Manuel d'utilisation de KASTL

Daouda Traoré
Equipe MOAIS INRIA-Montbonnot

4 septembre 2009

Résumé

KASTL (*Kaapi Adaptive Standard Template Library*) est une bibliothèque parallèle qui a été développée au dessus de KAAPI. Elle offre un ensemble d'algorithmes parallélisés de manière adaptative et facilement utilisable de la librairie standard C++ (*Standard Template Library* : STL). Elle utilise la même sémantique pour ses algorithmes que celle fournie par STL. Ce document décrit la manière d'utiliser les algorithmes de la librairie standard C++ (STL : *Standard Template Library*) implémentés dans KASTL et aussi la manière de construire ses propres algorithmes adaptatifs à partir de l'interface générique fournie par elle.

Table des matières

1 Installation de KAAPI et KASTL

Tout d'abord pour pouvoir utiliser KASTL, il faut installer la bibliothèque KAAPI dans laquelle elle y est déjà basée. Vous avez deux possibilités pour installer KAAPi et KASTL : vous pouvez installer soit la dernière version stable, soit celle de développement. Les étapes suivantes vous montrent les procédures à suivre.

1.1 Installation de la dernière version stable

- Téléchargez la version stable de KAAPI à l'adresse suivante :
  https ://gforge.inria.fr/frs/ ?group_id=94
- Consultez la documentation de KAAPI puis l'installer :
  http ://kaapi.gforge.inria.fr/doc
- Télécharger la dernière version stable de KASTL :
  https ://www-id.imag.fr/~traored/perso.html
- Décompressez l'archive de KASTL dans un répertoire.

1.2 Installation de la dernière version de développement

Pour installer la version de développement, suivez les étapes suivantes :

- Téléchargez la dernière version de développement de KAAPI incluant d'office KASTL :
  > **git clone git ://git.ligforge.imag.fr/git/kaapi/kaapi.git**
- Consultez la documentation de KAAPI puis l'installer :
  http ://kaapi.gforge.inria.fr/doc

1.3 Utilisation des librairies

1.3.1 Avec la version stable de KAAPI

Pour pouvoir utiliser KASTL avec la version stable, il faut donner deux répertoires au compilateur :

– **(kaapi)/install/include/kaapi** et **(kastl)/include**

ainsi qu'un répertoire à l'éditeur de liens :

– **(kaapi)/install/lib**

(kaapi) et **(kastl)** désignent les répertoires où vous avez installer KAAPI et KASTL.

1.3.2 Avec la version de développement de KAAPI

Pour pouvoir utiliser KASTL avec la version de développement, il faut donner deux répertoires au compilateur :

– **(kaapi)/install/include/kaapi** et **(kaapi)/install/kastl**

ainsi qu'un répertoire à l'éditeur de liens :

– **(kaapi)/install/lib**

(kaapi) désignent le répertoire où vous avez installer KAAPI.

2 Exemples d'utilisation des algorithmes de KASTL

2.1 Utilisation sans paramètre

Un algorithme de KASTL sans paramètre a la même signature (paramètres donné et résultat) que son équivalent STL. La figure 1 montre à partir d'un exemple, comment on peut facilement transformer un algorithme séquentiel STL à un algorithme parallèle adaptatif en KASTL. La figure 1(a) donne un exemple d'une fonction STL qui trie les éléments d'un vecteur d'entiers, et la figure 1(b) montre le même exemple avec le tri KASTL. La seule modification dans ce code (fig. 1 (b)) est le remplacement de **std** (fig. 1(a), ligne 12) par **kastl** (fig. 1(b), ligne 12), et le remplacement du fichier d'en-tête <**algorithm**> (fig. 1(a), ligne 2) par <**kastl/algorithm**> (fig. 1(b), ligne 2)..

```
1. #include <iostream>
2. #include <algorithm> //pour les algorithmes STL
3. #include <vector>
4.
5.
6. int main() {
7.
8.    int data[] = {12, 31, 2, 25, 16, 70, 33};
9.    std::vector<int> mydata(data, data+7);
10.
11.  //appel sort de la STL
12.  std::sort(mydata.begin(), mydata.end());
13. return 0;
14. }
```

