Présentation d'Athapascan-1-b

Mathias Doreille, François Galilée et Jean-Louis Roch

14 janvier 1997

Cette documentation présente l'interface de programmation d'*Athapascan-1* selon le plan :

1. Introduction : présentation générale de la bibliothèque.
2. Principe des tâches asynchrones : expression du parallélisme en *Athapascan-1*.
3. Mémoire locale et mémoire partagée : deux niveaux de mémoire.
4. Précédences entre tâches ayant des paramètres en mémoire partagée.
5. Containers.
6. Exemples de programmes. Expression de programmes complexes.
   (a) Découpe récursive : variations sur Fibonacci,
   (b) produit scalaire,
   (c) produit de matrices par blocs,
   (d) factorisation LU.

Préambule : cette documentation est plus axée sur la présentation d'*Athapascan-1* que sur sa justification. Néanmoins, le paragraphe 1.2 et les différents paragraphes intitulés "*Justification*" essaient de donner sommairement les arguments qui ont motivé certains choix de conception. Ils ne constituent pas à proprement parler une justification (qui nécessitera une présentation transversale à celle-ci) mais sont là pour faciliter la compréhension du modèle de programmation.

1 Introduction

Le but de ce paragraphe est de donner une idée générale de l'interface en montrant la programmation d'un exemple simple : le calcul d'un des termes de la suite de Fibonacci.

1.1 Un exemple de programme *Athapascan-1*

Athapascan-1 est une bibliothèque C++ qui permet d'exprimer le parallélisme par appel de procédure asynchrone à distance. Un tel appel est réalisé par la création d'une *tâche*. Une tâche décrit une procédure (prenant en argument des paramètres formels) ainsi que l'ensemble des paramètres effectifs à passer à cette procédure lors de son appel. L'exécution d'une tâche correspond alors à l'appel de la procédure sur les paramètres effectifs ; elle est réalisée indépendamment du processus qui a créé la tâche (appel asynchrone).

Les paramètres effectifs et formels d'une tâche peuvent être des objets en *mémoire locale* (mode de passage par copie) ou des objets en *mémoire partagée* (mode de passage par référence).

Différents *modes d'accès* sont définis pour un objet en mémoire partagée : *lecture*, *écriture*, *lecture/écriture* et *accumulation*. Les accès à la mémoire partagée, qui expriment les *dépendances de données*, définissent les contraintes à la fois de précédence et de localité entre tâches.

Remarque. *Athapascan-1* peut donc être vu comme une adaptation dans un cadre asynchrone du modèle théorique PRAM qui est synchrone : les synchronisations sur horloge globale de la PRAM sont remplacées par des synchronisations pour les accès aux données en mémoire partagée. Dans une PRAM, à chaque variable globale est associé un mode d'accès : EREW (Exclusive Read, Exclusive Write), CREW (Concurrent Read, Exclusive Write), ERCW (Exclusive Read, Concurrent Write) ou encore CRCW (Concurrent Read, Concurrent Write). En *Athapascan-1*, ce sont les accès aux données qui définissent les synchronisations.

Exemple 1. L'exemple suivant qui sera détaillé par la suite montre un programme parallèle simple : le calcul brutal d'un des termes de la suite de Fibonacci. Les mots-clefs de *Athapascan-1* sont préfixés par `a1_`.

Commentaires. Le programme principal lance de manière asynchrone une tâche de calcul – qui récursivement crée d'autres tâches de calcul jusqu'à un seuil d'arrêt – et une tâche d'affichage du résultat.
Il n'y a aucune contrainte de précédence entre les tâches de calcul (definie en ligne 7) qui demandent toutes des accès concurrents en écriture par accumulation (indiqués par le suffixe `_cw` sur `a1_shared` ligne 8) sur la variable res. Par contre la tâche d'affichage (definie en ligne 1) demande sur cette variable un accès en lecture (indiqués par le suffixe `_r` sur `a1_shared` ligne 2). Comme la date de création de la tâche d'affichage (crée en ligne 25) est postérieure à celle de la première tâche de calcul (crée en ligne 24), elle s'exécutera lorsque toutes les tâches d'écriture qui la précèdent seront achevées.

```
1  struct print_int : public a1_task {
2    void operator()(a1_shared_r<int> a) {
3      cout << "Resultat = " << *a << endl;  // affichage de la valeur de a
4    }
5  };
6
7  struct fibo : public a1_task {
8    void operator()(int n, a1_shared_cw<int> res) {
9      if( n<2 ) {
10       res += 1;
11     }
12     else {
13       a1_new_task(fibo(),n-1,res);
14       a1_new_task(fibo(),n-2,res);
15     }
16   }
17 };
18
19 a1_main() {
20   int n ;
21   a1_shared_rp_wp<int> res(0);
22   cout << "Fibo: n= " << endl;
23   cin >> n;  // lecture d'un entier au clavier
24   a1_new_task(fibo(),n,res);
25   a1_new_task(print_int(),res);
26 }
```

1.2 Les idées directrices dans la conception

Le principe de base d'*Athapascan-1* est qu'*il n'y a de synchronisation (et donc de communication) que lors du lancement de l'exécution d'une tâche*. Ce principe permet au système de :

- gérer complètement l'ordonnancement [1],
- d'avoir accès à la description complète des communications et donc des contraintes de localité d'une tâche[2].

[1]Notamment, des algorithmes d'ordonnancement en-ligne prouvés asymptotiquement optimaux comme les algorithmes de liste par exemple peuvent être mis en œuvre. On retrouve cette approche dans Cilk.

