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

Guy Tremblay

Dépt. d'informatique, Univ. du Québec à Montréal
C.P. 8888, Succ. Centre-Ville
Montréal, Qué., Canada
H3C 3P8

tremblay.guy@uqam.ca

RÉSUMÉ. *Cet article présente le langage Threaded-C, langage conçu pour la programmation de machines parallèles multi-contextes, plus précisément, de machines fondées sur l'architecture* EARTH. *Cette architecture, grâce à une hiérarchie à deux niveaux* threads*/fibres, supporte de façon efficace le parallélisme de granularité fine. Le langage Threaded-C, initialement conçu comme le « langage machine » de cette architecture, a évolué pour devenir le principal langage de programmation de cette architecture. C'est cette évolution que nous décrivons dans cet article, de même que les nouveaux éléments du langage (mécanismes d'exclusion et d'atomicité) introduits récemment dans le but de supporter la programmation de grappes de multiprocesseurs (*SMP clusters*). Nous décrivons aussi certaines stratégies de programmation utilisées pour la définition et la mise en œuvre de divers mécanismes de synchronisation, de même que des extensions du langage en cours de développement.*

ABSTRACT. *This paper presents the Threaded-C programming language, language initially designed for programming multi-threaded machines, more precisely, machines based on the* EARTH *architecture. This architecture, because of its two-level hierarchy of threads and fibers, efficiently supports fine-grain parallelism. Originally conceived as the "machine language" for the* EARTH *architecture, Threaded-C has now evolved to become the major programming language for this architecture. It is this evolution that is described in this paper, along with new features of the language (atomicity and mutual exclusion mechanisms) recently introduced in order to support programming on SMP parallel machines. Key programming strategies used to define and implement various synchronisation mechanisms are also presented, as well as additional language extensions currently under development.*

MOTS-CLÉS : *Modèle et langage de programmation parallèle, Architectures et programmation multi-contextes.*

KEYWORDS: *Parallel programming model and language, Multi-threaded architecture and programming.*

1. Introduction

Cet article décrit un langage, appelé Threaded-C, conçu pour la programmation d'architectures parallèles multi-contextes. Plus spécifiquement, Threaded-C a été développé pour la programmation de machines parallèles multi-contextes basées sur l'architecture EARTH [HUM 95]. Cette architecture a été conçue dans le but d'exécuter de façon efficace des programmes parallèles composés d'un grand nombre de tâches de fine granularité et telles que la structure du parallélisme ne soit pas nécessairement régulière. Un autre objectif était d'assurer que les machines construites sur la base de cette architecture puissent être réalisées à l'aide de processeurs classiques (*off-the-shelf computers*) [HUM 94]. De plus, les machines résultantes devaient aussi être « *scalable* ». Des machines de ce type possèdent généralement des mémoires distribuées et les communications entre les processeurs ont un impact majeur sur les performances. Il est donc important, tant pour les programmeurs que pour les concepteurs de compilateurs, d'avoir un contrôle sur les communications ainsi que sur la décomposition des programmes en fils d'exécution indépendants, la présence de fils multiples d'exécution et des changements rapides de contexte permettant de dissimuler le temps de latence des communications [HUM 95]. C'est donc dans le but de permettre l'expression de ces différents aspects que le langage Threaded-C a été conçu. En d'autres mots, comme nous le verrons à la section 3, Threaded-C est un langage définissant un modèle de programmation parallèle où les divers aspects de la programmation d'une application parallèle multi-contextes doivent être décrits explicitement [SKI 98].

Bien que Threaded-C définisse un modèle de programmation d'assez bas niveau, il est quand même possible d'améliorer le niveau d'abstraction du langage tout en restant *fidèle* — c'est-à-dire sans introduire un fossé sémantique majeur — au modèle de programmation requis pour l'architecture EARTH. C'est ce que nous allons essayer d'illustrer dans les pages qui suivent.

Cet article est structuré comme suit. La section 2 présente l'architecture EARTH et son modèle d'exécution. Ensuite, la section 3 décrit brièvement différents modèles de programmation parallèle et situe Threaded-C par rapport à ces langages et modèles. La section suivante présente la version initiale du langage Threaded-C, exposant certaines de ses faiblesses, et est suivie d'une section présentant la version révisée du langage. La section 6 présente ensuite certaines stratégies de programmation utiles pour la mise en œuvre de divers mécanismes de synchronisation, par exemple, sémaphores, I-structures, etc. Finalement, la dernière section, avant la conclusion, introduit diverses pistes pour l'évolution future du langage.

2. L'architecture EARTH et son modèle d'exécution

L'architecture EARTH (*Efficient Architecture for Running THreads*) trouve son origine dans les architectures à flux de données [ARV 86, LEE 94]. Dans ce type d'architectures parallèles à granularité fine, un programme est représenté par un graphe où chaque nœud correspond à une instruction et où les arcs représentent les données

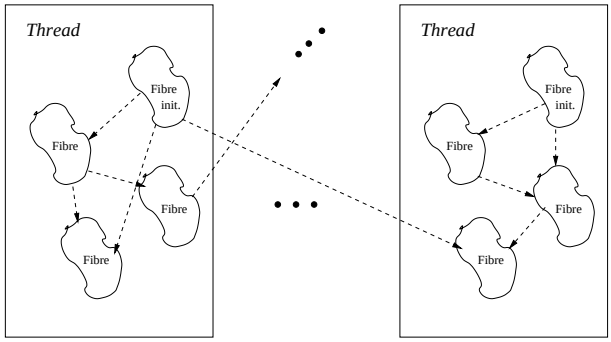

Figure 1. *Modèle d'exécution* EARTH *avec* threads *et fibres*

échangées entre les instructions. Du point de vue du parallélisme, l'intérêt d'une telle représentation est que seul un ordre *partiel* d'exécution est spécifié, ordre partiel *implicitement parallèle*. Le modèle d'exécution de l'architecture EARTH repose aussi sur la représentation d'un programme par un graphe, à la différence que les arcs représentent simplement des signaux de synchronisation plutôt que des transferts explicites de données (*argument-fetching principle* [GAO 89]) et, surtout, que les nœuds du graphe correspondent à des *groupes* d'instructions. De tels groupes d'instructions, qui s'exécutent selon la sémantique séquentielle usuelle, sont appelés des *fibres*.

2.1. *Fibres et* threads

La notion de fibre est différente de la notion de *thread* qu'on retrouve en Java [LEA 00] ou en *Pthreads* [BUT 97]. Une fibre, tout comme un *thread*, représente un fil d'exécution indépendant et de poids léger. Par contre, alors qu'un *thread* correspond à une activation de fonction/procédure, une fibre est simplement une séquence d'instructions *à l'intérieur* d'un *thread*. Comme les différentes fibres d'un *thread* partagent le même *contexte*, à savoir le bloc d'activation de la fonction/procédure (variables locales et arguments), l'amorce d'une nouvelle fibre peut donc se faire de façon rapide et peu coûteuse une fois le *thread* créé.

Le modèle d'exécution d'un programme multi-contextes EARTH repose donc sur une hiérarchie à deux niveaux (figure 1) : i) les *threads*, créés par l'activation parallèle de procédures ; ii) les fibres, qui s'exécutent dans le contexte d'un *thread*. Sauf pour la fibre dite d'initialisation (code au début de la procédure) qui est exécutée aussitôt que le *thread* est créé et amorce son exécution, l'exécution des fibres est contrôlée par l'envoi de signaux de synchronisation (lignes pointillées sur la figure 1). Alors que les *threads* sont des entités parallèles de granularité moyenne — par opposition, par exemple, à des processus Unix de plus forte granularité —, les fibres sont des entités de granularité fine.

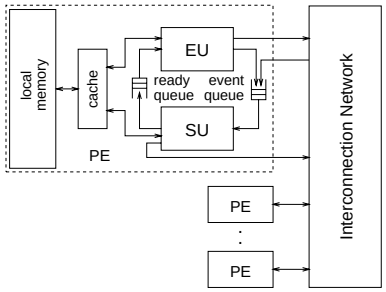

Figure 2. *Structure générale d'une machine* EARTH *(figure tirée de [THE 98])*

Dans le modèle EARTH, les fibres possèdent les propriétés suivantes :

– L'ordonnancement des fibres se fait de façon dynamique, essentiellement en fonction des dépendances de données (style *dynamic dataflow* [ARV 86]) : une fibre devient *prête* à s'exécuter lorsqu'elle a reçu tous les signaux de synchronisation appropriés.

– Les instructions à l'intérieur d'une fibre sont exécutées séquentiellement, selon la sémantique usuelle.

– Une fibre s'exécute de façon *non préemptive*. En d'autres mots, une fibre n'est jamais interrompue et *ne doit jamais bloquer*.

Cette dernière caractéristique, comme on le verra à la section 4, a un impact majeur sur le style de programmation (approche « *split-phase* »).

L'approche avec *threads* et fibres peut être vue comme un compromis entre le *dataflow* pur (nombre illimité de contextes de granularité très fine) et les architectures classiques (nombre limité de contextes poids lourd). Dans une machine *dataflow*, *chaque* instruction est un fil d'exécution indépendant. Un grand nombre de tâches s'exécutant en parallèle est donc généré, ce qui entraîne toutefois des coûts élevés de synchronisation. Le modèle EARTH permet d'obtenir des fils d'exécution de granularités diverses : une fibre peut regrouper des séquences (simple ou complexes) d'instructions ; un *thread* peut être composé d'une ou plusieurs fibres indépendantes (qui partagent le même contexte, le même bloc d'activation) ; un programme peut être formé de plusieurs *threads*.

La figure 2 présente la structure générale d'une machine EARTH. La propriété des fibres d'être non préemptive joue un rôle important car elle implique que les instructions d'un *thread* peuvent être exécutées par un pipeline normal d'exécution, représenté par l'unité *EU* (*Execution Unit*) dans la figure 2. Les tâches de synchronisation (traitement des signaux de synchronisation, ajout des fibres dans la file d'exécution (*ready queue*), etc.) sont reléguées à une unité indépendante, l'*unité de synchronisation* (*SU*, pour *Synchronisation Unit*). Une machine EARTH peut donc être construite à partir de processeurs ordinaires [HUM 94], en émulant l'unité de synchronisation en

logiciel ; des mises en œuvre existent sur des machines Manna, IBM SP-2, Beowulf, et des réseaux de stations Sun.

