Parallélisme et Distribution en C++
Une revue des langages existants

Nathalie FURMENTO - Yves ROUDIER - Günther SIEGEL

Rapport de Recherche I3S RR95-02

Parallelism and Distribution in C++
A survey of existing languages

Nathalie Furmento - Yves Roudier
I3S - CNRS - UNSA
Bât. ESSI, Route des Colles - B.P. 145
06903 Sophia Antipolis Cedex (FRANCE)
{furmento,roudier}@unice.fr

Günther Siegel
I.N.R.I.A. - Centre de Sophia Antipolis
2004, route des Lucioles - B.P. 93
06902 Sophia Antipolis Cedex (FRANCE)
Gunther.Siegel@sophia.inria.fr

Abstract: This report presents a survey of existing parallel and distributed extensions of the object oriented language C++.

First, we discuss the different programming models and analyse how the parallelism is introduced. In the analysis, the declaration of concurrent entity and the new parallel constructions are treated separately and thoroughly. We also study the integration of control mechanisms in object oriented features, and the constraints resulting from distribution.

The reader will find in appendix a synthetic presentation and a general description of over twenty extensions.

A short version [FRS95] of this report has been presented in *Septièmes Rencontres Francophones du Parallélisme* (RenPar'7).

Key-words: C++, Parallelism, Distribution, Concurrency.

Parallélisme et Distribution en C++
Une revue des langages existants

Nathalie Furmento - Yves Roudier
I3S - CNRS - UNSA
Bât. ESSI, Route des Colles - B.P. 145
06903 Sophia Antipolis Cedex (FRANCE)
{furmento,roudier}@unice.fr

Günther Siegel
I.N.R.I.A. - Centre de Sophia Antipolis
2004, route des Lucioles - B.P. 93
06902 Sophia Antipolis Cedex (FRANCE)
Gunther.Siegel@sophia.inria.fr

Résumé : Nous étudions dans ce rapport plusieurs extensions parallèles et distribuées du langage objet C++. Nous dégageons des critères permettant d'analyser leurs spécificités et l'impact qu'elles ont sur le modèle objet adopté dans C++.

Dans un premier temps, nous étudions les différents modèles de programmation rencontrés. Nous analysons ensuite le mode d'introduction du parallélisme, en traitant séparément l'aspect déclaration et l'aspect utilisation. Pour finir, nous abordons les abstractions de contrôle proposées ainsi que les mécanismes liés à la distribution des objets.

Le lecteur trouvera en annexe une synthèse des différents langages et une présentation individuelle plus détaillée.

Une version résumée [FRS95] de ce rapport a été présentée lors des *Septièmes Rencontres Francophones du Parallélisme* (RenPar'7).

Mots-clé : C++, Parallélisme, Distribution, Concurrence.

Table des matières

Liste des figures

Introduction

Le paradigme objet est de plus en plus utilisé lors du développement de systèmes parallèles et distribués. Les caractéristiques inhérentes à ce paradigme (abstraction, encapsulation, réutilisation, ...) en font un candidat naturel susceptible de résoudre les principales difficultés rencontrées lors de la programmation de tels systèmes. Le langage C++ [ES94] s'étant imposé comme un standard de fait, nous nous intéressons ici aux différents mécanismes proposés pour y introduire du parallélisme.

Une typologie habituelle du parallélisme (par exemple [Joh95] ou [DN93]) distingue principalement le parallélisme de tâches du parallélisme de données. Mais dans le modèle objet adopté dans C++, la classe est à la fois un type abstrait de données et un module regroupant des fonctionnalités, ce qui explique que l'on trouve simultanément ces deux genres de parallélisme dans certaines des extensions étudiées. Une étude selon ce seul critère est alors difficile.

Nous avons donc choisi de présenter ces langages en dégageant d'autres caractéristiques. Dans un premier temps, nous étudions les différents modèles de programmation rencontrés (Section 1). Nous analysons ensuite le mode d'introduction du parallélisme, en traitant séparément l'aspect déclaration (Section 2) et l'aspect utilisation (Section 3). Pour finir, nous abordons les abstractions de contrôle proposées (Section 4) ainsi que les mécanismes liés à la distribution des objets (Section 5). Notre travail ne vise pas la classification des modèles de parallélisme (voir notamment les travaux de Papathomas et Nierstrasz [PN91] ou de Hugues et Wallas [HW93]), mais plutôt à recenser et à analyser les techniques d'extensions de C++ utilisées pour la gestion du parallélisme.

D'autres extensions du langage C++, tout en y introduisant du parallélisme, s'intéressent à un domaine d'application ou à un modèle de programmation spécifique, et de ce fait sortent du cadre de notre étude. Nous pouvons, entre autres, citer les extensions temps réel (DROL [TT92], RTC++ [IMT90]) et celles basées sur le modèle acteur (ACT++ [KLM93], Broadway [Stu94]).

Cette étude n'a pas la prétention d'être exhaustive, les extensions de C++ qui en font l'objet sont les suivantes : Arjuna (page 23), Avalon/C++ (page 25), C** (page 27), CC++ (Compositional C++, page 29), Charm++ (page 31), Composites (page 33), Concurrent C++ (page 35), DC++ (Distributed C++, page 37), Dome (page 39), DoPVM (page 41), DPC++ (Data Parallel C++, page 43), KAROS (page 45), LITP-C++ (nom que nous donnons à l'extension C++ du LITP, page 47), Mentat (page 49), Mercury (page 51), Open C++ (page 53), Panda (page 55), pC++ (page 57), Presto (page 59), QPC++ (page 61), UC++ (page 63) et μC++ (Micro C++, page 65).

Le lecteur trouvera à l'annexe A une synthèse de ces différents langages et à l'annexe B une présentation individuelle plus détaillée.

1 Modèle de programmation

En fonction de l'extension utilisée, le programmeur devra structurer son application selon un modèle déterminé. Trois grands modèles semblent émerger : 1) structuration explicite en tâches concurrentes, 2) tâches structurées par les données ou encore 3) système d'objets communicants.

Certaines extensions comme pC++ ou QPC++ proposent, à un degré divers, plus d'un modèle de programmation. Ce fait est particulièrement intéressant dans la mesure où un programmeur utilise la même extension pour développer différents types d'application. De plus, une même application peut ainsi tirer partie des spécificités de chaque modèle. Pour ces raisons, nous pensons que les environnements offrant une programmation multi-modèles vont se généraliser.

1.1 Tâches concurrentes

Le premier modèle, le plus basique, consiste à structurer l'exécution du programme en un ensemble de tâches concurrentes manipulées de manière explicite. Le concept de tâche[1] doit être rapproché de celui de flot d'exécution. Les langages CC++, DC++, DoPVM, Mercury, pC++ et Presto proposent ce modèle.

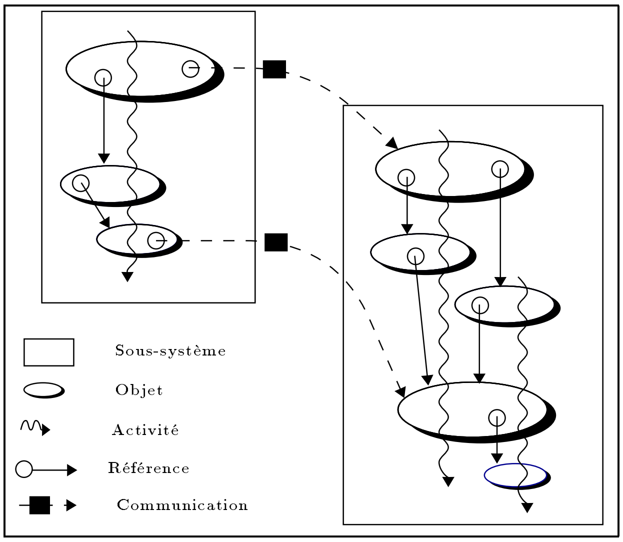

Figure 1: Modèle *tâches concurrentes*

1.2 Tâches structurées par les données

Le second modèle consiste à organiser le programme en un ensemble de tâches structurées par les données manipulées. C'est le cas du système Mentat dans lequel le programmeur indique les fonctions qui devront être exécutées en parallèle. Le compilateur organise ensuite le flot d'exécution en fonction des dépendances de données. D'autres langages comme C**, Composites, Dome, DPC++, pC++ ou QPC++ permettent de définir

[1] Dans la suite de ce document, nous utiliserons indifféremment les termes *thread*, *activité* ou *tâche*.

des *collections* d'objets dotées d'opérations globales qui s'exécuteront en parallèle sur la totalité des objets.

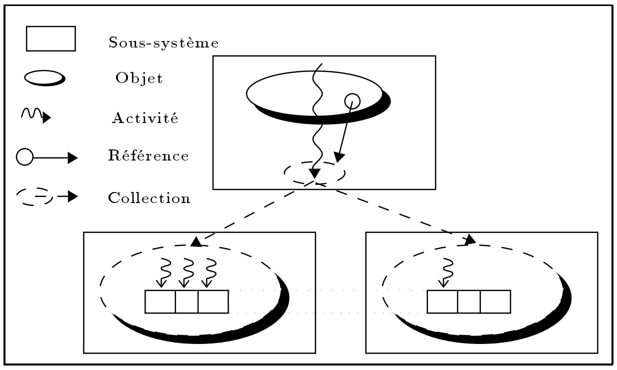

Figure 2: Modèle *tâches structurées par les données* (exemple de collection)

1.3 Objets actifs communicants

Dans le dernier modèle, l'application est composée d'un ensemble d'objets communicants. Ce modèle nous paraît le plus attractif dans la mesure où, comme en programmation séquentielle, l'objet reste une unité de traitement autonome. On peut distinguer l'approche hétérogène qui différencie objets passifs et objets actifs, de l'approche homogène (ou uniforme) où tout objet est actif.

Les extensions Arjuna, Avalon/C++, Charm++, Concurrent C++, KAROS, Open C++, Panda, QPC++, UC++ et μC++ proposent une approche hétérogène, tandis que LITP-C++ propose une approche homogène.

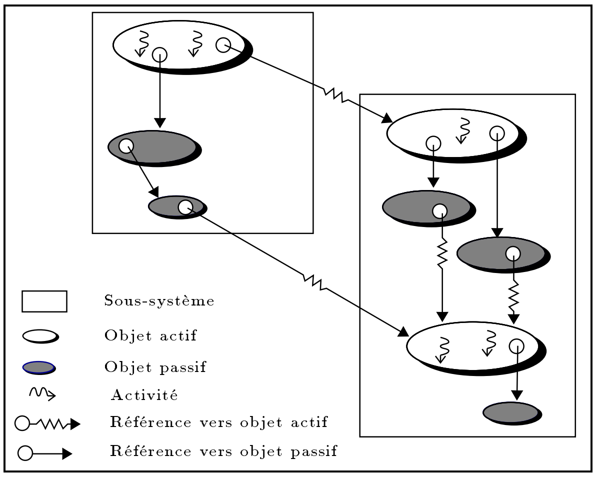

Figure 3: Modèle *objets actifs communicants* (approche hétérogène, multi-actifs)

Il est possible d'associer soit une activité à un objet, unifiant ainsi module, type et activité, soit plusieurs activités à un même objet, unifiant ainsi méthode et activité. Dans ce dernier cas, on parle alors d'objets multi-actifs, les extensions LITP-C++ et Panda offrent cette possibilité.

2 Déclaration du parallélisme

Nous nous intéressons dans cette section à la déclaration du parallélisme et aux modifications du langage C++ qui en découlent. Hormis CC++, DoPVM, LITP-C++ et UC++ qui ne nécessitent aucune déclaration spécifique (des déclarations pour les mécanismes de contrôle peuvent néammoins exister), les autres extensions étendent le système de typage afin d'y introduire le parallélisme.

Dans ce cas, le programmeur doit alors déclarer explicitement ses entités parallèles. Nous avons rencontré différentes approches concernant le typage de celles-ci. Certaines extensions permettent la définition de classes parallèles par réutilisation (héritage) et/ou extension de classes prédéfinies, d'autres permettent la construction de nouveaux types parallèles ou simplement l'utilisation de types de base existants.