[2]Ce principe permet de découpler la description des communications – lors de la création d'une tâche, elles correspondent aux paramètres de la procédure – de la synchronisation – préalable à l'exécution et où se déroulent les communications effectives –. Ce principe se retrouve dans BSP : les requêtes de communication sont non bloquantes mais ne sont effectuées qu'après une étape de synchronisation (globale ici) exprimée dans le programme.

Autrement dit, en *Athapascan-1*, toutes les contraintes de synchronisation apparaissent dans les créations de tâche à travers les paramètres effectifs en mémoire partagée. Une fois démarrée sur un processeur, une tâche peut poursuivre son exécution sans aucune synchronisation jusqu'à l'exécution de sa dernière instruction.

Les caractéristiques principales d'*Athapascan-1* sont les suivantes :

- L'exécution séquentielle d'un programme *Athapascan-1*, obtenue en remplaçant tous les appels asynchrones de procédures par des appels synchrones, est une exécution valide. Cette exécution définit alors un ordre total sur les tâches d'un programme *Athapascan-1* : l'*ordre séquentiel total de création*.
- Il y a deux niveaux de mémoire pour représenter les objets. A chaque tâche est associée une *mémoire locale* qui contient les objets qu'elle seule accède (en lecture ou écriture). Cette mémoire correspond à la mémoire locale du processeur sur lequel s'exécute la tâche. Elle est directement accessible en C++ (pile de la tâche et tas). La *mémoire partagée* contient les objets qui peuvent être accédés par différentes tâches. Les modes d'accès d'une tâche à un objet de la mémoire partagée (lecture et/ou écriture) sont complètement décrits dans le profil de la procédure associée à la tâche.
- Les tâches correspondent à des appels asynchrones de procédure à distance. Les relations de précédence sont définies par les *dépendances de données* sur les objets en mémoire partagée : l'*ordre séquentiel total de création des tâches* définit les précédences entre les tâches qui ne peuvent pas être effectuées en concurrence pour cause de conflit d'accès sur une donnée partagée commune.
- Granularité par appel synchrone local : puisque l'ordonnancement séquentiel, obtenu en remplaçant toutes les créations de tâches par des appels de procédure synchrones, est un ordonnancement valide, l'ordonnanceur peut adapter le grain de parallélisme : lorsqu'une tâche est exécutable, il est possible de l'exécuter séquentiellement en interprétant toutes les créations de tâches qu'elle effectue par des appels synchrones et locaux de procédure.
- Localité par type d'attributs : lorsqu'une tâche désire accéder à un objet en mémoire partagée, cet accès apparaît dans le profil formel de la tâche. Il est donc possible de connaître les contraintes de localité liées à l'exécution d'une tâche.

2 Type de tâches et appel asynchrone

Le but de cette section est de présenter dans un cadre élémentaire la déclaration de type de tâches et l'exécution de tâches en *Athapascan-1*.

2.1 Type de tâches

Une tâche *Athapascan-1* est une fonction-classe qui dérive de la classe `a1_task` et qui possède au moins l'opérateur "`()`". La définition est la suivante :

```
struct mon_type_de_tâche : public a1_task {
  // Champs obligatoires
  void operator () ( <Profil> )  {
    // Corps de la tâche : suite d'instructions
  }
};
```

Une information sur le type de tâche peut être fournie par l'utilisateur pour distinguer les tâches qui ne créent aucune autre tâche. Pour cela, la fonction-classe définissant la tâche doit dériver de la classe `a1_task_elem`.

D'autres membres peuvent être définis : des membres optionnels (ayant une implémentation par défaut dans `a1_task`) ou des membres définis directement par l'utilisateur.

Requis. Le type de chacun des paramètres formels apparaissant dans le profil doit être transmissible (cf §3). Dans cette partie, nous nous limiterons à l'utilisation des types de base `long`, `double` et `bool` qui sont tous transmissibles.

Exemple 2. Déclaration d'un type de tâche qui permet l'affichage d'un réel en mémoire partagée sur la sortie standard.

```
struct print_double : public a1_task_elem {
  void operator() (a1_shared_r<double> v)  {
     cout << "Le resultat est : " << *v << endl;
  }
};
```

Remarque. La tâche `print_double` ne créera pas d'autres tâches, elle peut aussi dériver de la classe `a1_task_elem`.

2.2 Création de tâche

La procédure `a1_new_task` permet de créer une tâche avec des paramètres effectifs instanciés.

```
a1_new_task ( a1_task mon_type_de_tâche(), < Paramètres Effectifs > );
```

La *création d'une tâche* définit un *appel asynchrone de procédure à distance* dont les paramètres peuvent être des objets en mémoire locale ou des objets en mémoire partagée. La date de début d'exécution et le site d'exécution sont choisis par l'ordonnanceur *Athapascan-1* associé à la tâche créatrice.

Une tâche est dite *achevée* lorsqu'elle a atteint sa dernière instruction. L'exécution étant asynchrone, aucun contrôle ne peut être fait par l'utilisateur sur cette date de fin. Une fois la tâche démarrée, elle exécute séquentiellement les instructions de son corps : aucune synchronisation ne peut apparaître dans cette séquence d'instructions. Seule la date de début d'exécution d'une tâche peut, selon la nature des paramètres, être soumise à des contraintes de synchronisation.

Cependant, lorsque les paramètres d'appels sont des objets en mémoire locale – cadre de ce paragraphe –, il n'y a aucune contrainte sur cette date d'exécution : l'exécution de la tâche peut donc commencer dès sa création.

