuit = lesFourmis[ev.second].circuitComplet();
172                 }
173                 modifMeilleur.unlock();
174 
175                 ajouterPh(lesFourmis[ev.second].circuitComplet(),lesFourmis[ev.second].
                        poidsDuCircuit());
176             //     cout << "Poids du circuit touvé : " << lesFourmis[ev.second].
                    poidsDuCircuit() << "   au temps t = " << ev.first << endl;
177                 lesFourmis[ev.second].retourDepart();
178             }
179             else if (noeud2 == "erreur"){
180                 exit(-1);
181             }
182             else if (noeud2 !="bloque"){
183                     ev.first += G->valeur_arc(noeud1,noeud2);
184             }
185             // Remet la fourmi à l'agenda.
186             // Dans le cas où la fourmi est dans un cul-de-sac, elle retourne au dé
                    part par elle-même.
187             // Elle est remis à l'agenda au même temps.
188             boost::unique_lock<boost::mutex> l(modifAgenda);
189             agenda.insert(ev);
190         }
191     }
192 
193 }
194 
195 
196 // Demande au graphe le poids des arcs voisins d'un noeud donné.
197 map<string,double> colonie::poidsDesArcsVoisins(string noeudActuel){
198     // Utilise la matrice de phéromone afin de connaître les noeuds voisins.
199     accesPhero.lock_shared();
200     map<string,double> poids = pheromone[noeudActuel];
201     accesPhero.unlock_shared();
202     map<string,double>::iterator it;
203     for (it=poids.begin();it!=poids.end();++it){
204         poids[it->first]=G->valeur_arc(noeudActuel,it->first);
205     }
206 
207     return poids;
```

```
208 }
209
210 map<string,double> colonie::calculProba(string noeud,const map<string,double>&
        voisins){
211     map<string,double> probabilites;
212     map<string,double>::iterator it;
213     probabilites = voisins;
214     double visibilite;
215
216     accesPhero.lock_shared();
217     for(it=pheromone[noeud].begin();it!=pheromone[noeud].end();++it){
218         visibilite =1/probabilites[it->first];
219         probabilites[it->first] =pow(visibilite,parametres.alpha)*pow(it->second,
              parametres.beta);
220     }
221     accesPhero.unlock_shared();
222
223     return probabilites;
224 }
225
226 void colonie::evapPh(){
227     map<string,map<string,double> >::iterator it1;
228     map<string,double>::iterator it2;
229     {
230             boost::unique_lock<boost::shared_mutex> lock(accesPhero);
231
232             for (it1=pheromone.begin();it1!=pheromone.end();++it1){
233                     for (it2=pheromone[it1->first].begin();it2!=pheromone[it1->first].
                          end();++it2){
234                         it2->second *=parametres.rho;
235                         if (it2->second < parametres.minPh){
236                                 it2->second = parametres.minPh;
237                         }
238                     }
239             }
240     }
241 }
242
243 // Ajout de la phéromone. Total : poids total du circuit.
244 void colonie::ajouterPh(const list<string>& chemin,double total){
245     // Utilisation de deux itérateurs décalés d'une position.
246     list<string>::const_iterator it1;
247     list<string>::const_iterator it2;
248     // Plus le circuit est court, plus il y a de phéromone.
249     double ratio = parametres.coeffPh/total;
250     it2 =chemin.begin();
251     ++it2;
252     boost::unique_lock<boost::shared_mutex> lock(accesPhero);
253
254     for(it1=chemin.begin();it2!=chemin.end();++it1){
255         pheromone[*it1][*it2] +=ratio ;
256         if (pheromone[*it1][*it2] > parametres.maxPh)
257         {
258                 pheromone[*it1][*it2] = parametres.maxPh;
259         }
260         ++it2;
```