2.1 Déclaration par réutilisation et extension de classes prédéfinies

La possibilité de construire de nouvelles classes parallèles par réutilisation et extension est conforme à la philosophie objet et est donc particulièrement intéressante. Selon l'implémentation retenue, les classes racines seront soit intégrées au compilateur, soit fournies sous forme de bibliothèques.

Dans les extensions Arjuna, Avalon/C++, KAROS et Panda, les classes d'objets actifs doivent dériver de classes prédéfinies. Dome, lui, fournit un mécanisme basé sur les templates. Les extensions Mercury et Presto fournissent la classe *Thread* qui est généralement utilisée comme type de base (Section 2.2.2), mais qui pourra être redéfinie selon les besoins de l'utilisateur.

2.2 Déclaration à l'aide de nouveaux types parallèles

Les nouveaux types parallèles peuvent être soit construits par le programmeur, soit déjà existants sous forme de type de base.

2.2.1 Types construits

Nous avons rencontré différentes approches pour la construction de nouveaux types parallèles. Certaines extensions introduisent de nouvelles formes syntaxiques pour la spécification de classes, d'autres de nouvelles abstractions parallèles similaires à la notion de classe.

Constructions syntaxiques

Dans certaines extensions, le programmeur préfixe ses déclarations de classe par un mot-clé indiquant au compilateur qu'il déclare un type spécifique (classe d'objets actifs, type parallèle, ...). Très souvent, les nouvelles classes d'objets actifs possèdent des restrictions (au niveau des membres statiques, publics, ...) et de nouvelles catégories de membres.

Par exemple :

- Dans Charm++, la déclaration de classes d'objets parallèles se fait en préfixant la classe par *chare*. Les classes *chare* possèdent des membres privés et des *points d'entrée* qui constituent l'interface des objets.

```
message BooleanObject { Boolean convergeValue;};

chare class Receiver {
entry:
      virtual ReceiveResult(BooleanObject *result);
private:
      int foo();
};
```

Figure 4: Classes *chare* de Charm++

- Les classes Mentat se déclarent en préfixant la classe par *persistent mentat*, *sequential mentat* ou *regular mentat*.

Dans d'autres extensions, le programmeur emploie une construction syntaxique entièrement nouvelle pour exprimer une construction parallèle. C'est par exemple le cas en C^{**}, où l'utilisateur peut déclarer ses propres agrégats (types tableaux parallèles) en postfixant la déclaration d'une classe par des crochets, et les fonctions membres parallèles par le mot-clé *parallel*. Le cas d'Open C++ est particulier : des constructions syntaxiques sont introduites pour les mécanismes réflexifs, et sont ensuite utilisées pour introduire une sémantique parallèle.

```
class matrix {
   float checksum() parallel;
} [] [];                                      // dimension 2, unknown size
class small_matrix { float value; } [5] [5];  // dimension 2, size 5x5
```

Figure 5: Agrégats de C^{**}

Abstractions similaires à la notion de classe

Certaines extensions ajoutent une abstraction parallèle similaire à la notion de classe. C'est le cas de Composites (*composite*), Concurrent C++ (*process*), pC++ (*Collection*), de QPC++ (*process class*, *process set*) et de μC++ (*uTask*).

```
Collection NameOfCollectionType:  Kernel {
   ...    // private & public & protected data fields
   ...    // and method functions
MethodOfElement:
   ...    // data and functions that are added to each element
};
```

Figure 6: *Collection* de pC++

2.2.2 Types de base

Dans le modèle tâches concurrentes, certaines extensions possèdent des classes permettant de déclarer des objets threads (*Thread* de Presto et de Mercury, *DcThread* de DC++).

L'extension DPC++, elle, définit de nouveaux types de base (par exemple pour la manipulation des entiers, des réels, ...), non héritables et fixés dans le langage ; il est impossible de rajouter un nouveau type parallèle et l'utilisateur ne dispose que de primitives prédéfinies.

```
parallel_int b(16,4);  // declare a parallel array of 16x4 integers
```

Figure 7: Type entier parallèle de DPC++

3 Utilisation du parallélisme

Dans la mesure où l'utilisation du parallélisme est fortement liée au modèle de programmation utilisé, nous reprenons dans cette section les trois modèles évoqués dans la section 1 en détaillant pour chacun d'eux les différents modes d'utilisation du parallélisme.

3.1 Tâches concurrentes

Dans ce modèle, le parallélisme est généré par la création de nouvelles activités. Ces activités peuvent être associées à un objet dédié, et sont alors contrôlables par invocation de fonctions membres. Elles peuvent aussi être créées à l'aide de primitives du langage ou de constructions syntaxiques particulières, et ne sont alors pas directement contrôlables.

3.1.1 Tâches associées à un objet dédié

Les extensions Presto et DC++ possèdent respectivement les classes *Thread* et *DcThread*. En Presto, l'activation d'une thread, préalablement créée, se fait par l'appel de la fonction membre *start()*. En DC++, l'activation se fait automatiquement à la création. Dans les deux langages, la tâche initiale et la tâche créée poursuivent leur exécution de manière indépendante.

```
Stack *s = new Stack (100);
Thread *t = new Thread(``Pusher'');   // Create a thread named ``Pusher''

t->start(s, s->push, 43);             // Let t be responsible for pushing
                                      // 43 onto the stack
```

Figure 8: Classe *Thread* de Presto

3.1.2 Tâches créées par des primitives du langage

L'extension DoPVM dispose d'une primitive, *initiate()*, qui permet de démarrer de nouveaux processus[2]. Par la suite, les processus communiquent entre eux à l'aide d'objets partagés.

```
main() {
   begin_do(``session1'');          // registering
   initiate(``worker'', ``SUN4'');  // create a process `worker''
   ...                              // on a SUN4
   end_do();                        // end
}
```

Figure 9: Processus de DoPVM

[2] Ce sont des processus Unix.

3.1.3 Tâches manipulées à l'aide de constructions syntaxiques

Parmi les extensions qui offrent des constructions syntaxiques permettant de créer automatiquement de nouvelles tâches, CC++ constitue une bonne illustration avec :

- Les blocs parallèles (mot clé *par*) dans lesquels les instructions de plus haut niveau sont exécutées de manière concurrente.
- Les boucles parallèles (mot clé *parfor*) où chaque itération crée une tâche qui exécute les instructions du bloc.
- Le parallélisme non structuré, et la possibilité de lancer en parallèle l'exécution d'une expression (mot-clé *spawn*).

```
par {
      eeny();
      meeny();
}
```

Figure 10: Bloc parallèle de CC++

3.2 Tâches structurées par les données

Dans la mesure où le parallélisme est structuré par les données manipulées, une fois ces dernières déclarées, l'utilisation de celui-ci est relativement implicite. Par exemple, pour C^{**} et pC++, l'appel d'une méthode "parallèle" sur un objet *Collection* va être automatiquement transformé en l'appel de la fonction sur chacun des éléments de la collection. Dans DPC++ et Dome, des primitives sont fournis sur les agrégats parallèles.
En général, la granularité est figée par la structure des données. Cependant dans le langage Composites, il est possible de la manipuler de manière explicite pour augmenter le parallélisme.

Dans le système Mentat, l'approche est différente. Les concepteurs sont partis du principe que les techniques d'analyse de code ne permettaient pas encore, dans le cadre général, de paralléliser automatiquement une application. C'est donc au programmeur d'indiquer les traitements qu'il serait avantageux d'exécuter de manière concurrente et distribuée. C'est ce qu'il fait en déclarant certaines de ses classes comme classe Mentat. Le compilateur organise ensuite la distribution et le flot d'exécution en fonction des dépendances de données. Les aspects communication et synchronisation sont entièrement pris en charge par le compilateur.

3.3 Objets actifs communicants

Dans ce modèle, le parallélisme est généré par la création d'objets actifs qui s'exécutent de manière concurrente et communiquent par envoi de message. Selon les extensions, nous avons les situations suivantes :

- Pour celles mono-actives, une nouvelle activité est générée à la création de l'objet actif. La concurrence provient des activités indépendantes des divers objets. On parle de concurrence inter-objet.

- Pour celles multi-actives, une activité est créée pour servir une requête, alors que l'objet peut déjà être en train d'en servir d'autres. En plus de la concurrence inter-objet, il y a de la concurrence intra-objet.

3.3.1 Création d'objets actifs

Les objets actifs peuvent être créés de différentes manières. La création peut être, soit identique à celle des objets C++, soit faite à l'aide de nouvelles primitives : *create* de Concurrent C++, *new_ chare()* de Charm++, ou *activenew* de UC++. Arjuna propose un mode de création standard, qui correspond en fait à l'instanciation et à la connexion d'un proxy avec un objet serveur distant.

```
consumer c;             // process of type consumer
c = create consumer();
```

Figure 11: Création d'objets actifs en Concurrent C++

3.3.2 Communication entre objets actifs

A l'exception de Charm++, qui définit explicitement un type *message*, les mécanismes d'appel de méthode et d'envoi de message sont généralement unifiés. Certains langages proposent par défaut une activation synchrone (Arjuna, Open C++ et Panda), d'autres une activation asynchrone (KAROS, UC++) ; quelques-uns offrent les deux modes, en explicitant soit l'un, soit l'autre (Concurrent C++, QPC++). L'invocation asynchrone permet d'obtenir de la concurrence à travers la communication.

```
process class ActiveObj {
public:
    void set_mean(int,int);
};
ActiveObj ao;

ao.set_mean(3,12);                // synchronous call :  waiting
asynchronous ao.set_mean(4,5);   // asynchronous call
...                              // process is not stopped
```

Figure 12: Appels synchrones et asynchrones en QPC++

3.3.3 Concurrence

Le degré de concurrence qu'il est possible d'obtenir en pratique dépend du support d'exécution des objets actifs. Selon les extensions, nous pourrons avoir la correspondance suivante :

1. Un processus Unix pour un ensemble d'objets actifs.
2. Un processus Unix pour un objet actif.
3. Un processus léger pour un objet actif, dans ce cas plusieurs objets actifs pourront être regroupés dans un même processus Unix.

Ce choix relève de l'implémentation et ne devrait pas intervenir lors de la programmation (si ce n'est la synchronisation qui est traitée à la section 4), toutefois la granularité du parallélisme déterminera le contenu des classes d'objets actifs et donc la structuration générale du programme. En effet, la troisième approche permet d'avoir un nombre d'objets actifs beaucoup plus important et tire profit des nouveaux systèmes d'exploitation et des machines multiprocesseurs. La concurrence pourra également être augmentée si les objets sont multi-actifs.

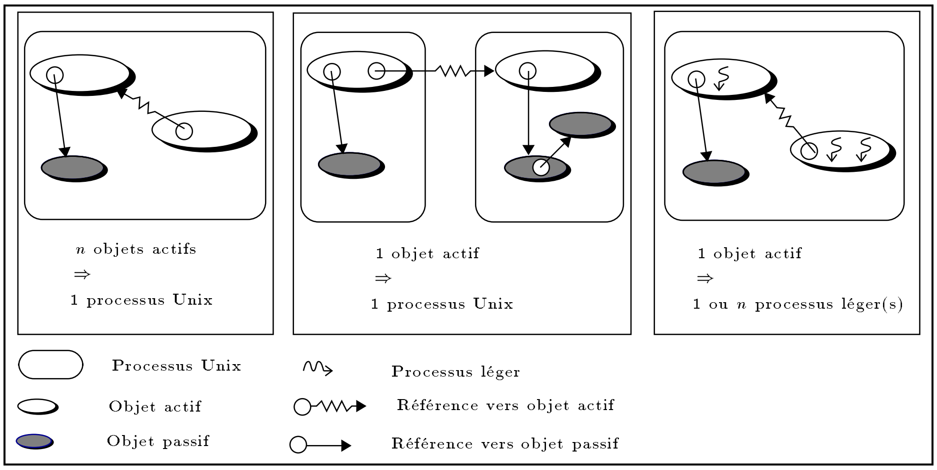

Figure 13: Granularité du parallélisme

4 Contrôle

Nous étudions dans cette section les différents mécanismes de contrôle proposés ainsi que leur intégration dans le modèle objet de C++. Les fonctionnalités introduites portent sur les mécanismes transactionnels, et sur la synchronisation. Pour celle-ci, nous avons distingué la synchronisation dirigée par les activités de la synchronisation dirigée par les messages.