Exemple 3. Création d'une tâche `print_double` avec en entrée le réel `6.3`. L'exécution de la tâche écrit ce réel sur la sortie standard. La tâche est exécutée de manière asynchrone sans contrainte de synchronisation.

```
a1_main() {
  double k = 6.3;
  a1_new_task ( print_double() , k );
}
```

Justification 1.

1. *L'appel asynchrone [2.2] permet la construction dynamique du graphe de flot de données. Les appels peuvent être ordonnancés selon différents algorithmes prouvés compétitifs (par exemple à la demande –* receiver initiated *– ou statique) sans nécessiter un mécanisme de migration d'une tâche en cours d'exécution.*

2.3 Barrière de synchronisation globale

Dans le processus séquentiel initial (exécution de la fonction `main()`) et uniquement dans ce processus initial, la création d'une tâche par l'opération `a1_new_task` est un appel *synchrone* de la procédure associée à cette tâche. La synchronisation est effectuée sur la terminaison de toutes les tâches générées à partir de cette tâche.

Le programme est alors *terminé* lorsque le processus séquentiel initial a effectué sa dernière instruction.

Justification 2.

1. *Cette synchronisation permet d'écrire des procédures séquentielles dont le corps est implanté en parallèle (et qui donc lance des tâches de manière asynchrone) et qui se terminent lorsque tous les calculs parallèles sont achevés. Ainsi, il est possible de faire appel à de telles procédures au sein d'un programme séquentiel.*

2. *Par suite, il est possible de décrire en* Athapascan-1 *un programme parallèle comme une succession séquentielle de pas, chaque pas étant exécuté en parallèle.*

2.4 Contrôle de l'ordonnancement d'une tâche

Chaque tâche est ordonnancée par un objet de type `a1_sched` qui implémente une politique d'ordonnancement qui lui est propre[3].

La fonction externe : `a1_sched get_self_sched()` permet d'accéder à l'objet qui ordonnance la tâche en cours d'exécution.

Deux paramètres optionnels peuvent être fournis lors de la création d'une tâche (appel à `a1_new_task`) : un ordonnanceur et une priorité.

```
a1_new_task( a1_task f, <Paramètres Effectifs>);
a1_new_task_s( a1_sched s, a1_task f, <Paramètres Effectifs>);
a1_new_task_p( a1_prior p, a1_task f, <Paramètres Effectifs>);
a1_new_task_sp( a1_sched s, a1_prior p, a1_task f, <Paramètres Effectifs>);
```

- `s` est un objet de type `a1_sched` qui doit prendre en charge l'ordonnancement de la tâche. Par défaut, cet objet est celui retourné par `a1_self_sched()`.
- `p` est un entier qui donne à l'ordonnanceur une information de priorité sur la tâche[4]. Le type `a1_priority` possède un constructeur prenant en entrée un `int`. Informellement, pour un ensemble de tâches concurrentes et de même coût, la tâche de plus grande priorité est prioritaire.
  La fonction externe `a1_priority a1_self_priority()` permet d'obtenir la priorité de la tâche en cours d'exécution. Par défaut, la valeur de priorité donnée dans `a1_new_task` est `a1_self_priority()`.

2.5 Information de coût

Il est possible d'associer à un type de tâche des fonctions membres optionnelles pour indiquer différentes informations sur la tâche :

1. `a1_cost cost ( <paramètres de taille>);`
2. `a1_cost total_cost ( <paramètres de taille>);`
3. `a1_task_count task_generate( <paramètres de taille>);`

Chacune de ces fonctions prend un nombre variable d'arguments tous de type `a1_size_t`. Le $k^{\text{ème}}$ argument correspond à la taille du $k^{\text{ème}}$ paramètre de la tâche, i.e. de `operator()`.
La fonction `cost_info` retourne le coût d'exécution de la tâche, `total_cost` retourne le coût d'exécution de la tâche augmenté du coût total d'exécution de l'ensemble des tâches qu'elle génère (donc le coût d'exécution de la tâche par un ordonnancement séquentiel), et `task_generate` retourne le nombre de tâches créées par la tâche.

Des défauts sont fournis pour ces trois fonctions ; les fonctions défauts retournent la valeur
`<type_de_retour>::unknown()`.

3 Mémoire locale et mémoire partagée

La mémoire d'*Athapascan-1* est structurée en 2 niveaux :

- La *mémoire locale* associée à chaque tâche, qui contient les objets qu'elle seule manipule.
- La *mémoire partagée*, qui contient les objets pouvant être accédés par différentes tâches.

[3] Le respect des contraintes de synchronisation exprimées dans le programme *Athapascan-1* est garanti par toute politique d'ordonnancement.

[4] Cette information est informelle : elle ne peut en aucun cas modifier les précédences qui sont garanties par tout ordonnanceur. Elle permet à l'ordonnanceur de faire un choix local entre plusieurs tâches n'ayant pas de contraintes de synchronisation.

3.1 Type transmissible

Tout objet placé en mémoire partagée ou passé en paramètre d'une tâche doit pouvoir être communiqué. Sur un support distribué, le type de l'objet doit donc posséder un opérateur de recopie à distance : il est alors *transmissible*.

On appelle *type transmissible* T tout type muni des fonctions :

a1_ostream& operator<< (a1_ostream& out, const T&) pour envoyer un objet de type T dans le flot out.

a1_istream& operator>> (a1_istream& in, T&) pour recevoir un objet de type T du flot in.

T::(const T&) pour copier un objet de type T.

Toutes les communications d'objets se font au sens de ces deux opérateurs.

