Runtime: Moteur d'exécution

Airbus Group Innovations

18 août 2014

Résumé

Ce document donne des détails sur les principes de base et les algorithmes implémentés dans RUNTIME, un moteur d'exécution utilisé par la bibliothèque `hmat`.

Table des matières

Acronymes

FIFO *First In, First Out*

1 Introduction

RUNTIME est un moteur d'exécution basé sur le modèle de graphe de tâches énuméré, similaire à celui utilisé par Quark et StarPU. Il présente les caractéristiques suivantes :

- Mémoire partagée ;
- MPI ;
- Support limité de l'OOC (*Out-Of-Core*) ;
- Différents ordonnanceurs, gestion des priorités (similaires à l'ordonnanceur `prio` de StarPU).

L'objectif de ce document est de documenter le fonctionnement de cet ordonnanceur, afin de faciliter ses évolutions et sa maintenance. Après une présentation du modèle utilisé (ETF), nous donnerons son implémentation simple en mémoire partagée, puis documenterons les extensions nécessaires pour la gestion de la mémoire distribuée, et le support limité de l'OOC.

2 Graphe de tâches enuméré

Tâches et données Un algorithme peut être décomposé en un ensemble fini ordonné de tâches $T := \{t_i \mid i = 1, \ldots, N\}$, opérant sur un ensemble de données $D = \{d_j \mid j = 1, \ldots, M\}$. À chaque tâche $t \in T$ est associé deux ensembles de données respectivement lues et écrites par cette tâche, $In(t) \subset D$ et $Out(t) \subset D$.

Exemple 1 (Produit Matrice-Matrice)**.** *Considérons l'opération* GEMM(C, α, A, B, β) *définie par :*

$$C \leftarrow \alpha AB + \beta C$$

Ici, $D = \{A, B, C\}$*,* $t =$ GEMM*,* $In(t) = \{A, B, C\}$ *et* $Out(t) = \{C\}$*. Par ailleurs, si* $\beta = 0$*, alors* $In(t) = \{A, B\}$*.*

Cohérence séquentielle Le modèle de cohérence utilisé ici est celui de la cohérence séquentielle. Ceci signifie que le résultat final de toute exécution valide de l'algorithme est le même que celui d'une exécution dans laquelle les tâches sont exécutées séquentiellement dans l'ordre d'énumération.

La cohérence séquentielle peut être respectée en déterminant les tâches dépendantes, et en ajoutant une contrainte de précédence entre ces opérations, dont le sens est donné par l'ordre d'énumération des tâches. Ces contraintes sont déterminées à partir des données lues et écrites par les tâches.

Plus précisément, soit $d \in D$,

- Toute **lecture** de d nécessite que les **écritures** précédentes soient terminées ;
- Toute **écriture** de d nécessite que les **lectures** et **écritures** précédentes soient terminées.

Une tâche t est une lecture de d si et seulement $d \in In(t)$ (idem pour les écritures). Notons $t_i \to t_j$ la relation "t_i doit précéder t_j". Alors,

$$
\begin{aligned}
t_i \to t_j \iff\ & \exists d \in D,\ (d \in Out(t_j)) \wedge (d \in In(d_i)) \wedge (i < j) \\
& \text{ou} \\
& \exists d \in D,\ (d \in In(t_j) \cup Out(t_j)) \wedge (d \in Out(d_i)) \wedge (i < j)
\end{aligned} \tag{1}
$$

La première relation traduit la condition "une lecture doit attendre les écritures précédentes", la seconde la condition "une écriture doit attendre les lectures et écritures précédentes".

Quelques remarques :

— Une tâche ne peut pas dépendre d'une tâche future (condition $i < j$) ;

— Si $i \to j$ et $j \to k$, alors $i \to k$ (transitivité).

On définit alors le graphe de tâches par :

Définition 2 (Graphe de tâches)**.** Soit T et D des ensembles de tâches et données définis tels que précédemment. On définit un graphe de tâche $G = (V, E)$ dont l'ensemble des sommets V est T, et les arêtes E sont $\{(t_i, t_j) \in T \times T \mid t_i \to t_j\}$. C'est un **graphe acyclique dirigé** (DAG).

La transitivité de la relation $\cdot \to \cdot$ permet de simplifier l'expression du graphe (réduire son nombre d'arêtes), en introduisant :

— Le dernier écrivain $W(d, t_i) := \{\max\{t_j \mid t_j < t_i \wedge d \in Out(t_j)\}\}$

— L'ensemble des derniers lecteurs $R(d, t_i) := \{t_j \| \ d \in In(t_j) \wedge W(d, i) < t_j < t_i\}$

Le dernier écrivain de d à l'instant de l'énumération de t_i est la dernière tâche ayant écrit dans d avant la tâche t_i, et les derniers lecteurs sont les lecteurs depuis la dernière écriture.

