Slide 1

Évolution d'un langage pour la programmation parallèle multi-contextes : Threaded-C

Guy Tremblay
Dépt. d'informatique
UQAM

Séminaire du dépt. d'informatique
14 sept. 2001

Slide 2

0 Contexte

Projet EARTH (Groupe *CAPSL, Univ. of Delaware*):

- EARTH = *Efficient Architecture for Running THreads*

= Architecture multi-contextes supportant le parallélisme de fine granularité

Langage Threaded-C (version 1.0) = langage conçu . . .

- pour la programmation de l'architecture EARTH
- par des architectes (de niveau matériel)

=> très bas niveau d'abstraction (langage "machine")

Objectif de mon mandat (congé sabbatique, jan. à juin 2000)

= Simplifier et améliorer le langage, mais en préservant le contrôle sur la machine ;

= Augmenter le langage pour supporter les machines SMP

Slide 3

Sommaire:

1. L'architecture EARTH et son modèle de programmation
2. Le langage initial : Threaded-C (version 1.0)
3. Le langage révisé : Threaded-C (version 2.0)
4. Évolution future de Threaded-C
5. Conclusion

Slide 4

1 L'architecture EARTH et son modèle de programmation

Origine de l'architecture EARTH = architectures à flux de données :

- Programme = graphe de flux de données
  - Noeud du graphe = instruction
  - Arc = donnée échangée entre deux instructions

Propriétés intéressantes :

- Noeud = instruction => granularité très (trop) fine
- Ordre partiel => implicitement parallèle

Modèle EARTH : programme = graphe, mais . . .

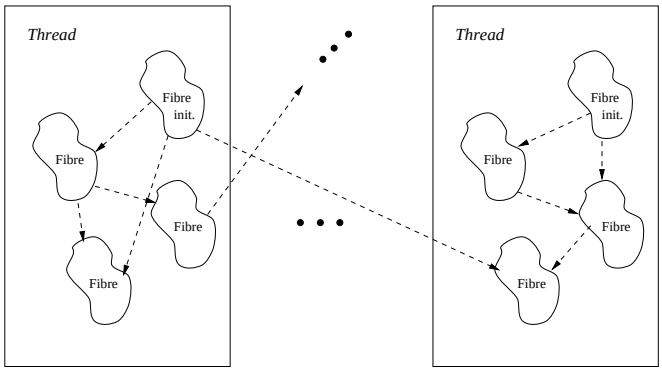
- Noeud = *groupe* d'instructions (fibre)
- Arc = signal de synchronisation (pas de *flux* de jetons)

Slide 5

Threads vs. fibres

Modèle EARTH => hiérarchie à deux niveaux *threads* et fibres :

- *Thread* = activation parallèle d'une procédure (comme Pthreads ou Java)
- Fibre = séquence d'instructions provenant d'un même *thread*



Slide 6

Approche EARTH avec *threads* et fibres = compromis entre . . .

- Machine *dataflow* pure => nombre illimité de contextes de granularité fine
- Architecture classique => nombre très limité de contextes poids lourd

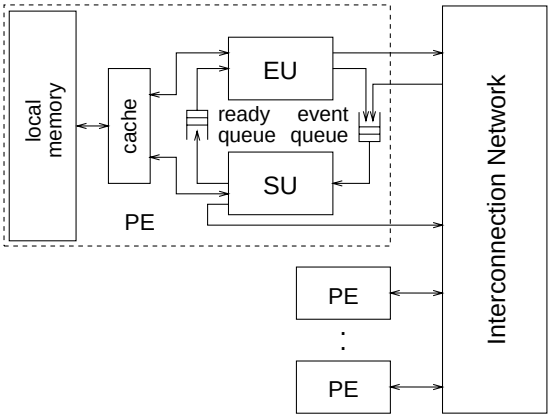
Caractéristiques des fibres EARTH :

- Les fibres d'un *thread* partagent le même contexte
  (bloc d'activation de la procédure: arguments, vars locales, etc.)
- Une fibre est activée par la réception de signaux de synchronisation
  (*dataflow scheduling*)
- Les instructions d'une fibre sont exécutées séquentiellement
  (sémantique usuelle)
- Une fibre s'exécute sans jamais bloquer et sans jamais être interrompue
  (*non-blocking and non-preemptive*)

Slide 7

Impact sur l'architecture des caractéristiques des fibres :

- Pipeline standard pour l'exécution des fibres (EU)
- Unité indépendante de synchronisation (SU)