3.1.1 Sémantique

La sémantique de << et >> est définie à partir de la sémantique du constructeur par recopie de C++.
Soit T un type transmissible. Considérons le programme suivant :

```
T a = ...;     // Déclaration d'un objet dans la pile
T x(a);        // x est construit comme une copie (dans la pile) de a

a1_iostream f ;
T y;
f << a;
f >> y;        // y est construit à partir de a par utilisation des
               // opérateurs << et >>
```

Alors le test booléen (x == y) retourne la valeur true.

3.1.2 Types de base transmissibles

Tous les types fondamentaux de C++ sont transmissibles (les opérateurs << et >> sont fournis par la bibliothèque *Athapascan-1*). De plus, les types définis par *Athapascan-1* sont aussi transmissibles à l'exception du type générique a1_shared<mode_accès>.

3.1.3 Containers transmissibles

Tout container de la STL (Standard Template Library) instancié sur un type transmissible est transmissible.
exemple : vector<int>, list<vector<int> > ...

3.1.4 Taille d'une donnée et coût de communication

Tout type transmissible possède une fonction externe qui retourne la taille de l'objet. Cette taille, exprimée en octets, permet de connaître le coût de la communication de l'objet (recopie à distance). Cette taille est aussi utilisée pour connaître le coût d'une tâche, la taille du $k^{ème}$ paramètre étant passée en $k^{ème}$ paramètre de la fonction membre a1_cost cost(<taille des données>) de la tâche.

```
a1_size_t a1_size(const T&);
```

Une fonction par défaut existe : a1_size_t a1_size(const T& t) { return sizeof(T); }

Justification 3.

3.1 L'opérateur de copie à distance est décrit de manière bidirectionnelle (<< et >>) ce qui permet d'optimiser la granularité des communications. Dans le cas d'objets structurés (couples, enregistrements), même si l'emballage et le déballage de plusieurs objets sont effectués, une seule communication physique peut être mise en œuvre.

3.1.2 Les opérateurs de copie des types de base, tous de taille constante, sont fournis par le système pour éviter une encapsulation inutile. Ils sont implémentés par fonctions externes.

3.1.3 Les containers permettent de rendre facilement communicable une séquence d'objets transmissibles.

3.2 Objets en mémoire partagée et modes d'accès

Athapascan-1 permet de définir des *objets en mémoire partagée*. Ces objets permettent au système *Athapascan-1* de déterminer les contraintes de précédence (dépendance par rapport à un objet en mémoire partagée) et de localité entre les tâches.

Quatre principaux modes d'accès existent pour un objet en mémoire partagée :

- lecture : `a1_shared_r`,
- écriture : `a1_shared_w`,
- lecture/écriture : `a1_shared_r_w`,
- accumulation : `a1_shared_cw` (cumulative write). L'écriture est alors faite de manière distribuée par accumulation de la valeur à écrire sur la valeur déjà existante de l'objet.

Des restrictions d'accès peuvent être apportées à ces quatre modes pour préciser que l'accès effectif sur la donnée ne sera pas directement effectué par la tàche mais par une (ou plusieurs) tâche(s) de sa descendance. On rajoute pour cela le suffix `p` (pour *postponed*) : `a1_shared_rp`, `a1_shared_wp`, `a1_shared_rp_w`, `a1_shared_r_wp`, `a1_shared_rp_wp` et `a1_shared_cwp`.
Ces informations permettent d'optimiser la gestion des copies de données ainsi que de relâcher plus encore les contraintes de précédences sur les tâches.

Pour tous les modes décrits ci-dessus, les accès aux objets se font par les opérateurs membres C++ suivants :

- pour la lecture :
  - `const T& operator*()`
- pour l'écriture :
  - `void operator=(const T& t)` pour un objet alloué dans la pile (implique une copie de l'objet par *Athapascan-1*).
  - `void operator=(T* t)` pour un objet alloué dans le tas (la destruction de l'objet pointé par `t` sera géré par *Athapascan-1*, l'utilisateur ne doit donc pas effectuer de `delete` sur cet objet).
- pour l'accumulation :
  - `operator+=(const T&).`

Des vérifications à la compilation sont effectuées pour assurer la validité de l'utilisation de ces opérateurs en fonction du mode et du type d'accès de la donnée partagée.

La déclaration d'un objet en mémoire partagée de type `T` se fait alors de l'une des manières suivantes :

```
a1_shared_[mode d'accès]<T> [nom de l'objet];
a1_shared_[mode d'accès]<T> [nom de l'objet]([valeur d'initialisation]);
```

Exemple 4. Création de deux objets de type `int` en mémoire partagée, de mode lecture differée `a1_shared_rp` et lecture/ecriture `a1_shared_r_w`. Initialisation du deuxième objet par la valeur 3. Accès au deuxième objet par l'operateur *.

```
a1_shared_rp_<int> a;
a1_shared_r_w<int> b(3);
int c = *b;
*b = 3 + *b * c;
```

4 Dépendance de données entre tâches

4.1 Changement explicite du mode d'accès

Le mode d'accès d'une variable partagée reste inchangé tout au long de l'exécution d'une tâche. Il est cependant possible, en cours d'exécution d'une tâche, de restreindre les accès d'une donnée partagée aux seules tâches filles par appel, sur la donnée partagée, d'une des fonction suivante.

