SCIENCES SORBONNE UNIVERSITÉ

Examen Réparti 1 PSCR
Master 1 Informatique Nov 2024

UE 4I400

Année 2024-2025

2 heures – **Une page A4 recto verso manuscrite comme seul document papier autorisé**
Tout appareil de communication électronique interdit (téléphones...).

Introduction

- Le barème est sur 20 points et est donné à titre indicatif.
- Dans le code C++ demandé vous prendrez le temps d'assurer que la compilation fonctionne.
- On vous fournit une archive contenant un projet eclipse CDT, qu'il faudra modifier. Décompresser cette archive dans votre home, de façon à avoir un dossier `~/exam/`
- Pour importer ce projet dans Eclipse, le plus facile :"File->Import->General->Existing Projects into Workspace",
- Si vous préférez utiliser un autre IDE ou la ligne de commande, on vous fournit un Makefile trivial. Attention en termes de configuration si vous refaites vous même, il faut les flags "-pthread -lexpat" au link et "-std=c++20" à la compilation.
- Le makefile actuel produit deux cibles, "bin/mc" compilé avec optimisations et "debug/mc" compilé en -O0 -g. Pensez à utiliser la version optimisée pour vos mesures.
- A la fin de la séance, fermez simplement votre session. On ira par script récupérer les fichiers dans ce fameux dossier `~/exam/`. Assurez vous donc de bien suivre ces instructions.

Le sujet est composé de parties relativement indépendantes qu'on pourra traiter dans l'ordre qu'on souhaite. Pour la majorité des questions il s'agit de fournir un code compilable correct. On demande dans vos réponses de se limiter seulement aux *primitives* pour la concurrence offertes en C++ : thread, mutex, condition_variable et atomic.

L'application et les structures de données sont quasiment entièrement codé dans des header; c'est délibéré pour faciliter la substitution d'une implantation par une autre. On recommande donc quand on a traité une question de changer les include dans le fichier principal pour utiliser la nouvelle version de la classe.

1 Model checking

On considère le problème de l'exploration concurrente de l'espace d'états d'un système. Le système est décrit par son état initial et par une relation de transition, qui associe à un état l'ensemble de ses successeurs. Les états sont donc organisés dans un (grand) graphe.

System.hh

```
1 #ifndef SYSTEM_H
2 #define SYSTEM_H
3
```

```
4  #include "State.hh"
5  #include <vector>
6  #include <string>
7
8  class System
9  {
10 public:
11   // Returns the initial state
12   virtual State initial_state () const = 0;
13
14   // Generates successors of a given state
15   virtual std::vector<State> successors (const State &s) const = 0;
16
17 public:
18   System () {};
19   virtual ~System () {};
20 };
21
22 #endif // SYSTEM_H
```

On ne vous demande pas de développer l'implantation précise du System : on vous fournit une implantation qui permet d'explorer l'espace d'états d'un réseau de Petri (Petri net).

Pour obtenir une instance de `System` on invoquera `System *system = PetriNet::buildNet(path);` où `path` désigne un fichier PNML (format standard pour représenter un Petri net). Ensuite on se contente d'invoquer les opérations de System.

Le programme principal doit explorer l'ensemble des états accessibles du système (supposé fini), en partant de l'état initial. Pour que le processus converge, il est nécessaire de distinguer :

- `seen` : les états déjà visités dans le parcours. `StateSet` permet d'ajouter `bool add(const State& s)` un état, et rend vrai si cet état était nouveau.
- `todo` une file des états nouvellement rencontrés et donc qu'il faut explorer. `StateQueue` porte deux opérations `void push(const State& s)` qui ajoute un état dans la file, et `bool pop(State& s)` qui rend vrai, extrait un élément de la file, et l'affecte dans l'argument s si la file n'est pas vide. Dans le cas contraire, la fonction rend false et ne modifie pas l'argument.

Le boucle principale est donc :

```
1   // Build the system using the builder API
2   System *system = PetriNet::buildNet (argv[1]);
3
4   State initial = system->initial_state ();
5
6   // Data structures
7   StateQueue todo;
8   StateSet seen;
9
10  // Initialize
11  todo.push (initial);
12  seen.add (initial);
13
14  State s;
15
16  while (todo.pop (s)) {
17    std::vector<State> successors = system->successors (s);
18
19    for (const State &succ : successors) {
20      if (seen.add (succ)) {
21        todo.push (succ);
```

```
    }
  }
}
std::cout << "Total states explored: " << seen.size << std::endl;
```

L'objectif est de paralléliser ce code, en particulier en introduisant des versions de `seen` et `todo` utilisables en multi-thread.