On définit alors le graphe de tâche réduit par :

Définition 3 (Graphe de tâches réduit)**.** Soit T et D des ensembles de tâches et données définis tels que précédemment. On définit un graphe de tâche réduit $G = (V, E)$ dont l'ensemble des sommets V est T. L'ensemble des arêtes E de G est construit ainsi :

$$
\begin{aligned}
&\forall t_i \in T, \\
&\qquad \forall d \in In(t), E \leftarrow E \cup W(d, t_i) \text{ et} \\
&\qquad \forall d \in Out(t), E \leftarrow E \cup W(d, t_i) \cup R(d, t_i)
\end{aligned}
$$

Ce graphe réduit encode les mêmes contraintes de précédence que le graphe initial, nous ne considérons que celui-ci dans la suite.

3 Graphe de tâches et moteur d'exécution

Exécution parallèle d'un graphe de tâches L'exécution parallèle des tâches du graphe de la figure 1 est claire : il est possible d'exécuter les tâches 2 et 3 en parallèle, mais il est également possible d'exécuter la tâche 4 en parallèle de la tâche 3, suivant les mêmes règles que celles définissant un parcours valide séquentiel. Une tâche t_i est

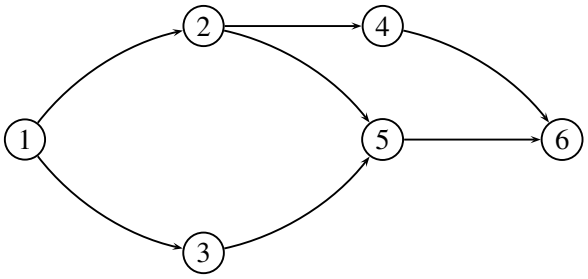

FIGURE 1 – Exemple de graphe de tâches

considérée éligible pour exécution si toutes ses dépendances $\{t_{i'} \mid (i', i) \in A\}$ ont été exécutées. Dans ce cas, l'exécution parallèle du graphe de tâches respecte la contrainte de cohérence séquentielle, et le résultat est le même quel que soit le nombre de processeurs. Une propriété supplémentaire intéressante est même obtenue : les résultats sont dits exactement reproductibles. Cette garantie n'est cependant plus valable dans le cas de l'utilisation d'extensions de ce formalisme, comme par exemple les réductions (qui ne sont pas supportées par `runtime`), les opérations de la norme IEEE-754 n'étant pas associative.

3.1 Moteur d'exécution

Le formalisme de graphe de tâches exposé dans la section précédente a de nombreux avantages. Il est expressif, simple, et la formalisation des contraintes permet une grande liberté d'exécution parallèle. En effet, certaines méthodes de parallélisation sont centrées autour de l'expression des régions parallèles, et de l'identification des tâches pouvant être effectuées de façon concurrente. Le formalisme présenté ici est inversé, puisqu'il exprime les contraintes, et évite dans certains cas d'imposer des barrières artificielles liées à la programmation manuelle d'une exécution parallèle.

Cependant, le graphe de tâches ne permet pas de déterminer une exécution parallèle. Cette tâche est déléguée à un système opérant « en ligne » lors de l'exécution du programme. Il est chargé d'explorer le graphe de dépendances dans un ordre valide en parallèle, de manière à maximiser l'utilisation des ressources.

La spécification des tâches et de leurs dépendances se fait au travers d'une fonction `insert_task()` dont l'utilisation est de la forme (pour l'opération GEMM de l'exemples 1) :

```
1 insert_task(gemm, C, IN | OUT, A, IN, B, IN, ...);
```

Cet appel est **non bloquant**, et ajoute une tâche instanciée par l'exécution de la fonction `gemm` dans le graphe de tâches. Cette fonction accède à la variable `C` en lecture et écriture, et aux variables `A` et `B` en lecture. La tâche est un sommet du graphe de tâches, et les dépendances entre les tâches sont déduites des dépendances sur les données `A`, `B` et `C`. Aucun code n'est exécuté à cette étape.