Slide 8

Modèle de mémoire EARTH

Espace global *d'adressage*

+

Mémoire physiquement distribuée (*NUMA*)

+

Fibres non-bloquantes

=>

Notion de pointeur `GLOBAL` (accès non-local)

Opérations spéciales (*split-phase*) de communication :

```
PUT_SYNC   ( src,     adr_dst,            fente_de_sync );
GET_SYNC   ( adr_src, adr_dst,            fente_de_sync );
BLKMOV_SYNC( adr_src, adr_dst, nb_octets, fente_de_sync );
```

Slide 9

2 Le langage initial : Threaded-C (version 1.0)

Threaded-C (version 1.0) :

- Pas supposé être utilisé pour la programmation
  (langage cible pour compilateurs)
- Conçu comme *le langage machine* de l'architecture EARTH

=> Niveau d'abstraction assez faible

Langage pour programmation de haut niveau = EARTH-C, mais

- Écart sémantique important entre EARTH-C et EARTH
- Absence de contrôle sur la granularité et les communications

=> EARTH-C fut rapidement abandonné en faveur de Threaded-C

Slide 10

```
 1  THREADED fib( int n, int *GLOBAL resultat, SPTR termine )
 2  {
 3      SLOT SYNC_SLOTS[1];
 4      int r1, r2;
 5
 6      INIT_SYNC(0, 2, 2, 1);
 7
 8      if (n <= 1) {
 9          DATA_RSYNC_L( 1, resultat, termine );
10      } else {
11          TOKEN( fib, n-1, TO_GLOBAL(&r1), SLOT_ADR(0) );
12          TOKEN( fib, n-2, TO_GLOBAL(&r2), SLOT_ADR(0) );
13          END_THREAD();
14
15      THREAD_1:
16          DATA_RSYNC_L( r1 + r2, resultat, termine );
17      }
18      END_FUNCTION();
19  }
```

Figure 1: La procédure récursive `fib` en Threaded-C (version 1.0)

Slide 11

Faiblesses et difficultés de la première version du langage

De nombreuses applications furent développées mais . . . langage pas facile à utiliser :

- Identification des fentes de synchronisation et des fibres à l'aide de *numéros* ;
- Association *non-locale* entre une fibre, une fente et le nombre de signaux ;
- Instructions *non-génériques* de communication et synchronisation :

```
GET_SYNC_B  GET_SYNC_S  GET_SYNC_L  GET_SYNC_F  GET_SYNC_D  GET_SYNC_G
GET_RSYNC_B GET_RSYNC_S GET_RSYNC_L GET_RSYNC_F GET_RSYNC_D GET_RSYNC_G
```

- Pas de convention uniforme sur les noms (THREAD/END_THREAD, _SYNC) ;
- Instruction `END_THREAD()` redondante ;
- Notion de pointeur `GLOBAL` difficile à maîtriser ;

Slide 12

3 Le langage révisé : Threaded-C (version 2.0)

```
 1  THREADED fib( int n, int *GLOBAL resultat, SPTR termine )
 2  {
 3    int r1, r2;
 4
 5    if (n <= 1) {
 6      PUT_SYNC( 1, resultat, termine );
 7      TERMINATE;
 8    } else {
 9      TOKEN( fib, n-1, TO_GLOBAL(&r1), TO_SPTR(RETOURNER_RES) );
10      TOKEN( fib, n-2, TO_GLOBAL(&r2), TO_SPTR(RETOURNER_RES) );
11    }
12
13    FIBER RETOURNER_RES <* 2 *> {
14      PUT_SYNC( r1 + r2, resultat, termine );
15      TERMINATE;
16    }
17  }
```

Figure 2: La fonction récursive `fib` en Threaded-C (version 2.0)

Slide 13

Simplification et uniformisation du langage

- Identificateurs symboliques ;
- Façon plus uniforme de nommer les instructions, par ex., suffixe `SYNC =>` envoi d'un signal de synchronisation ;
- Déclaration *implicite* des fentes et déclaration *locale* du nombre de signaux ;
- Fin de fibre *implicite* : exécution selon la sémantique usuelle du langage C, jusqu'à ce que `FIBER` ou `TERMINATE` soit rencontré ;
- Instructions *génériques* de synchronisation et transfert de données : `PUT_SYNC`, `GET_SYNC`, `BLKMOV_SYNC` et `SYNC`.

La notion de pointeur `GLOBAL` a été préservée pcq. fondamentale à cause des fibres non-bloquantes.