Vous pouvez le tester en lançant sur un petit exemple comme "bin/mc example/Airplane.pnml" qui possède 43463 (40 k) états. Les autres modèles exemple ont respectivement 243 états pour Philosophers, 2546432 (2 M) pour Kanban, 4051732 (4 M) pour AutonomousCar.

Les graphes produits sont grands d'où l'intérêt d'une exploration concurrente.

1.1 StateSet (6 points)

Question 1. (2 points) Dans une copie `StateSetMT.hh` de `StateSet.hh` écrire une version thread safe de la classe. On gardera le nommage intact, de façon à pouvoir utiliser cette nouvelle version en changeant simplement `#include "StateSet.hh"` par `#include "StateSetMT.hh"` dans le main. On demande dans cette question de n'utiliser qu'un seul verrou.

Question 2. (3 points) Dans une autre copie `ConcurrentStateSet.hh` écrire une version thread safe de la classe qui utilise un verrouillage fin, à l'aide d'un verrou par *bucket* dans la table. De nouveau on gardera le nom de la classe intact. On recommande d'introduire un `vector<mutex>` plutôt que de modifier le typage de `buckets`. On prendra garde à ne pas introduire de data race autour de la manipulation de *size*, tout en limitant la contention (i.e. on préfère ne pas utiliser un lock supplémentaire).

Question 3. (1 point) **Répondez dans le fichier "questions.txt" à la racine du projet.** Expliquez pourquoi la version avec verrouillage fin peut être meilleure que l'approche avec un seul verrou.

1.2 Boucle principale (4 points)

Question 4. Mettez à jour le main :

- Il doit lire un entier N dans son deuxième argument
- Il instancie N threads ; chacun d'eux doit réaliser la boucle actuelle : pop un état, calcul des successeurs, ajout/test dans `seen`, si nouveau ajout dans `todo`. Les threads se terminent si `pop` rend false.
- Il attend la fin des threads démarrés avec join
- Il affiche le nombre d'états calculé et le temps.

NB: Le programme est incomplet à ce stade, il manque une implantation de `StateQueue` thread-safe/bloquante ainsi qu'une façon correcte de détecter la terminaison; mais répondez à la question comme si ces problèmes étaient déjà résolus.

1.3 State Queue (8 points)

Question 5. (2 point) Dans une copie `StateQueueMT.hh` de `StateQueue.hh` écrivez une version thread safe de la classe, munie d'une sémantique bloquante sur pop si vide.

Question 6. (1 point) **Répondez dans le fichier "questions.txt" à la racine du projet.** Dans cette version de la classe `StateSet`, peut on se contenter de `notify_one` au lieu de `notify_all` ? Justifiez votre réponse.

Question 7. (2 point) Dans une copie `StateQueueTerm.hh` de `StateQueueMT.hh` ajoutez un flag booléen "isFinished" initialement faux, et une fonction `setFinished()` permettant d'indiquer que le traitement est terminé et donc basculer le flag à vrai. On a donc une version thread safe de la classe,

munie d'une sémantique bloquante sur pop si vide et que "isFinished" est faux, mais si "isFinished" est vrai, pop n'est pas bloquant et rend false. On fera attention à notifier aux bons endroits pour ne pas laisser des threads bloqués si la situation a changé.

Question 8. (3 points) Dans une nouvelle copie `ConcurrentStateQueue.hh` du fichier, ajoutez le comportement suivant:

- On ajoute un entier "active" en attribut de la classe; on lui passe la valeur à la construction. Il représente le nombre de threads actifs, initialement le main passera donc le N correspondant au nombre de threads qu'il crée.
- On ajoute une nouvelle `condition_variable` utilisée pour attendre la fin de toute activité.
- On ajoute une méthode "void await()" qui sera invoquée par le main et permet de se bloquer sur la nouvelle condition jusqu'à ce que "active" soit zéro.
- On met à jour "active" de la manière suivante dans `pop`:
  - Juste avant de dormir sur wait(), parce que la file est vide, on décrémente active. S'il atteint zéro, on notifie le(s) thread potentiellement endormi dans `await()`.
  - Dès le réveil en sortant de wait() on réincrémente active

1.4 Assemblage final

Question 9. (1 point) Mettez à jour le main concurrent pour utiliser ces mécanismes : après avoir lancé les threads, le main `await` que le processus soit fini, débloque la file avec `setFinished` et enfin attend les threads avec `join`.

Question 10. (1 point) **Répondez dans le fichier "questions.txt" à la racine du projet.** Mesurer le temps de calcul et exhiber une trace pour 1, 4, 8, 16 threads pour l'exemple "Kanban_5.pnml" qui possède 2.5 millions d'états.