(a) code STL

```
1. #include <iostream>
2. #include <kastl/algorithm> //pour les algorithmes KASTL
3. #include <vector>
4.
5.
6. int main() {
7.
8.    int data[] = {12, 31, 2, 25, 16, 70, 33};
9.    std::vector<int> mydata(data, data+7);
10.
11.  //appel sort de KASTL
12.  kastl::sort(mydata.begin(), mydata.end());
13.return 0;
14. }
```

(b) Transformation en code KASTL

FIG. 1 – Exemple d'utilisation sur le code KASTL kastl : :sort

Remarque : Dans la STL il y a deux classes d'algorithmes : une destinée aux applications non numériques (représentée par <**algorithm**>) et l'autre pour les applications numérique (représentée par <**numeric**>). Dans KASTL, il y a aussi ces deux classes, les non numériques sont représentées par <**kastl/algorithm**> et les numériques par <**kastl/numeric**>.

2.2 Utilisation avec paramètre dynamique

Pour optimiser les performances, d'autres paramètres sont fournis par KASTL. Ils sont fixés par défaut, mais un utilisateur qui souhaite les ajuster peut les faire en utilisant ses propres valeurs qu'il juge optimales pour ses applications. L'utilisation de ces paramètres nécessite seulement une modification légère de la fonction STL équivalente, c'est à dire ces paramètres sont juste ajoutés après les paramètres de fonction STL équivalente. Les paramètres les plus importants sont : le seuil d'amortissement de l'exécution séquentielle (**extract_seq**), et le seuil d'extraction de parallélisme (**extract_par**). KASTL peut utiliser ces paramètre d'une manière **dynamique**.

paramètres pour l'extraction séquentielle : KASTL utilise un schéma algorithmique qui adapte le grain en fonction de la taille restante de l'exécution en cours. Mais des phénomènes pratiques comme par exemple les contentions mémoires ou les effets cache peuvent influer le choix du grain par défaut. Pour obtenir davantage de performance ,

l'utilisateur peut ajuster ce choix en ajustant juste le facteur utilisé par KASTL pour adapter dynamiquement le grain. L'utilisateur peut choisir des valeurs comprises entre 1 et 500 pour ce choix. Par exemple si l'utilisateur décide de choisir un facteur égal à 10 pour la fonction *transform*, il peut appeler celle-ci de la manière suivante :
kastl : :transform(input.begin(), input.end(), output.begin(), op, 10), juste un paramètre de plus est ajouté par rapport à std.

paramètres d'amortissement du surcoûts de parallélisme : Les algorithmes comme : `adjacent_find`, `equal`, `find`, `find_if`, `find_end`, `find_first_of`, `mismatch`, `search`, `search_n`, `includes`, `partial_sum`, `remove`, `remove_if`, `remove_copy`, `remove_copy_if`, `set_difference`, `set_intersection`, `set_symmetric_difference`, `set_union`, `unique`, `unique_copy`, `stablepartition` introduisent tous des surcoûts arithmétiques pour leurs parallélisations. KASTL amortit ces surcoûts en utilisant un paramètre initial défini par défaut, mais l'utilisateur peut ajuster ce paramètre pour obtenir plus de parallélisme. La figure 2 montre comment on utilise ce paramètre sur l'algorithme `partial_sum` (ligne 9). Pour modifier le paramètre dans chacun de ces algorithmes cités ci-dessous, on initialise juste **kastl : :nom_algo_macroloop_initsize**.

```
1. #include <iostream>
2. #include <kastl/numeric> //pour numeric KASTL
3. #include <vector>
4.
5. int main() {
6.
7.    std::vector<int> mydata(data, 1);
8.   //initialisation du paramètre  pour amortir le surcoût arithmétique
9.   kastl::partiall_sum_macroloop_initsize = 30;
10.  //appel partial_sum de KASTL
11.  kastl::partial_sum(mydata.begin(), mydata.end());
12.return 0;
13. }
```

FIG. 2 – Exemple d'utilisation du paramètre d'amortissement séquentiel sur kastl : :partial_sum