Slide 14

Programmation multi-contextes de machines *SMP*

Autre objectif de la révision du langage :

- Comportement correct des programmes s'exécutant sur des *SMP clusters* :

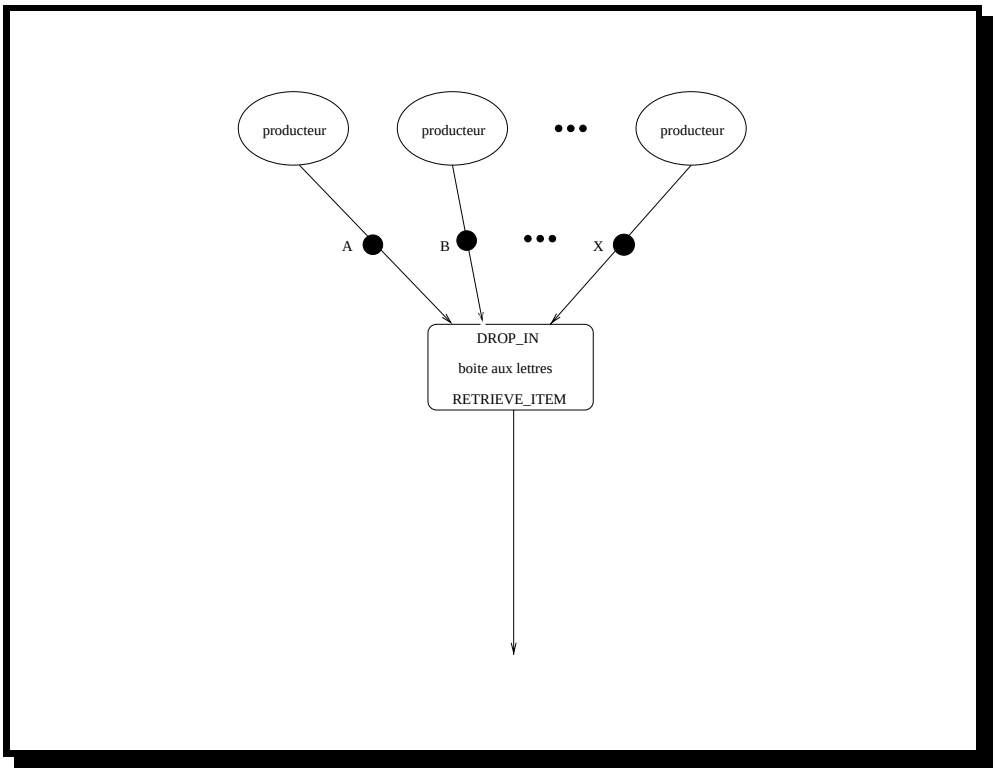
  => *Plusieurs* fibres peuvent accéder à la même mémoire en même temps

  => Besoin de mécanismes d'exclusion

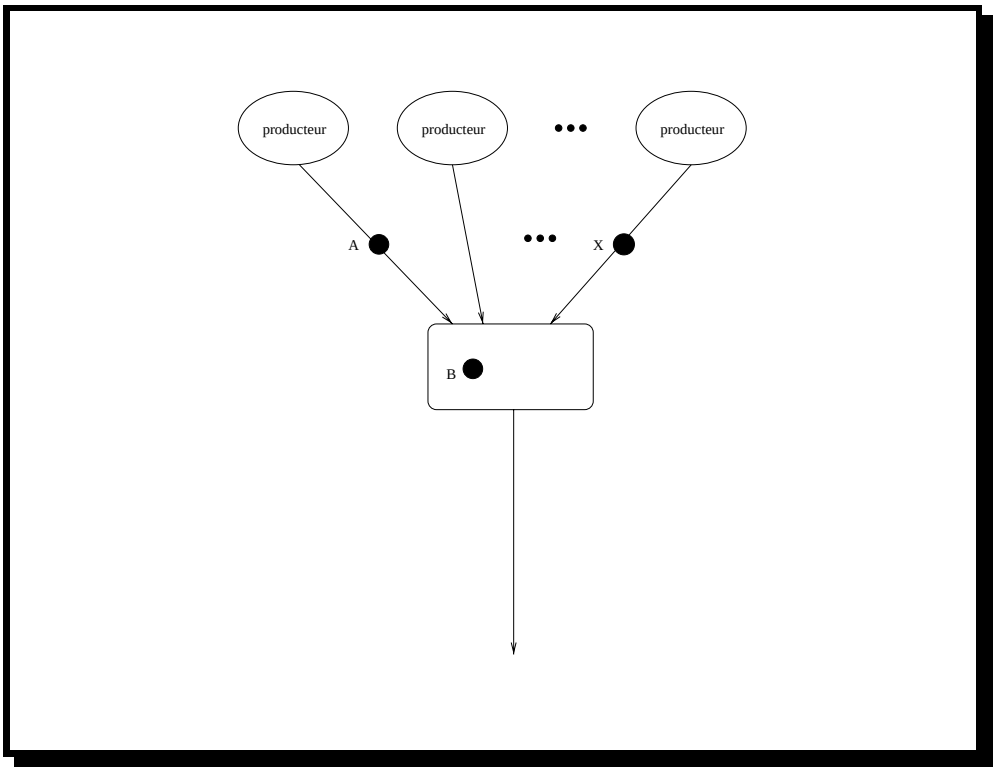
Deux mécanismes ont été introduits :