4.1 Mécanismes transactionnels

Une transaction atomique (ou action atomique), mécanisme couramment utilisé dans les bases de données, regroupe un ensemble d'instructions et a les propriétés suivantes : 1) sérialisation (les transactions semblent s'exécuter en série), 2) consistence (une transaction réussit complètement ou échoue), 3) persistance (une transaction confirmée survit aux pannes). Les mécanismes transactionnels permettent de développer des applications distribuées qui peuvent résister aux pannes aussi bien du réseau de communication que des machines elles-mêmes.

Ces caractéristiques sont introduites par le biais de classes spécifiques dotées de primitives correspondantes à celles que l'on trouve dans les gestionnaires de bases de données. Avalon/C++ possède les classes *recoverable*, *atomic* et *subatomic* ayant entre autres les primitives *abort()* et *commit()*. Arjuna définit la classe *AtomicAction* et les primitives *begin()*, *end()*, *abort()*, etc. Dome n'offre pas de mécanismes transactionnels mais des mécanismes automatiques de sauvegarde et de reprise en cas de panne.

4.2 Synchronisation dirigée par les activités

La synchronisation dirigée par les activités peut intervenir de trois manières différentes : 1) au sein d'un groupe d'objets, on a alors une synchronisation globale, 2) à l'interface d'un objet, on a alors une synchronisation locale, ou 3) par le biais d'entités spécialisées, la synchronisation sera globale ou locale selon la programmation.

4.2.1 Gestion au niveau d'un groupe d'objets

Dans certains langages, la notion de groupe d'objets recouvre celle de localité de données. On a un regroupement physique des objets. Par exemple, lorsque des objets sont rassemblés dans un même *domaine* DC++, l'exécution de leurs méthodes est sérialisée.

Une autre approche consiste à grouper logiquement des objets : pour les langages C**, Composites et pC++, l'exécution d'une méthode en parallèle sur l'ensemble des objets d'une collection est vue extérieurement comme une exécution atomique. DPC++ a plutôt une approche SIMD et offre une synchronisation au niveau instruction sur les traitements effectués à l'intérieur d'un tableau parallèle.

4.2.2 Gestion par l'interface d'un objet

La synchronisation à l'interface d'un objet consiste à ordonnancer l'exécution de ses méthodes par la mise en place d'une politique de service. Cet ordonnancement peut être fixé, c'est-à-dire statique et implicite, ou programmable. Dans le second cas, nous pouvons aussi distinguer le contrôle centralisé (son expression est regroupée en un point de l'objet, et est totalement séparée de celle des fonctions membres), du contrôle décentralisé (son expression est répartie en plusieurs points, en général, dans chacune des fonctions membres).

L'extension UC++ est un exemple d'ordonnancement statique implicite dans la mesure où l'exécution des méthodes est automatiquement sérialisée.

Au sujet de l'ordonnancement programmable, nous avons rencontré différentes approches qui se distinguent par les choix adoptés (composante syntaxique et sémantique) et par la puissance d'expression qu'elles autorisent.

Les files d'attentes de DC++, sorte de barrière de synchronisation, que le programmeur doit associer à une fonction puis manipuler explicitement sont un exemple de programmation décentralisée du contrôle. Le pouvoir d'expression de ces files d'attente reste assez limité et ne permet qu'un ordonnancement rudimentaire.

```
Fork::Fork() {
   DcMakeDelayQueue(Fork::eat);   // creation of a delay queue
                                  // associated to eat()
}
void Fork::eat() {
   DcDQClose(Fork::eat);          // closing the delay queue
   ...                            // work
   DcDQOpen(Fork::eat);           // opening the delay queue
}
```

Figure 14: Files d'attente de DC++

Par contre, les points d'entrée de Concurrent C++ (*accept*) et de QPC++ (symbolisés par $<$.) sont beaucoup plus puissants (on dispose notamment d'un *select* indéterministe) et restent encore très explicites dans la sélection des requêtes à servir.

```
process class simpleMindedBuffer {
public:
   void get() { ... };
   void put() { ... };
   ...
   $simpleMindedBuffer() { // process body
      while(1) { // serve put() then get()
            all < . put;
            all < . get;
      }
   }
};
```

Figure 15: Points d'entrée de QPC++

Enfin, les conditions d'attente de LITP-C++ sont utilisables de manière beaucoup plus implicite et souple. Le pouvoir d'expression d'un contrôle déclaratif (dont le mode de sélection des requêtes est fixé) peut être moins grand que celui d'un contrôle entièrement explicite, mais la réutilisation de la synchronisation est souvent plus importante ou plus facile (à condition que le contrôle soit centralisé).

```
class ConcurrentQueue :  public FixedQueue {
    delay :
       full() delay putback(Item);
       empty() delay getfront(Item&);
};
```

Figure 16: Conditions d'attente de LITP-C++

La réutilisation de la synchronisation au niveau de l'interface d'un objet est en effet un problème difficile, notamment parce qu'elle interfère avec le mécanisme normal d'héritage. Ce problème, connu sous le nom d'anomalie d'héritage, a été largement étudié [MY93, Ber94], mais aucune solution définitive n'y a été apportée pour l'instant.

4.2.3 Gestion par le biais d'entités spécialisées

La synchronisation peut aussi être introduite par le biais d'entités spécialisées dont le seul rôle est de gérer (suspendre, réactiver, ...) une activité par rapport à son environnement (classe *Lock* de Presto ou de Mercury, *uMonitor* et *uSemaphore* de μC+).

```
// l is a reference to a Lock (Lock *l)
l->lock();
...             // critical code (the caller holds the lock)
l->unlock();
```

Figure 17: Verrous de Presto

4.3 Synchronisation dirigée par les messages

4.3.1 Mécanismes de communication

Dans les extensions à modèle objets actifs communicants, la communication est déclenchée par un appel de méthode sur un objet distant. Si le mode de communication (synchrone ou asynchrone) est en général prédéterminé (Charm ++), on peut distinguer les langages qui permettent de le sélectionner soit à la définition du type parallèle, comme Open C++, de ceux qui, comme QPC++ ou UC++, le font à l'appel de la méthode.

4.3.2 Mécanismes de variable future

Un futur (ou variable future) est une variable résultant de l'appel asynchrone d'une fonction sur un objet distant, et dont la valeur n'est pas encore connue. A la consultation[3], l'activité sera bloquée jusqu'à ce que la donnée soit disponible. On peut distinguer les futurs explicites (le programmeur précise qu'il souhaite ce type de fonctionnement) des futurs automatiques (tous les appels asynchrones ayant une valeur de retour donnent

[3]Par consultation, nous entendons lecture ou écriture d'une donnée membre, ou activation d'une méthode.

lieu à des futurs).

QPC++ et KAROS implémentent un mécanisme de futur implicite, alors que dans CC++, il est explicite (variables *sync*). Par contre, les futurs de QPC++ intègrent l'attente par nécessité (l'appel est donc complètement transparent) alors que ceux de KAROS ne peuvent être testés ou bloqués qu'explicitement.

```
res=Call(...);
...
Bool arrived = Awaited(res);  // testing the result arrival
...
Wait(res);                    // blocking wait of res
```

Figure 18: Futurs de KAROS

4.3.3 Mécanismes réflexifs

Il est possible d'intégrer au langage une part de connaissance et de possibilité d'influer sur lui-même, ce qui permet au programmeur de modifier ce comportement (au même titre que le comportement de l'application).

Ce type de mécanisme peut permettre, d'une part, de traiter les problèmes d'anomalie d'héritage, par exemple en surimposant un axe de synchronisation au mécanisme d'héritage standard, comme c'est le cas en Hector [BFPS94, BFB90]. D'autre part, et surtout, la réflexivité permet d'avoir accès directement au mécanisme d'envoi de message standard et de le modifier en vue d'obtenir une communication dotée d'une sémantique parallèle; cette approche permet aussi de contrôler de manière générique des informations sur le système, comme sa répartition. Le langage Open C++ constitue une bonne illustration des possibilités (par exemple, l'introduction de nouveaux méta-objets suffit à modifier le mode de communication ou le mode de programmation parallèle), et aussi du coût (en performance surtout, parfois en réutilisation) de telles techniques d'extension. L'introduction de tels mécanismes reste par contre assez délicate en C++.

5 Nature distribuée de l'application

5.1 Restrictions imposées par les langages

Les langages permettant de développer sur des environnements à mémoire distribuée et offrant une vision "machine virtuelle" à espace d'adressage unique imposent presque tous des restrictions (en terme de performances et/ou d'utilisation) au niveau du partage de données. Selon le modèle de programmation adopté et les contraintes de réalisation rencontrées, celles-ci sont d'importances diverses (déclaration des entités globalement accessibles avec le mot-clé *global* en CC++, variables *shared_ t* de Mercury, manipulation de variables partagées par le biais d'un identifiant en DoPVM, déclaration d'objets en mémoire partagée dans Panda par le mot-clé *DSM*).

5.2 Equilibrage de charge et ordonnancement

Pratiquement aucun des langages étudiés n'offre de mécanisme d'équilibrage de charge. Charm++ propose des stratégies automatiques ou programmables pour le placement dynamique d'objets. Dome offre un équilibrage dynamique par migration de données. Mercury et μC++ fournissent à l'utilisateur des primitives de migration d'activités (*migrate()* et *Umigrate()*). Panda permet, quant à lui, une migration des objets qui sont placés en mémoire partagée ou des activités, mais au choix explicite du programmeur (à la déclaration pour les objets et à l'utilisation pour les activités).

```
class Manager :  UserThread {
    ...
};

Manager *mgr;
...
mgr->migrate(machine);
```

Figure 19: Migration d'activité en Panda

En effet, le problème d'équilibrage de charge est difficile (voire NP-complet) en général. De bonnes solutions n'existent que pour des applications spécifiques. C'est le cas de Dome, qui est orienté vers des applications SPMD régulières, ce qui permet d'évaluer raisonnablement la charge en fonction des calculs déjà effectués.

Conclusion

Dans le cadre de l'avant-projet Sloop [BMa94], nous nous intéressons à la conception et à la réalisation en C++ de méthodes et d'outils permettant l'utilisation efficace de machines multi-processeurs pour la simulation de systèmes à événements discrets, et plus généralement d'applications non-régulières.

L'objectif de cette étude était de recenser un échantillon représentatif des extensions parallèles de C++, et d'analyser les différents mécanismes permettant d'y introduire le parallélisme. Ce travail préliminaire va nous permettre soit de sélectionner une extension satisfaisant nos spécifications, soit de profiter de l'expérience et/ou de l'acquit d'autres projets. De plus, nous sommes actuellement en train d'identifier des mécanismes originaux, compatibles avec notre modèle de parallélisme, et que nous pourrions donc intégrer.

Chacune des extensions parallèles présentées offre des variations par rapport à son modèle de programmation, ces derniers étant néanmoins peu nombreux. Ceci dit, l'utilisation généralisée du langage C++ pour la programmation de systèmes parallèles et distribués passe par un effort de standardisation. Celui-ci devra porter à la fois sur l'aspect syntaxique et sur les abstractions parallèles introduites. La granularité du parallélisme, la réutilisation, l'équilibrage de charge devront être particulièrement étudiés.
Plusieurs groupes travaillent actuellement, de manière indépendante, à la définition en vue d'une normalisation d'une extension parallèle de C++. Nous pouvons notamment citer le consortium européen Europa[4] et le projet américain HPC++[5]. Bien sûr, la standardisation d'une version parallèle de C++ passe par la standardisation du langage C++ lui-même, processus actuellement en cours dans le cadre du travail de l'ISO [Ket95, Acc95].

