Parallel Programming Models: Where does Threaded-C stand?

G. Tremblay

UQAM

Presented at the University of Delaware

August 16, 2001

Slide 1

Outline

1. Programming *paradigms* and parallel programming
2. Various issues and goals for parallel programming models
3. Evolution of `Threaded-C`: past . . . and future

Slide 2

Slide 3

1 Programming *paradigms* and parallel programming

Paradigme de programmation $\approx$ general way of approaching a programming problem, generally using a language that support appropriate abstraction mechanims

- Paradigme procédurale => procédures et sous-routines
- Paradigme orienté objets => classe d'objets
- Paradigme fonctionnel => valeurs et fonctions

Paradigm as a way to look at the world

"*When the only tool one knows is a hammer, one sees nails everywhere!*"

Slide 4

Concurrent programming

Concurrent program = program that contains one or more tasks that *cooperate* and, thus, *communicate*

Concurrent programming = Multi-threaded programming *or* Parallel programming *or* Distributed programming

Parallel programming = different tasks execute on different processors and collaborate in order to solve a given problem more rapidly (or, dually, solve a larger problem)

Two major ways to communicate:

- By using shared variables
- By exchanging messages

Slide 5

A possible way to categorize the various parallel programming paradigms

Implicit parallelism	Explicit parallelism	
Shared variables / Messages	Shared variables / Messages	Imperative [Procedural] [OO]
Shared variables / Messages	Shared variables / Messages	Declarative [Functional] [Logic]

Slide 6

Explicit vs. implicit parallelism

- Explicit
  - Instructions spéciales de création de *threads*, de synchronisation et communication, etc.
  - Exemples: `fork/join`, `cobegin/coend`, `forall`
- Implicit
  - Programme *ordinaire*: c'est le compilateur qui se charge d'identifier et d'exploiter le parallélisme
  - Exemples: languages fonctionnels, *dusty decks*

Slide 7

Imperative vs. declarative

- Impératif
  - Basé sur l'utilisation de structures de contrôle et d'affectations (notion d'*état modifiable*)
  - L'ordre d'exécution est spécifiée par le programmeur
- Déclaratif
  - Basé principalement sur l'utilisation d'expressions, de valeurs et de fonctions
  - L'ordre d'exécution est souvent non-spécifié: basé sur les dépendances de données

Slide 8

Shared variables vs. Message passing

- Variables partagées
  - Les processus partagent un état global modifiable
  - L'information est communiquée entre processus en modifiant les variables partagées
- Échanges de messages
  - Chaque processus n'a accès qu'à ses propres variables locales
  - Les processus communiquent en s'envoyant des messages

2 Various issues and goals for parallel programming models

Key (abstract) steps in developping parallel programs:

Slide 9

1. Decomposition into tasks and processes
2. Orchestration of the necessary data access, communication, and synchronization
3. Mapping (binding) of tasks and processes to processors

Decomposition into tasks

Key strategy = *divide-and-conquer*

But . . . may take many different forms

- Recursive parallelism
- Iterative parallelism
- Workers parallelism (bag of tasks)
- Specialist parallelism

Slide 10

Not all programming models or languages easily support all decomposition strategies

Example: Computing the sum of the elements of an array

Slide 11

Orchestration of data access, communication, and synchronization

Clearly depend on the major information exchange paradigm

- Shared memory
- Message passing

Shared memory

- Some control on the distribution of data (to enhance locality)?
- How is exclusive access (critical sections) supported?
  - Locks and barriers
  - Semaphores
  - Monitors
- How is condition synchronization supported?
  - Semaphores
  - Condition variable

Slide 12

An example of a monitor implementing a semaphore:

```
monitor Semaphore {
  int s = 0;
  cond pos;

  procedure Psem() {
    while (s == 0) wait(pos);
    s = s - 1;
  }

  procedure Vsem() {
     s = s + 1;
     signal(pos);
  }
}
```

Slide 13

Message passing

Key abstraction = *communication channel* = defines a path (a link) between tasks

Different styles of message passing:

- Static vs. dynamic specification of the channels
- Synchronous vs. asynchronous:
  - Synchronous = Both processes must be ready to communicate
  - Asynchronous
- Uni-directional or bi-directional:
  - Uni-directional: From the sender to the receiver
  - Bi-directional: Symetrical exchange of information (e.g., RPC)

Slide 14

Key goals when choosing a programming model

1. Expressiveness
2. Performance
3. Portability

But . . . these are generally *conflicting* goals

- Very expressive, high-level model => *mismatch* (large semantic gap) *between programming model and machine model* (better portability, lower performance)
- Low-level programming model => *mismatch between application structure and programming model* (lower portability, improved performance)