2.2. *Modèle de mémoire*

Une autre caractéristique importante de l'architecture `EARTH` est son modèle de mémoire. Le modèle `EARTH` suppose la présence d'un espace global *d'adressage*, de même qu'un modèle de mémoire physiquement distribué (*NUMA*). Cette caractéristique est importante puisqu'une fibre ne doit pas bloquer lorsqu'une opération avec un long temps de latence est effectuée. Une notion de pointeur *global* doit donc être supportée par le langage. De tels pointeurs permettent de référer à des adresses situées dans n'importe quel module mémoire. Toutefois, l'accès au contenu d'un pointeur ne peut s'effectuer qu'à l'aide d'opérations spéciales de communication — par exemple, `PUT_SYNC` ou `GET_SYNC` — de façon à rendre explicites ces communications avec un long temps de latence et, donc, à décomposer ces accès en deux phases distinctes associées à des fibres différentes : la requête, dans une première fibre, suivie de la réception dans une autre fibre (approche *split-phase*).

Les sections 4 et 5 décrivent de façon plus précise les principales caractéristiques du langage utilisé pour programmer la machine `EARTH`, à savoir le langage Threaded-C. Auparavant, nous allons toutefois tenter de mieux situer Threaded-C parmi les divers langages et modèles de programmation parallèle existants.

3. Modèles de programmation parallèle

3.1. *Classification des modèles de programmation*

Un premier critère souvent utilisé pour caractériser les modèles de programmation parallèle est la façon dont les tâches communiquent entre elles, c'est-à-dire soit par l'intermédiaire de variables partagées, soit par l'échange de messages [FOS 95, AND 00]. Dans le premier cas, différents mécanismes d'exclusion mutuelle doivent être introduits (verrous, sémaphores, moniteurs, etc.), alors que le deuxième cas demande l'introduction de mécanismes de communication (synchrone/asynchrone, uni-/bi-directionnelle, définition statique/dynamique des canaux, etc.) [AND 00]. On peut aussi distinguer entre les modèles basés sur le parallélisme de données (le parallélisme est obtenu en exécutant une même opération qui s'applique à plusieurs données distinctes) par opposition au parallélisme de tâches (le parallélisme est obtenu en générant plusieurs tâches indépendantes travaillant sur des données distinctes) [QUI 90, AND 00].

Une autre classification des différents modèles de programmation parallèle est celle présentée par Skillicorn et Talia [SKI 98]. Ces deux auteurs définissent un modèle de programmation comme « une interface qui sépare les propriétés de haut niveau de celles de bas niveau ». Plus concrètement, un modèle de programmation définit

« une machine abstraite fournissant certaines opérations au programmeur et nécessitant la mise en œuvre de ces opérations sur les architectures sous-jacentes ». De tels modèles visent donc à fournir des abstractions facilitant le travail des programmeurs, tout en assurant une certaine indépendance, et stabilité, face aux machines elles-mêmes.

Skillicorn et Talia définissent leurs différentes catégories de modèles de programmation parallèle en se référant aux divers aspects qui doivent être traités lors de l'écriture d'un programme parallèle :

– Décomposition du programme en *threads* ;

– Association (*mapping*) *threads*/processeurs ;

– Organisation des communications entre les *threads* ;

– Synchronisation entre les *threads*.

Leurs différentes catégories sont alors définies en fonction de ce qui, dans un modèle, doit être exprimé explicitement ou non. Au niveau le plus abstrait, on trouve des modèles où rien n'est explicite, c'est-à-dire même le parallélisme est implicite — par exemple, certains langages fonctionnels. À l'extrême opposé, on a des modèles où tout est explicite, donc où les programmeurs doivent spécifier tous les détails de décomposition, communication, etc. Entre ces deux extrêmes, on trouve des modèles intermédiaires où seuls certains éléments sont explicites, par exemple décomposition et association explicites, mais traitement implicite des communications et synchronisations.

À l'intérieur de ces diverses catégories, Skillicorn et Talia différencient aussi les différents modèles selon le degré de contrôle sur les *threads* et les communications : structure dynamique par opposition à statique des *threads*, organisation limitée ou non des communications. Palagatti introduit lui aussi une dimension semblable, à savoir la présence ou non de restrictions sur la structure des activités parallèles [PAL 98] : les modèles les plus généraux n'imposent aucune restriction sur la structure des activités parallèles (graphe de calcul arbitraire), alors que d'autres modèles, généralement pour des raisons d'efficacité de la mise en œuvre (par le compilateur), imposent diverses restrictions sur la structure des activités (par exemple, structure statique reflétant la topologie de la machine, approche par gabarits ou squelettes, etc.).

Finalement, on peut aussi distinguer, bien que cela puisse être considéré comme orthogonal à la question du *modèle* de programmation lui-même, entre les *langages* de programmation et les *interfaces* de programmation, une interface de programmation pouvant généralement être mise en œuvre dans plusieurs langages — par exemple, des *bindings* de MPI [PAC 97] sont disponibles pour Fortran, C, Java, etc.

3.2. *Mise en perspective du modèle de programmation de Threaded-C*

Le modèle de programmation de Threaded-C a évolué de façon étroitement liée au modèle d'exécution de l'architecture EARTH. Ces deux modèles ont pour héritage le

modèle *dataflow*, plus précisément le modèle *argument-fetching dataflow* [GAO 89]. Dans ce modèle, signaux de contrôle et transferts de données sont *découplés*. Ainsi, bien qu'une instruction soit activée par la réception de *signaux* de synchronisation et que ces signaux soient généralement (mais pas nécessairement) associés à des dépendances de données, il n'y a aucune notion de flux explicite de données ou de jetons comme dans les architectures *dataflow* classiques [DEN 85, ARV 86]. Cette évolution du modèle *dataflow* a conduit à un rapprochement vers le modèle von Neumann classique, donc à un modèle hybride d'exécution. Cette hybridité se reflète aussi dans le langage Threaded-C, un langage impératif basé sur C mais qui incorpore de nombreux concepts du modèle *dataflow*.

Threaded-C est un *langage* de programmation qui se situe, selon la classification de Skillicorn et Talia, dans la catégorie des langages de bas niveau d'abstraction où tous les détails de décomposition, association, communication et synchronisation doivent être exprimés de façon explicite. Dans sa nouvelle version, Threaded-C supporte autant la programmation par variables partagées (pour les fibres et *threads* s'exécutant sur un même noeud) que celle basée sur l'échange de messages (par la manipulation asynchrone et *split-phase* de pointeurs globaux ou par l'utilisation de boîtes aux lettres). De plus, aucune restriction n'est imposée sur la structure du calcul, des liens arbitraires de communication pouvant être créés entre les divers *threads*. Threaded-C, dont le parallélisme en est un de tâches plutôt que de données, supporte une hiérarchie à deux niveaux (*threads* et fibres) de tâches parallèles, ce qui permet un contrôle fin de la granularité du parallélisme. La principale propriété des fibres `EARTH`, à savoir exécution non bloquante, propriété nécessaire pour la mise en œuvre du modèle `EARTH` sur des processeurs standards, se réflète aussi au niveau du langage Threaded-C, conduisant à une approche *split-phase* des opérations bloquantes (section 6).

Threaded-C se distingue d'autres langages pour la programmation parallèle multi-contextes supportant le parallélisme de tâches et à granularité fine d'abord et avant tout par sa hiérarchie *threads*/fibres. Contrairement à Cilk [FRI 98], Threaded-C n'est pas limité au parallélisme récursif de style *diviser-pour-régner* : en Threaded-C, il est possible d'établir des liens bi-directionnels entre *threads*, de réaliser des pipelines logiciels, etc. De plus, Cilk supporte uniquement le modèle de programmation par variables partagées. Split-C [CUL 93] et UPC [CAR 99] sont deux autres langages basés sur C reposant sur une approche *split-phase* des opérations de communication — par exemple, les instructions *split-phase* d'affectation de Split-C permettent de superposer communications et calculs, ce qui permet, comme en Threaded-C, de dissimuler la latence des accès non locaux. Toutefois, en Split-C et UPC, la distinction *thread*/fibre n'est pas explicite, non plus que les communications associées à certaines communications non locales. Le modèle *Pthreads* POSIX [BUT 97], une bibliothèque de *threads* pour C/Unix, quant à lui, ne supporte ni la hiérarchie *threads*/fibres, ni le comportement *split-phase*. Là aussi, le modèle de programmation supporté est celui par variables partagées.

Un autre langage dont les racines proviennent elles aussi, comme Threaded-C, du modèle *dataflow* est le langage pH [NIK 01]. Toutefois, contrairement à Threaded-C, pH est un langage fonctionnel et non un langage impératif. Plus précisément, pH est une version parallèle et non paresseuse du langage Haskell avec un certain nombre d'extensions non fonctionnelles (I-structures [ARV 89] et M-structures [BAR 91]). Selon la classification de Skillicorn et Talia, pH est un langage où seul le parallélisme est explicite, donc à un plus haut niveau d'abstraction que Threaded-C — le programmeur n'a donc aucun contrôle sur les communications et synchronisations ni sur la décomposition en *threads* et fibres. Par contre, en pH contrairement à Haskell, certaines constructions du langage permettent d'exprimer plus explicitement le parallélisme — boucles, I- et M-structures — alors que la sémantique indulgente du langage (non stricte mais non paresseuse) permet de générer un plus grand nombre de tâches parallèles [TRE 01a].

MPI (*Message Passing Interface*) est une *interface* standard pour la programmation parallèle supportant strictement le modèle de programmation par échange de messages [PAC 97]. Au cours des dernières années, MPI est devenu de plus en plus utilisé dans l'industrie et supporté sur de nombreuses machines, ce qui permet d'écrire des programmes parallèles généralement portables sur diverses architectures. Dans leur classification, Skillicorn et Talia associent MPI à la classe des modèles « tout explicite », donc à faible niveau d'abstraction. Contrairement à Threaded-C, toutefois, MPI n'offre aucun support pour la programmation par variables partagées. De plus, MPI est surtout utilisé pour le parallélisme de forte granularité. En fait, le mécanisme utilisé pour identifier les tâches n'est pas défini dans le standard (MPI version 1.0) et ces tâches — de granularité plus forte qu'un simple *thread* — sont générées de façon statique, au début de l'exécution du programme. Par contre, MPI fournit une vaste collection d'opérations de communication globale, e.g., `broadcast`, `scatter`, `gather`, etc. De telles opérations ne font pas partie du langage Threaded-C mais pourraient être définies sous forme de bibliothèques (voir section 6).