[4] `http://www.lpac.ac.uk:80/europa/`
[5] `http://www.extreme.indiana.edu/hpc++/`

A Récapitulatif

A.1 Modèle de programmation

	Tâches concurrentes	Tâches structurées par les données	Objets actifs communicants
Arjuna			Hétérogène Mono-actif
Avalon/C++			Hétérogène Mono-actif
C**		X	
CC++	X		
Charm++			Hétérogène Mono-actif
Composites		X	
Concurrent C++			Hétérogène Mono-actif
DC++	X		
Dome		X	
DoPVM	X		
DPC++		X	
KAROS			Hétérogène Mono-actif
LITP-C++			Homogène Multi-actif
Mentat		X	
Mercury	X		
Open C++			Hétérogène Mono-actif
Panda			Hétérogène Multi-actif
pC++	X	X	
Presto	X		
QPC++		X	Hétérogène Mono-actif
UC++			Hétérogène Mono-actif
μC++			Hétérogène Mono-actif

A.2 Déclaration du parallélisme

	A l'aide de nouveaux types parallèles	**Par réutilisation et extension de classes prédéfinies**
Arjuna		Hérite de *LockManager*
Avalon/C++		Hérite de *recoverable* ou *atomic* ou *subatomic*
C**	Types tableaux parallèles Fonctions membres parallèles	
Charm++	Classes préfixées par *chare*	
Composites	Abstraction *composite*	
Concurrent C++	Abstraction *process*	
DC++	Instance de *DcThread*	
Dome		Mécanisme de template
DPC++	Types parallèles	
KAROS		Hérite de *ActiveObject* ou *DataObject*
Mentat	Classes préfixées par *persistent mentat*, *sequential mentat* ou *regular mentat*	
Mercury	Instance de *Thread*	Redéfinition possible de *Thread*
Open C++	Constructions syntaxiques pour les mécanismes réflexifs Bibliothèques de méta-classes	
Panda		Hérite de *UserThread*
pC++	Abstraction *Collection*	
Presto	Instance de *Thread*	Redéfinition possible de *Thread*
QPC++	Abstractions *process class* et *process set*	
μC++	Abstraction *uTask*	

Remarque : Les extensions CC++, DoPVM, LITP-C++ et UC++ ne nécessitent aucune déclaration spécifique pour les entités parallèles.

A.3 Utilisation du parallélisme

A.3.1 Modèle tâches concurrentes

	Objet dédié	**Primitives du langage**	**Constructions syntaxiques**
CC++			*par{ }, parfor{ }, spawn*
DC++	Instance de *DcThread*		
DoPVM		*initiate()*	
Mercury	Instance de *Thread*		
Presto	Instance de *Thread*		

Remarque : Pour pC++, aucun mécanisme d'utilisation n'a été rajouté.

A.3.2 Modèle tâches structurées par les données

	Caractéristiques
C**	Fonction membre parallèle définie par l'utilisateur
Composites	Granularité modifiable, fonctions de groupe
Dome	Primitives de base fournies par les templates
DPC++	Primitives de base sur les types parallèles
Mentat	Approche data-flow
pC++	Fonction membre parallèle définie par l'utilisateur
QPC++	Mécanisme de diffusion à un groupe

A.3.3 Modèle objets actifs communicants

	Création	**Communication**	**granularité**
Arjuna	Standard à l'aide d'un mécanisme de proxi	Activation de méthode Synchrone (RPC)	A
Avalon/C++	Standard	Activation de méthode Synchrone	
Charm++	*new_ chare()*	Envoi de message explicite Asynchrone	A
Concurrent C++	*create*	Activation de méthode Synchrone/asynchrone	
KAROS	Standard	Activation de méthode Asynchrone	
LITP-C++	Standard	Activation de méthode Synchrone/asynchrone	
Open C++	Standard	Activation de méthode Synchrone	A
Panda	Standard	Activation de méthode Synchrone	B
QPC++	Standard	Activation de méthode Synchrone/asynchrone	B
UC++	*activenew*	Activation de méthode Asynchrone	A
μC++	Standard	Activation de méthode Synchrone	B

A $\Rightarrow$ 1 objet actif pour 1 processus Unix
B $\Rightarrow$ 1 objet actif pour 1 ou n processus léger(s)

A.4 Contrôle

	Synchronisation dirigée par les activités	**Synchronisation dirigée par les messages**
C**	Gestion groupe logique	
CC++		Futur explicite Attente par nécessité
Charm++		Messages asynchrones
Composites	Gestion groupe arborescente	RdV entre nœuds d'un même arbre
Concurrent C++	Gestion par l'interface	
DC++	Gestion groupe physique	
Dome	Synchronisation de groupe SPMD	
DPC++	Synchronisation de groupe SIMD	
KAROS		Futur implicite Attente explicite
LITP-C++	Condition d'attente	Futur explicite Attente par nécessité
Mentat	Condition d'attente	
Mercury	Entité spécialisée	Continuation
Open C++		Mécanisme réflexif
Panda	Entité spécialisée	
pC++	Gestion groupe logique	
Presto	Entité spécialisée	Futur explicite Attente explicite
QPC++	Gestion par l'interface	Futur implicite Attente par nécessité
UC++	Ordonnancement statique implicite Sérialisation	
μC++	Entité spécialisée	

Remarque : Arjuna et Avalon/C++ implémentent un mécanisme transactionnel.

B Présentation des langages

B.1 Arjuna

Arjuna est développé à l'Université de Newcastle (Angleterre) et est utilisé pour la programmation d'applications distribuées résistantes aux pannes.

B.1.1 Le modèle de programmation

Arjuna offre un mode de programmation distribuée orientée objet de type RPC. Le système est organisé en un ensemble d'objets serveurs, répartis sur les machines. En l'absence d'application, les objets serveurs sont inactifs et stockés en mémoire secondaire. Un objet serveur est réactivé par le système à l'invocation distante d'une de ses fonctions membres. L'activation distante se fait comme une activation locale grâce à un mécanisme de stub (proxy).

Pour activer un objet serveur, le système crée un nouveau processus qui charge l'état de l'objet et sert la requête. Une fois le service effectué, l'état de l'objet est à nouveau stocké et le processus détruit. Un mécanisme de verrou et d'historique est rajouté assurant ainsi l'atomicité des transactions.

B.1.2 Déclaration du parallélisme

Les classes serveurs doivent hériter de la classe *LockManager* et être pré-traitées par le générateur de stub. Ce dernier, à partir de la déclaration C++, génère une classe utilisable côté client et une classe utilisable côté serveur. La classe côté client contient notamment dans ses constructeur et destructeur les appels permettant d'initialiser et de terminer une connexion.

Dans les fonctions membres, le programmeur doit explicitement faire appel aux primitives de gestion des verrous et des transactions (voir Section B.1.4).

B.1.3 Utilisation du parallélisme

Le parallélisme s'utilise de manière relativement transparente et la concurrence provient des activités indépendantes des différents serveurs.

B.1.4 Contrôle

Le mécanisme transactionel est possible grâce aux propriétés de la classe *LockManager*: primitives gérant la persistance et l'atomicité des actions. L'historique est géré par les instances de la classe *AtomicAction*. Arjuna adopte une approche entièrement objet dans laquelle chaque enregistrement d'opération est un objet instance d'une classe système (*LockRecord*, *RecoveryRecord*, etc.).

```
Outcome SpreadSheet::Set(int x, int y, int v) {
   AtomicAction SetAction;
   ...
   SetAction.begin();                          // begin action
   if(setlock(new Lock(WRITE))==GRANTED) {     // try for lock
      ...                                      // work
      if(SetAction.End()!=COMMITTED)           // try to commit action
          ...
    }
   else
      SetAction.Abort();                       // lock refused, abort action
   ...
}
```

B.1.5 Gestion de la distribution de l'application

La programmation avec Arjuna ne permet pas la création dynamique de nouveaux objets serveurs, mais simplement la connexion avec des serveurs déjà créés. De plus, la nature distribuée de l'application n'est pas entièrement transparente dans la mesure où de nombreuses restrictions et contraintes existent :

- Les classes serveurs ne peuvent avoir de membres données publiques ou statiques, de fonctions membre inline ou à paramètres variables.
- La transmission des paramètres et du résultat des fonctions se fait toujours par copie. Le programmeur devant doter ses types de primitives *pack()* et *unpack()*.

B.1.6 Renseignements pratiques

Arjuna comprend un générateur de stubs et utilise le support d'exécution RPC *Rajdoot* [PS88]. Une version peut être obtenue gratuitement à des fins de recherche.

Site www	:	`http://arjuna.ncl.ac.uk/`
Bibliographie	:	[Par, DPS91]

B.2 Avalon/C++

Avalon/C++ est développé à l'Université de Carnegie Mellon (USA) et est utilisé pour la programmation d'applications distribuées transactionnelles résistantes aux pannes.

B.2.1 Le modèle de programmation

Un programme Avalon/C++ est organisé en un ensemble de serveurs encapsulant des objets et exportant des opérations. Les serveurs communiquent entre eux, via les opérations exportées, selon une sémantique RPC.
Au sein d'un serveur, 2 catégories d'objets cohabitent : ceux pouvant être restaurés suite à une panne (objets stables ou persistants) et ceux n'ayant pas cette propriété (objets volatiles).

B.2.2 Déclaration du parallélisme

Un serveur est instance d'une classe ayant hérité directement ou indirectement d'une des classes système *recoverable*, *atomic* ou *subatomic* (voir Section B.2.4).

B.2.3 Utilisation du parallélisme

Chaque serveur est une entité autonome servant des appels distants. Différentes primitives permettent de lancer l'exécution en séquence ou en parallèle d'un ensemble de transactions et d'annuler (*abort()*) ou de confirmer (*commit()*) une transaction.

B.2.4 Contrôle

Le mécanisme transactionnel est possible grâce aux classes système qui garantissent la sérialisation, la consistance et la persistance.

- La classe *recoverable*, assure la persistance. Lors d'une restauration, l'instance est restaurée dans l'état correspondant à la dernière transaction confirmée, le programmeur devant indiquer explicitement les opérations qui modifient l'objet (primitives *pin()*, *unpin()* et *pinning() { }*).

```
class rec_X : public recoverable {
   X_type X;
public:
   void modify() {
         pinning(){/*...modify X...*/}
    }
};
```

- La classe *atomic*, supporte un mécanisme de verrous lecture/écriture (primitive *write_lock()* et *read_lock()*) ainsi que la récupération sur panne.

- La classe *subatomic*, permet au programmeur d'affiner les mécanismes de verrou et de sauvegarde/restauration, afin d'exploiter certaines caractéristiques des objets et augmenter le degré de concurrence.

```
class at_array: public atomic {
   ...
public:
   void store(int index, int value) {
      write_lock();
      ...
   };
   int fetch(int index) {
      read_lock();
      ...
   };
};
```

B.2.5 Gestion de la distribution de l'application

La création de serveurs est dynamique et plusieurs serveurs peuvent être placés sur une même machine. Cette dernière est indiquée à la création du serveur.

Lors d'un appel de fonction membre d'un autre serveur (appel distant), la transmission des paramètres et du résultat se fait par copie.

B.2.6 Renseignements pratiques

Avalon/C++ est implémenté sous la forme d'un pré-processeur et utilise le runtime *camelot* et le système d'exploitation *Mach* [ABG+86] .

Bibliographie : [DHW88]

B.3 C**

C** est un langage data-parallèle à grain moyen développé à l'Université du Wisconsin-Madison (USA). Il est essentiellement orienté vers le calcul massivement parallèle, tout en fournissant un modèle de manipulation des données assez général. Le langage se place dans une optique de mémoire partagée.

B.3.1 Le modèle de programmation

Le parallélisme est introduit par des structures de donnés manipulables de manière concurrente, les *agrégats*. Ces structures de données parallèles autorisent le parallélisme imbriqué, c'est-à-dire qu'elles peuvent contenir d'autres structures de données parallèles et que le grain d'exécution s'ajuste en conséquence.

Un agrégat (qui conserve une structure matricielle) peut être découpé dans une de ses dimensions; ses éléments peuvent être accédés de manière réellement concurrente.

B.3.2 Déclaration du parallélisme

Les agrégats (types tableaux parallèles) sont déclarés par l'ajout de crochets à la fin de la définition d'une classe :

```
class matrix { float value; } [] [] ;           // dimension 2, unknown size
class small_matrix { float value; } [5] [5] ;   // dimension 2, size 5x5
```

B.3.3 Utilisation du parallélisme

Le constructeur d'un agrégat initialise tous ses éléments, contrairement à celui d'un tableau, dans lequel celui de chaque élément doit être invoqué.

L'utilisateur peut écrire des fonctions membres d'agrégats ou des fonctions membres parallèles.

