Slide 1

# THREADED-C

## Where is it coming from?

## Where is it going?

Guy Tremblay
Dépt. d'informatique, UQAM
Montréal, Qué.

CAPSL Seminar
Friday, Feb. 25, 2000

Slide 2

## Outline of presentation

```
EARTH-C                     [High-level]
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
    |
THREADED-C (version 1.1)    [Assembly language]
    |
THREADED-C (version 1.0)    [Machine language]
```



Slide 3

## Outline of presentation

```
EARTH-C                     [High-level w/ wide semantic gap]
    |
    |
    |
    |
    |    X-THREADED-C       [High-level w/ narrow sem. gap?]
    |     /                 [Specialized libraries?]
    |    /
    |   /
THREADED-C (version ?.?)    [Cleaned-up assembly language]
    |                       [Frame-based?]
    |
THREADED-C (version 1.1)    [Assembly language]
    |
THREADED-C (version 1.0)    [Machine language]
```

Slide 4

## 1 EARTH-C VS. THREADED-C

### A bit of history

EARTH-C $\Rightarrow$ For high-level programming of EARTH

= C with high-level parallel constructs:

- Ordinary, parallel, or sequential statements; `forall` loops;
- Pointers always remote $\Rightarrow$ communication = compiler's job)
  (but with `local` hint)
- Annotation to indicate target processor for a procedure activation
  ($\approx$ `INVOKE`)

THREADED-C = direct interface with EARTH machine

- *not* intended for high-level application development
- to be used as EARTH-C compiler target language

Slide 5

### Why EARTH-C did not become *the* EARTH language

- Wide *semantic gap* between EARTH-C and THREADED-C:
  - No direct control of low-level (fiber-level) parallelism
    $\Rightarrow$ various analysis required for thread partitioning
  - No direct control over communication (implicit)
- Compiler long to come out (did it ever?)

**Net effect** All demonstration applications ended-up being developed in THREADED-C

Slide 6

## 2 From THREADED-C (1.0) to THREADED-C (1.1)

**THREADED-C (1.0)**

= Language described in CAPSL Technical Memo 19 (March 1998)

- First explicit and official description of THREADED-C
- In fact: still the current *official* description of THREADED-C ;-(

Some characteristics

- *Machine* language: slot and threads numbers
- Global handles to allow non-MANNA implementation, yet still some dependences on MANNA
  (some example programs might not work for non-MANNA)
- A few other things (see next slide)



Slide 7

**THREADED-C (1.1)**

= Language described in CAPSL Technical Note 05 (October 1998)

**"Annoying" things in** THREADED-C (1.0) **fixed in** THREADED-C (1.1)

- Threads and synchronization slots were identified exclusively by numbers
- Use of `RETURN` for `CALL`ed functions vs. use of `END_FUNCTION` for `INVOKE`d ones
- `END_FUNCTION` had to be the last physical instruction
- `static` variables could not be used in threaded functions
- Local variables could not be initialized in their declaration
- Little error detection regarding use of synchronization slots

Slide 8

**Remaining annoying features of** THREADED-C
**(informal survey with students and post-docs)**

(**) `GLOBAL` handles, which are not regular pointers
(no dereference, no pointer arithmetic)

(*) Difficulty in setting up producer/consumer communication

- Semantics of global variables
- Low-level handling of `SYNC_SLOTS`.
  Sync. slot initialization separate from thread definition
- Non-generic `DATA_SYNC`: `DATA_[R]SYNC_[BSLFDG]`
- No easy way to have multiple copies of a thread working on different data

(**) No debugging support

Slide 9

## 3 From THREADED-C (1.1) to THREADED-C (?.?): Cleaning-up THREADED-C

**Goal** Have a public release (open source style) of EARTH and THREADED-C

**Assumption** THREADED-C = *direct* interface to the EARTH execution model

**Some issues having to be addressed**

**(C.)** Frame-based addressing for synchronization slots and/or data location?

**(D.)** Use of global handles for remote references?

Interference between concurrent fibers within a given processor?