Un langage d'un tout autre ordre associé initialement à l'architecture `EARTH` est EARTH-C [HEN 97, TAN 97]. Ce langage supporte un modèle de mémoire partagée et un plus haut niveau d'abstraction : c'est le compilateur qui est responsable du parallélisme de fine granularité — décomposition en fibres, communications de valeurs simples entre ces fibres, etc. — alors que le programmeur s'occupe de spécifier les tâches de plus forte granularité — instructions séquentielles ou parallèles, boucles `forall`, distribution et communication d'aggrégats, etc. EARTH-C fut initialement conçu pour le développement d'applications sur l'architecture `EARTH`, son compilateur devant générer du code Threaded-C. Toutefois, pour diverses raisons — entre autres, écart sémantique important entre EARTH-C et `EARTH`, absence de contrôle sur la granularité des fibres et sur les communications, coûts élevés des analyses requises pour l'identification et synchronisation des fibres (*alias analysis*) —, EARTH-C fut peu à peu abandonné au profit de Threaded-C. Mentionnons toutefois que des travaux récents ont montré que EARTH-C pouvait malgré tout être compilé de façon efficace sur l'architecture `EARTH`, et ce de façon à traiter différentes classes d'application avec parallélisme irrégulier [ZOP 01].

```
 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 3. *La procédure récursive* `fib` *en Threaded-C (version 1.0)*

Après ce bref tour d'horizon de quelques modèles et langages de programmation parallèle, nous allons maintenant présenter de façon plus détaillée le langage Threaded-C. Comme on le verra, il est clair qu'il s'agit, au sens des modèles de Skillicorn et Talia, d'un langage avec un niveau d'abstraction relativement faible. Toutefois, nous allons montrer que le niveau d'abstraction, même en restant de style « tout explicite », peut quand même être amélioré de façon à rendre le langage plus simple à utiliser. De plus, nous allons montrer que le langage peut aussi être amélioré pour le rendre indépendant de certains détails de mise en œuvre de l'architecture `EARTH`, permettant entre autres la mise en œuvre de cette architecture sur des grappes de multiprocesseurs.

4. Le langage initial : Threaded-C (version 1.0)

Le langage Threaded-C a été initialement conçu comme *le langage machine* de l'architecture `EARTH`. En fait, tel qu'indiqué à la section précédente, Threaded-C n'était pas supposé être utilisé pour la programmation mais devait plutôt servir de langage cible à des compilateurs. Toutefois, pour les raisons mentionnées plus haut, Threaded-C devint rapidement le principal langage de programmation des machines `EARTH`.

Threaded-C ayant été conçu comme un langage machine, la première mouture de ce langage comportait donc un niveau d'abstraction assez faible. La figure 3 présente le code Threaded-C pour une procédure récursive `fib` : cette procédure est `THREADED`, ce qui signifie qu'elle peut être activée en parallèle *et* peut contenir des fibres. Elle reçoit trois arguments, dont un pointeur vers le résultat (`resultat`) — Threaded-C ne

permet que de définir des *procédures* parallèles, donc tout résultat doit être transmis par l'intermédiaire d'un argument passé par référence — et un pointeur (`termine`) vers une fente de synchronisation (*SPTR*) où sera transmis le signal indiquant que le résultat est disponible.

Ici, les activations de la procédure `fib` seront exécutées sur différents processeurs : l'instruction `TOKEN` (lignes 11 et 12) crée un nouveau *thread* associé à l'activation de `fib`, activation s'exécutant sur un processeur choisi par le système d'exécution — il est aussi possible de spécifier explicitement le processeur où une procédure doit s'exécuter, en utilisant l'instruction `INVOKE(` *num_processeur, procedure, ...*`)`. Cette distribution des activations de `fib` sur les divers processeurs a pour conséquence que le pointeur vers le résultat doit être un pointeur `GLOBAL` (obtenu, lignes 11 et 12, avec `TO_GLOBAL`), puisqu'il peut référer à un espace mémoire d'un autre processeur.

Pour bien comprendre cet exemple, soulignons quelques autres points :

– Une fibre (sauf la fibre d'initialisation, c'est-à-dire la suite d'instructions au début de la procédure) est identifiée par une étiquette telle que `THREAD_1`.[1] Une fibre doit être associée à une fente de synchronisation (*sync slot*). L'état d'une telle fente indique le nombre de signaux à recevoir avant que la fibre soit prête à s'exécuter et, donc, qu'elle soit mise dans la file d'exécution (*ready queue*, figure 2). L'association entre une fibre et une fente de synchronisation, ainsi que la spécification du nombre de signaux à recevoir, se fait à l'aide de l'instruction `INIT_SYNC`. Dans notre exemple, (ligne 6), la fente `0` est associée à la fibre `1` et deux signaux (le premier `2`) doivent être reçus avant que la fibre soit activée la première fois. Le deuxième `2` (appelé *reset count*) indique le nombre de signaux requis pour activer les instances *subséquentes* de la fibre : dans le modèle `EARTH`, une fibre donnée peut être activée à plusieurs reprises, et ce dans le même contexte d'exécution (*thread*) — on verra un exemple à la section 5.

– L'exécution d'une fibre se termine lorsque l'instruction `END_THREAD()` est rencontrée ou lorsque la procédure se termine à l'aide de `END_FUNCTION()`.

– Les fentes de synchronisation doivent être déclarées explicitement, par exemple, `SLOT SYNC_SLOTS[N] ;`, `N` indiquant le nombre total de fentes requises. En règle générale, le nombre de fentes est le même que le nombre de fibres (association statique entre fibres et fentes). Toutefois, il est aussi possible que plusieurs fentes réfèrent à une même fibre ou encore qu'une même fente soit utilisée pour plusieurs fibres distinctes (association dynamique). Notons qu'à partir d'un *numéro local* de fente, l'opérateur `SLOT_ADR` (lignes 11 et 12) retourne un pointeur global (`SPTR` = *Slot Pointer*) pouvant être utilisé à partir d'un autre *thread* (lignes 9 et 16).

– Une valeur peut être transmise à une adresse globale uniquement à l'aide d'instructions spéciales de communication. Ici (lignes 9 et 16), `DATA_RSYNC_L` transmet la valeur indiquée dans l'adresse (globale) spécifiée par `resultat` et signale ensuite la

1. Initialement, dans la première version du langage Threaded-C, le terme « *thread* » était utilisé pour indiquer une fibre. La distinction entre *thread* et fibre a été introduite parce que plus conforme à la façon dont le terme « *thread* » est généralement utilisé dans la littérature.

fente de synchronisation `termine` (reçue en argument). Ainsi, lorsque le parent (appelant) aura reçu les deux signaux provenant de ses enfants (appelés), sa fibre 1 (ligne 15) sera alors activée.

4.1. *Faiblesses et difficultés du langage*

Plusieurs applications ont été développées avec cette première version du langage, et ce dans de nombreux domaines : calcul de grilles d'éléments finis [HEB 98], finance computationnelle [THU 99], transformation discrète 2D d'ondelettes [KHO 99], tranformation de Fourier [THU 00], solution de systèmes d'équations linéaires (méthode du *conjugate gradient*) [THE 00], comparaison de génomes [MAR 01].

Toutefois, l'expérience a montré que le langage n'était pas d'utilisation facile, ce qui fut amplement confirmé lors d'entrevues effectuées auprès d'utilisateurs du langage [TRE 00a] :

– Les instructions de communication et de synchronisation n'étaient pas génériques. Ces instructions se divisaient en quatre classes (`DATA`, `GET`, `BLKMOV` et synchronisation pure) et, dans chaque classe, des variantes avec des noms distincts devaient être utilisées. Ainsi, 12 variantes étaient disponibles pour transférer une valeur scalaire, selon que la fente à synchroniser était locale (`SYNC`) ou globale (`RSYNC` pour R*emote*) et selon le type de la valeur à transmettre (par exemple, `DATA_SYNC_X` avec `X` = `B`, `S`, `L`, `F`, `D`, `G`, pour `byte`, `short`, etc.). Au total, on avait donc 28 opérations différentes.

– Les fentes de synchronisation et fibres devaient être déclarées et manipulées à l'aide de numéros plutôt qu'à l'aide de symboles et l'initialisation d'une fente et son association à une fibre devait se faire au début de la procédure, donc loin de la fibre. De plus, le cas le plus courant (association statique entre fente et fibre, où chaque fibre est associée à une unique fente et vice-versa) ne recevait aucun support spécial, ce qui alourdissait les programmes. Par exemple, une procédure complexe contenant une dizaine de fibres devait alors débuter par une longue suite d'instructions comme suit :