- Les fonctions membres d'agrégat encapsulent des opérations globales à tout l'agrégat, mais effectuées de manière séquentielle (par exemple une fonction de réduction, ou une entrée/sortie).
- Les fonctions membres parallèles ont une exécution simultanée, comme leur nom l'indique, sur tous les éléments de l'agrégat. Elles se déclarent par l'ajout du mot-clé *parallel* à la fin de la déclaration de la fonction, après la liste d'arguments :

```
class matrix {
 float checksum() parallel;
 friend transpose (parallel matrix) parallel;
  // one argument must be prefaced by ``parallel'' in a friend function
} [] [];
```

  Tous les arguments d'une telle fonction sont d'abord copiés, puis le calcul est fait avec ces copies, et on met finalement à jour les emplacements auxquels correspondent les copies.

B.3.4 Contrôle

La synchronisation est réalisée lors du calcul lui-même, par les fonctions parallèles qui s'exécutent sur les agrégats. Une fonction parallèle s'exécute simultanément sur chaque élément de l'agrégat, de manière atomique (l'appel s'effectue dans un même instant logique et sans concurrence avec les appels sur les éléments voisins). Il est aussi possible, de manière classique, d'effectuer des réductions sur les résultats d'une fonction parallèle (avec le nouveau mot-clé *return%+*).

L'enchaînement de fonctions (d'exécution parallèle) sur un agrégat est, lui, purement séquentiel, ce qui en permet une interprétation très intuitive; on peut assurer un certain recouvrement calcul/communication au niveau de ces phases d'enchaînement. Par rapport au grain et à la synchronisation, on peut dire que le modèle de programmation est plutôt SPMD que SIMD.

B.3.5 Gestion de la distribution de l'application

Le modèle de mémoire est global et les agrégats sont tous utilisés de manière "séquentielle" (vu de l'extérieur). Le programmeur ne dispose pas de primitive de placement des éléments d'une collection, et par ailleurs, le parallélisme est d'un grain fin.

B.3.6 Renseignements pratiques

Bibliographie : [LRV92]

B.4 CC++ (Compositional C++)

Le langage CC++, développé au California Institute of Technologie (CALTECH, USA), est une extension de C++, tout programme C++ est un programme CC++ et seuls 5 nouveaux mots clés ont été rajoutés.

B.4.1 Le modèle de programmation

Un programme CC++ est organisé en un ensemble *d'objets processeurs* au sein desquels plusieurs tâches s'exécutent.

B.4.2 Déclaration du parallélisme

Aucune déclaration préalable à l'utilisation du parallélisme n'est nécessaire. Certains mécanismes de distribution et de contrôle sont introduits par le biais du système de typage.

B.4.3 Utilisation du parallélisme

Trois constructions permettent d'utiliser le parallélisme :

- Les blocs parallèles (mot clé *par*) dans lesquels les instructions de plus haut niveau sont exécutées de manière concurrente.

```
par {
      eeny();
      meeny();
}
```

- Les boucles parallèles (mot clé *parfor*) où chaque itération crée une tâche qui exécute les instructions du bloc. Il y aura donc autant de tâches que de tours de boucle.

```
parfor (int i=0,j=10; i < 10 ; i++,j--) {
      eeny(i);
      meeny(j);
}
```

- Le parallélisme non structuré, avec la possibilité de lancer en parallèle l'exécution d'une expression CC++ (mot-clé *spawn*). La tâche initiale et la tâche créée poursuivent de manière indépendante.

```
spawn take_off(i);
```

B.4.4 Contrôle

Les instructions suivant un bloc *par* ou *parfor* ne sont exécutées que quand l'ensemble des tâches du bloc s'est terminé. Deux autres mécanismes permettent de synchroniser les tâches :

- Les variables de synchronisation (mot clé *sync*), qui peuvent être de n'importe quel type et peuvent être vues comme une variable future. A la création, elles n'ont pas de valeur et toute tentative de lecture est alors bloquante. Une fois initialisées, les tâches en attente sont débloquées, et la variable se comporte comme une constante.

```
sync int i;       // a sync integer
int * sync ptr;   // a sync pointer to an integer
```

- Les fonctions atomiques (mot clé *atomic*) garantissent que leur exécution se déroulera en exclusion mutuelle. Une fonction atomique définit donc la granularité d'entrelacement à l'exécution. Afin de conserver cette sémantique, certaines restrictions sont imposées aux fonctions atomiques (qui ne peuvent, par exemple, accéder à des variables de synchronisation, etc.).

```
class Queue {
public:
     atomic enqueue(QueueElement &);
     atomic QueueElement dequeue();
};
```

B.4.5 Gestion de la distribution de l'application

La notion d'espace d'adressage correspond au concept d'objet processeur. Un objet processeur est une instance d'une classe quelconque déclarée comme étant globale (mot clé *global*). Une classe globale est transformée à la compilation en exécutable. A l'instanciation, le programmeur précise la machine cible et la localisation du code exécutable.

Une tâche s'exécutant au sein d'un objet processeur accède de manière efficace aux données définies ou allouées au sein de cet objet. Le partage de données entre objets processeurs se fait en déclarant les pointeurs comme étant globaux (mot clé *global*).

```
int * global iptr;    // a global pointer to an integer

global class Test {   // a global class
public:               // instance = processor object
     int foo();
};
```

B.4.6 Renseignements pratiques

CC++ est implémenté sous la forme d'un pré-processeur et utilise le runtime *nexus* [FKOT94, FGT94, FKT94b, FKT94a]. Une version gratuite de CC++ est disponible.

Site www	:	`http://www.compbio.caltech.edu/`
Site ftp	:	`cs.caltech.edu`
Bibliographie	:	[CK92a, CK92b, CCK93]

B.5 Charm++

Charm++ est développé à l'Université d'Illinois (USA). Le parallélisme se programme explicitement à partir de quelques extensions à C++.

B.5.1 Le modèle de programmation

Un exécutable Charm++ peut être vu comme un ensemble d'entités actives, réparties sur des machines, et communiquant par envoi de messages non bloquants. Charm++ distingue les objets séquentiels des objets parallèles, et le modèle d'exécution est basé sur l'envoi de message. Le partage de données entre objets parallèles est explicite et est introduit à travers des objets particuliers. Charm++ offre la possibilité de réaliser du partage de charge dynamique.

B.5.2 Déclaration du parallélisme

Une entité active est instance d'une catégorie de classes : les classes *chare*. Une classe *chare* peut comprendre :

- Des membres privés (fonctions ou données).
- Des *points d'entrées* qui sont l'interface avec les autres objets *chare*. Un point d'entrée peut être vu comme une méthode accessible à distance et possédant un unique paramètre qui est le message (type *message*).

```
message BooleanObject { Boolean convergeValue;};

chare class Receiver {
entry:
      virtual ReceiveResult(BooleanObject *result);
private:
      int foo();
};
```

Une catégorie particulière de classes *chare* est la catégorie *branched chare*, qui à l'instanciation, crée un représentant sur chaque processeur. L'objet *branched chare* accédé est toujours l'instance locale. Une classe *branched chare* peut avoir des fonctions membres publiques, leur activation est bloquante.

B.5.3 Utilisation du parallélisme

Le parallélisme découle de la distribution des instances *chare* sur l'ensemble des machines et de la communication asynchrone. L'envoi de message a une syntaxe propre, proche de l'activation de fonctions membres :

```
x=>foo(Msg);
```

La création des instances *chare* se fait à l'aide de la primitive *new_ chare()*.

B.5.4 Contrôle

L'appel d'un point d'entrée d'un objet *chare* est non bloquant, et le code associé est exécuté de façon atomique. De plus, un objet *chare* sérialise le service des requêtes. Plusieurs politiques sont disponibles pour la gestion des files de messages en attente de service (FIFO, priorité, etc.).

B.5.5 Gestion de la distribution de l'application

A la création d'un objet *chare*, le programmeur peut spécifier la machine cible, ou il peut laisser le système effectuer le placement en fonction d'une politique dynamique de gestion de charge. Un objet *chare* a son propre espace d'adressage et communique avec les autres objets *chare* par envoi de messages.

Charm++ interdit les membres de données statiques et les variables globales ne peuvent être de n'importe quelle classe. Différents modes prédéfinis de partage de données entre les objets *chare* sont disponibles :

- Les objets en lecture seule (*read-only objects*).
- Les objets initialisés une seule fois (*write-once objects*).
- Les objets accumulateurs (*accumulator objects*) qui possèdent uniquement 2 méthodes *add()* et *combine()* définies par le programmeur.
- Les objets (*monotonic*) qui peuvent être mis à jour par une opération idempotente, commutative et associative fournie par le programmeur. L'accès en lecture ne retourne pas nécessairement la dernière valeur écrite.

B.5.6 Renseignements pratiques

Charm++ est implémenté sous la forme d'un pré-processeur et utilise un runtime propriétaire. Une version gratuite de Charm++ est disponible.

Site www	:	`http://charm.cs.uiuc.edu/`
Site ftp	:	`cs.uiuc.edu:pub/CHARM`
Bibliographie	:	[KK93]

B.6 Composites

Le langage Composites est développé à l'Université du Delaware (USA), et introduit un modèle d'agrégat à base d'arbre, à parallélisme imbriqué et à granularité variable. Le langage est orienté vers la programmation de systèmes distribués, spécialement les réseaux de stations de travail.

B.6.1 Le modèle de programmation

Le langage s'appuie sur un modèle de programmation dans lequel les tâches sont structurées par les données tout en introduisant des échanges explicites entre tâches parallèles.

Le programmeur définit des agrégats qui s'agencent de manière séquentielle et encapsulent toute exécution parallèle. Les objets qui constituent ces agrégats sont exécutés de manière parallèle, sous le contrôle de la structure en forme d'arbre dont ils constituent les feuilles. Cette structure autorise également un parallélisme imbriqué (un noeud parallèle recevant une instruction parallèle la répercute aussi sur ses fils). Les objets contenus dans l'arbre peuvent aussi communiquer entre eux sous forme d'envoi de message explicite.

B.6.2 Déclaration du parallélisme

Le programmeur définit tout d'abord l'arbre comme structure de base :

```
composite Tree {
    int operation1(int argument);
    int operation2(float argument);
}
```

Les noeuds d'un arbre *composite* peuvent être de types différents. Ils sont définis comme :

```
node Tree::anode {
                                                    // some data structure
    anode(int value) { ... };                       // constructor
    int operation1(int argument) { return ... };
    int operation2(float argument) { return ... };
};
```

B.6.3 Utilisation du parallélisme

Des opérations préfixées permettent de sélectionner les objets qui se trouvent dans l'arbre et qui répondent à une certaine condition, ils recevront seuls le message :

```
   [*]children.op(); // all the elements of the aggregate
   [3]children.op(); // element #3 of the aggregate
   [[*]children.booleanop()]children.op();
      // all the elements that comply to condition booleanop
```

L'utilisateur dispose d'une pseudo-variable particulière, symbolisée par le caractère #; elle est passée en argument d'un message à un élément fils d'un noeud, et sera remplacée par son numéro d'index au niveau du noeud (premier fils, deuxième, etc.)

B.6.4 Contrôle

La structure d'arbre offre un premier niveau de contrôle, séquentiel à chaque noeud de l'arbre. Les appels préfixés permettent de sélectionner seulement certains éléments pour effectuer une opération. De plus, la présence de plusieurs niveaux dans l'arbre permet d'utiliser le parallélisme imbriqué, c'est à dire de recouvrir les calculs et les communications par un *pipelinage* des messages.

Il est possible de contrôler la granularité du parallélisme de l'arbre, ce qui permet de faire varier l'imbrication (transmissions de messages, etc.). La primitive *->internal()* permet de transformer une feuille en un nouveau nœud interne, en redistribuant ses éléments sur les nouvelles feuilles.

```
node DistributedList::leaf {
   LinkedList local;
   int numel;

   void insert(string key, int x) {
      if (numel > MAXEL) {
         ->internal();
         [rand()]children->insert(key, x);
      }
      else
         local.insert(key, x); numel++;
   }
};
```

Les nœuds d'un arbre *composite* peuvent altérer son flot de contrôle par un rendez-vous explicite réalisé par les primitives *send()* et *receive()* (toutes deux bloquantes); elles peuvent être utilisées aussi bien pour synchroniser deux exécutions que pour communiquer des valeurs.