2.3 Utilisation avec paramètre statique

paramètres pour l'extraction séquentielle à pas constant : Si un utilisateur veut que son extraction séquentielle se fait par pas constant, il n'a qu'à passer juste un nombre négatif ou zéro à l'algorithme sur lequel il veut que ça se passe. Par exemple, si l'utilisateur veut que l'algorithme `transform` extrait un par un les éléments du tableau, il n'a qu'à appeler de la manière suivante :
kastl : :transform(input.begin(), input.end(), output.begin(), op, 0)
Et si il veut que l'extraction se fasse par pas de 100, il n'a qu'à appeler l'algorithme de la manière suivante :
kastl : :transform(input.begin(), input.end(), output.begin(), op, -99)

paramètres pour les algorithmes de tris : Comme la STL, KASTL fournit aussi deux algorithmes génériques de tri : `sort` et `stable_sort` ce dernier garantissant l'ordre relatif de deux éléments de même valeur par rapport à la relation d'ordre choisie. Ces deux algorithmes utilisent un grain par défaut pour amortir le surcoût de créations de tâches. Pour pouvoir ajuster la valeur de ce grain, l'utilisateur peut les faire en affectant des valeurs à ces variables : **kastl : :sort_grain_size** (pour le tri non stable) et **kastl : :stable_sort_grain_size**. La figure 3 montre comment on utilise ce paramètre sur l'algorithme `stable_sort` (ligne 9).

```
1. #include <iostream>
2. #include <kastl/algorithm> //pour algorithm KASTL
3. #include <vector>
4.
5. int main() {
6.
7.    std::vector<int> mydata(data, 1);
8.   //initialisation du paramètre  pour amortir le surcoût arithmétique
9.   kastl::stable_sort_grain_size = 500;
10.  //appel partial_sum de KASTL
11.  kastl::stable_sort(mydata.begin(), mydata.end());
12.return 0;
13. }
```

FIG. 3 – Exemple d'utilisation du paramètre grain kastl : :stable_sort

3 L'interface générique

Dans cette section, nous montrons l'utilisation de l'interface générique fournie par KASTL. Cette interface fournit trois classes génériques qui peuvent être étendues pour développer des algorithmes parallèles adaptatifs. Ces trois classes sont : la classe **WorkAdapt**, la classe **NextJump**, et la classe **FinalizeWork**.

La classe **WorkAdapt** permet de construire un descripteur de travail adaptatif. Elle fournit 13 fonctions qui sont décrites dans le tableau 1. Un utilisateur souhaitant développer un algorithme parallèle doit instancier (spécialiser) les fonctions dont il aura éventuellement besoin. Ces fonctions seront implémentées sans aucun verrou de synchronisation car ils sont gérés par le moteur exécutif de KASTL. La classe **WorkAdapt** utilise la classe **NextJump**. L'utilisateur doit étendre la classe **NextJump** pour définir sa propre classe qui permet de faire le saut et la classe **FinalizeWork** pour définir sa propre classe de finalisation. Par exemple si nous considérons qu'un WorkAdapt est décrit par un intervalle $w = [f, l[$ où f est un indice de début et l un indice de fin, et si on suppose que w a été préempté à l'indice k, alors on peut définir l'intervalle $[k, l[$ comme un **NextJump**. Toutes ces fonctions ne seront pas complétées par l'utilisateur, il les remplira selon ses besoins. Les fonctions **bool extract_nextseq()**, **bool extract_par()**, **void localcompute()**, **void setjump(NextJump*&)**, **void jump(NextJump*)** sont virtuelles pures et doit être obligatoirement spécialisées par l'utilisateur.

Fonction	Spécification
bool extract_nextseq()	permet d'extraire une partie du travail localement, retourne faux s'il n'y a plus de travail à extraire.
bool extract_par()	permet de voler une partie du travail sur le travail restant à faire, retourne faux s'il n'y a plus de travail à extraire.
void localcompute()	permet d'exécuter localement l'algorithme séquentiel optimal.
bool is_empty()	doit retourner vrai si le travail est vide sinon faux.
void join(const WorkAdapt* sw)	Permet de faire la fusion du résultat du voleur avec le résultat de la victime.
void setjump(NextJump*&)	Permet d'effectuer le saut sur le travail qui a été fait par le voleur.
void jump(NextJump*)	Permet d'affecter le travail restant à faire par le voleur qui a été préempté à la victime qui sera son nouveau travail.
bool extract_next_macrostep()	Permet d'extraire la taille de la macro-loop
bool get_finalize_work(FinalizeWork*& fw)	Permet de récupérer le travail à finaliser par le voleur qui a été préempté.
void get_main_result()	Permet de récupérer le résultat local de la victime.
void get_result()	Permet de récupérer le résultat local du voleur.
bool must_jump()	Permet d'autoriser tous les exécuteurs à faire un saut sur leurs voleurs respectifs.
bool is_finalize_work() const	retourne vrai si le calcul doit être finalisé sinon faux.