- `a1_shared_r` : conversion vers `a1_shared_rp` par appel de la fonction membre `change_to_rp()` ;
- `a1_shared_w` : conversion vers `a1_shared_wp` par appel de la fonction membre `change_to_wp()` ;
- `a1_shared_cw` : conversion vers `a1_shared_cwp` par appel de la fonction membre `change_to_cwp()` ;
- `a1_shared_r_w` : conversion vers
  - `a1_shared_rp_w` par appel de la fonction membre `change_to_rp_w()`,
  - `a1_shared_r_wp` par appel de la fonction membre `change_to_r_wp()`,
  - `a1_shared_rp_wp` par appel de la fonction membre `change_to_rp_wp()` ;
- `a1_shared_rp_w` : conversion vers `a1_shared_rp_wp` par appel de la fonction membre `change_to_rp_wp()` ;
- `a1_shared_r_wp` : conversion vers `a1_shared_rp_wp` par appel de la fonction membre `change_to_rp_wp()`.

Justification 4. *La limitation des conversions au seul passage d'un accès effectifs vers son accès différés provient de :*

- *l'interdiction de synchronisation pendant l'exécution de la tâche,*

4.2 Passage de paramètre à une tâche

Lors de la création de tâche (`a1_new_task`), les paramètres formels (en mémoire locale ou partagée) d'un type de tâche doivent être instanciés avec des objets (en mémoire locale ou partagée) ayant des modes de partage compatibles.

Plus précisément, soit `T` un type transmissible.

La liste suivante décrit l'ensemble des conversions autorisées lors de la création d'une tâche : on trouve à gauche de la flèche ($\longrightarrow$) le mode d'accès de la donnée pour la tâche créatrice et à droite celui de la tâche créée.

- `a1_shared_r` ou `a1_shared_rp` $\longrightarrow$ `a1_shared_r` ou `a1_shared_rp`;
- `a1_shared_w` $\longrightarrow$ `a1_shared_w` ou `a1_shared_wp`;
- `a1_shared_wp` $\longrightarrow$ `a1_shared_w`, `a1_shared_wp`, `a1_shared_cw` ou `a1_shared_cwp`;
- `a1_shared_cw` ou `a1_shared_wp` $\longrightarrow$ `a1_shared_cw` ou `a1_shared_cwp`;
- `a1_shared_r_w` ou `a1_shared_r_wp` $\longrightarrow$ `a1_shared_r` ou `a1_shared_rp`;
- `a1_shared_rp_w` ou `a1_shared_rp_wp` $\longrightarrow$ tous les modes d'accès.

De plus, dans le processus séquentiel initial (exécution de la fonction `main()`) les modes d'accès `a1_shared_r_w` et `a1_shared_r_wp` peuvent être convertis vers tous les autres modes d'accès.

4.3 Précédence entre tâches

Les relations de précédence et de localité entre les tâches sont définies par les objets en mémoire partagée qu'elles accèdent.

Les conflits d'accès entre tâches sur un même objet en mémoire partagée sont résolus (par des synchronisations entre tâche) de manière à ce que les tâches accédant en lecture à cet objet lisent la valeur qu'elles auraient lue lors d'une exécution séquentielle.

Une tâches est donc exécutable lorsque toutes les données partagées qu'elle accède en lecture ne sont plus accédées en écriture (simples, par accumulation et éventuellement différées) par les taches qui la précèdent selon l'ordre d'exécution séquentiel (ordre total).

Exemple 5. Cet exemple montre un cas de précédence par conversion du mode d'accès de donnée en mémoire partagée lors de la création de tâche. Ici, l'appel à `init_double` convertit `a` qui est accédé en référence : conversion de `a1_shared_rp_wp` vers `a1_shared_w<double>`; l'appel à `print_double` convertit `a` de `a1_shared_rp_wp` vers `a1_shared_r`.

La tâche `print_double` accède à l'objet partagé `a`, dans le mode `a1_shared_r`. Deux autres tâches créées avant celle-ci accèdent à cette donnée : `a1_main` et `init_double`. `init_double` accède cette donnée dans le mode `a1_shared_w`. Ainsi la tâche `print_double` sera *exécutable* après la terminaison de la tâche `init_double` (sinon la valeur lue ne serait pas celle lue lors d'une exécution séquentielle du programme).

```
struct print_double : public a1_task_elem {
  void operator() (a1_shared_r<double> v)  {
    cout <<  "Le resultat est : " << *v << endl;  // affichage de la valeur de v
  }
};

struct init_double : public a1_task {
  void operator()(a1_shared_w<double> v) {
    v = sqrt(2.0) / 2 ;
  }
};

struct a1_main : public a1_task {
  void operator() (...)
    a1_shared_rp_wp<double> a;   // creation d'une variable de type int en mémoire partagée
    a1_new_task( init_double(), a);  // création d'une tâche prenant a en parametre
    a1_new_task( print_double(), a); // création d'une tâche prenant a en parametre
  }
};
```

5 Container en mémoire partagée

La mise en mémoire partagée d'un container permet de regrouper dans un même objet (le container) un ensemble de données de même type. On distingue deux types de containers, selon la localisation de leurs éléments qui sont soit tous en mémoire locale soit tous en mémoire partagée. Une utilisation directe est de permettre de décrire des types de tâches dont le nombre d'arguments en mémoire partagée est variable, ce nombre étant fixé lors de la construction de la tâche.

Ce paragraphe précise la définition de ces deux types containers en *Athapascan-1*.