```
261     }
262 }
```

C Code *Boost*

C.1 *colonie.h*

```
1  /*
2   *  Classe colonie : fichier d'en-tête
3   *  Créé par Samuel Boutin
4   *  Janvier 2012
5   */
6  #pragma once
7  #include <map>
8  #include <list>
9  #include <vector>
10 #include <string>
11 /*
12 #include "../../../boost_1_46_1/boost/thread/thread.hpp"
13 #include "../../../boost_1_46_1/boost/thread/detail/thread_group.hpp"
14 #include "../../../boost_1_46_1/boost/thread/mutex.hpp"
15 #include "../../../boost_1_46_1/boost/thread/shared_mutex.hpp"
16 */
17 #include <boost/thread.hpp>
18
19 #include "fourmi.h"
20 #include "graphe.h"
21
22 using namespace std;
23 class graphe;
24
25 // Création d'une structure pour contenir l'ensemble des paramètres de gestion de
       la colonie.
26 struct paramsColonie
27 {
28     int nb_fourmis; // Nombre de fourmis par noeud.
29     double minPh;   // Quantité minimale de phéromone par arc.
30     double maxPh;   // Quantité maximale de phéromone par arc.
31     double delai;   // Intervalle de temps auquel la phéromone est évaporée.
32     double rho;     // Coefficient d'évaporation de la phéromone.
33     double coeffPh; // Quantité de phéromone a distribué.
34     double alpha;   // Exposant de l'indice de visibilité.
35     double beta;    // Exposant de la phéromone.
36 };
37
38 class colonie{
39 public:
40     //Méthodes public:
41     colonie(const graphe&);
42     list<string> trouverCircuit(double);
43     void trouver(double);
44
45
46 private:
47     //Attributs de la classe :
48     vector<fourmi> lesFourmis;
49     map<string,map<string,double> > pheromone;
```

```
50      list<string> meilleurCircuit;
51      double meilleurPoids;
52      multimap<double,int> agenda;
53      // Structure contenant les paramètres de la colonie :
54      paramsColonie parametres;
55      //Pointeur vers le graphe que l'on cherche à solutionner.
56      const graphe *G;
57
58      //Méthodes privées :
59
60      // Chaque fourmi demande à la colonie le poids de chacun des arcs voisins.
61      map<string,double> poidsDesArcsVoisins(string);
62      // Chaque fourmi demande à la colonie la probabilité de chacun des arcs voisins.
63      map<string,double> calculProba(string,const map<string,double>&);
64      // Évaporation de la phéromone :
65      void evapPh();
66      // Chaque fourmi demande à la colonie de distribuer de la phéromone sur son
            circuit lorsqu'il est complété.
67      void ajouterPh(const list<string>&,double);
68
69          // Variable de synchoronisation
70          // 1) Synchronisation de l'agenda:
71          boost::mutex modifAgenda;
72          // 2) Synchronisation de la phéromone :
73          boost::shared_mutex accesPhero;
74          // 3) Synchronisation du meilleur circuit :
75          boost::mutex modifMeilleur;
76          // 4) Synchoronisation des affichages à l'écran (temporaire)
77          boost::mutex ecran;
78
79
80
81  };
```

C.2 *colonie.cpp*

```
1   /*
2    *  Code de la classe colonie
3    *  Créé par Samuel Boutin
4    *  Janvier 2012
5    */
6
7   #include "colonie.h"
8   #include "graphe.h"
9
10  #include <map>
11  #include <string>
12  #include <set>
13  #include <list>
14  #include <iostream>
15  #include <cmath>
16  #include <unistd.h>
17  #include <cstdlib>
18
19  #include <boost/thread.hpp>
20
21
```