Exemple 4 (Produit Matrice-Matrice en tuiles). *Supposons trois matrices A, B, et C, décomposées en un tableau de $n \times n$ tuiles,* `A[i][j]` *représentant la tuile (i, j). Alors une écriture de la multiplication $C \leftarrow AB + C$ peut se donner dans ce formalisme par :*

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      insert_task(gemm, C[i][j], IN | OUT, A[i][k], IN, B[k][j], IN);
    }
  }
}
go();
```

L'insertion des tâches est non bloquante et construit uniquement le graphe de tâches, l'exécution est faite dans le corps de la fonction `go()`, qui est bloquante.

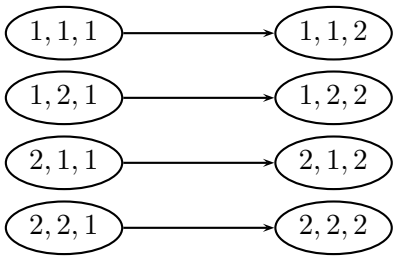

FIGURE 2 – Graphe de tâches induit pour un produit matriciel.

Dans le cas où $n = 2$ dans l'algorithme précédent, le graphe de tâches est donné par la figure 2, sur laquelle la tache (i, j, k) représente l'opération (séquentielle) $C_{ij} \leftarrow C_{ij} + A_{ik}B_{kj}$. L'ordre d'exploration du graphe n'est pas spécifié, et 4 parties connexes donc totalement indépendantes du point de vue de l'exécution apparaissent.

Remarque 5 (Accumulation). *Dans l'exemple précédent, les tâches dans la boucle la plus interne sont commutatives*[1]*. L'expression des dépendances formulée dans l'exemple précédent ne permet pas d'exposer cette caractéristique. Il est possible dans certains moteurs d'exécution (y compris QUARK et StarPU) de donner cette indication, sans laquelle les tâches de la boucle la plus interne ont des dépendances trop fortes (ordre strict au lieu d'exclusion mutuelle).*

3.2 Principes de base

Le moteur d'exécution comprend quatres parties principales :

1. Création du graphe de tâches à partir des appels à `insert_task()` ;
2. Détermination des tâches éligibles à l'exécution ;

1. En arithmétique exacte

3. Ordonnancement des tâches éligibles ;
4. Gestion des *workers*.

3.2.1 Construction du graphe de tâches

La première étape (construction du graphe se décompose ici de la manière suivante) :
— Construction des ensembles de dernier écrivain et derniers lecteurs à l'insertion des tâches ;
— Ajout des arêtes et des sommets au graphe de tâche.

Une fois le graphe de tâche déterminé, les structures de données nécessaires aux étapes suivantes sont construites. Plus précisément, les structures de données sont similaires à celles nécessaires pour effectuer un **tri topologique** d'un graphe acyclique dirigé. Pour chaque sommet du graphe, on construit :
— Le degré entrant de l'arête (pour le sommet t_i, nombre d'arêtes de la forme $t_j \to t_i$ dans le graphe) ;
— La liste des successeurs de t_i, *i.e.* l'ensemble $\{t_k \mid t_i \to t_k\}$.

3.2.2 Détermination des tâches éligibles à l'exécution

Les tâches éligibles sont les sommets du graphe dont le degré entrant est égal à 0. On note que ceci ne permet de construire que l'ensemble des tâches éligibles au début de l'exécution, il sera nécessaire de remettre à jour les structures de données au cours de l'exécution pour déterminer les autres tâches éligibles. Par ailleurs, cet ensemble est non vide, puisque la première tâche insérée n'a pas de dépendance.

3.2.3 Ordonnancement des tâches éligibles

Les tâches éligibles sont ajoutées à un ensemble de tâches dont l'exécution est possible, en attente de gestion par un des *worker*. L'ordre d'exécution effectif de ces tâches est laissé libre, puisqu'il ne peut pas avoir d'influence sur le résultat de l'algorithme, du fait de l'expression des dépendances supposée exhaustive dans le graphe de tâches. Le modèle le plus simple est celui d'une simple file d'attente *First In, First Out* (FIFO), mais d'autres politiques d'ordonnancement sont possibles. Néanmoins, du point de vue du reste du moteur d'exécution, l'ordonnanceur fournit deux opérations :
— `push(task)` : ajoute une tâche à l'ensemble des tâches disponibles ;
— `try_pop()` : tente de récupérer une tâche, l'enlevant ainsi de l'ensemble des tâches en attente.

La première opération est appelée par le mécanisme de résolution des dépendances, la seconde par un des *workers*.

Deux exemples d'ordonnanceurs sont illustrés par la figure 3, un ordonnanceur étant une simple FIFO, le second prenant en compte des priorités relatives entre les tâches.

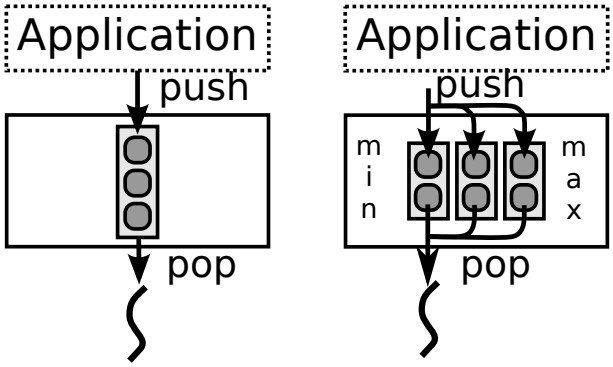

FIGURE 3 – Ordonnancement sans (`eager`) et avec priorités (`prio`).

3.2.4 Gestion des workers

Le cœur du moteur d'exécution instancie un nombre configurable de *workers*, typiquement un par cœur sur la machine. Chaque *worker* est un *thread*, créé et terminé dans `go()`, exécutant la boucle de l'algorithme 1 :

Algorithme 1 Boucle d'exécution de tâches par un processeur.