1. Fibre `EXCLUSIVE =>` dans un *thread*, *une seule fibre exclusive* à la fois peut s'exécuter ;
2. Boîtes aux lettres atomiques = *opération de fusion* non-déterministe
   (combiner des valeurs provenant de différentes sources) :
   (a) `INIT_MAILBOX` : allocation et initialisation ;
   (b) `DROP_IN` : transmission (à distance) d'un item et envoi d'un signal ;
   (c) `RETRIEVE_ITEM` : retrait (local) d'un item (arbitraire) ;

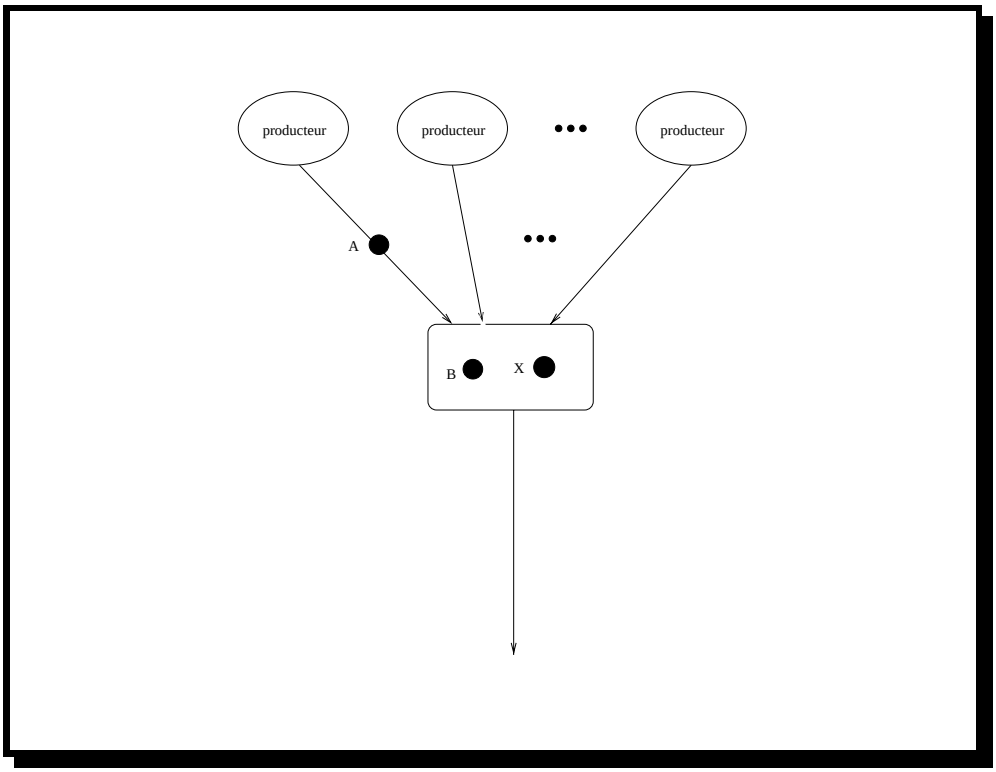
Slide 15



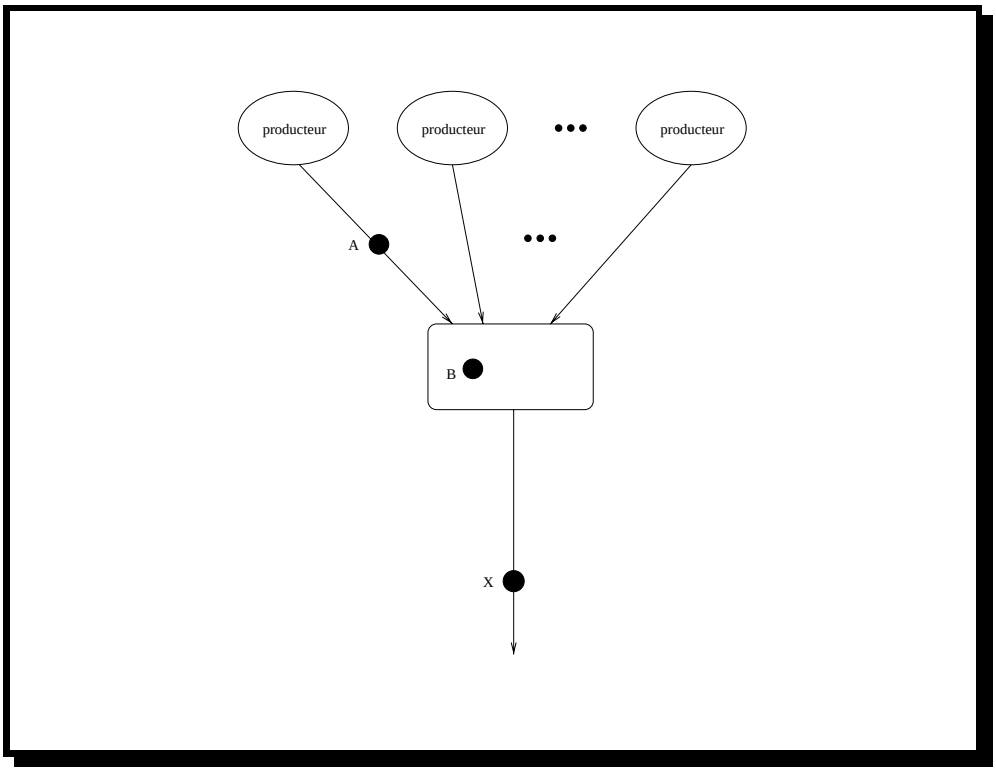
Slide 16



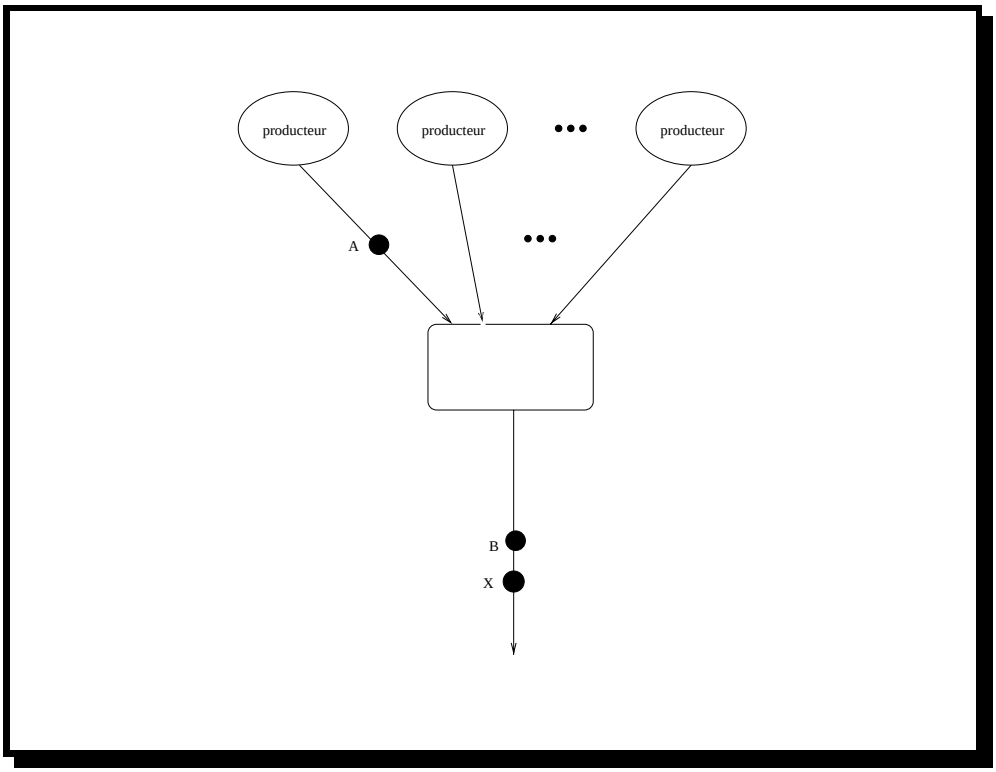
Slide 17



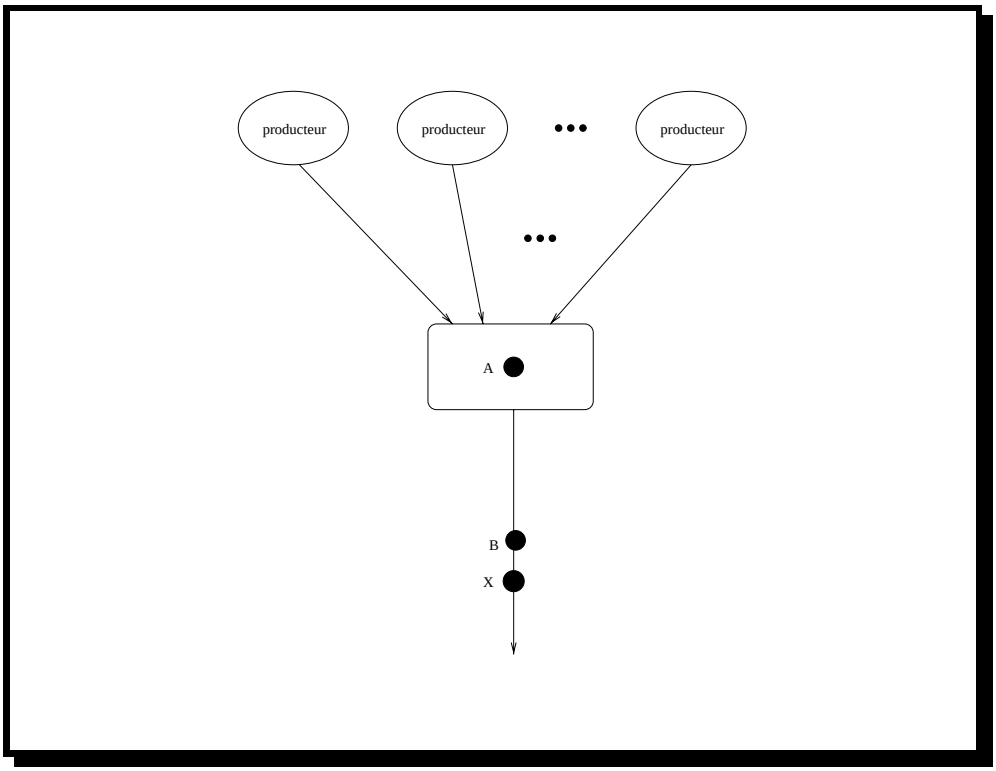
Slide 18



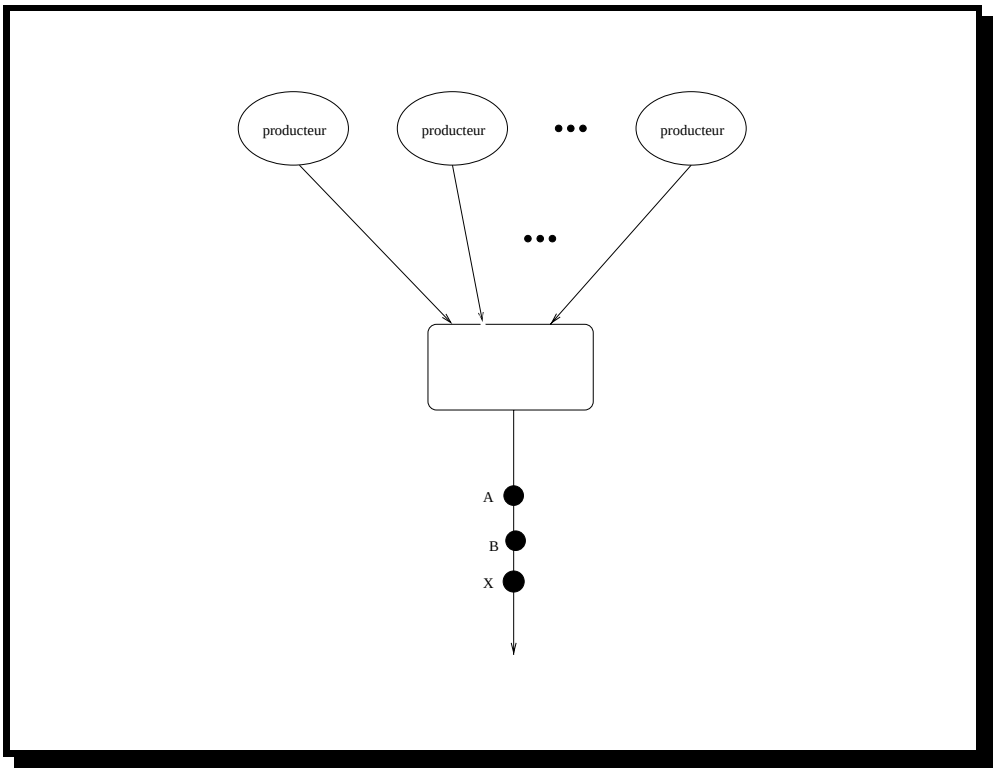
Slide 19



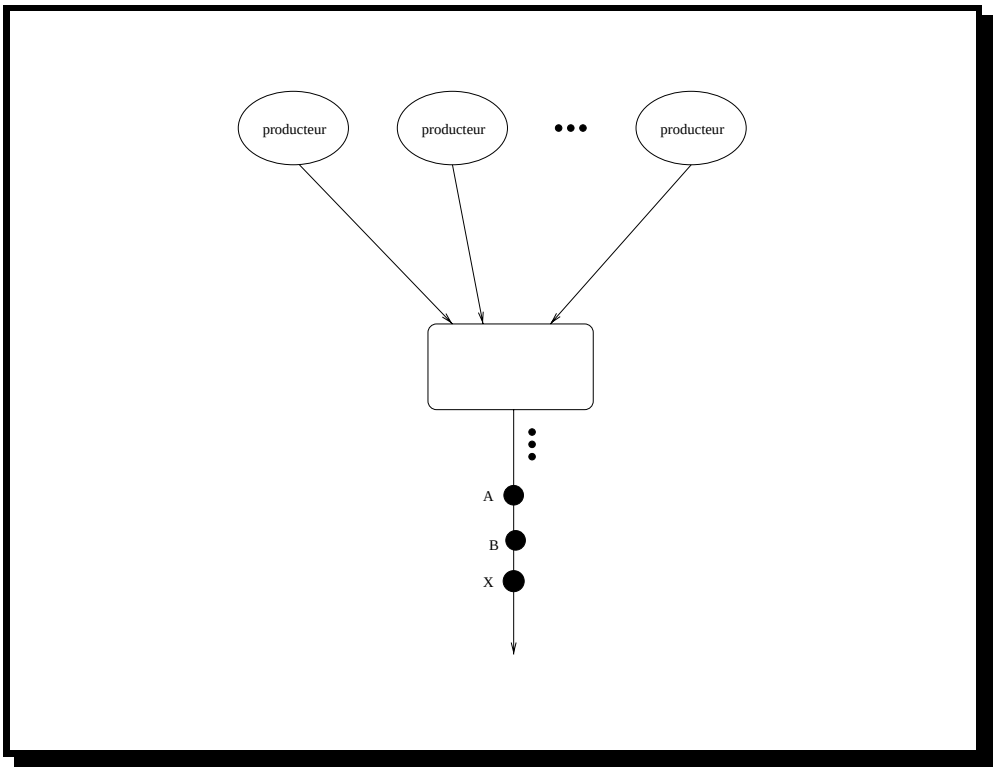
Slide 20



Slide 21



Slide 22



Slide 23

```
THREADED producteur( MAILBOX *GLOBAL mb )
{
  ...; DROP_IN( mb, &n, sizeof(int) ); ...
}


/*********************************************************/

  MAILBOX mb;

  INIT_MAILBOX( &mb, TRAITER_ITEM );

  for( i = 0; i < NUM_NODES; i++ )
    INVOKE( i, producteur, TO_GLOBAL(&mb) );


  EXCLUSIVE FIBER TRAITER_ITEM <* 1 *> {
    int v;

    RETRIEVE_ITEM( mb, &v );
    total += v;
    ...
  }
```

Slide 24

Propriétés de l'exemple précédent:

- Le processus maître reçoit les valeurs produites par les différents producteurs
- La mise à jour de la variable `total` est faite de façon atomique parce que la fibre `TRAITER_ITEM` est EXCLUSIVE
- L'atomicité des ajouts/retraits dans la boîte aux lettres est assurée par le RTS
- Cette solution requiert un nombre minimum de communications inter-processeurs = une (1) communication par `producteur`