```
INIT_SYNC( 0, i0, r0, 1 );
INIT_SYNC( 1, i1, r1, 2 );
...
INIT_SYNC( 9, i9, r9, 10 );
```

– Bien que l'instruction `END_THREAD()` était généralement redondante, puisque la fin d'une fibre correspondait habituellement au début de la fibre suivante, elle devait être indiquée explicitement, son omission créant un résultat semblable à celui associé, en C, à un `break` omis dans un `switch` (*fall through*).

– La mise en œuvre initiale du compilateur Threaded-C (version 1.0), mise en œuvre réalisée à l'aide d'un pré-processeur, avait introduit des restrictions qui, en termes conceptuels, s'expliquaient difficilement. Ainsi, seule une procédure `THREADED` pouvait contenir des instructions de synchronisation et de communication, rendant ainsi difficile, dans certains cas, l'introduction d'abstractions procédurales. Cette limitation pouvait être contournée en définissant une procédure `THREADED` qui était ensuite

appelée à l'aide d'une instruction `CALL` plutôt qu'à l'aide d'une instruction `INVOKE` ou `TOKEN`. Toutefois, la sémantique de l'instruction `CALL` était incorrectement décrite comme une instruction d'appel séquentiel, ce qu'elle n'était pas vraiment.

– La notion de pointeur global était considérée difficile à maîtriser. Plusieurs utilisateurs auraient souhaité pouvoir manipuler un pointeur global comme n'importe quel autre pointeur, par exemple, permettre l'arithmétique de pointeurs, pouvoir déréférencer un pointeur global sans utiliser d'opérations spéciales (communications implicites).

Dans le but de corriger un certain nombre de ces problèmes, une nouvelle version du langage Threaded-C a donc été développée [TRE 00b], version que nous décrivons dans la prochaine section.

5. Le langage révisé : Threaded-C (version 2.0)

Un objectif important du développement de la nouvelle version du langage Threaded-C était de simplifier et améliorer le langage. Un autre objectif tout aussi important était d'assurer le fonctionnement correct des programmes Threaded-C s'exécutant sur diverses mises en œuvre de l'architecture `EARTH`, y compris des grappes de multiprocesseurs où plusieurs processeurs peuvent partager une même mémoire. Toutefois, ces modifications ainsi que leur mise en œuvre devaient se faire en respectant certaines contraintes importantes :

– Des travaux portant sur la compilation de langages de haut niveau ciblés vers l'architecture `EARTH` ont été faits et continuent à se faire, entre autres dans le cadre du langage EARTH-C [TAN 97, GAO 97, ZOP 01] et du langage pH [MSH 00]. Toutefois, l'objectif premier du projet `EARTH` est d'abord et avant tout celui de développer une nouvelle architecture parallèle multi-contextes et, par la même occasion, de définir un modèle de programmation de bas niveau, une forme de « langage machine évolué » qui soit approprié pour ce nouveau type d'architecture. L'écart sémantique (*semantic gap*) entre le langage révisé et l'architecture devait donc rester suffisamment étroit pour que les coûts associés à une instruction ou opération puissent être facilement *prévisibles*, tant par le programmeur que par le concepteur d'un compilateur pour un langage de haut niveau ciblé vers l'architecture `EARTH`. En d'autres mots, la nouvelle version de Threaded-C devait continuer de permettre un contrôle direct des ressources de la machine, plus précisément, permettre de contrôler tant la décomposition en *threads* et fibres que les communications et les synchronisations, mais si possible d'une façon plus simple et abstraite, en laissant implicites certains détails.

– La révision du langage ainsi que la mise en œuvre de ces modifications au sein du compilateur et système d'exécution devaient pouvoir s'effectuer, par un petit groupe de personnes (l'auteur ainsi que deux étudiants), à l'intérieur d'un laps de temps réduit (quelques mois).

C'est donc pour ces raisons que la nouvelle version du langage Threaded-C (version 2.0) constitue clairement, comme on pourra le constater dans ce qui suit, une

```
 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 4. *La fonction récursive* `fib` *en Threaded-C (version 2.0)*

évolution du langage par rapport à la version initiale, non une révolution. Nous examinerons ultérieurement (à la section 7) d'autres modifications du langage, certaines plus significatives, actuellement en développement ou envisagées pour le futur.

Notons que bien que la deuxième série de contraintes ne soit pas d'ordre strictement scientifique, de telles contraintes doivent inévitablement être prises en compte dans tout projet d'ingénierie, où des contraintes d'ordres pratiques entraînent inévitablement certains compromis.

5.1. *Simplification et amélioration du langage*

La figure 4 présente la procédure `fib` écrite dans la nouvelle version du langage. Les principales différences d'avec la version précédente sont les suivantes :

– Une procédure n'a besoin d'être déclarée `THREADED` (ligne 1) que si elle sera activée de façon parallèle ou que si elle contient des fibres. Une procédure ordinaire, appelée à l'aide de l'instruction d'appel du langage C, peut maintenant contenir des opérations de communication ou synchronisation, ce qui n'était pas possible dans la version initiale du langage. Dans ce cas, les fentes utilisées doivent évidemment être reçues en arguments (type `SPTR`) puisque la notion de fibre ou de fente locale ne prend son sens que dans le contexte d'une procédure `THREADED`.

– Une déclaration de fibre (ligne 13) introduit un identificateur dénotant le nom de la fibre et associe à cette fibre, de façon implicite, une fente de synchronisation de même nom (`RETOURNER_RES`). Le nombre de signaux requis pour activer la fibre la première fois (*init count*) est indiqué par `<*` k `*>`, donc de façon locale. Par défaut,

si un seul nombre est indiqué comme c'est le cas ici, alors le nombre de signaux requis pour les activations subséquentes de la fibre (le *reset count*) est le même que le nombre initial de signaux. Par contre, il est aussi possible d'indiquer explicitement un *reset count*, par exemple, `<* 2, 1 *>` indique que la première activation nécessitera la réception de deux signaux alors que les activations subséquentes n'en auront besoin que d'un seul. Ceci est utile, entre autres, pour certains programmes qui définissent des pipelines logiciels : dans l'état stable du pipeline, certaines fibres ont alors besoin d'un nombre de signaux inférieur à celui requis pour l'amorce (le remplissage) du pipeline.

Notons que la spécification de façon *locale* du nombre de signaux est particulièrement intéressante pour les procédures contenant plusieurs fibres, car on évite ainsi les longues séries d'instructions `INIT_SLOT` au début de la procédure. L'instruction `INIT_SLOT` peut évidemment être utilisée si plusieurs fentes peuvent référer à la même fibre ou si l'association fibre/fente doit se faire de façon dynamique, par exemple :

```
INIT_SLOT( SLOT1, 2, 1, FIBRE1 );  /* Plusieurs fentes, une fibre. */
INIT_SLOT( SLOT2, 2, 1, FIBRE1 );

while ( condition ) {
   INIT_SLOT( SLOT, 2, 2, FIBRE ); /* Association dynamique. */
   ...
}
```

– La fin d'une fibre (lignes 7, 12 et 15) est maintenant *implicite*, l'exécution d'une fibre se déroulant selon la sémantique usuelle du langage C, à la différence que l'exécution de la fibre se termine aussitôt que le mot-clé `FIBER` (qui identifie la prochaine fibre) ou l'instruction `TERMINATE` (qui termine le *thread*) est rencontrée. De plus, contrairement à la version initiale du langage, l'instruction `TERMINATE` (auparavant `END_FUNCTION`) peut, comme une instruction `return` normale, apparaître n'importe où dans la procédure et un message d'erreur est signalé si elle est absente.

– Des instructions *génériques* de synchronisation et de transfert de données sont maintenant disponibles : `PUT_SYNC`, `GET_SYNC`, `BLKMOV_SYNC` et `SYNC`. La même instruction (surchargée) peut donc être utilisée, peu importe le type de fente à synchroniser (locale ou globale) ou le type de l'élément à transmettre (entier, flottant, etc.). Par exemple, pour l'instruction `PUT_SYNC`, qui transfère une donnée (de type `T`) vers la destination `dest` (indiquée par un pointeur global vers un objet de type `T`), on a les deux versions (elles mêmes surchargées) suivantes, selon que la fente à synchroniser est locale (*`SLOT_ID`*, c'est-à-dire un identificateur dénotant une fente locale), ou globale (`SPTR`) :

```
void PUT_SYNC( T datum, T *GLOBAL dest, SLOT_ID s )
void PUT_SYNC( T datum, T *GLOBAL dest, SPTR s )
```

De façon similaire, l'instruction `GET_SYNC`, qui permet de transférer une donnée d'une adresse source vers une adresse destination, est définie comme suit :

```
void GET_SYNC( T *GLOBAL src, T *GLOBAL dest, SLOT_ID s )
void GET_SYNC( T *GLOBAL src, T *GLOBAL dest, SPTR s )
```

L'instruction `BLKMOV_SYNC`, quant à elle, est semblable à `GET_SYNC` mais effectue des transferts par bloc, un argument supplémentaire permettant de spécifier le nombre d'octets à transmettre.

Soulignons finalement que, même si difficile à maîtriser, la notion de pointeur `GLOBAL` a été préservée dans la nouvelle version du langage, car elle est fondamentale dans le contexte de l'architecture `EARTH`. Comme on l'a mentionné précédemment, une fibre `EARTH` est non préemptive et, de plus, la mémoire est généralement physiquement distribuée entre les divers processeurs. Il est donc crucial que le langage supporte la distinction entre accès local (court temps de latence ne nécessitant pas de changement de contexte) et accès global (temps de latence possiblement long, nécessitant un changement de contexte), de façon à ce que les fibres et les coûts de communication soient explicites et demeurent sous le contrôle du programmeur. Notons toutefois que la version précédente du compilateur Threaded-C ne permettait pas les opérations sur les pointeurs globaux telles que `gptr++`, `gptr+i` ou `&gptr->a`. Cette restriction a été éliminée puisque de telles opérations manipulent exclusivement des adresses et, contrairement à des expressions telles que `*gptr` ou `gptr->a`, n'entraînent aucune communication implicite lorsque `gptr` réfère à une adresse non locale.

5.2. *Programmation multi-contextes de grappes de multiprocesseurs*

Outre la simplification et l'uniformisation du langage, un autre objectif de la révision de Threaded-C était d'assurer que les programmes s'exécutent correctement sur diverses machines parallèles, y compris des machines avec plusieurs processeurs partageant une même mémoire. Sur de telles machines, il est possible que *plusieurs* fibres qui accèdent à la même mémoire soient actives en même temps, y compris des fibres provenant d'une même procédure. Deux mécanismes ont donc été introduits pour assurer l'atomicité et l'exclusion mutuelle :

1) Fibres mutuellement exclusives : une fibre `EXCLUSIVE` est assurée d'être *la seule fibre* à s'exécuter parmi les fibres `EXCLUSIVE`s du *thread* dont elle fait partie. Ce mot-clé joue donc un rôle semblable au `synchronized` de Java [LEA 00].

2) Boîtes aux lettres atomiques : le modèle de synchronisation de Threaded-C s'inspire du modèle *dataflow*. Une notion utile dans ce modèle est celle d'une opération de fusion (*merge*) non déterministe permettant de combiner, en un point unique, des valeurs provenant de différentes sources.

Une opération de fusion non déterministe peut être utile, entre autres, pour mettre en œuvre un processus de réduction. Un tel processus consiste à appliquer une opération, généralement un opérateur binaire associatif et commutatif, à une collection de valeurs produites dans un ordre arbitraire par différents processus. Par exemple, la réduction à l'aide de l'opérateur « `+` » permet de calculer la somme d'une série de valeurs, alors que l'utilisation d'un opérateur « `max` » permet de trouver la valeur maximum.