TAB. 1 – Interface fournie par la classe WorkAdapt

Fonction	Spécification
bool extract_nextseq()	permet d'extraire une partie du travail localement, retourne faux s'il n'y a plus de travail à extraire.
bool extract_par()	permet de voler une partie du travail sur le travail restant à faire, retourne faux s'il n'y a plus de travail à extraire.
void localcompute()	permet d'exécuter localement l'algorithme séquentiel optimal.

TAB. 2 – Interface fournie par la classe FinalizeWork

3.1 Exemple : `partial_sum`

La fonction **kastl : :run_adapt**(work) permet de lancer le moteur adaptatif sur le travail construit (work).

```
//The finalization Work
template<class T, class BinOp>
class Partial_Sum_FinalizeWork : public kastl::FinalizeWork {

    T _first, _last;
    BinOp _op;
    Distance_type _beg, _end;
    typedef typename std::iterator_traits<T>::value_type val_t;
    val_t _val;
    Distance_type _beg_local, _end_local;
    Distance_type _c, _pargrain;

public :
    Partial_Sum_FinalizeWork(T first, T last, BinOp op, Distance_type beg, val_t val=0,
                             Distance_type c=100, Distance_type pargrain=2):
       FinalizeWork(), _first(first),  _last(last), _op(op), _beg(beg), _end(last-first),
        _val(val), _c(c), _pargrain(pargrain) { }

    bool extract_nextseq() {
        if( _beg < _end ) {
            _beg_local = _beg;
            _beg += (_c > 0)?(_c * kastl::log_2(_end-_beg+1)):(-_c+1);
            if( _beg > _end)  _beg = _end; _end_local = _beg;
           return true;
         }
       return false;
    }

    bool extract_par( FinalizeWork*& sw) {
          if((_end - _beg) > _pargrain) {
             Distance_type i, j;
             Distance_type mid = _end - ((_end - _beg)/2);
             i = mid; j = _end; _end = mid;
             sw = new  Partial_Sum_FinalizeWork<T, BinOp>( _first, _first+j, _op, i,
                                                           _val, _c, _pargrain);
               return true;
          } else {
            sw = 0; return false; }
     }

    void localcompute() {
       std::transform(_first + _beg_local, _first + _end_local, _first + _beg_local,
                      UnaryOp<val_t, BinOp>(_val, _op)) ;
    }

   Util::OStream& pack(Util::OStream& os) const { return os ; }
   Util::IStream& unpack(Util::IStream& is, FinalizeWork*& target) {  return is; }
};
```

FIG. 4 – Spécialisation de la classe FinalizeWork pour `partial_sum`

```
template<class In, class Out, class BinOp>
class PARTIAL_SUM_WORK : public kastl::WorkAdapt
{
    In _first,_last;
    Out _res;
    BinOp _op;
    typedef typename std::iterator_traits<Out>::value_type val_t;
    val_t _local_res;
    Distance_type _beg, _end;
    const Distance_type _init_beg, _init_end;
    Distance_type _c, _pargrain;
    Distance_type _beg_local,_end_local;
    val_t _main_res,_thieft_res;

    public:

    PARTIAL_SUM_WORK(In first, In last, Out res, BinOp op, Distance_type beg,
                     Distance_type c =100, Distance_type pargrain=2): WorkAdapt(),
      _first(first),  _last(last), _res(res), _op(op), _local_res(val_t(0)), _beg(beg),
      _end(last-first), _init_beg(beg),_init_end(last - first), _pargrain(pargrain), _c(c)
    { }

    bool extract_nextseq() {
        if( _beg < _end ) {
            _beg_local = _beg;
            _beg += (_c > 0)?(_c * kastl::log_2(_end-_beg+1)):(-_c+1);
            if( _beg > _end)  _beg = _end; _end_local = _beg;
           return true;
         }
       return false;
    }

    bool extract_par( WorkAdapt*& sw) {
           if((_end - _beg) > _pargrain) {
             Distance_type i, j;
             Distance_type mid = _end - ((_end - _beg)/2);
             i = mid; j = _end; _end = mid;
             sw = new  PARTIAL_SUM_WORK<In, Out, BinOp>(_first, _first+j, _res, _op, i,
                                                        _c, _pargrain);
                   return true;
           } else {
            sw = new  PARTIAL_SUM_WORK<In, Out, BinOp>(_first, _first, _res, _op,0);
            return false;
          }
    }
```