**(A. B.)** Other issues mentioned previously

Slide 10

## Possible language modifications

**A. Simplify the common case**

- fiber activated by a unique sync. slot
- reset count = init count

```
// No need for explicit INIT_SYNC
FIBER_FOO <<3>>:
  ...
END_THREAD();
```



Slide 11

**B. Allow multiple copies of a thread through parameterized symbols**

```
THREAD_INIT:
  for( d = 0; d < N; d++ )
    GotBundle[d] = SLOT_ADR(UNWRAP[d]);
  RSYNC( slotsSetUpDone );
  END_THREAD();


THREAD_UNWRAP[i: 0..N-1]:            // Indexed symbols
  UnwrapBundle( Bundle, i );         // Will RSYNC GotBundle[i]
  SYNC( UNWRAPS_DONE );
  END_THREAD();
```

Slide 12

**C. Frame-based approach for synchronization slots**

**D.** `GLOBAL` **handles for "ordinary" memory references**

- Pointers basic to standard C programming
- "*Clean*" pointers (to dynamic, i.e., `malloc`ed memory) are awkward to deal with using frame-based approach (see below)
- `GLOBAL` annotation associated with distributed memory
  ⇒ common in many high-level programming languages
  ... unless remoteness is completely transparent (virtual shared memory)
- If goal of frame-based approach for data is to facilitate data exchange between fibers, than make it explicit (see below)

Slide 13

**Owner-based handling of dynamic data structures**

```
typedef TreeNode {
    T val;
    TreeNode *GLOBAL left,
             *GLOBAL right;
};
 typedef
   TreeNode *GLOBAL
   Tree;
```

```
THREADED traverseTree( Tree t, SPTR done_slot )
{
  /* ASSERT( OWNER_OF(t) == NODE_ID ) */

  SLOT SYNC_SLOTS[1];
  TreeNode *local_t;

  INIT_SYNC( 0, 2, 2, 1 );
  local_t = TO_LOCAL( t );
  if( local_t == NULL ) {
    SPAWN(DONE);
  } else { // Preorder traversal.
    processVal( local_t->val );
    INVOKE( OWNER_OF(local_t->left), traverseTree,
            local_t->left, SLOT_ADR(DONE) );
    INVOKE( OWNER_OF(local_t->right), traverseTree,
            local_t->right, SLOT_ADR(DONE) );
  }
  END_THREAD();
```

Slide 14

**Frame-based approach to allow for direct access to local data leads to** *content coupling*

**Typical pattern for producer-consumer relationship**

- $F$ invokes $F'$: $F'$ copies addresses of some local var of $F$ into its own local vars
- $F'$ sends back to $F$ the address of some of its own local var, so $F$ can have direct access

Now, they each have access to data location within each other, so they can exchange information

$\Rightarrow$ *content coupling*!

**Level of coupling =** *strength* of the connections between the modules

**The lower the better** Modularity



Slide 15

**Levels of coupling** *from low (good) to high (bad)*

| | | |
|---|---|---|
| :-) | **Data** | Data is non-structured and passed through parameter list |
| :-) | **Structure** | Data is structured and passed through parameter list |
| :-/ | **Control** | One routine passes data to the other that tells it what to do |
| :-\ | **Global** | Two routines make use of the same global data |
| :-( | **Content** | One uses the code inside another or one alters the local data used inside another |

- It is one thing to have the language make it *possible* to introduce bad form of coupling (typical in C with &)
- It is another thing to have the language make it the *typical* way of dealing with a certain class of problems!

Slide 16

## 4 From THREADED-C (?.?) towards X-THREADED-C?

**Major goal of an** X-THREADED-C **=** Improve *programmability*

**Major requirement** Preserve simple mapping to basic THREADED-C

- $\equiv$ keep narrow semantic gap
- $\equiv$ leave some details implicit but with predictable costs

**Support separation of concern** Allow separation of "basic" work from communication setup work

**Support modularity** Make it possible to avoid bad coupling

Slide 17

**Some possible ideas (see examples below)**

- Data flow style programming based on data dependencies?
- Special constructs for setting-up typical communication patterns?
  - through some form of non-strict parameterization?
  - through library of skeletons?
- Message-passing between fibers to prevent content coupling?