Dans la version initiale de Threaded-C, un tel processus de réduction ne pouvait être programmé de façon simple que de l'une ou l'autre des façons suivantes :

– En fixant, de façon statique, le nombre de processus producteurs, chacun d'entre eux transmettant alors son résultat dans un espace destination dédié.

– En associant un verrou d'exclusion mutuelle à une destination unique partagée par l'ensembles des producteurs.

La première solution empêchait toutefois l'opération d'être appliquée en parallèle, le processus réducteur devant attendre que toutes les valeurs aient été transmises. Quant à la deuxième solution, un premier désavantage est qu'elle entraînait des coûts inutilement élevés de communication : tant l'accès au verrou que la transmission de la valeur peuvent nécessiter une communication. Un autre désavantage, plus fondamental celui-là, est que la mise en œuvre en Threaded-C (version 1.0) de verrous ou de tout autre mécanisme d'exclusion mutuelle reposait sur la propriété qu'à tout instant une seule fibre pouvait être active sur un nœud donné de la machine. Or, cette propriété, valide dans les *mises en œuvre* initiales de l'architecture `EARTH`, n'était pas une propriété de l'*architecture* `EARTH` elle-même. Ainsi, une mise en œuvre de l'architecture `EARTH` sur une grappe de multiprocesseurs avec mémoire partagée par plusieurs processeurs violerait inévitablement cette propriété, rendant ainsi caduques les mises en œuvre de l'exclusion mutuelle.

Dans la version révisée du langage, un nouveau type `MAILBOX` (*atomic mailboxes* = boîtes aux lettres atomiques) a donc été introduit dans le but de résoudre les problèmes d'exclusion mutuelle et de supporter un style d'opérations de fusion non déterministe typique du modèle *dataflow*. Ce nouveau type supporte, entre autres, les opérations suivantes (voir [TRE 00b] pour d'autres opérations) :

1) `INIT_MAILBOX` : alloue et initialise, sur le processeur local, une boîte aux lettres. Cette opération a aussi pour effet d'associer à la boîte aux lettres une fente de synchronisation à laquelle un signal sera automatiquement envoyé à chaque fois qu'une nouvelle donnée sera transmise (avec `DROP_IN`).

2) `DROP_IN` : transmet une donnée dans une boîte aux lettres. Cette opération peut être exécutée à distance, à partir d'un autre processeur (par l'intermédiaire d'un pointeur global vers une boîte aux lettres). Lorsque la donnée arrive à destination, un signal est automatiquement envoyé à la fente spécifiée lors de l'appel à `INIT_MAILBOX`.

3) `RETRIEVE_ITEM` : retire une donnée (arbitraire) d'une boîte aux lettres. Contrairement à `DROP_IN`, cette opération *doit* s'exécuter sur le processeur où a été allouée la boîte aux lettres.

La figure 5 présente un exemple utilisant une boîte aux lettres et une fibre exclusive : un processus maître (`MAIN`) crée un *thread* `producteur` sur chacun des processeurs disponibles (lignes 14–15, où `NUM_NODES` indique le nombre de processeurs disponibles sur la machine) et attend ensuite que chacun lui transmette une valeur, dans un ordre arbitraire. Chacun des producteurs transmet (ligne 4) son numéro d'identification (`NODE_ID`, qui indique le numéro du processeur courant) en utilisant le pointeur global vers la boîte aux lettres `mb` reçue en argument (ligne 1). Lorsque la donnée arrive finalement à destination, un signal est automatiquement envoyé à la fente `TRAITER_ITEM` (ligne 17), fente qui a été associée à la boîte aux lettres lors de l'initialisation (ligne 13). Chaque signal envoyé à cette fente crée une *nouvelle activation* de

```
 1  THREADED producteur( MAILBOX *GLOBAL mb )
 2  {
 3    int n = NODE_ID;
 4    DROP_IN( mb, &n, sizeof(int) );
 5    TERMINATE;
 6  }
 7
 8  THREADED MAIN()
 9  {
10    MAILBOX mb;
11    int i, total = 0;
12
13    INIT_MAILBOX( &mb, TRAITER_ITEM );
14    for( i = 0; i < NUM_NODES; i++ )
15      INVOKE( i, producteur, TO_GLOBAL(&mb) );
16
17    EXCLUSIVE FIBER TRAITER_ITEM <* 1 *> {
18      int v;
19
20      RETRIEVE_ITEM( mb, &v );
21      total += v;
22      SYNC(TERMINER);
23    }
24
25    FIBER TERMINER <* NUM_NODES *> {
26      printf( "total = %d\n", total );
27      TERMINATE;
28    }
29  }
```

Figure 5. *Un processus de réduction avec producteurs multiples, boîte aux lettre atomique et fibre exclusive*

la fibre de même nom — on a ici un exemple d'une même fibre activée à *plusieurs* reprises, donc une fibre pour laquelle plusieurs instances sont créées dynamiquement. Le code associé à la fibre `TRAITER_ITEM` permet alors de retirer une des données reçues (choisie arbitrairement, ligne 20) et ensuite de mettre à jour la variable `total` (ligne 21). Soulignons que les manipulations de la variable `total` sont faites de façon atomique puisque le mot-clé `FIBER` de la fibre `TRAITER_ITEM` est précédé du mot-clé `EXCLUSIVE` : si plusieurs données arrivent « en même temps », plusieurs instances de la fibre seront mises dans la file d'activation, mais une seule instance à la fois pourra être exécutée.

Finalement, pour compléter l'explication de cet exemple, notons qu'à chaque fois qu'un élément est reçu et qu'une instance de la fibre `TRAITER_ITEM` s'exécute, un signal est envoyé à la fente `TERMINER` (ligne 22). Lorsque tous les éléments ont été reçus

et cumulés dans la variable `total` (un par processeur, donc `NUM_NODES` tel qu'indiqué à la ligne 25), le contenu de la variable `total` est alors imprimé et le programme se termine.

Un point important à souligner concernant le traitement des boîtes aux lettres atomiques est que, tout comme pour les signaux de synchronisation, on considère que c'est la tâche de l'unité de synchronisation de gérer les boîtes aux lettres et de s'assurer de l'atomicité des opérations. Dans les mises en œuvre actuelles de l'architecture `EARTH`, cette unité de synchronisation est en fait réalisée par l'intermédiaire d'un système dynamique d'exécution (*RTS = Run-Time System*). En pratique, c'est donc le *RTS* qui s'assure que les opérations effectuées sur les boîtes aux lettres se font de façon atomique, que pour un *thread* donné une seule fibre exclusive à la fois s'exécute. Pour une description plus détaillée du fonctionnement du *RTS* développé pour la version 2.0 de Threaded-C, le lecteur peut consulter [MOR 01].

Finalement, notons que dans le but de rendre accessible aux programmeurs diverses informations sur la structure de la machine, informations pouvant être utiles pour améliorer les performances et l'efficacité d'un programme, divers opérateurs sont disponibles pour identifier les propriétés des pointeurs globaux : `OWNER_OF`, `IS_LOCAL`, `SHARE_MEMORY`, etc. Pour plus de détails, le lecteur peut consulter [TRE 00b].

6. Stratégies de programmation pour la définition de mécanismes de synchronisation

Comme on l'a vu dans la section précédente, les mécanismes de synchronisation de base de Threaded-C sont les opérations de communication (`PUT_SYNC`, `GET_SYNC`, `BLKMOV_SYNC`), les boîtes aux lettres atomiques, et les fibres exclusives. À l'aide de ces mécanismes de base, il est possible de définir d'autres mécanismes de synchronisation. Les modules de bibliothèque suivants ont ainsi été définis en utilisant les mécanismes mentionnés plus haut :[2]

– Verrous d'exclusion mutuelle ;

– Sémaphores ;

– I-structures [ARV 89] ;

– Canaux de communication [AND 00] ;

– Boites de réduction parallèle ;

– Liens pour communications bi-directionnelles entre pairs (*peers links*) ;

– Opération atomique de transfert de bloc de données (`ATOMIC_BLKMOV`) ;

En fait, lors des discussions ayant conduit à la révision du langage, différents mécanismes de support à l'exclusion mutuelle avaient été examinés. Initialement, les deux

2. Un certain nombre de ces modules sont disponibles à l'URL suivant :

`http://www.capsl.udel.edu/EARTH/LIBRARY-DOC/`

approches concurrentes étaient principalement l'inclusion dans le langage de verrous d'exclusion mutuelle et l'ajout d'une instruction spéciale de transfert de blocs de données (`ATOMIC_BLKMOV`). Chacun de ces mécanismes avait toutefois des désavantages majeurs :

– L'utilisation de verrous conduit à un style de programmation qui n'est pas celui du modèle *dataflow* sous-jacent au langage Threaded-C. De plus, comme on l'a mentionné brièvement à la section précédente, l'utilisation d'un verrou peut entraîner une augmentation significative du nombre de communications. Ainsi, une approche naïve au transfert atomique d'une valeur vers une destination partagée protégée par un verrou peut requérir plusieurs communications inter-processeurs : l'accès au verrou (deux communications : requête et confirmation), le transfert de la valeur, la confirmation que la valeur a été reçue, suivi de la libération du verrou.

– L'ajout d'une instruction spéciale de transfert atomique (`ATOMIC_BLKMOV`) aurait demandé une modification importante à la façon dont une fente de synchronisation est manipulée. Lorsque la valeur (nombre de signaux restant) associée à une fente de synchronisation tombe à 0, la fibre associée est ajoutée à la file d'exécution et la valeur associée à la fente est aussitôt mise à jour en utilisant le *reset count* spécifié lors de la déclaration de fibre/fente. Or, l'ajout de l'instruction `ATOMIC_BLKMOV` aurait demandé qu'un nouveau type de fente de synchronisation soit défini, fente dont la réinitialisation aurait été sous le contrôle explicite du programmeur (à l'aide d'une nouvelle instruction `RESET_SLOT`) plutôt qu'implicite et sous le contrôle de l'unité de synchronisation.

Or, tant dans le cas des verrous que dans le cas de l'instruction atomique de transfert, il a pu être montré que les boîtes aux lettres atomiques permettaient de définir des modules de bibliothèque mettant en œuvre ces autres approches. Dans ce qui suit, nous allons donc tenter d'expliquer les grandes lignes de la stratégie de programmation utilisée pour la mise en œuvre de tels mécanismes de synchronisation. Nous allons aussi expliquer l'impact qu'a, sur l'interface des opérations, le style de fibre supporté par l'architecture `EARTH`.