B.6.5 Gestion de la distribution de l'application

Les variables globales sont refusées par le modèle, ce qui peut poser problème en C++, avec certaines applications. Cependant, la répartition, comme le parallélisme, est transparente vu de l'extérieur des agrégats (qui l'encapsulent complètement).

B.6.6 Renseignements pratiques

Bibliographie : [CP94]

B.7 Concurrent C++

Le langage Concurrent C++ est développé au laboratoire Bell AT&T (USA). Il représente une synthèse des langages C++ et Concurrent C [GR86], ce qui permet d'obtenir un langage qui ait à la fois des concepts d'*abstraction de données* et de *programmation parallèle*.

B.7.1 Le modèle de programmation

La concurrence est introduite dans le langage à l'aide de la notion de processus qui vont s'exécuter en parallèle. Les différents processus d'une application pourront interagir entre eux à l'aide de transactions. Une transaction est associée à un processus (le serveur), et peut être vue comme un service appelé par un autre processus (le client). On distingue deux types de transactions : bloquantes ou non-bloquantes.

B.7.2 Déclaration du parallélisme

La définition d'un processus est constitué de deux parties : sa spécification et son implémentation. Le type *process* ainsi que les deux mots clés *spec* et *body* ont été rajoutés au langage. Le processus va pouvoir être invoqué par le biais de transactions, qui peuvent être vues comme des points d'entrée du processus. Ces transactions sont introduites dans la définition du processus par le mot clé *trans*, et dans l'implémentation par *accept* ou *select*.

```
process spec consumer() {
    trans void put(int);
}
process body consumer() {
    int c;
    for (;;) {
       accept put (a)
          c = a;
    if (c == EOF) break;
    putchar(c);
    }
}
```

La déclaration du processus se fait alors de la manière suivante :

```
consumer c;
c = create consumer();
```

B.7.3 Utilisation du parallélisme

Une fois le processus créé, l'appel de transaction sur ce processus se fait avec la même syntaxe qu'un appel de méthode.

```
c.put('g');
```

B.7.4 Contrôle

La synchronisation de tâches n'existe pas de manière explicite dans le langage, mais on peut la programmer à l'aide de transactions bloquantes. En fait, une transaction bloquante peut être utilisée uniquement pour établir une synchronisation (une transaction peut n'avoir aucun paramètre et ne rien renvoyer).

Il est possible de préciser une priorité à la création d'un processus (*by priority(p)*) et de la changer en cours d'exécution à l'aide de primitives du langage. Les transactions peuvent avoir des gardes ou être temporisées, et un processus peut également se mettre en attente sur plusieurs transactions.

```
accept get_lock(id) suchthat (isfree(id)) {
...
}
accept diskop(cyl, ...)  by (abs(cyl - curpos)) {
...
}
```

B.7.5 Gestion de la distribution de l'application

La première version du langage ayant été implementée sur une machine mono-processeur, aucune restriction ne s'impose pour la gestion de données.

B.7.6 Renseignements pratiques

Concurrent C++ est écrit à l'aide d'un préprocesseur qui génère du C++.

Bibliographie : [GR88]

B.8 DC++ (Distributed C++)

DC++, développé à l'Université d'Utah (USA), offre un mode de programmation explicite du parallélisme et est orienté vers les systèmes faiblement couplés. DC++ applique les concepts introduits dans Concurrent Scheme [KS90].

B.8.1 Le modèle de programmation

Un exécutable DC++ est organisé en un ensemble de *domaines* et en un ensemble de tâches s'exécutant au sein de ces domaines.

B.8.2 Déclaration du parallélisme

Un *domaine* est à la fois un espace logique d'adressage et une entité de contrôle. A la création, le programmeur précise le processeur physique où allouer le domaine. Le domaine peut être vu comme un processeur abstrait.

```
DcDomain* domain[num_domains_needed];
for(i=0; i < num_domains_needed; i++)
   domain[i]=new DcDomain(i);
```

Les instances des classes ayant hérité directement ou indirectement de la classe système *DcGateway* peuvent être créées sur des domaines distants. Le domaine cible est précisé à la création, par défaut l'objet est créé localement. L'activation d'une de leurs méthodes se fera alors selon une sémantique RPC (voir Section B.8.5). Ces fonctions accessibles à distance sont considérées comme étant des *points d'entrée* du domaine.

```
class Philosopher :  public DcGateway {
   Fork left, right;
public:
   Philisopher(int i, Fork l, Fork r);
   void Live();
};
```

B.8.3 Utilisation du parallélisme

Le parallélisme est obtenu en créant explicitement des tâches (instances de *DcThread*). Une tâche correspond à l'exécution d'une méthode et se déroule au sein du domaine où est situé l'objet.

```
Philosopher *tab_philo[num_phil];
for(i=0;i< num_phil;i++)
   new DcThread(*tab_philo[i],Philosopher::Live);
```

B.8.4 Contrôle

Une seule tâche est active à la fois au sein d'un domaine. L'ensemble des tâches en attente d'exécution est géré par une file FIFO.

Les tâches se synchronisent implicitement lors de l'activation d'un point d'entrée. Le programmeur peut aussi associer à chaque point d'entrée une barrière de synchronisation sous la forme d'un drapeau doté d'une file d'attente (primitive *DcMakeDelayQueue()*). Il contrôle explicitement l'accès aux fonctions membres en *ouvrant* (*DcDQOpen()*) ou en *fermant* (*DcDQClose()*) le drapeau.

```
Fork::Fork(int i) {
   self=i;
   DcMakeDelayQueue(Fork::PickUpFork);  // creation of a delay queue
                                        // associated to PickUpFork()
}
```

```
void Fork::PickUpFork(int phil) {
   DcDQClose(Fork::PickUpFork);      // closing the delay queue
}                                    // associated to PickUpFork()

void Fork::PutDownFork(int phil) {
   DcDQOpen(Fork::PickUpFork);       // opening the delay queue
}                                    // associated to PickUpFork()
```

Une tâche accédant à la valeur de retour d'une autre tâche est bloquée tant que cette valeur n'est pas encore disponible.

B.8.5 Gestion de la distribution de l'application

DC++ supporte un nommage global inter-domaines pour les instances des classes descendantes de la classe système *DcGateway*. Les références à ces instances peuvent être transmises entre domaines, toute autre donnée sera passée par copie. A la création d'une instance d'une classe descendante de *DcGateway*, le programmeur précise le domaine de création. L'activation d'une fonction membre sur ce type d'objet est gérée par un RPC, s'il est situé sur un autre domaine.

DC++ permet, en héritant de la classe système *DcValue*, de passer certaines données par *copie profonde* lors d'une transmission inter-domaines par copie (paramètres ou résultat de fonctions).

B.8.6 Renseignements pratiques

DC++ est implémenté sous la forme d'un pré-processeur et utilise un runtime propriétaire. Une version gratuite de DC++ est disponible.

Site ftp	:	`cs.utah.edu:pub/dc++`
Bibliographie	:	[CKS93c, CKS93a, Car94, CKS93b, Car93b, Car93a]

B.9 Dome

Le langage Dome est développé à l'Université Carnegie Mellon (USA). Il s'agit d'un langage spécialement orienté vers les réseaux de machines hétérogènes, aussi bien de stations de travail que de calculateurs; il suppose une architecture matérielle à mémoire distribuée. Le langage fournit aussi des primitives pour implémenter des mécanismes de tolérance aux pannes, et éviter de recommencer intégralement un calcul.

B.9.1 Le modèle de programmation

Le langage, qui est assez orienté vers des applications de calcul, propose une programmation SPMD du parallélisme (un programme sur des données multiples).
Les objets encapsulent des données, distribuées sur les processeurs, et sur lesquelles on effectue des traitements en parallèle.

B.9.2 Déclaration du parallélisme

Le langage fournit des classes prédéfinies comme des vecteurs parallèles : ces classes permettent, par un mécanisme de templates, de transformer des objets séquentiels en objets répartis :

```
#include "dome.h"
#include "dVector.h"
#include "domeobj.h"
...
// each of these vectors will be spread across all processes :
dVector<double> v1(tdome, vector_size);
dVector<double> v2(tdome, vector_size);
dVector<double> prod(tdome, vector_size);
```

Ces classes peuvent être étendues en redéfinissant l'empaquetage pour la couche de communication (PVM en l'occurrence).

B.9.3 Utilisation du parallélisme

Les objets répartis sont accessibles par des opérateurs surchargés ou de nouvelles fonctions (somme globale par exemple) :

```
...
// compute a product of v1 and v2, using the overloaded vector product operator :
prod = v1 * v2;

// compute the sum of all the elements of dVector<double> prod :
double dot_product=prod.gsum();
```

B.9.4 Contrôle

La synchronisation est également SPMD et s'effectue au niveau des opérations globales. On ne dispose que d'une synchronisation dirigée par les données, ceci pour simplifier l'écriture du programme. Cependant, l'aspect tolérance aux pannes assure également une certaine synchronisation dans les points de reprise des différentes tâches si l'on veut pouvoir assurer une restauration cohérente d'un calcul interrompu.

B.9.5 Gestion de la distribution de l'application

Le partitionnement et le placement des données est réalisé automatiquement à l'intérieur de chaque objet; l'utilisateur peut cependant préciser la fragmentation de ses données (il utilisera des classes différentes). Tous les processeurs sont utilisés par défaut, à moins que l'utilisateur ne restreigne le nombre de tâches démarrées.

L'équilibrage de charge est réalisé par une migration des données automatique, à l'occasion de points de rééquilibrage réguliers, dont la fréquence est fonction du nombre d'instructions adressées à chaque objet. Ce mécanisme peut également être modifié par l'utilisateur.

B.9.6 Renseignements pratiques

Dome est implémenté en C++ et utilise PVM pour la communication et la distribution des données (les fragments d'objets sont affectés à des tâches PVM).

Bibliographie : [BSS94]

B.10 DoPVM

DoPVM est une interface développée à l'Université d'Emory (Atlanta, USA) au dessus du langage C++ et de PVM [Sun90]. C'est un environnement distribué orienté objet, qui, à l'aide de macros et de librairies, permet à l'utilisateur de programmer de façon distribuée en utilisant des objets partagés.

B.10.1 Modèle de programmation

Le modèle de base est celui de processus communicants à l'aide d'objets partagés. Différents modes de programmation sont accessibles au dessus de ce modèle. Les plus traditionnels sont :

- Le mode "mémoire partagée" : fourni directement par l'interface, il est construit au dessus du système d'échange de messages de PVM.
- Le mode "partage de travail", où un système de files d'attente permet de régler des problèmes de synchronisation, d'ordonnancement ou de partitionnement.
- Le mode "pipeline" : ce modèle est utilisé lorsqu'une application peut être fragmentée en plusieurs niveaux, chacun d'eux étant pris en charge par un ou plusieurs processus dédiés. Les données passent du niveau n au niveau $n+1$ à travers un pipeline.

B.10.2 Déclaration du parallélisme

Il n'y a pas de déclaration du parallélisme dans ce langage, les seules entités parallèles sont des processus qui nécessitent seulement une création (voir Section B.10.3). Seuls les objets partagés doivent être déclarés.

Cette déclaration va se faire en utilisant des primitives de base du système et en héritant d'une classe particulière gérant le partage de données (*shared*). Une instance d'une telle classe va alors être référencée par un identificateur unique, qui sera par la suite utilisé par les différents processus de l'application pour pouvoir accéder à l'instance.

B.10.3 Utilisation du parallélisme

La création de nouveaux processus se fait à l'aide de la primitive `initiate()`. On lui donne en paramètre le nom du processus à exécuter, et éventuellement l'architecture ou la machine cible sur laquelle celui-ci doit s'exécuter. Pour pouvoir communiquer entre eux via des objets partagés, les processus auront besoin de se connecter à une "session DoPVM" à l'aide de la primitive `begin_do()`.

```
main() {
   begin_do(``session1);            // registering
   initiate(``worker'', ``SUN4'');  // create a process ``worker''
                                    // on a SUN4
   ...
   end_do();
}
```

L'utilisation d'un objet partagé ou d'une file d'attente distribuée se fait de la même manière qu'un objet séquentiel. Il suffit juste de le déclarer, et de lui assigner un identifiant pour pouvoir le manipuler comme un objet séquentiel.

```
test x;           // Class test inherits from shared
x.set_id(ID_1);   // binding to the system object
                  // whose ID is ID_1
x.foo(4);         // Call method on object x
```

Une file d'attente distribuée est une instance de la classe *fqueue* (FIFO) ou *lqueue* (LIFO). Elle est notamment utilisée pour des applications où l'on doit partager le travail entre différents processus.