Slide 18

**Other possible extensions or modifications**

- Compiler handling of `GLOBAL` handles (e.g., local <−> global conversions)
- User-control over `TOKEN` migration
- Broadcasting
- Non-deterministic thread/synchronization slot
- Parallel I/O
- OO programming THREADED-C ⇒ THREADED-C++
- Dynamic allocation, initialization and binding of synchronization slots



Slide 19

**Data flow style programming**

Let the compiler handle most of the work related with the explicit creation and initialization of sync. slots

```
THREADED fib( int n, int GLOBAL* ??result )
{
   int r1, r2;

   if ( n < 2 ) {
      r1 = 1; SYNC(!!r1);
      r2 = 0; SYNC(!!r2);
   } else {
      TOKEN( fib, n-1, TO_GLOBAL(&!!r1) );
      TOKEN( fib, n-2, TO_GLOBAL(&!!r2) );
   }
   END_THREAD();

 THREAD <<??r1, ??r2>>:
   DATA_SYNC_L( r1 + r2, result, !!result );
   RETURN();
}
```

```
THREADED MAIN( int argc, char *argv[] )
{
   int n, res;

   n = atoi(argv[1]);
   TOKEN( fib, n, TO_GLOBAL(&!!res) );
   END_THREAD();

 THREAD <<??res>>:
   printf( "fib(%d) = $d\n", n, res );
   RETURN();
}
```

Slide 20

**Special parameterisation for communication set-up**

Allow a threaded function to be parameterised relative to the identities of others fibers with which it will synchronize

```
THREADED MAIN( int argc, char *argv[] )
{
   Item buffer;

   BEGIN_SETUP(the_producer, the_consumer);
     !!the_producer <== TOKEN( producer, TO_GLOBAL(&!!buffer) )
                              [the_consumer::GET_ITEM];
     !!the_consumer <== TOKEN( consumer, TO_GLOBAL(&!!buffer) )
                              [the_producer::PRODUCE_ITEM];
   END_SETUP();
   END_THREAD();

  THREAD <<??the_producer, ??the_consumer, 2*??buffer>>:
    RSPAWN( the_producer::PRODUCE_ITEM );
    END_THREAD();
}
```

Slide 21

```
THREADED producer(Item GLOBAL* ??buffer_adr)[the_consumer]
{
   Item item;
   Item GLOBAL* item_buffer;

  THREAD_INIT <<??the_consumer>>:
   DATA_RSYNC_G( buffer_adr, TO_GLOBAL(&item_buffer), !!buffer_adr );
   END_THREAD();

 PUBLIC THREAD_PRODUCE_ITEM:
   item = produce_item();
   BLKMOV_RSYNC( TO_GLOBAL(&item), item_buffer, sizeof(Item), the_consumer );
   END_THREAD();
}
```

Slide 22

**Another example: a pipeline of processes**

```
   BEGIN_SETUP( ps[i: 0..N-1] )
     i = 0 =>
       ps[0] <== TOKEN( proc, ... )
                 [ ps[N-1]::ACK, ps[1]::ITEM_READY ];
     0 < i < N-1 =>
       ps[i] <== TOKEN( proc, ... )
                 [ ps[i-1]::ACK, ps[i+1]::ITEM_READY ];
     i = N-1 =>
       ps[N] <== TOKEN( proc, ... )
                 [ ps[N-2]::ACK, ps[0]::ITEM_READY ];
   END_SETUP();
```



Slide 23

**Parameterised fibers to prevent content coupling**

```
THREADED proc( ... )
{
  // Declarations of variables;
  ...

  // Init. statements in fiber 0
  ...
  END_THREAD()

  PUBLIC
  THREAD_HANDLE_INFO( T1 a1, T2 a2 ):
  // Spawned only when
  // a1 and a2 are available.
     ...
  END_THREAD();
}
```

```
THREADED other_function( ... )
{
   ...
   proc_tok <== TOKEN( proc, ... );

   ...

   v1 = ...;
   v2 = ...;
   RSEND( proc_tok::HANDLE_INFO, v1, v2 );
   ...
}
```

Slide 24

## 5 Conclusion and future work

Personal (honest) opinion on current style THREADED-C:

- Has shown EARTH architectural concept is interesting and viable
- Has not shown such a machine is easily programmable: very difficult programs to read!

Definitively requires higher-level abstractions

- Although it comes from data flow, THREADED-C currently looks like a low-level control-oriented machine language
- Mix up some shared memory and message-passing elements, but in an unclear way (`BLMKVOV_RSYNC` for locality management)

**Slide 25**

Future work

- Look at other existing, higher-level languages
- Decide on the level of language research "*investment*"
  - Develop new X-THREADED-C language?
  - Develop compiler for existing language targeted to THREADED-C?
- Clarify various issues: frame-based? global handles? concurrent threads?