6.1. *Approche* split-phase *des opérations potentiellement bloquantes*

Le premier point à souligner concernant la définition et la mise en œuvre de divers mécanismes de synchronisation à l'aide de bibliothèques Threaded-C est que, à cause des caractéristiques des fibres de l'architecture `EARTH`, les opérations de synchronisation potentiellement bloquantes deveniennent des instructions dites *split-phase* : une requête doit tout d'abord être transmise et un signal, asynchrone, doit par la suite être envoyé pour indiquer que la requête a été satisfaite. Ainsi, l'appel pour obtenir l'accès à un verrou *split-phase* a l'allure suivante, l'accès exclusif au verrou n'étant assuré

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

Figure 6. *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 );
```

Figure 7. *API des sémaphores en Threaded-C*

qu'après que la fente `VERROU_OBTENU` ait été signalée, et non pas immédiatement après que l'instruction `LOCK_SYNC` ait été exécutée :

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

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

Un autre exemple illustrant l'approche *split-phase* est présenté dans les figures 6 et 7. Les interfaces POSIX [BUT 97] des opérations pour la manipulation de sémaphores sont présentées à la figure 6. Rappelons qu'à un sémaphore (de type `sem`) est associée une variable entière non négative, où `sem_wait` (aussi appelée `P`) permet de diminuer la valeur de la variable alors que `sem_post` (opération `V`) permet de l'augmenter. Notons toutefois que l'opération `sem_wait` ne peut s'exécuter et se compléter que lorsque la valeur du sémaphore est supérieure à 0 ; dans le cas contraire, le *thread* POSIX qui exécute `sem_wait` *bloque* jusqu'à ce que la variable redevienne supérieure à 0, à quel point elle peut alors être diminuée.

Les interfaces des opérations de la version Threaded-C de sémaphores semblables à ceux de la bibliothèque POSIX sont présentées à la figure 7. Les interfaces de ces opérations diffèrent de celles des sémaphores POSIX et ce pour plusieurs raisons :

– Tout d'abord, un `SEMAPHORE` est un objet qui est alloué et initialisé sur un processeur donné (avec `SEM_INIT_SYNC`) mais qui peut être utilisé sur n'importe quel processeur. Un `SEMAPHORE` doit donc être un pointeur `GLOBAL` (cf. le `typedef`).

– Ensuite, en Threaded-C, une fibre *ne peut jamais bloquer*, puisque les fibres sont non préemptives. Lorsqu'il est possible qu'une opération puisse ne pas complé-

ter immédiatement, comme c'est le cas avec l'opération `sem_wait`, alors une approche *split-phase* est utilisée, telle qu'indiquée par l'opération `SEM_WAIT_SYNC`[3] : le deuxième argument (`wait_termine`) spécifie une fente de synchronisation (`SPTR`) à laquelle un signal sera transmis lorsque l'opération d'accès au sémaphore sera complétée. De façon générale, donc, on peut dire qu'en Threaded-C, au lieu d'avoir un seul *thread* qui, possiblement, bloque à plusieurs reprises, on génère plutôt plusieurs fibres distinctes, chacune d'entre elles étant non bloquante.

– Finalement, dans la version Threaded-C, l'opération de création et d'initialisation d'un sémaphore est elle aussi une opération *split-phase*, et ce bien que l'opération équivalente POSIX ne soit pas bloquante. Ceci s'explique par la façon, décrite dans les prochains paragraphes, dont les sémaphores sont mis en œuvre.

6.2. *Mise en œuvre de mécanismes de synchronisation à l'aide de* moniteurs actifs

En Threaded-C, un sémaphore est associé à un *thread* (une instance d'une procédure *threaded* nommée `semaphore_handler`) dont la tâche est de gérer les requêtes pour manipuler le sémaphore (par l'intermédiaire du pointeur global associé). À un sémaphore est aussi associée, sur le processeur où le sémaphore a été alloué, une structure de données locale (de type `struct SEMAPHORE`) contenant la valeur courante du sémaphore ainsi que d'autres informations de synchronisation :

```
struct SEMAPHORE {
  MAILBOX wait_mailbox;   /* To receive the WAIT requests. */
  SPTR    post;           /* To be signaled by POST requests. */
  int     nb_waiting;
  int     value;
};
```

Il est important de réaliser que lorsqu'une opération doit être effectuée sur un sémaphore, la requête pour cette opération peut évidemment se faire à partir d'un autre processeur que celui où le sémaphore a été créé. De façon à minimiser le nombre de communications lors d'une opération, il faut donc faire en sorte que le traitement s'effectue sur le processeur parent (approche *owner based*). Une approche de style délégation avec mandataire (*proxy*) est donc utilisée. Ainsi, l'opération `SEM_WAIT_SYNC` exportée par l'API présentée à la figure 7 est en fait une macro dont le seul rôle est d'invoquer l'opération appropriée sur le processeur où est situé le sémaphore :

```
#define SEM_WAIT_SYNC(sm, sl)\
        INVOKE(OWNER_OF((sm)), _SEM_WAIT_SYNC, (sm), (sl))
```

3. Notons que le nom de l'opération se termine par le suffixe `SYNC` : par *convention*, dans la version révisée du langage Threaded-C, une opération qui transmet un signal de synchronisation possède un nom qui se termine par ce suffixe, par exemple, `GET_SYNC`, `PUT_SYNC`, `SYNC`, etc.

À son tour, l'opération `_SEM_WAIT_SYNC` est définie comme suit, donc n'a pour seul rôle que celui d'activer la fibre appropriée du *thread* qui gère le sémaphore :

```
THREADED _SEM_WAIT_SYNC( SEMAPHORE sem, SPTR slot )
{
 DROP_IN(TO_GLOBAL(&TO_LOCAL(sem)->wait_mailbox), &slot, sizeof(SPTR));
 TERMINATE;
}
```

Dans ce cas, une boîte aux lettres est utilisée puisqu'il est nécessaire de transmettre l'identité (`SPTR`) de la fente à synchroniser lorsque le sémaphore aura été obtenu.

La figure 8 présente le code définissant le *thread* `semaphore_handler` — notons que pour simplifier la présentation, le code associé à l'opération de destruction du sémaphore a été omis. Ce sont les champs de la structure de données associée à un sémaphore qui permettent, directement (par exemple, champ `post`) ou indirectement (champ `wait_mailbox`), de signaler des fibres locales au *thread* `semaphore_handler` créé pour gérer le sémaphore. Selon le cas, ce sera la fibre `WAIT` (ligne 23) qui sera activée, signalée indirectement par l'intermédiaire de la boîte aux lettres `wait_mailbox` associée à cette fibre (ligne 19), ou encore la fibre `POST` (ligne 34), signalée par le biais de la fente associée au champ `post` (ligne 18). Ce sont donc ces fibres exclusives, locales au *thread* `semaphore_handler`, qui effectuent le travail approprié de manipulation du sémaphore et qui assurent, à l'aide des opérations de synchronisation de base, qu'un sémaphore est manipulé de façon atomique.

Cette stratégie de programmation, développée de façon indépendante et qui s'impose de façon naturelle dans le contexte du langage Threaded-C, est aussi décrite par Andrews [AND 00, Chap. 7], qui parle alors d'une approche dite de *moniteur actif*, c'est-à-dire où un processus actif, une forme de *serveur*, gère les requêtes d'accès à une structure de données encapsulée par le moniteur. Dans le contexte de Threaded-C, cette approche s'impose parce que les échanges d'information entre processeurs ne peuvent se faire qu'en utilisant des opérations explicites de communication. La façon la plus directe d'assurer la manipulation atomique d'une structure de données consiste alors à effectuer ces manipulations grâce à des fibres mutuellement exclusives d'une même activation de procédure. Lorsque cela est nécessaire, par exemple, le traitement est trop complexe pour pouvoir être effectué en une seule fibre, il est alors possible d'utiliser des verrous d'exclusion mutuelle. Dans ce cas, il est toutefois préférable d'utiliser un verrou qui soit local, de façon à minimiser le nombre de communications inter-processeurs.

L'approche avec un moniteur actif gérant de façon atomique et exclusive les requêtes d'accès et de manipulation d'une structures de données privée est utilisée pour plusieurs des modules de bibliothèque : les verrous d'exclusion mutuelle (dits *split-phase locks* [TRE 00b]), les I-structures, les canaux de communication inter-processus, etc.

```
 1  THREADED semaphore_handler
 2  (
 3    SEMAPHORE sem_handle,
 4    int init_value,
 5    SPTR created
 6   )
 7  {
 8    struct SEMAPHORE* sem;
 9    SPTR              slot_to_sync;
10    int               size;
11
12    assert( IS_LOCAL(sem_handle) );
13    sem = TO_LOCAL(sem_handle);
14
15    /* Initialize the various fields. */
16    sem->value      = init_value;
17    sem->nb_waiting = 0;
18    sem->post       = TO_SPTR(POST);
19    INIT_MAILBOX( &(sem->wait_mailbox), TO_SPTR(WAIT) );
20    SYNC(created);
21    /* Initialization completed: wait for requests to arrive. */
22
23    EXCLUSIVE FIBER WAIT <* 1 *> {
24      if ( sem->value > 0 ) {
25        sem->value-;
26        RETRIEVE_ITEM( sem->wait_mailbox, &slot_to_sync );
27        SYNC( slot_to_sync );
28      } else {
29        /* Already 0: leave demander in mailbox. */
30        sem->nb_waiting++;
31      }
32    }
33
34    EXCLUSIVE FIBER POST <* 1 *> {
35      sem->value++;
36      if ( sem->nb_waiting > 0 ) {
37        /* Some other process is waiting for it. */
38        RETRIEVE_ITEM( sem->mailbox, &slot_to_sync );
39        sem->nb_waiting-;
40        sem->value-;
41        SYNC( slot_to_sync );
42      }
43    }
44  }
```

Figure 8. *Le moniteur actif* `semaphore_handler`