```
22 #define nbThread 8
23
24 using namespace std;
25
26 colonie::colonie(const graphe& _G){
27    // 1) Conserve un pointeur vers le graphe afin d'avoir accès à la matrice des
       poids via la méthode valeur_arc.
28    G = &_G;
29
30    // 2) Initialisation des paramètres :
31    // (Il serait également possible de construire un autre constructeur acceptant
       directement la structure en paramètre).
32    parametres.nb_fourmis = 20;  // Nombre de fourmis par noeud
33    parametres.minPh = 1;        // Quantité minimale de phéromone sur chaque arc
34    parametres.maxPh = 200;      // Quantité maximale de phéromone sur chaque arc
35    parametres.rho   = 0.5;      // Taux d'évaporation de la phéromone
36    parametres.coeffPh = 100;    // Lors de la distribution de la phéromone, une
       quantité coeffPh/lopngueur_arc sera distribuée sur chaque arc
37    parametres.alpha = 1;        // Exposant de l'inverse de la longueur de l'arc
       dans le calcul des probabilités
38    parametres.beta =1;          // Exposant de la phéromone dans le calcul des
       probabilités.
39
40
41    // 3) Initialisation du vecteur de fourmis :
42    int totalFourmi = parametres.nb_fourmis*G->size();
43    lesFourmis.reserve(totalFourmi);
44
45
46    // 4) Initialisation de chaque fourmi :
47    // Afin d'initialiser les fourmis, récupère l'ensemble des noeuds du graphe.
48    set<string> tousLesNoeuds;
49    set<string>::iterator it;
50    for (it=G->begin();it!=G->end();++it){
51       tousLesNoeuds.insert(*it);
52    }
53
54    // Boucle sur les noeuds :
55    for(it=G->begin();it!=G->end();++it){
56       // Boucle sur le nombre de fourmis
57       for(int j=0;j<parametres.nb_fourmis;++j){
58          lesFourmis.push_back(fourmi(*it,tousLesNoeuds));
59       }
60    }
61
62
63    // 5) Initialisation de la matrice creuse de phéromone :
64    set<string>::iterator it1;
65    set<string>::iterator it2;
66    // Boucle sur l'ensemble des noeuds
67    for (it1=G->begin();it1!=G->end();++it1){
68       // Boucle sur l'ensemble des noeuds
69       for (it2=G->begin();it2!=G->end();++it2){
70          if (G->valeur_arc(*it1,*it2) != -1){
71             pheromone[*it1][*it2] = parametres.minPh;
72          }
```

```
73          }
74      }
75
76
77      // 6) Initialisation de la fréquence d'évaporation de la phéromone :
78      map<string, map<string,double> >::iterator itmap1;
79      map<string, double>::iterator itmap2;
80      itmap1 = pheromone.begin();
81      itmap2 = (itmap1->second).begin();
82      parametres.delai = G->size()*G->valeur_arc(itmap1->first,itmap2->first)*5.;
83
84
85      // 7) Initialisation de l'agenda :
86      // Un événement pour chaque fourmi
87      for (int j=0;j<totalFourmi;++j){
88          agenda.insert(pair<double,int>(0.,j) );
89      }
90      // Événement pour l'évaporation de la phéromone :
91      agenda.insert(pair<double,int>(parametres.delai,-1));
92  }
93
94  list<string> colonie::trouverCircuit(double tempsEvol){
95          // Création des fils travailleurs.
96          vector<boost::thread*> lesTravailleurs(nbThread);
97          for (int i=0;i<nbThread;++i)
98          {
99                  lesTravailleurs[i] = new boost::thread(&colonie::trouver,this,
                        tempsEvol);
100         }
101
102
103         // Attend la fin de chaque fil.
104         for (int i=0;i<nbThread;++i)
105         {
106                 lesTravailleurs[i]->join();
107                 delete lesTravailleurs[i];
108         }
109
110         return meilleurCircuit;
111 }
112
113
114 void colonie::trouver(double tempsEvol)
115 {
116
117         boost::thread::id id =  boost::this_thread::get_id();
118         ecran.lock();
119                 cout << "Thread " << id << endl;
120         ecran.unlock();
121
122   // Trouve le prochain événement dans l'agenda :
123    multimap<double,int>::iterator i;
124   /*
125    modifAgenda.lock();
126             i=agenda.begin();
127             // Insère une condition d'arrêt :
```

```
128             agenda.insert(pair<double,int>(i->first+tempsEvol,-2));
129     modifAgenda.unlock();
130 */
131 {
132         boost::unique_lock<boost::mutex> lock(modifAgenda);
133             i=agenda.begin();
134             // Insère une condition d'arrêt :
135             agenda.insert(pair<double,int>(i->first+tempsEvol,-2));
136 }
137     double t;
138     string noeud1,noeud2;
139     map<string,double> prob;
140     map<string,double> arcs;
141     pair<double,int> ev; // Événement à traiter.
142 
143     while(true){
144         modifAgenda.lock();
145                 i=agenda.begin();
146                 ev = *i;
147                 agenda.erase(i);
148         modifAgenda.unlock();
149         if (ev.second ==-2){
150             // Fin du temps de recherche
151             return;
152         }
153         if (ev.second == -1){
154             // Évaporation de la phéromone
155             evapPh();
156             ev.first+=parametres.delai;
157         }
158         else{
159             // Fait avancer une fourmi :
160             noeud1 = lesFourmis[ev.second].noeudActuel();
161             arcs = poidsDesArcsVoisins(noeud1);
162             prob = calculProba(noeud1,arcs);
163             // La colonie fournie à la fourmi la probabilité et la longueur des arcs
                    voisins.
164             noeud2 = lesFourmis[ev.second].avancer(prob,arcs);
165 
166             if (noeud2=="circuit"){
167                 // Vérifie si le circuit trouver est le meilleur :
168                 modifMeilleur.lock();
169                 if((lesFourmis[ev.second].poidsDuCircuit() < meilleurPoids) ||
                        meilleurCircuit.empty()){
170                     meilleurPoids = lesFourmis[ev.second].poidsDuCircuit();
171                     meilleurCircuit = lesFourmis[ev.second].circuitComplet();
172                 }
173                 modifMeilleur.unlock();
174 
175                 ajouterPh(lesFourmis[ev.second].circuitComplet(),lesFourmis[ev.second].
                        poidsDuCircuit());
176             //     cout << "Poids du circuit touvé : " << lesFourmis[ev.second].
                    poidsDuCircuit() << "   au temps t = " << ev.first << endl;
177                 lesFourmis[ev.second].retourDepart();
178             }
179             else if (noeud2 == "erreur"){
```