Slide 25

Autres mécanismes de synchronisation

Les mécanismes de base peuvent être utilisés pour en définir d'autres :

- Verrous d'exclusion mutuelle ;
- Sémaphores ;
- I-structures ;
- Canaux de communication inter-processus ;
- Opérations parallèles de réduction ;

Caractéristique importante = les opérations bloquantes deviennent *split-phase* :

```
LOCK_SYNC( verrou, TO_SPTR(VERROU_OBTENU) );
/* A ce point, le verrou n'a *pas* encore ete obtenu. */
...

FIBER VERROU_OBTENU <* 1 *> {
   /* Debut de la section critique. */
   ...
   UNLOCK(verrou)
   /* Fin de la section critique. */
}
```

Slide 26

Autre exemple : Sémaphores à la POSIX

```
void sem_init   ( sem* s, int val_init );
void sem_wait   ( sem* s );
void sem_post   ( sem* s );
void sem_destroy( sem* s );
```

API des sémaphores POSIX

```
typedef struct SEMAPHORE *GLOBAL SEMAPHORE;

void SEM_INIT_SYNC( SEMAPHORE* sem, int val_init, SPTR sem_cree );
void SEM_WAIT_SYNC( SEMAPHORE  sem, SPTR wait_termine );
void SEM_POST     ( SEMAPHORE  sem );
void SEM_DESTROY  ( SEMAPHORE  sem );
```

API des sémaphores en Threaded-C

Slide 27

Stratégie pour la mise en oeuvre des sémaphores (et autres mécanismes) :

- Chaque nouveau sémaphore est associé à une instance d'un *thread* = procédure `semaphore_handler`
- Chaque opération sur le sémaphore est réalisée par la combinaison deux composants (style *stub*/*skeleton*) :
  1. Un mandataire (*proxy*) qui transmet une requête au *thread* associé au sémaphore, la requête étant un message transmis via une *sync slot* ou une boîte aux lettres
  2. Une fibre (exclusive) du *thread* du sémaphore traite la requête

Note: le `semaphore_handler` peut être considéré comme un *moniteur actif* = un processus actif (par opposition à une structure de données passive) qui encapsule et gère une ressource en assurant l'atomicité des opérations.

Slide 28

4 Évolution future de Threaded-C

Threaded-C supporte différentes stratégies de programmation :

- Mémoire partagée (fibres d'un même *thread*, *threads* sur même noeud);
- Échange de messages (entre *threads* via opérations de communication).

Mais, en ce qui concerne les communications . . .

- L'utilisation de pointeurs globaux peut conduire à une forme de *couplage pathologique* (*content coupling*) ;
- La structure et le contenu des échanges n'est pas clair ou explicite.

Une solution possible = Introduction de fibres avec arguments et points d'entrée :

- Rend plus explicite le contenu des communications ;
- Diminue le couplage ;

Slide 29

```
THREADED fib( int n, ENTRY(int) resultat )
{
  if (n <= 1) {
    SEND( resultat, 1 );
    TERMINATE;
  } else {
    TOKEN( fib, n-1, TO_ENTRY(RES1) );
    TOKEN( fib, n-2, TO_ENTRY(RES2) );
  }

  FIBER RES1( int r1 )
        RES2( int r2 ) {
    SEND( resultat, r1+r2 );
    TERMINATE;
  }
}
```

Figure 3: La fonction récursive `fib` en Threaded-C avec points d'entrée et fibres avec arguments

Slide 30

Un prototype est déjà disponible :

- Pré-processeur écrit en `perl` qui génère du Threaded-C (version 2.0) utilisant des boîtes aux lettres ;
- Développé par Julien Sauvageau (aut. 2000), dans le cadre d'un stage (bacc. en info. de gestion) ;
- Principale limite : n'effectue aucune vérification de types :(

Slide 31

5 Conclusion

Caractéristiques de Threaded-C :

- Hiérarchie à deux niveaux de parallélisme avec *threads* et fibres
  => meilleur contrôle de la *granularité* du parallélisme ;
- Programmation par mémoire partagée ou par échanges de messages ;
- Liens de communication arbitraires entre *threads*
  (producteur–consommateur, pipeline logiciel, liens bi-directionnels) ;

Objectifs du mandat atteints :

- Langage résultant “plus simple”, mais sans introduire un important fossé sémantique ;
- Modifications du langage et mise en oeuvre complétées en un temps relativement court par une petite équipe ;
- Va permettre de continuer d’explorer les propriétés de l’architecture EARTH.