```
 1  THREADED fib( int n, ENTRY(int) resultat )
 2  {
 3    if (n <= 1) {
 4      SEND( resultat, 1 );
 5      TERMINATE;
 6    } else {
 7      TOKEN( fib, n-1, TO_ENTRY(RES1) );
 8      TOKEN( fib, n-2, TO_ENTRY(RES2) );
 9    }
10
11    FIBER RES1( int r1 )
12          RES2( int r2 ) {
13      SEND( resultat, r1+r2 );
14      TERMINATE;
15    }
16  }
```

Figure 9. *La fonction récursive* `fib` *en Threaded-C (prototype d'une version 3.0)*

7. Évolution future du langage Threaded-C

Threaded-C supporte autant les stratégies de programmation fondées sur la mémoire partagée que sur l'échange de messages. Par exemple, les fibres d'une même activation peuvent s'échanger des informations en utilisant le bloc d'activation associé à leur *thread*. De même, des fibres ou *threads* s'exécutant sur le même nœud peuvent communiquer directement par l'intermédiaire de la mémoire. De façon générale, toutefois, des *threads* indépendants communiquent plutôt en s'échangeant des messages. Dans la version actuelle du langage, la structure et le contenu des échanges n'est pas toujours très clair, puisque les communications se font essentiellement à l'aide d'opérations sur des pointeurs globaux, donc des adresses. Or, de telles adresses, héritage de C, peuvent être utilisées tant pour le transfert de simples valeurs que pour le transfert de tableaux de valeurs, ce qui peut parfois conduire à une forme de *couplage pathologique* [MCC 93] entre l'appelant et l'appelé.

Ainsi, soit la fonction suivante (définie seulement en partie) :