```
180             exit(-1);
181         }
182         else if (noeud2 !="bloque"){
183                 ev.first += G->valeur_arc(noeud1,noeud2);
184         }
185         // Remet la fourmi à l'agenda.
186         // Dans le cas où la fourmi est dans un cul-de-sac, elle retourne au dé
                part par elle-même.
187         // Elle est remis à l'agenda au même temps.
188         boost::unique_lock<boost::mutex> l(modifAgenda);
189         agenda.insert(ev);
190       }
191     }
192
193 }
194
195
196 // Demande au graphe le poids des arcs voisins d'un noeud donné.
197 map<string,double> colonie::poidsDesArcsVoisins(string noeudActuel){
198     // Utilise la matrice de phéromone afin de connaître les noeuds voisins.
199     accesPhero.lock_shared();
200     map<string,double> poids = pheromone[noeudActuel];
201     accesPhero.unlock_shared();
202     map<string,double>::iterator it;
203     for (it=poids.begin();it!=poids.end();++it){
204         poids[it->first]=G->valeur_arc(noeudActuel,it->first);
205     }
206
207     return poids;
208 }
209
210 map<string,double> colonie::calculProba(string noeud,const map<string,double>&
        voisins){
211     map<string,double> probabilites;
212     map<string,double>::iterator it;
213     probabilites = voisins;
214     double visibilite;
215
216     accesPhero.lock_shared();
217     for(it=pheromone[noeud].begin();it!=pheromone[noeud].end();++it){
218         visibilite =1/probabilites[it->first];
219         probabilites[it->first] =pow(visibilite,parametres.alpha)*pow(it->second,
              parametres.beta);
220     }
221     accesPhero.unlock_shared();
222
223     return probabilites;
224 }
225
226 void colonie::evapPh(){
227     map<string,map<string,double> >::iterator it1;
228     map<string,double>::iterator it2;
229     {
230             boost::unique_lock<boost::shared_mutex> lock(accesPhero);
231
232             for (it1=pheromone.begin();it1!=pheromone.end();++it1){
```

```
233                     for (it2=pheromone[it1->first].begin();it2!=pheromone[it1->first].
                            end();++it2){
234                         it2->second *=parametres.rho;
235                         if (it2->second < parametres.minPh){
236                                 it2->second = parametres.minPh;
237                         }
238                     }
239             }
240     }
241 }
242
243 // Ajout de la phéromone. Total : poids total du circuit.
244 void colonie::ajouterPh(const list<string>& chemin,double total){
245     // Utilisation de deux itérateurs décalés d'une position.
246     list<string>::const_iterator it1;
247     list<string>::const_iterator it2;
248     // Plus le circuit est court, plus il y a de phéromone.
249     double ratio = parametres.coeffPh/total;
250     it2 =chemin.begin();
251     ++it2;
252     boost::unique_lock<boost::shared_mutex> lock(accesPhero);
253
254     for(it1=chemin.begin();it2!=chemin.end();++it1){
255         pheromone[*it1][*it2] +=ratio ;
256         if (pheromone[*it1][*it2] > parametres.maxPh)
257         {
258                 pheromone[*it1][*it2] = parametres.maxPh;
259         }
260         ++it2;
261     }
262 }
```