5.1 Container non distribué en mémoire partagée

Pour les objets en mémoire partagée vus jusqu'à présent, l'utilisateur fournit, dans la déclaration du type, le type de l'objet et le mode d'accès sur cet objet. La déclaration d'un container "partagé" de type `Container`, avec le mode de partage `Mode_Partage` et dont les éléments, de type transmissible `T`, ne sont pas "partagés" s'écrit de la même manière :

```
a1_shared_[Mode_Partage]<Container< T > > objet;
```

Voici un exemple de déclaration et d'utilisation de ce type de container :

```
a1_shared_r_w<vector< int > > a;
int e = (*a)[3]; //accés au 3éme élément du container
```

5.2 Collection d'objets en mémoire partagée

Un container dont les éléments sont également en mémoire partagée est appelé *collection*. Pour la définition du type *collection* l'utilisateur doit indiquer à la fois le mode d'accès du container et le mode d'accès de ses éléments. En reprenant les mêmes notations, la déclaration d'un container "partagé" de type `Container`, avec le mode de partage `Mode_Partage_Container` et dont les éléments, de type transmissible `T`, sont "partagés" avec le mode `Mode_Partage_Element` s'écrit :

```
a1_shared_[Mode_Partage_Container]< Container<
        a1_shared_[Mode_Partage_Element]< T > > > objet;
```

Voici quelques exemples de déclaration et d'utilisation de ce type de container :

```
a1_collection_r_w<vector<a1_shared_rp_wp<int > > > a;
 a1_collection_r_w<vector<a1_shared_r_w<int > > > b;
a1_collection_r<vector<a1_shared_cw<int > > > c;
int e = *(*a)[3];  //accés au 3éme élément du container
```

Exemple 6. L'exemple didactique suivant montre sur le calcul de la norme d'un vecteur de `complexe` différentes créations de tâches imbriquées et en séquence. Il y a précédence entre l'initialisation d'une des composantes du vecteur et le calcul de son module qui est sommé dans la variable en écriture concurrente `norme_v`. Les calculs concernant deux composantes différentes peuvent être effectués en parallèle.
La tâche d'affichage de la norme est ordonnancée lorsque toutes les autres tâches sont terminées. Il est à noter que les composantes du vecteur ne sont pas communiquées lors de l'exécution de la tâche `norme2`.

```
struct carre  : public a1_task {
  // accumulation du carre d'un reel
  void operator()(double x, a1_shared_cw<double> res ) {
    res += x * x ;
  }
}

struct module : public a1_task {
  // accumulation du module d'un complexe
  void operator()(a1_shared_r<complex> x, a1_shared_cw<double> res) {
     a1_new_task ( carre(), (*x).imag, res ) ;
     a1_new_task ( carre(), (*x).real, res ) ;
  }
}

struct norme2 : public a1_task {
  // accumulation du carre de la norme euclidienne d'un vecteur de complexes
  void operator()(a1_collection_r<vector<a1_shared_rp<complex> > > x,
                  a1_shared_cw<double> res {
    for (int i=0 ; i < (*x).size(); i++ )
      a1_new_task( module(), (*x)[i], res ) ;
  }
}


a1_main() {
  // création d'une variable en mémoire partagée initialisée à 0
  a1_shared_rp_wp<double > norme_v( 0.0 );
  int n ;

  cout << "Taille du vecteur : ";
  cin  >> n ;

  // création d'un container (tableau) partagé distribué v de n éléments de type complex
  a1_collection_rp_wp<vector<a1_shared_rp_wp<complex > > > v(n);

  // initialisation du tableau partagé distribué v
  for (int i=0 ; i < n ; i++ )
    a1_new_task( init_double(), (*v)[i] );

  // création de la tâche calculant la norme sur le tableau partagé distribué v
  a1_new_task( norme2(), v, norme_v );

  // création de la tâche affichant le résultat
  a1_new_task( print_double(), norme_v );
}
```

6 Exemples de programme

6.1 Variations sur Fibonacci

Ces versions sont données à titre d'exemple pour montrer un calcul récursif simple :

- la première version donne le programme séquentiel de départ.
- la deuxième version est une parallélisation de type découpe récursive avec synchronisation : division, sous-calculs parallèles et fusion.
- la troisième version est une parallélisation sans synchronisation : elle utilise un accumulateur en mémoire partagée puisque la fusion dans la version précédente est associative et commutative.

6.1.1 Version récursive séquentielle

```
struct fibo {
  void operator() (int n, int& res) {
    if( ( n<2 ) {
      res = 1 ;
    }
    else {
      int res1, res2 ;
      fibo() ( n-1, res1 ) ;
      fibo() ( n-2, res2 ) ;
      res = res1 + res2 ;
    }
  }
};

main () {
  int n ;
  cout << "Fibo: n= " ;
  cin >> n ;

  int res ;
  fibo()( n, res ) ;
  cout << "Resultat = " << res << endl ;
}
```

6.1.2 Version récursive parallèle avec synchronisation

```
struct print_int : public a1_task_elem {
  void operator() (a1_shared_r<int> a) {
    cout <<   "Resultat = " << a << endl ;
  }
};

struct somme : public a1_task_elem {
  void operator() (a1_shared_r<int> a, a1_shared_r<int> b,  a1_shared_w< int> res) {
    res = a + b ;
  }
};

struct fibo : public a1_task {
  void operator() (int n, a1_shared_w< int> res) {
    if( n<2 ) {
      res = 1 ;
    }
    else {
      a1_shared_rp_wp<int> res1, res2 ;
      a1_new_task ( fibo(), n-1, res1 ) ;
      a1_new_task ( fibo(), n-2, res2 ) ;
      // mutation (conversion) de la variable res vers le mode sans accès ``shared''
      // pour pouvoir la passer à une autre tâche.
      a1_new_task ( somme(), res1, res2, res ) ;
    }
  }
};

a1_main () {
  int n ;
  cout << "Fibo: n= " ;
  cin >> n ;

  a1_shared_rp_wp<int> res ;
  a1_new_task ( fibo(), n, res ) ;
  a1_new_task ( print_int(), res ) ;
}
```