```
THREADED foo( int *GLOBAL gptr, SPTR fin ) {
  int tab[N];
  ...
  BLKMOV_SYNC( tab, gptr, N*sizeof(int), fin );
  ...
}
```

Un appel « `TOKEN(foo, TO_GLOBAL(&n), TO_SPTR(SLT))` », où `n` serait une simple variable entière plutôt qu'un tableau, serait tout à fait légal mais conduirait très certainement à une erreur d'exécution.

Dans le but de diminuer le recours à l'utilisation de pointeurs pour le transfert de

valeurs tout en préservant le plus plus possible la correspondance avec les communications et fibres de l'architecture sous-jacente, une nouvelle extension du langage Threaded-C a récemment été développée. L'objectif de cette nouvelle extension est de rendre plus claires et plus explicites les communications entre fibres de *threads* indépendants tout en réduisant le couplage. Plus précisément, une notion de *fibres avec arguments*, inspirée de la notion de *entry point* du langage Charm [KAL 95] et de la notion d'*inlet* de la machine TAM [CUL 91], a été introduite. Cette nouvelle extension du langage permet alors d'écrire la procédure `fib` sous la forme présentée à la figure 9, dont les éléments clés sont les suivants :

– La procédure `fib` (ligne 1), plutôt que de recevoir un pointeur vers le résultat et un pointeur vers une fente de synchronisation, reçoit simplement un point d'entrée `resultat` de type `ENTRY(int)`, c'est-à-dire une référence à une fibre avec argument permettant de recevoir un `int`. Un tel point d'entrée peut donc être vu comme une forme de canal ou port de communication par l'intermédiaire duquel il est possible d'envoyer des valeurs à une fibre dans le but de satisfaire ses dépendances de données et, donc, de l'activer.

– À une même fibre peuvent être associés plusieurs points d'entrée, en fonction du nombre de messages devant être reçus pour que la fibre s'exécute. Ici, la fibre qui additionne (ligne 11) les résultats produits par les deux appels récursifs (lignes 7 et 8) requiert deux résultats intermédiaires (`r1` et `r2`, produits par deux *threads* distincts) avant de pouvoir être activée. Des points d'entrée appropriés sont donc spécifiés et transmis aux deux *threads* enfants pour chacun des résultats (`TO_ENTRY(RES1) et TO_ENTRY(RES2)`, lignes 7 et 8).

– Pour transmettre une valeur à une fibre avec arguments par l'intermédiaire d'un point d'entrée, l'opérateur `SEND`, typique des approches avec canaux ou ports de communication, est alors utilisé (lignes 4 et 13). Ici, le premier argument indique le point d'entrée, alors que le ou les arguments suivants indiquent la ou les valeurs à transmettre. Notons que la transmission des valeurs est alors accompagnée implicitement d'un signal de synchronisation approprié transmis à la fibre à laquelle est associée le point d'entrée.

Comme l'illustre cet exemple, le couplage appelant/appelé entre *threads* est réduit par le fait que le nombre d'arguments à transmettre est diminué, un point d'entrée combinant destination et signal de synchronisation. De plus, plusieurs valeurs peuvent aussi être transmises dans une même communication, par exemple, un point d'entrée `ENTRY(int, int)` serait utilisé avec une instruction `SEND(pe, v1, v2)`. Un point d'entrée peut donc être vu comme une forme de canal de communication qui permet de rendre *explicites* les valeurs transmises lors des communications tout en rendant *implicites* les détails des signaux de synchronisation.

Autre avantage intéressant, l'utilisation de points d'entrée et fibres avec arguments permet aussi de spécifier les contraintes d'activation d'une fibre autrement qu'en indiquant simplement le nombre de signaux requis (i.e., avec `<* k *>`), ce qui permet donc de rendre plus explicites les dépendances de données associées à l'activation d'une fibre — « quelles sont les valeurs requises ? » plutôt que simplement « combien de valeurs doivent être reçues ? ».

Notons que, étant donné le mécanisme d'appel par valeur du langage C et la façon dont les tableaux sont manipulés en C, le transfert de tableaux doit quand même continuer à se faire en utilisant des pointeurs et des instructions `BLKMOV_SYNC` (à moins que la taille du tableau soit connue de façon statique et que le tableau soit déclaré comme champ d'un `struct`). Notons aussi qu'un point d'entrée peut lui-même être transmis en argument, ce qui permet la configuration dynamique des structures de communication, permettant entre autres un couplage indirect dans le style des *call-backs* [SAU 01].

Dernier point à signaler concernant les points d'entrées et fibres avec arguments : pour l'instant, cette extension du langage n'est encore qu'à l'état de prototype, mis en œuvre à l'aide d'un pré-processeur `perl` [SAU 01], et n'a pas été incorporée officiellement dans le compilateur, Ce pré-processeur, qui génère du code Threaded-C (version 2.0) — à un point d'entrée est associée une ou plusieurs boîtes aux lettres atomiques servant à recevoir les divers groupes d'arguments —, n'effectue pour l'instant aucune vérification sur les types des arguments transmis et reçus par le biais des points d'entrée.

Finalement, d'autres extensions possibles du langage Threaded-C sont encore à l'étude. Par exemple, il serait utile de pouvoir spécifier différents niveaux de priorité pour les fibres, principalement pour les fibres exclusives qui doivent manipuler des ressources partagées — dans la pure tradition du modèle *dataflow*, le modèle `EARTH` ne spécifie aucune contrainte sur la façon dont la prochaine fibre sélectionnée pour exécution doit être choisie. Une notion de définition de *fibres indexées*, qui permettrait de définir un groupe de fibres exécutant le même code, est aussi à l'étude, ce qui serait utile, entre autres, pour définir des boucles `forall` typiques des programmes avec parallélisme de données. Finalement, la présence d'une mémoire partagée entre plusieurs *threads* et fibres s'exécutant en parallèle doit nous conduire inévitablement à spécifier plus formellement le modèle de consistence mémoire supporté par le langage, et ce par le biais d'un modèle relaxé de mémoire, plus particulièrement le modèle LC [GAO 00].

8. Conclusion

Comme on a pu le voir dans les sections précédentes, l'approche de programmation de Threaded-C, bien que de style impératif et procédural, s'inspire aussi du modèle *dataflow*. Threaded-C se distingue d'autres langages pour la programmation parallèle multi-contextes à granularité fine principalement par sa hiérarchie à deux niveaux avec *threads* et fibres — qui permet un meilleur contrôle de la granularité du parallélisme —, ainsi que par son approche *split-phase* des opérations bloquantes — qui assure une correspondance fidèle avec le modèle d'exécution `EARTH`.

Le langage Threaded-C, il est clair, n'est pas un langage général appelé à remplacer d'autres langages ou interfaces de programmation déjà existants comme Java ou MPI. En fait, ce n'est pas dans ce but qu'il a été conçu. Threaded-C a plutôt été conçu, et c'est aussi dans cet esprit qu'il a été amélioré et qu'il continuera à l'être, pour ser-

vir de véhicule d'expérimentation pour la programmation et l'utilisation de machines parallèles multi-contextes à granularité fine, principalement l'architecture EARTH. En ce sens, il a jusqu'à présent très bien joué son rôle, puisqu'il a rendu possible le développement de nombreuses applications (voir section 4.1) qui ont permis de mieux comprendre les avantages — entre autres, exploitation efficace du parallélisme irrégulier de fine granularité et dissimulation du temps de latence — de cette architecture. De plus, Threaded-C a aussi été utilisé comme langage cible pour la compilation de langages de programmation parallèle plus abstraits, par exemple, EARTH-C [ZOP 01] et pH [MSH 00] et pourrait très bien être utilisé comme langage cible pour d'autres langages, par exemple, OpenMP [DAG 98].

Bien que le langage Threaded-C se veuille un langage d'assez bas niveau, au sens des modèles de Skillicorn et Talia, il est quand même possible, et c'est ce que nous avons essayé de montrer dans cet article, d'améliorer le niveau d'abstraction du langage tout en préservant certaines caractéristiques clés requises par le modèle EARTH, à savoir fibres non bloquantes et hiérarchie *threads*/fibres. Dans quelle mesure la version révisée du langage facilitera le travail de développement de nouvelles applications ou de compilateurs ciblés vers EARTH reste toutefois encore à voir, son adoption n'en étant encore qu'à ses débuts. Toutefois, quelques expériences récentes [TRE 01b] nous portent à croire que ce sera le cas et que, entre autres, l'utilisation de boîtes aux lettres simplifiera grandement l'écriture de certains programmes en permettant un découplage plus clair entre *threads* dans les problèmes nécessitant une organisation producteur/consommateur.

Remerciements

L'auteur tient à remercier le Prof. G.R. Gao (*Dept. of ECE, University of Delaware*) pour l'avoir accueilli au sein de son équipe (*CAPSL laboratory*), où la majeure partie du travail décrit dans cet article a été effectuée. Merci à tous les membres de cette équipe, J.N. Amaral, K.B. Theobald, C.J. Morrone, M.D. Butala et plusieurs autres, qui ont été impliqués dans la révision du langage Threaded-C. Des remerciements aussi à I. Maffezzini (Dépt. d'informatique, UQAM) pour ses commentaires sur la première version de cet article, ainsi qu'aux lecteurs anonymes pour leurs commentaires et suggestions. Merci à J. Sauvageau, stagiaire du bacc. en informatique de gestion (UQAM) pour son travail sur le pré-processeur décrit à la section 7. Finalement, soulignons que ce travail a en partie été rendu possible grâce à une subvention du CRSNG (Conseil de Recherche en Sciences Naturelles et en Génie du Canada).

9. Bibliographie

[AND 00] ANDREWS G., *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 2000.

[ARV 86] ARVIND, CULLER D., « Dataflow Architectures », *Ann. Reviews in Comp. Sci.*, vol. 1, 1986, p. 225–253.

[ARV 89] ARVIND, NIKHIL R., PINGALI K., « I-Structures : Data Structures for Parallel Computing », *TOPLAS*, vol. 11, n° 4, 1989, p. 598-632.

[BAR 91] BARTH P., NIKHIL R., ARVIND, « M-Structures : Extending a Parallel, Non-strict, Functional Languages with State », CSG Memo n° 327, March 1991, MIT.

[BUT 97] BUTENHOF D., *Programming with POSIX Threads*, Addison-Wesley, 1997.

[CAR 99] CARLSON W., DRAPER J., CULLER D., YELICK K., BROOKS E., WARREN K., « Introduction to UPC and Language Specification », CCS-TR n° 99-157, 1999, George Mason University.

[CUL 91] CULLER D., SAH A., SCHAUSER K., VON EICKEN T., WAWRZYNEK J., « Fine-grain Parallelism with Minimal Hardware Support : A Compiler-Controlled Threaded Abstract Machine », *Proc. of the Fourth Int'l Conf. on ASPLOS*, 1991, p. 164–175.

[CUL 93] CULLER D., AL., « Parallel Prog. in Split-C », *Proc. of Supercomp. '93*, 1993, p. 262–273.

[DAG 98] DAGUM L., MENON R., « OpenMP : An Industry-Standard API for Shared-Memory Programming », *IEEE Comp. Science & Eng.*, vol. 5, n° 1, 1998, p. 46–55.

[DEN 85] DENNIS J., « Data Flow Computation », BROY M., Ed., *Control Flow and Data Flow : Concepts of Distributed Programming*, p. 345–398, Springer-Verlag, 1985.

[FOS 95] FOSTER I., *Designing and Building Parallel Programs*, Addison-Wesley, 1995.

[FRI 98] FRIGO M., AL., « The Implementation of the Cilk-5 Multithreaded Language », *PLDI '98*, ACM, 1998.

[GAO 89] GAO G., YATES R., « The Argument-Fetching Dataflow Architecture Project : A Status Report », *Canadian Conf. on Elec. and Comp. Eng.*, Montreal, Sept. 1989.

[GAO 97] GAO G., TANG X., WANG J., THEOBALD K., « Thread Partitioning and Scheduling Based on Cost Model », *SPAA '97*, Newport, Rhode Island, June 1997, p. 272–281.

[GAO 00] GAO G., SARKAR V., « Location consistency — A new memory model and cache consistency protocol », *IEEE Trans. on Comp.*, vol. 49, n° 8, 2000, p. 798–813.

[HEB 98] HEBER G., BISWAS R., THULASIRAMAN P., GAO G., « Using Multithreading for the Automatic Load Balancing of Adaptive Finite Element Meshes », *Symp. on Solving Irregularly Structured Problems in Parallel*, p. 132–143, Springer-Verlag, LNCS-1457, 1998.

[HEN 97] HENDREN L., TANG X., ZHU Y., GHOBRIAL S., GAO G., XUE X., CAI H., OUELLET P., « Compiling C for the EARTH Multithreaded Architecture », *International Journal of Parallel Programming*, vol. 25, n° 4, 1997, p. 319–347.

[HUM 94] HUM H., THEOBALD K., GAO G., « Building Multithreaded Architectures with Off-the-Shelf Microprocessors », *8th IEEE Int'l Parallel Processing Symposium*, 1994.

[HUM 95] HUM H., AL., « A Design Study of the EARTH Multiprocessor », *PACT '95*, ACM Press, 1995, p. 59–68.

[KAL 95] KALE L., RAMKUMAR B., SINHA A., GURSOY A., « The Charm Parallel Programming Language and System : Part I — Description of Language Features », http ://NSCP01.PHYSICS.UPENN.EDU/parallel/environments/charm, 1995.

[KHO 99] KHOKHAR A., HEBER G., THULASIRAMAN P., GAO G., « Load Adaptive Algorithms and Implementations for the 2D Discrete Wavelet Transform on Fine-Grain Multithreaded Architectures », *Parallel Processing Symp.*, San Juan, Puerto Rico, April 1999, p. 458–462.

[LEA 00] LEA D., *Concurrent Programming in Java (2nd Ed.)*, Addison-Wesley, 2000.

[LEE 94] LEE B., HURSON A., « Dataflow Architectures and Multithreading », *IEEE Computer*, vol. 27, n° 8, 1994, p. 27–39.

[MAR 01] MARTINS W., DEL CUVILLO J., USECHE F., THEOBALD K., GAO G., « A Multithreaded Parallel Implementation of a Dynamic Programming Algorithm for Sequence Comparison », *Pacific Symposium on Biocomputing*, Mauna Lani, Hawaii, January 2001, World Scientific, p. 311–322.

[MCC 93] MCCONNELL S., *Code Complete — A Practical Handbook of Software Construction*, Microsoft Press, Redmond, WA, 1993.

[MOR 01] MORRONE C., AMARAL J., TREMBLAY G., GAO G., « A Multi-Threaded Runtime System for a Multi-Processor/Multi-Node Cluster », KENT R., Ed., *Int'l Symp. on High Performance Comp. Syst. and Appl.*, Kluwer Academic Publ., June 2001.

[MSH 00] MSHEIK H., « Code Generation for a Functional Language Targeted to a Parallel Multithreaded Machine », Master's thesis, Dépt. d'Informatique, UQAM, Août 2000, `http://www.info.uqam.ca/ ~tremblay/Maitrises/msheik.html`.

[NIK 01] NIKHIL R., ARVIND, *Implicit Parallel Programming in pH*, Morgan Kaufmann Publ., 2001.

[PAC 97] PACHECO P., *Parallel Programming with MPI*, Morgan Kaufman Publ., 1997.

[PAL 98] PALAGATTI S., *Structured Development of Parallel Programs*, Taylor & Francis, 1998.

[QUI 90] QUINN M., HATCHER P., « Data-Parallel Programming on Multicomputers », *IEEE Software*, vol. 7, n° 5, 1990, p. 69–76.

[SAU 01] SAUVAGEAU J., « Extension du langage Threaded-C : Fibres avec arguments et points d'entrée », Rapport de projet, bacc. en info. de gestion (prog. coop.), mai 2001.

[SKI 98] SKILLICORN D., TALIA D., « Models and Languages for Parallel Computation », *ACM Computing Surveys*, vol. 30, n° 2, 1998, p. 123–169.

[TAN 97] TANG X., GHIYA R., HENDREN L., GUANG G., « Heap Analysis and Optimizations for Threaded Programs », *Conf. on Parallel Arch. and Compilation Techniques*, San Francisco, CA, 1997, IEEE Computer Society Press, p. 2–13.

[THE 98] THEOBALD K., AMARAL J., HEBER G., MAQUELIN O., TANG X., GAO G., « Overview of the Threaded-C Language », CAPSL Technical Memo n° 19, March 1998, Univ. of Delaware.

[THE 00] THEOBALD K., AGRAWAL G., KUMAR R., HEBER G., GAO G., STODGHILL P., PINGALI K., « Landing CG on EARTH : A Case Study of Fine-Grained Multithreading on an Evolutionary Path », *Proc. of SC2000 : High Performance Networking and Computing*, Dallas, TX, Nov. 2000.

[THU 99] THULASIRAM R., GAO G., « A Multithreading Parallel Computational Approach for Valuing Derivatives », *First WAFA Finance Research Conference*, Fairfax, VA, April 1999.

[THU 00] THULASIRAMAN P., THEOBALD K., KHOKHAR A., GAO G., « Multithreaded Algorithms for the Fast Fourier Transform », *Proc. of the 12th annual ACM Symposium on Parallel Algorithms and Architectures*, Bar Harbor, Maine, July 2000, SIGACT/SIGARCH and EATCS, p. 176–185.

[TRE 00a] TREMBLAY G., « Threaded-C Release 2.0 : Motivation, Description, and Rationale », CAPSL Technical Note n° 09, June 2000, Univ. of Delaware.

[TRE 00b] TREMBLAY G., THEOBALD K., MORRONE C., BUTALA M., AMARAL J., GAO G., « Threaded-C Language Reference Manual (Release 2.0) », CAPSL Technical Memo n° 39, Sept. 2000, Univ. of Delaware.

[TRE 01a] TREMBLAY G., MALENFANT B., « Lenient evaluation and parallelism », *Computer Languages*, vol. 26, n° 1, 2001, p. 27–41.

[TRE 01b] TREMBLAY G., MORRONE C., AMARAL J., GAO G., « Implementation of the EARTH programming model on SMP clusters : a multi-threaded language and runtime system », Submitted to Concurrency and Computation : Practice and Experience, Nov. 2001.

[ZOP 01] ZOPPETTI G., « Compiling Several Classes of Irregular Applications on Multithreaded Architectures », PhD thesis, Comp. Sc. Dept., Univ. of Delaware, Newark, DE, 2001.

Article reçu le 27 août 2001.
Version révisée le 3 janvier 2002.
Rédacteur responsable : GIL UTARD

Guy Tremblay *est professeur agrégé au département d'informatique de l'Université du Québec à Montréal. Ses activités de recherche portent sur la programmation parallèle et les méthodes formelles de spécification et vérification : spécification de modèles faibles de mémoire, langages pour la programmation parallèle multi-contextes, vérification parallèle de modèles. Il a aussi publié un manuel sur les méthodes formelles de spécification et a contribué au* Guide to the SWEBOK *(chapitre sur la conception des logiciels).*