```
while true do
    ok ← try_pop(t)
    if ¬ok then
        exponential_backoff()
        continue
    end if
    if t == SENTINEL then
        break
    end if
    EXECUTE(t)
    UPDATEDEPENDENCIES(t)
end while
```

Dans cet algorithme, la fonction UPDATEDEPENDENCIES est la fonction responsable de la mise à jour des structures de dépendances. Plus précisément, elle retire la tâche t du graphe de tâches, et ajoute les tâches nouvellement éligibles à la file d'attente.

4 Runtime

Le moteur d'exécution est écrit en C++11, et n'a de dépendance que sur un compilateur C++11, la STL associée et MPI pour la mémoire distribuée. Il repose essentiellement sur les classes suivantes :

— Classes "externes"

- `TaskScheduler` : Singleton exposant les fonctions membre `insertTask()` et `go()` ;
- `Task`

— Classes “internes” :

- `Scheduler` : politique d'ordonnancement, exposant `push()` et `tryPop()`
- `Worker`

De plus, la gestion de MPI et de l'OOC a nécessité l'ajout d'une classe `Data` représentant une donnée, qui n'est pas présente dans les premières version, fonctionnant uniquement en mémoire partagée.

4.1 Insertion des tâches

4.1.1 Tâches

Les tâches sont dans RUNTIME des classes dérivées de la classe `Task`, une classe abstraite de base, dont les éléments publics sont :

```
class Task {
public:
  Priority priority;
  Task(std::string name="Task");
  virtual void call() = 0;
};
```

Autrement dit, une tâche est une classe ayant une méthode `call()` permettant d'exécuter la tâche. Les paramètres de cette tâche doivent être passés autrement, par exemple dans le constructeur de la classe dérivée.

Une instance de `Task` est allouée par le code appelant, en revanche la destruction est assurée par le moteur d'exécution.

Par exemple, une tâche `GemmTask` peut être implémentée ainsi :

```
class GemmTask : public Task {
private:
  double alpha, beta;
  double *c, *a, *b;

public:
  GemmTask(double* c, double alpha, double* a, double* b, double beta)
    : alpha(alpha), beta(beta), c(c), a(a), b(b) {}
  void call() {
    cblas_dgemm(...);
  }
};
```

4.1.2 InsertTask

Utilisation Le prototype de la fonction membre `insertTask()` est :

```
enum AcessMode {READ, WRITE};
void insertTask(Task* task, const vector<pair<Data*, AccessMode>>& params);
```

et peut être utilisée (en C++11) ainsi :

```
TaskScheduler s;
s.insertTask(new GemmTask(c, alpha, a, b, beta),
             {{a, READ}, {b, READ}, {c, WRITE}});
```

en supposant que le type `Data` est un alias de `void`.

Description Le suivi des accès aux données est fait au travers de la class `AccessTracker` et du membre `dataAccess` :