6.1.3 Version récursive parallèle sans synchronisation

```
struct print_int : public a1_task_elem {
  void operator() (a1_shared_r<int> a) {
    cout << "Resultat = " << *a << endl ;
  }
};

struct fibo : public a1_task {
  void operator() (int n, a1_shared_cw<int> res) {
    if( n<2 ) {
      res += 1 ;   // accumulation locale du résultat  dans l'accumulateur res
    }
    else {
      a1_new_task ( fibo(), n-1, res ) ;
      a1_new_task ( fibo(), n-2, res ) ;
    }
  }
};

a1_main () {
  int n ;
  a1_shared_rp_wp<int> res ;
  cout << "Fibo: n= " ;
  cin >> n ;
  a1_new_task( fibo(), n, res ) ;
  a1_new_task ( print_int(), res ) ;
}
```

6.2 Produit scalaire de vecteurs

Voici un exemple de produit scalaire de deux vecteurs avec partitionement des vecteurs par blocs.

6.2.1 Version séquentielle

```
double scalar_product(vector<double>& A, vector<double>& B)
{
  double tmp = 0;
  for(int i=0; i<A.dim() ;i++)
    tmp += A[i]*B[i];
  return tmp;
}

a1_main()
{
  vector<double> IN_A, IN_B;
  cin >> IN_A >> IN_B;
  cout << scalar_product(A,B);
}
```

6.2.2 Version parallèle

```
struct task_scalar_product : public a1_task_elem {
  void operator()(a1_shared_r<vector<double> > a, a1_shared_r<vector<double> > b,
                  a1_shared_cw<double> c ) {
    c += scalar_product(*a, *b);
  }
};

struct par_scalar_product : public a1_task {
  void operator()( a1_collection_r<vector<a1_shared_rp<vector<double> > > > A,
                   a1_collection_r<vector<a1_shared_rp<vector<double> > > > B,
                   a1_shared_cwp<double> C) {
     for(int i=0; i<(*A).dim() ;i++)
        a1_new_task(task_scalar_product<double>(), (*A)[i], (*B)[i], C);
  }
};

a1_main()
{
  vector<double> IN_A, IN_B;
  cin >> IN_A >> IN_B;
  a1_collection_rp_wp<vector<a1_shared_rp_wp<vector<double> > > > A =
          IN_A.partitioning(sqrt(IN_A.dim()));
  a1_collection_rp_wp<vector<a1_shared_rp_wp<vector<double> > > > B =
          IN_B.partitioning(sqrt(IN_A.dim()));
  a1_shared_rp_wp<double> C(0);
  a1_new_task(par_scalar_product<double>(),A,B,C);
  a1_new_task(print<double>(), C);
}
```

6.3 Tri de tableaux

6.4 Produit de matrices par blocs

Voici un exemple de multiplication de matrices avec partitionnement des matrices par blocs bidimensionel.

6.4.1 Version séquentielle

```
void matrix_times(const matrix<double>& A, const matrix<double>& B, matrix<double>& C)
{
  for(int i=0; i<A.row_dim() ;i++)
    for(int j=0; j<B.col_dim(); j++)
      for(int k=0; k<A.col_dim(); k++)
         C(i,j) += A(i,k) * B(k,j);
}

main()
{
  matrix<double> A, B, C;
  cin >> A >> B;
  matrix_times(A,B,C);
  cout << C;
}
```

6.4.2 Version séquentielle par blocs bidimensionnels

```
void bloc_matrix_times(const matrix<matrix<double> >& A,
                       const matrix<matrix<double> >& B,
                       matrix<matrix<double> >& C)
{
  for(int i=0; i<A.row_dim() ;i++) {
    for(int j=0; j<B.col_dim(); j++) {
      for(int k=0; k<A.col_dim(); k++)
        C(i,j) += A(i,k) * B(k,j);
    }
  }
}

main()
{
  matrix<double> IN_A, IN_B, C;
  cin >> A >> B;
  // construction des 3  matrices (de dimension sqrt(n)) de blocs (de taille sqrt(n))
  // à partir des matrices IN_A et IN_B.
  matrice<matrice<double> > A = IN_A.two_dim_partitioning(sqrt(IN_A.coldim()));
  matrice<matrice<double> > B = IN_B.two_dim_partitioning(sqrt(IN_A.coldim()));
  matrice<matrice<double> > C (sqrt(IN_A.coldim()));
  bloc_matrix_times(A,B,C);
  cout << C;
}
```