B.10.4 Contrôle

Il n'existe qu'une seule copie d'un objet partagé et les accès concurrents en écriture vont être sérialisés dans un ordre indéterminé. Toutefois, un processus aura la possibilité de bloquer l'accès (lecture et/ou écriture) à un objet partagé.

Pour gérer la consistence de données, chaque objet partagé dispose en interne d'un statut qui lui permet de savoir s'il doit être mis à jour, si son accès ne nécessite pas une communication, ou s'il peut être modifié.

B.10.5 Gestion de la distribution de l'application

Chaque processus dispose d'un espace mémoire privé et d'un démon pour gérer la mémoire partagée.

A noter que les objets peuvent être partagés sur des machines ayant des architectures différentes.

B.10.6 Renseignements pratiques

Une version gratuite de cette interface (sources et exemples) est disponible par ftp anonyme.

Site ftp	:	`mathcs.emory.edu:/pub/skip`
e-mail	:	`skip@mathcs.emory.edu`
Bibliographie	:	[HS93]

B.11 DPC++ (Data Parallel C++)

DPC++ est une extension data-parallèle de C++ développée au LPAC (Londres, Angleterre). Il vise à offrir en C++ les extensions de tableaux disponibles en Fortran 90; il est ciblé pour les systèmes MasPar.

B.11.1 Le modèle de programmation

Une application DPC++ est décrite par ses structures de données parallèles : le programmeur manipule des tableaux parallèles, les éléments de ces tableaux n'étant pas eux-mêmes des objets parallèles.

B.11.2 Déclaration du parallélisme

Un type parallèle (tableau à n dimensions) est formé en ajoutant *parallel_* devant un type séquentiel (trois types séquentiels sont rajoutés : des booléens, *logical*, et des nombres complexes, *complex* et *double_ complex*).

```
parallel_int b(16,4);       // declare a parallel array of 16x4 integers
                            // the limit is 7 parallel dimensions ...

parallel_short d(64)[10];   // d is a serial array of 10 objects
                            // each of which is a parallel vector of 64 elements
```

B.11.3 Utilisation du parallélisme

Le langage fournit des primitives de manipulation globale des tableaux, aussi bien arithmétiques (+, +=, =, *sin()*, ...) ou logiques (&&, ||, ...) que de sélection (technique de masquage par des vecteurs de booléens), de réduction (préfixes, ...), de stockage, de décalage ou de permutation.

```
float a;
parallel_float b(1024);
...
a=sum(b);                 // scalar result
```

B.11.4 Contrôle

Une synchronisation intervient implicitement à l'exécution de chaque instruction qui travaille sur des tableaux parallèles. Ce modèle de synchronisation rattache DPC++ aux langages à sémantique SIMD.

B.11.5 Gestion de la distribution de l'application

Les tableaux parallèles sont placés sur un maximum de processeurs sans qu'aucune intervention explicite de l'utilisateur soit nécessaire (ni possible, d'ailleurs, mis à part grouper des blocs de données).

B.11.6 Renseignements pratiques

L'implémentation du langage a été réalisée sur machine Maspar. Un programme étendu est compilé de C++ à C où sont utilisées des primitives spécifiques à la machine (bibliothèques MPL).

Bibliographie : [Gil92]

B.12 KAROS

KAROS est une extension parallèle de C++ définie au CEA à Saclay (France). Elle est orientée vers les applications distribuées et comprend notamment un protocole (ACS) pour la gestion des transactions.

B.12.1 Le modèle de programmation

KAROS a un modèle d'objet hétérogène : les objets actifs, les *ActiveObject*, peuvent être référencés à distance et s'échanger des messages; les objets passifs, les *DataObjects*, sont locaux aux *ActiveObjects*.

B.12.2 Déclaration du parallélisme

Le programmeur définit ses classes par héritage soit d'*ActiveObject*, soit de *DataObject*:

```
class ConferenceManager :  public ActiveObject {
    ...
  method(query) {
    ...
    reply(...);
  };
};
```

On notera que les fonctions membres, susceptibles d'être activées à distance, sont déclarées de manière différente de C++ (au moins dans la version présentée), et retournent un résultat par la primitive *reply()*.

B.12.3 Utilisation du parallélisme

La communication s'effectue par envoi de message asynchrone (par exemple un *Call()*, voir B.12.4) :

```
class ConferenceManager :  public ActiveObject {
  method(organize_conference) {
    ...
    reply(result); // if success, returns a result
                   // or possibly fail();
  };
};
...
res=Call(cm, ConferenceManager, organize_conference) << Arg1 << ...;
     // asynchronous message passing
...  // keep on working while the message is being sent
```

B.12.4 Contrôle

Le langage permet d'utiliser toute variable normale comme futur implicite. Il est possible d'interroger l'état d'un futur (primitive *Awaited()*) et d'attendre le résultat effectif

(*Wait()*). Ce mécanisme permet aussi, en KAROS, de rendre compte des échecs d'une invocation (primitive *Failure()*) :

```
res=Call(...);
...
Bool arrived = Awaited(res); // testing the result arrival
...
Bool failed = Failure(res); // testing for a service failure
```

Le protocole ACS (*Apply*, *Call*, *Send*) définit trois genres de messages asynchrones, en commençant par le plus sûr :

- Si un message *Apply* provoque une erreur au niveau du serveur (l'appelé), le client (l'appelant) échoue aussi.
- Si un message *Call* échoue au niveau du serveur, le client en est averti (test avec *Failure()*) et peut choisir d'échouer ou de poursuivre.
- Un message *Send* n'implique aucune réponse de la part du serveur, ni d'accusé de bon fonctionnement.

Le langage fournit aussi une classe *AtomicObject* qui doit être héritée par les objets passifs pour assurer leur atomicité.

B.12.5 Gestion de la distribution de l'application

Les seuls objets qui peuvent être distribués de manière transparente sont les *ActiveObjects*.

B.12.6 Renseignements pratiques

Un prototype a été développé en C++, avec des techniques de surcharge d'opérateurs et des macros. Il fonctionne sur station de travail Sun, avec Unix BSD 4.0.

Bibliographie : [GCLR92]

B.13 LITP-C++

Hayssam Saleh et Philippe Gautron (Rank Xerox France et Université Paris VI - LITP) proposent un modèle de contrôle concurrent pour C++ qui repose sur la mise en attente des requêtes non traitables immédiatement. Une bibliothèque fournit également des appels de méthodes unifiés aux communications asynchrones ainsi que des futurs (utilisables par intanciation explicite au moyen de templates). Le langage est constitué pour une part d'une extension de C++ (syntaxique et sémantique), et pour une autre part d'une bibliothèque qui complète et encapsule les concepts parallèles.

B.13.1 Le modèle de programmation

LITP-C++ offre un modèle homogène d'objets actifs communicants, avec plusieurs flots d'exécution possibles dans un objet. Le langage supporte bien sûr les communications asynchrones entre ces objets actifs.

Le parallélisme s'exprime à l'intérieur des objets parallèles (au niveau des méthodes), de manière explicite :

- Les méthodes "const", qui ne modifient pas l'objet, peuvent s'exécuter simultanément avec d'autres méthodes *const* (à noter que cette parallélisation est implicite dans le cas d'une réutilisation de code séquentiel).
- Méthodes pouvant modifier l'objet : on peut explicitement autoriser leur exécution avec une autre méthode par une déclaration commune avec la même condition de mise en attente.

B.13.2 Déclaration du parallélisme

Tous les objets sont actifs, leurs méthodes se trouvant par défaut en exclusion mutuelle (sauf pour les méthodes const qui ne modifient pas les donnés de l'objet). Le programmeur n'a donc pas de déclaration particulière pour introduire le parallélisme (il agit plutôt au niveau du contrôle parallèle des objets).

B.13.3 Utilisation du parallélisme

Les communications asynchrones sont encapsulées dans la classe template *Asynchronous-Call* de la bibliothèque :

```
ConQueue *aConQueue = new ConQueue; // instantiates a concurrent queue

AsynchronousCall<ConQueue> asyncQueue(aConQueue);
    // (1) instantiates a new class type :  AsynchronousCall<ConQueue>
    // (2) instantiates this class :  asyncQueue
    // (3) initializes the instance with aConQueue

Item item; // arbitrary item
aConQueue->putback(item); // synchronous call
asyncQueue->putback(item); // asynchronous call unified with method call
```

B.13.4 Contrôle

La synchronisation porte sur la réception des messages (synchrone). Les requêtes sont synchronisées par un mécanisme de mise en attente conditionnelle : l'exécution d'une requête sur une fonction membre de l'interface de l'objet parallèle est retardée (mot-clé *delay*) si une des fonctions membres booléennes qui lui sont attachées s'évalue à vrai. La condition d'attente peut être paramétrée par un argument de la méthode dont on contrôle l'exécution.

```
class ConQueue :  public FixedQueue {
    delay:
     full() delay putback(Item);
     empty(int num) delay putback(num, Item&);  // method parameter
};
```

Il est possible de favoriser l'exécution de certaines méthodes par rapport aux autres fonctions membres grâce à un mécanisme de priorités (opérateurs $>$ et $<$, avec un ordre partiel).

La bibliothèque permet d'introduire des futurs explicites par un mécanisme de templates (classe *Future*). L'utilisation de l'opérateur de coercion sur un futur entraîne une "attente par nécessité".

```
class AsynchronousQueue :  public ConQueue {
    public:
      getfront(Future<Item>&); // result of asynchronous call
    delay:
      empty() delay getfront(Future<Item>&);
};
    ...
// allocation of resources :
AsynchronousQueue *aQueue = new AsynchronousQueue;
AsynchronousCall<AsynchronousQueue> asyncQueue(aQueue);
Future<Item> result;

asyncQueue->getfront(result); // asynchronous call
...
Item it = result; // blocking call of the coercion operator
```

B.13.5 Gestion de la distribution de l'application

Les objets sont référençables de manière globale dans le système.

B.13.6 Renseignements pratiques

Le langage a été implémenté sur un T.Node à 16 transputers, sous le système d'exploitation Helios.

Bibliographie : [SG91]

B.14 Mentat

Mentat est un environnement pour la programmation distribuée, développé à l'Université de Virginie (USA), comprenant un langage de programmation et un environnement d'exécution (*runtime-system*).

B.14.1 Le modèle de programmation

Dans le système Mentat, le programmeur prend les décisions concernant le partitionnement des données et les traitements qui devront être exécutés en parallèle. Le compilateur se charge des aspects communication, synchronisation et dépendances de données.

B.14.2 Déclaration du parallélisme

La granularité du parallélisme est l'exécution de méthodes, ce parallélisme intervient à deux niveaux :

- Une méthode est exécutée de manière concurrente dès que ses paramètres sont disponibles. Le compilateur se charge de gérer les synchronisations dues aux dépendances de données. L'ordre d'activation des fonctions n'est alors plus respecté.
- L'exécution du code d'une méthode peut lui même contenir du parallélisme.

C'est au programmeur d'indiquer les méthodes qui devront être exécutées en parallèle : ce sont les méthodes des *classes Mentat*. Un objet Mentat possède un espace d'adressage propre, une tâche chargée du contrôle et un nom global.

Parmi les classes Mentat, on distingue :

- La classe *persistent*, qui conserve un état entre les appels.
- La classe *sequential*, qui conserve un état entre les appels et respecte l'ordre d'activation (l'ordre d'exécution est identique à l'ordre d'activation).
- La classe *regular*, qui ne conserve pas d'état entre les appels ; le système pouvant instancier un nouvel objet à chaque appel.

```
persistent mentat class p_matrix {
// private members
public:
   void initialize (matrix *);
   matrix* mpy(matrix *);
};
```

B.14.3 Utilisation du parallélisme

Le parallélisme est implicite dans la mesure où l'activation d'une fonction d'une instance Mentat donne lieu à une exécution concurrente. Les résultats des fonctions doivent être retournés à l'aide de la primitive *rtf()* (remplace le *return()* séquentiel). Cette primitive permet de retourner un résultat *attendu* (résultat d'une autre activation concurrente en cours) et gère les continuations.