FIG. 5 – Spécialisation de la classe WorkAdapt pour `partial_sum`

```
  bool  extract_next_macrostep(){
       static Distance_type alpha = partial_sum_macroloop_initsize ;
       bool extractmacro = (_beg  < _init_end);
       if(extractmacro){

         Distance_type completed_size = _beg - _init_beg ;
         Distance_type lg_sz = kastl::log_2(kastl::log_2( completed_size + 4 ) ) ;
         Distance_type size_macrostep= alpha*completed_size / lg_sz ;
         if (size_macrostep < partial_sum_macroloop_initsize)
             size_macrostep = partial_sum_macroloop_initsize ;
         _end = _beg + size_macrostep ;

         if(_end > _init_end) _end = _init_end;
       }
     return extractmacro;
}

void join(const WorkAdapt* jw){
     const  PARTIAL_SUM_WORK * src = dynamic_cast< const  PARTIAL_SUM_WORK*> ( jw );
     val_t thief_res = src->get_result();
     _local_res = _op(_local_res, thief_res) ;
     Distance_type thieft_end = src->get_end();
        *((_res+thieft_end)-1)   = _local_res;
 }

void  get_main_result(const WorkAdapt* jw ) {
     const  PARTIAL_SUM_WORK * src = dynamic_cast< const  PARTIAL_SUM_WORK*> ( jw );
     _main_res = src->get_result();
}

void  localcompute() {
    val_t val = ( (_local_res==val_t(0))? *(_first+_beg_local) : _op(_local_res, *(_first
   *(_res+_beg_local)   = val ;
    _local_res = *(partial_sum_local_compute(_first + _beg_local, _first + _end_local,
                                             _res+_beg_local, _op)) ;
}

inline bool is_finalize_work() const { return true; }

void get_finalize_work(kastl::FinalizeWork*& fw) {
     fw = new Partial_Sum_FinalizeWork<Out, BinOp>(_res, (_res+_beg)-1, _op,
                 _init_beg, _main_res, _c, _pargrain);
}

val_t get_result() const { return _local_res ;}
Distance_type get_end() const { return _end; }
```

FIG. 6 – Spécialisation de la classe WorkAdapt pour `partial_sum` (suite)

```
   void setjump(kastl::NextJump*& j) {
         Distance_type end = _end; _end = _beg;
         j = new kastl::NextJumpWork(_beg, end);
   }

   void jump(const kastl::NextJump* j) {
        const  kastl::NextJumpWork * nextj = dynamic_cast< const  NextJumpWork*> ( j );
        _beg = nextj->get_beg();_end = nextj->get_end();
   }

   Util::OStream& pack(Util::OStream& os) const { return os ; }
   Util::IStream& unpack(Util::IStream& is, WorkAdapt*& target) { return is;}
};

////////////////////Adaptive parallel partial_sum/////////////////////////////

template <typename RandomAccessIterator1,  typename RandomAccessIterator2, typename BinOp>
inline RandomAccessIterator2 partial_sum_adapt(RandomAccessIterator1 first,
                                               RandomAccessIterator1 last,
                                               RandomAccessIterator2 res,
                                               BinOp op,
                                               std::random_access_iterator_tag,
                                               std::random_access_iterator_tag,
                                               Distance_type seqgrain=100,
                                               Distance_type pargrain=2) {
    if(first==last) return res;
    typedef typename std::iterator_traits<RandomAccessIterator2>::value_type val_t;
    PARTIAL_SUM_WORK<RandomAccessIterator1, RandomAccessIterator2, BinOp>* work;
    work = new  PARTIAL_SUM_WORK<RandomAccessIterator1, RandomAccessIterator2, BinOp>(first
                last, res, op, 0, seqgrain, pargrain);

    kastl::run_adapt(work);

  return res + (last-first) ;
}
```

FIG. 7 – Spécialisation de la classe WorkAdapt pour `partial_sum` (suite)