6.4.3 Version parallèle

```
//-----------------------------------------------------------------------
//encapsulation des expressions et indication des coûts de calcul

struct times_equal : public a1_task_elem {
  void operator()(a1_shared_r<matrix<double> > a, a1_shared_r<matrix<double> > b,
                  a1_shared_cw<matrix<double> > c) {
    matrix<double> tmp;
    matrix_times(*b, *c, tmp);
    c += tmp;  // accumulation (locale) du résultat dans l'accumulateur c.
  }
  cost_task cost(a1_size_t a, a1_size_t b, a1_size_t c) { return a*a*a; }
};

//-----------------------------------------------------------------------
//algorithme de multiplication de matrice

struct par_bloc_matrix_times : public a1_task {
  void operator()(a1_collection_r<matrix<a1_shared_rp<matrix<double> > > > A,
                  a1_collection_r<matrix<a1_shared_rp<matrix<double> > > > B,
                  a1_collection_r<matrix<a1_shared_cwp<matrix<double> > > > C)
  {
    for(int i=0; i<(*A).row_dim();i++) {
      for(int j=0; j<(*B).col_dim(); j++)
        for(int k=0; k<(*A).col_dim(); k++)
           a1_new_task(times_equal<double>(),(*A)(i,k), (*B)(k,j), (*C)(i,j));
       a1_flush();
    }
  }
};

a1_main()
{
  matrix<double> IN_A, IN_B;
  cin >> IN_A >> IN_B;
  a1_collection_rp_wp<matrix<a1_shared_rp_wp<matrix<double> > > > A =
          IN_A.two_dim_partitioning(sqrt(IN_A.coldim()));
  a1_collection_rp_wp<matrix<a1_shared_rp_wp<matrix<double> > > > B =
          IN_B.two_dim_partitioning(sqrt(IN_A.coldim()));
  a1_collection_rp_wp<matrix<a1_shared_rp_wp<matrix<double> > > > C;
  a1_new_task( par_bloc_matrix_times<double>(), A,B,C);
  a1_new_task(print<double>(), C);
}
```

6.5 Factorisation LU

Voici un exemple de factorisation LU avec partitionnement de la matrice par blocs bidimensionels.

6.5.1 Version séquentielle

```
void factorization_LU(matrix<double>& A)
{
  for (int k=0 ; k < A.row_dim() ; k++ ) {
    double pivot = 1/A(k,k);
    for (int i=k+1 ; i < A.row_dim() ; i++ ) {
      A(i,k) *= pivot;
      for (int j=k+1 ; j < A.col_dim() ; j++ )
        A(i,j) -=  A(i,k) * A (k,j);
    }
  }
}
```

6.5.2 Version séquentielle par blocs bidimensionels

```
void bloc_factorization_LU (matrix<matrix<double> >& A)
{
  for (int k=0 ; k < A.row_dim() ; k++ ) {
    factorization_LU( A(k,k) );             // A(k,k) = L( A(k,k) ) * U( A(k,k) )
    for (int i=k+1 ; i < A.col_dim() ; i++ )
       A(i,k).this_times_inverse_U(A(k,k));  // A(i,k) = A(i,k) * inv( U( A(k,k) ) )
    for (int j=k+1 ; j < A.row_dim() ; j++ )
      A(k,j).inverse_L_times_this(A(k,k));   // A(k,j) = inv( L( A(k,k) ) ) * A(k,j)
    for (int i=k+1 ; i < A.col_dim() ; i++ )
      for (int j=k+1 ; j < A.row_dim() ; j++ )
        A(i,j).minus_times_equal( A(i,k), A(k,j) ); // A(i,j) = A(i,j) - A(i,k) * A(k,j)
    }
  }
}

main()
{
  matrice<double> IN;
  cin >> IN; // lecture depuis l'entrée standard de la matrice IN

  // construction de la matrice (de dimension sqrt(n)) de blocs (de taille sqrt(n))
  // à partire de la matrice IN
  matrice<matrice<double> > A = IN.two_dim_partitioning(sqrt(IN.coldim()));
  bloc_factorization_LU(A);
  cout << A;
}
```

6.5.3 Version parallèle

Chacune des trois expressions des trois boucles de l'algorithme sont encapsulées dans trois tâches différentes.

```
//-------------------------------------------------------------------------
//encapsulation des expressions et indication des coûts de calcul.

struct factorisation_LU : public a1_task_elem {
  void operator()(a1_shared_r_w<matrix<double> > a) {
    (*a).factorization_LU();
  }
  cost_task cost(a1_size_t a) { return a*a*a*2/3; }
};

struct m_times_inverse_U : public a1_task_elem {
  void operator()(a1_shared_r_w<matrix<double> > m, a1_shared_r<matrix<double> > u) {
    (*m).this_times_inverse_U(*u);
  }
  cost_task cost(a1_size_t a, a1_size_t b) { return a*a*a; }
};

struct inverse_L_times_m : public a1_task_elem {
  void operator()(a1_shared_r_w<matrix<double> > m, a1_shared_r<matrix<double> > l) {
    (*m).inverse_L_times_this(*l);
  }
  cost_task cost(a1_size_t a, a1_size_t b) { return a*a*a; }
}

struct minus_times_equal : public a1_task_elem {
  void operator()(a1_shared_rp_w<matrix<double> > a, a1_shared_r<matrix<double> > b,
                  a1_shared_r<matrix<double> > c) {
    (*a).minus_times_equal( *b, *c );
  }
  cost_task cost(a1_size_t a, a1_size_t b, size c) { return a*a*a; }
};

//-------------------------------------------------------------------------
//algorithme d'élimination de gauss.

struct par_bloc_factorization_LU : public a1_task {
  void operator() (a1_collection_r<matrix<a1_shared_rp_wp<matrix<double> > > > A)
  {
    for (k=0 ; k < (*A).row_dim() ; k++ ) {
      a1_new_task(factorisation_LU<double>(), (*A)(k,k));
      for (i=k+1 ; i < (*A).col_dim() ; i++ ) {
        a1_new_task(m_times_inverse_U<double>(), (*A)(i,k), (*A)(k,k));
      for (int j=k+1 ; j < (*A).row_dim() ; j++ )
        a1_new_task(inverse_L_times_m<double>(), (*A)(k,j), (*A)(k,k));
        for (j=k+1 ; j < (*A).row_dim() ; j++ )
          a1_new_task(minus_times_equal<double>(), (*A)(i,j), (*A)(i,k), (*A)(k,j));
      }
      a1_flush();
    }
  }
};
```