```
struct AccessTracker {
  Task* lastWrite;
  deque<Task*> lastReads;
};
unordered_map<Data*, AccessTracker> dataAccess;
```

La classe `AccessTracker` suit le dernier accès en écriture et les derniers accès en écriture à une donnée, et `dataAccess` fait le lien entre une donnée et les accès.

Pour chaque dépendance `Data* d` d'une tâche `Task* t`, selon le type d'accès :

READ Si `dataAccess[d].lastWrite != NULL`, alors ajouter une dépendance `dataAccess[d].lastWrite` $\rightarrow$ `t`, et ajouter `t` à `dataAccess[d].lastReads` ;

WRITE Ajouter des dépendances :
- `dataAccess[d].lastWrite` $\rightarrow$ `t`
- `t2` $\rightarrow$ `t` pour tout `t2` dans `dataAccess[d].lastReads`

Puis vider `dataAccess[d].lastReads`, et faire `dataAccess[d].lastwrite` $\leftarrow$ `t`

Toutes les dépendances de toutes les tâches sont collectées dans le membre `vector<pair<Data*, Data*>> deps` qui est l'ensemble des arêtes du graphe de tâches.

4.2 Exécution

Une fois toutes les tâches insérées, `TaskScheduler::go(nThreads)` est appelé. Cette fonction doit :
- Préparer les structures de données et les tâches initialement disponibles ;
- Lancer les *workers*
- Attendre la fin de l'exécution.

4.2.1 Préparation

L'étape précédente permet de construire le graphe de tâches. Il est ensuite nécessaire de préparer les structures de données importantes pour le suivi des dépendances, ce qui

est fait dans `TaskScheduler::prepare()`. Les informations à suivre sont :

— Degré entrant des sommets ;

— Liste des successeurs.

Ces informations sont dans `map<Task*, TaskSuccessor> succ`, avec

```
struct TaskSuccessors {
  int count; // degre entrant
  deque<Task*> successors;
};
```

La construction de cette structure est simple :

— Pour tout $t_i \to t_j \in$ `deps`,

— `succ[`t_j`].count += 1`

— `succ[`t_i`].successors.push_back(`t_j`)`

Une fois cette structure construite, le graphe de tâche est inutile. Toutes les tâches ayant pour degré entrant 0 sont poussées dans la liste des tâches disponibles.

4.2.2 Workers

Un `worker` est une instance de cette classe :

```
class Worker {
private:
  Scheduler& q;
  TaskScheduler& scheduler;

public:
  Worker(Scheduler& q, TaskScheduler& scheduler);
  void mainLoop();
};
```

`Scheduler` est ici l'ordonnanceur (par exemple, FIFO ou avec gestion des priorités) qui est un conteneur intérieurement synchronisé (les fonctions membres `push` et `tryPop` sont plus ou moins équivalentes à des méthodes `synchronized` en Java). La fonction membre `mainLoop()` implémente l'algorithme 1, et la valeur sentinelle est `NULL` ; c'est celle qui est lancée par le moteur d'exécution à la création du THREAD.

Les lignes importantes de cette fonction sont :

```
task->call();
scheduler.postTaskExecution(task);
```

La première est l'appel à la tâche, qui n'a besoin d'aucune synchronisation. La seconde est la mise à jour des dépendances et la libération de `t`.

4.2.3 Mise à jour des dépendances

Lorsqu'une tâche à été exécutée, elle est retirée du graphe de tâches. Ceci correspond à décrémenter le degré entrant de tous ses successeurs, ce qui est fait dans `TaskScheduler::postTaskExecution()`. Puisque cette fonction peut être appelée par n'importe quel *worker*, elle est protégée par un *mutex*.

Plus précisément :

— Pour toutes les tâches t_2 dans `succ[`t`].successors`
  — `succ[`t_2`].count--` ;
  — Si `succ[`t_2`].count == 0`, on l'ajoute dans la liste des tâches prêtes.

Le reste de la fonction s'occupe de nettoyage des structures, et de vérifier qu'il reste encore des tâches à exécuter. Dans le cas contraire, une tâche "sentinelle" (`NULL` ici) est poussée par *worker*, de manière à signaler la terminaison. C'est également cette fonction qui détruit l'instance de la tâche.

4.2.4 Verrouillage

Il y a peu de verrous dans le moteur d'exécution. En effet, l'exécution des tâches ne nécessite aucun verrouillage, sous réserve que les dépendances de données soient correctement spécifiées. Les élements nécessitant une protection sont :

— L'ordonnanceur. L'opération `push()` n'est appelée que par un *thread* à la fois puisque son accès est fait depuis `postTaskExecution()`. En revanche, l'opération `tryPop()` est appelée par tous les *workers*, et doit être protégée.
— Les structures contenant les dépendances entre les tâches, modifiées par `postTaskExecution()`.

Ordonnanceur Extérieurement, l'ordonnanceur est une file d'attente de type *Single Producer, Multiple Consumers* (SPMC). Elle est néanmoins implémentée ici comme une file d'attente synchronisée intérieurement, et supportant un modèle *Multiple Producers, Multiple Consumers* (MPMC). De plus, il n'y a qu'un seul verrou pour les écrivains et les lecteurs, ce qui n'est pas optimal, mais plus simple.

Gestion des dépendances Tous les appels à `postTaskExecution()` sont protégés par un *mutex*, ce qui donne l'équivalent d'une méthode `synchronized` en Java.