Slide 15

3 Evolution of Threaded-C: past . . . and future

3.1 The initial language: Threaded-C (version 1.0)

`Threaded-C` (version 1.0) :

- Not supposed to be used for programming
  (target language for compilers)
- Designed as the *machine language* of the EARTH architecture

=> Low-level of abstraction

Language for high-level programming = `EARTH-C`, but

- Important semantic gap between `EARTH-C` et EARTH
- Lack of control on the granularity of parallelism and on communications (compiler responsibility)

=> `EARTH-C` was abandoned in favor of `Threaded-C`

Slide 16

```
THREADED fib( int n, int *GLOBAL resultat, SPTR termine )
{
    SLOT SYNC_SLOTS[1];
    int r1, r2;

    INIT_SYNC(0, 2, 2, 1);

    if (n <= 1) {
        DATA_RSYNC_L( 1, resultat, termine );
    } else {
        TOKEN( fib, n-1, TO_GLOBAL(&r1), SLOT_ADR(0) );
        TOKEN( fib, n-2, TO_GLOBAL(&r2), SLOT_ADR(0) );
        END_THREAD();

    THREAD_1:
        DATA_RSYNC_L( r1 + r2, resultat, termine );
    }
    END_FUNCTION();
}
```

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

Slide 17

Weaknesses and difficulties of Threaded-C 1.0

Not easy to write/read programs:

- Sync slots and fibers identified with *numbers* ;
- *Non-local* binding between a fiber, a slot and the slot properties ;
- *Non-generic* communication and synchronisation instructions :

```
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
```

- No uniform naming convention (`THREAD`/`END_THREAD`, `_SYNC`) ;
- Redundant `END_THREAD()` instruction ;
- Difficult notion of `GLOBAL` handle ;

Slide 18

4 The revised language : Threaded-C (version 2.0)

```
THREADED fib( int n, int *GLOBAL resultat, SPTR termine )
{
  int r1, r2;

  if (n <= 1) {
    PUT_SYNC( 1, resultat, termine );
    TERMINATE;
  } else {
    TOKEN( fib, n-1, TO_GLOBAL(&r1), TO_SPTR(RETOURNER_RES) )
    TOKEN( fib, n-2, TO_GLOBAL(&r2), TO_SPTR(RETOURNER_RES) )
  }

  FIBER RETOURNER_RES <* 2 *> {
    PUT_SYNC( r1 + r2, resultat, termine );
    TERMINATE;
  }
}
```

Slide 19

Simplification and uniformisation

- Symbolic identifiers ;
- More uniform naming convention for instructions, for ex., `SYNC` suffix => sending of a sync signal ;
- *Implicit* declaration of slots and *local* binding of properties ;
- *Implicit* end of fiber : execution is done using the usual C language semantics, until `FIBER` or `TERMINATE` is encountered ;
- *Generic* synchronisation and data transfer instructions : `PUT_SYNC`, `GET_SYNC`, `BLKMOV_SYNC` et `SYNC`.

The `GLOBAL` handle concept has been preserved because it is fundamental (non-blocking fibers).

Slide 20

Multithreaded programming on *SMP* machines

Other key objective of language revision =

> Ensure correct behavior of programs running on *SMP clusters* :
>
> => *Many* fibers can access the same memory at any given tim
>
> => Need for mutual exclusion mécanismes

Two mécanismes were introduced :

1. `EXCLUSIVE` fiber => within a *thread*, *only one exclusive fiber* can be executing ;
2. Atomic mailboxes = non-deterministic *merge*
   (to combine values coming from differents sources) :
   (a) `INIT_MAILBOX` : allocate and initialize ;
   (b) `DROP_IN` : send (remotely) an item and implicitly send a sync signal ;
   (c) `RETRIEVE_ITEM` : remove (locally) an (arbitrary) item ;

Slide 21

```
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 22

Implementing other synchronization mécanismes

The basic mécanismes can be used to implement other mechanisms :

- Mutual exclusion locks ;
- I-structures ;
- Parallel reduction boxes ;
- Atomic block moves with explicit reset of sync slot ;
- Peers links (unbuffered, bi-directional communication channels) ;

- Sémaphores ;
- (Uni-directional, buffered) Communication channels ;
- (Simplified) Linda-style tuple space ;

Slide 23

Key caractéristique = potentially blocking opérations become *split-phase* :

- Locks

```
LOCK_SYNC( verrou, TO_SPTR(SECTION_CRITIQUE) );

FIBER SECTION_CRITIQUE <* 1 *> {
   ...
   UNLOCK(verrou)
}
```

- Communication channel

```
typedef struct CHAN *GLOBAL CHAN;

void CHAN_INIT_SYNC( CHAN *chan, int max_items, SPTR chan_created );

void CHAN_SEND_SYNC( CHAN chan, ItemChan item, SPTR slot );

void CHAN_SEND_EOC( CHAN chan );

void CHAN_RECEIVE_SYNC( CHAN chan, ItemChan* item, bool* eoc, SPTR slot );
```

Slide 24

Key programming strategy = *active monitor*

Classical monitor

- *Passive* software component that encapsulates some shared resource and that *implicitly* provides mutual exclusion
- Exports a set of operations to manipulate the resource
- A call to such an operation becomes part of the calling thread

Active monitor

- Monitor implemented as an active process
- Uses some form of communication channel to receive requests
- Handles each request one at a time (mutual exclusion) then sends back an appropriate response

Slide 25

```
THREADED semaphore_handler
(
  SEMAPHORE sem_handle,
  int init_value,
  SPTR created
 )
{
  struct SEMAPHORE* sem;
  SPTR              slot_to_sync;
  int               size;

  assert( IS_LOCAL(sem_handle) );
  sem = TO_LOCAL(sem_handle);

  /* Initialize the various fields. */
  sem->value      = init_value;
  sem->nb_waiting = 0;
  sem->post       = TO_SPTR(POST);
  INIT_MAILBOX( &(sem->wait_mailbox), TO_SPTR(WAIT) );
  SYNC(created);
  /* Initialization completed: wait for requests to arrive. */
```

Slide 26

```
  EXCLUSIVE FIBER WAIT <* 1 *> {
    if ( sem->value > 0 ) {
      sem->value--;
      RETRIEVE_ITEM( sem->wait_mailbox, &slot_to_sync );
      SYNC( slot_to_sync );
    } else {
      /* Already 0: leave demander in mailbox. */
      sem->nb_waiting++;
    }
  }

  EXCLUSIVE FIBER POST <* 1 *> {
    sem->value++;
    if ( sem->nb_waiting > 0 ) {
      /* Some other process is waiting for it. */
      RETRIEVE_ITEM( sem->mailbox, &slot_to_sync );
      sem->nb_waiting--;
      sem->value--;
      SYNC( slot_to_sync );
    }
  }
}
```

Slide 27

Limitations and weaknesses of Threaded-C 2.0

What were the key requirements when developing `Threaded-C` 2.0?

- The semantic gap between programming model and EARTH execution model had to be kept *narrow*
- The proposed modifications had to be implemented rapidly (a few months by a very small team)

Thus, this is why `Threaded-C` is still a low-level language

Can the level be raised and yet preserve a *not too wide* semantic gap?

- One example: make the data flow more explicit (instead of implicit through pointer and sync slot manipulation)

  One possible solution (*proof-of-concept prototype* exists): Fiber with entry points and arguments

Slide 28

```
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;
  }
}
```

Slide 29

```
ENTRY_TYPE RecevoirRes( int );

THREADED reduire
  (
   int val_init,
   int (*op)(int, int),
   ENTRY( ENTRY(int, SPTR), ENTRY(ENTRY RecevoirRes) ) init
  )
{
  int val_courante = val_init;

  SEND( init, TO_ENTRY(APPLIQUER_OP), TO_ENTRY(ENVOYER_RES) );

  EXCLUSIVE FIBER APPLIQUER_OP( int val, SPTR fait ) {
    val_courante = (*op)(val_courante, val);
    SYNC( fait );
  }

  FIBER ENVOYER_RES( ENTRY RecevoirRes rcv_resultat ) {
   SEND( rcv_resultat, val_courante );
   TERMINATE;
  }
}
```

Slide 30

Issues/features discussed last year but not implemented (for lack of time or lack of *consensus*)

- Indexed fibers = static set of fibers

```
FIBER FIB[i: 0..N-1] <* 1 *> {
  int x = init[i];
  x +=  a[i];
  printf( "x = %d\n", x );
}
```

- Forall loops
- Fiber scheduling and priorities
- Split-phase function calls

Slide 31

```
THREADED int fib( int n, SPTR done )
{
  int r1, r2;

  if (n <= 1) {
    r1 = 0; r2 = 1;
    SPAWN( READY );
  } else {
    r1 := TOKEN( fib, n-1, TO_SPTR(READY) );
    r2 := TOKEN( fib, n-2, TO_SPTR(READY) );
  }

  FIBER READY <* 2 *> {
    RETURN( r1+r2, done );
  }
}
```

Slide 32

Other issues that could be addressed:

- Memory consistency model?
- Explicitly dealing with the memory hierarchy (percolation)?
- ...
- ...

Key questions:

- How far away can/should the `Threaded-C` programming model be from the EARTH model?
- What are the issues that should be handled/expressed by the programming model (and compiler) vs. by the programmer?
- Have direct control over everything ... except those things that we know compilers are good at?