B.14.4 Contrôle

La synchronisation est en général entièrement transparente dans la mesure où :

- Un objet Mentat ne sert qu'une méthode à la fois (sérialisation).
- Les données communes doivent être des objets Mentat.
- Le compilateur gère automatiquement les dépendances des données.

Le programmeur peut aussi associer une condition booléenne aux méthodes d'une classe Mentat. L'objet ne servira une demande à cette méthode, que si la condition est vérifiée.

```
mselect {
            :  maccept int func1(int arg1);
            break;
[delay>0]   :  maccept int func2();
            break;
}
```

B.14.5 Gestion de la distribution de l'application

Un objet Mentat constitue un espace d'adressage indépendant. A la création d'une instance (fonction membre *create()*), le programmeur peut préciser la machine cible (numéro de machine, *même machine que*).

```
mat1.create();                  // on the current host
mat1.create(another_M_object);  // colocate the object
mat1.create(int on_host);       // place on a specific host
```

Le passage de données entre objets Mentat se fait par copie. Lors du passage d'un pointeur, les zones mémoires sont automatiquement copiées. Seules les références à un objet Mentat peuvent être partagées. Une donnée commune devra donc être un objet Mentat.

L'interaction avec un objet Mentat ne peut se faire qu'à travers une méthode. Les classes Mentat ne peuvent avoir d'attribut dans leur section publique ni de membre statique (aucune mémoire commune).

B.14.6 Renseignements pratiques

Une version gratuite de Mentat est disponible.

Site www	:	`http://uvacs.cs.virginia.edu/~mentat/`
Site ftp	:	`uvacs.cs.virginia.edu`
e-mail	:	`mentat@uvacs.cs.virginia.edu`
Bibliographie	:	[TMRG94a, TMRG94b]

B.15 Mercury

Le langage Mercury est développé à l'Université de Rochester (USA). Il s'appuie sur une extension de C++ par l'adjonction de processus légers (threads) non préemptifs, et l'étend par l'ajout de continuations qui permettent de programmer dans un style proche du data-flow. Il est plus particulièrement destiné à la programmation de multi-processeurs à mémoire partagée.

B.15.1 Le modèle de programmation

Mercury (qui s'appuie sur le système Presto) a un modèle à tâches explicites et distinctes de la notion d'objet. L'ajout de *continuations* (voir B.15.3) permet l'enchaînement de synchronisations sur les objets et se rapproche donc d'une structuration par les données.

B.15.2 Déclaration du parallélisme

L'utilisateur déclare une activité comme un objet de type Thread (qu'il devra démarrer par la primitive *start*).

```
Thread *t, *p;
p = new Thread;
t = new Thread(``thr'',1,1024);   // name, id, stack_size
```

B.15.3 Utilisation du parallélisme

Le parallélisme est introduit à deux niveaux :

- Les threads : ils ne sont pas préemptifs, mais peuvent décider de se suspendre. On dispose d'opérations de duplication (*fork*), de jointure (*join*) et de migration (*migrate*) sur ces processus. Chaque thread est associé à un objet qui contient des données. Le langage considère les processus Unix comme des processeurs parallèles virtuels sur lesquels les processus légers sont placés

- Les continuations : elles permettent de décrire des opérations particulières d'enchaînement de threads (optimisé au niveau de la pile, notamment) et de synchronisations. Une continuation désigne son successeur, qui recevra ses données à la fin de son exécution. Une continuation ne peut démarrer son exécution, c'est-à-dire être instanciée en tant que thread à part entière, que si toutes ses dépendances de données sont résolues.

B.15.4 Contrôle

On obtient deux niveaux de synchronisation, qui correspondent aux deux niveaux de parallélisme :

- Synchronisations de threads, par jointure (comme en PRESTO).

- Synchronisations de
  - Continuations : la jointure dépend du type du successeur (continuation, groupe de continuation ou thread).

– Groupes de continuations (jointure en une opération).

De plus, le langage possède des objets spécifiquement dédiés à la synchronisation :

- Verrous (éventuellement bloquants).
- Moniteurs et variables conditionnelles (sections critiques).
- Barrières (attente d'un certain nombre de threads).

B.15.5 Gestion de la distribution de l'application

Deux types de données spécialisés sont introduits[6] :

- Les variables privées, définies par le mot-clé *private_t* (qui peut être omis, il s'agit du mode par défaut).
- Les variables partagées, définies par le mot-clé *shared_t*: elles sont accessibles à tous les processeurs virtuels (processus lourds).

Le placement des threads associés à une même donnée peut être effectué sur le même processeur pour accroitre les performances (object affinity scheduling, que l'on spécifie en dérivant un objet de la classe *uses_OAS*). Il est aussi possible d'indiquer une préférence pour le processeur virtuel qui traitera le thread (mais l'affectation n'est pas assuré, en raison de l'équilibrage de charge assuré par l'environnement) :

```
Thread *q;
q = new Thread((short)1, ``thr'', 1, 1024);
   // preference for virtual processor 1
```

B.15.6 Renseignements pratiques

Le système se compose d'une bibliothèque et d'un ensemble de pré-processeurs et d'outils qui fonctionnent en combinaison avec le compilateur C++ standard. Il nécessite aussi la bibliothèque parallèle Iris. Mercury est disponible par ftp.

Site ftp	:	`ftp://ftp.cs.rochester.edu/pub/u/kthanasi/`
Bibliographie	:	[Kon93]

[6] On notera que tous les objets de Mercury sont dérivés d'une classe Object qui contient des informations de type et des gestionnaires d'erreur.

B.16 Open C++

Open C++ est développé à l'Université de Tokyo (Japon) et consiste d'abord en une extension de C++ par un protocole à méta-objets (MOP). La réflexivité du langage est mise à profit pour introduire le parallélisme dans le langage grâce à une bibliothèque (Object Communities) modifiant la sémantique des constructions habituelles de C++.

B.16.1 Le modèle de programmation

Une application Open C++ se compose d'objets réflexifs, à chacun desquels est associé un méta-objet, et d'objets courants dont les méthodes sont exécutées normalement. Un objet réflexif peut contenir des méthodes normales et des méthodes réflexives; celles-ci sont déroutées vers son méta-objet qui peut appeler la méthode de l'objet ou exécuter un autre traitement.

On peut considérer qu'un objet réflexif associé à un méta-objet dont le comportement est dédié au parallélisme correspond à un objet actif communicant en Open C++. Les méthodes du méta-objet sont exécutées séquentiellement, mais il peut déclencher des activités multiples au niveau de son objet.

Le modèle de la bibliothèque parallèle consiste à définir des communautés d'objets distribués sur plusieurs machines. Les appels de méthodes entre objets actifs sont effectués de manière coopérative au niveau des méta-objets.

B.16.2 Déclaration du parallélisme

Les mécanismes réflexifs sont introduits par annotation en commentaire (//MOP), ce qui est également le cas pour le parallélisme qui est introduit par une bibliothèque réflexive :

```
//MOP reflect class X : M;
     // definition of the class - metaclass association

class X {
  public:
//MOP reflect:
    int a;
    X();
//MOP reflect:
    int func(int); // func(int) will be reflexive
};
    ...
  refl_X objx; // refl_X is the new name of the class
```

Le programmeur doit déclarer à la fois les classes d'objets réflexifs, les méta-objets parallèles qu'il va leur associer, et les méthodes qui seront prises en charge par le MOP et donneront lieu à un appel et à une exécution parallèle. Les métaclasses qui peuvent être associées font partie de la bibliothèque Object Communities (qui constitue un MOP supplémentaire).

B.16.3 Utilisation du parallélisme

L'utilisation est transparente au niveau des objets pour le programmeur, car toute la connaissance sur la distribution se trouve au niveau des méta-objets.

Cependant, les accès à un objet distant qui n'ont pas lieu via un accesseur (fonction membre) doivent être modifiés, par exemple dans le cas de la lecture d'un attribut public (les surcharges d'opérateurs de C++ ne sont pas suffisantes).

```
z=objx.a(); // read the content of a
```

La communication peut être de type point-à-point (RPC) ou de type diffusion à tous les objets de la communauté.

B.16.4 Contrôle

La bibliothèque Object Communities, qui contient les différents types de méta-objets, implémente des mécanismes de transactions et d'appels RPC. Elle se destine essentiellement au collecticiel et aux applications distribuées.

Le méta-objet contrôle la concurrence interne de son objet. Il peut ignorer ou retarder l'invocation d'une méthode jusqu'à ce qu'une condition soit satisfaite par le système.

La classe *OcLockMetaObj* permet d'implémenter des transactions atomiques, mais ne supporte pas les points de retour (rollback). Les services des méthodes sont mis en attente tant que l'exécution est verrouillée; ce verrouillage est réalisé par une méthode du méta-objet qui donne lieu à une diffusion.

B.16.5 Gestion de la distribution de l'application

La définition des objets n'inclut pas de concept distribué : ceux-ci apparaissent seulement dans la définition des méta-objets. Les objets distants peuvent être accédés au travers de pointeurs, le compilateur utilisant une technique de "smart pointers".

Les données partagées sont implémentées par la classe *OcMetaObj* : elles sont répliquées entre les objets qui appartiennent à la même communauté, et leur consistence est conservée par les méta-objets.

B.16.6 Renseignements pratiques

Open C++ est implémenté sous forme d'une extension du compilateur Cfront d'AT&T qui se charge de gérer les annotations réflexives. Le coût d'un appel réfléchi correspond à environ 8 fois le coût d'un appel virtuel C++. L'implémentation de la communication parallèle fait appel aux bibliothèques RPC de Sun.

Site ftp	:	`ftp://utsun.s.u-tokyo.ac.jp/lang/openC++/`
Bibliographie	:	[CM92, Chi93]

B.17 Panda

Panda est développé à l'Université de Kaiserslautern (Allemagne) et se présente comme un sur-ensemble strict de C++. Il introduit non seulement le parallélisme et la distribution mais aussi des mécanismes de persistence des objets et de ramasse-miettes.

B.17.1 Le modèle de programmation

Un programme Panda est organisé en un ensemble d'objets communicants; ceux-ci sont dotés de méthodes qui peuvent être déclenchées par un appel et entraîner l'exécution d'une activité (thread). Tout objet non spécialement protégé peut exécuter simultanément plusieurs méthodes (parallélisme intra-objet).

B.17.2 Déclaration du parallélisme

Le programmeur introduit une notion de parallélisme en héritant de la classe *UserThread*. Après cette déclaration, l'initialisation d'un objet déclenchera en fait l'instanciation d'un objet actif (avec un thread associé).

```
class Increaser :  UserThread {
    ...
};

main {
    incPtr = new Increaser (10);
    ...
};
```

B.17.3 Utilisation du parallélisme

Une méthode *code()* joue le rôle d'un constructeur spécifique pour la partie parallèle de l'objet actif. Elle est exécutée par une nouvelle activité (thread) de celui-ci, et reste active tant que l'objet n'est pas détruit. On unifie ici implicitement appel de méthode et requête à un objet distant grâce à la présence d'un espace d'adressage global (voir Section B.17.5).

B.17.4 Contrôle

Le langage offre des objets synchronisants tels les sémaphores, les sémaphores de lecture/écriture distribués et les signaux ainsi que des moniteurs; ceux-ci permettent de restreindre l'accès concurrent aux objets. Le niveau d'exclusion est celui des méthodes d'un objet actif (ce qui correspond à des objets à parallélisme interne possible).

B.17.5 Gestion de la distribution de l'application

Panda introduit un mécanisme d'adressage global : une adresse virtuelle est un identificateur unique dans tout le système. Ce système impose quelques restrictions sur les machines à adressage 32 bits.

Le langage offre à la fois une mobilité des objets et des activités des objets (les deux par un mécanisme de faute sur l'accès à une page mémoire).

- Panda permet de placer n'importe quel objet en mémoire partagée distribuée, ce qui autorise la mobilité automatique (pendant l'utilisation) des objets. Un objet n'est placé en mémoire partagée que sur demande explicite, en raison du coût important de l'accès à ce type d'objet, par la suite. La syntaxe d'une telle création étend l'initialisation courante par le mot clé *DSM* :

```
class Any;
...
Any *pointer = new DSM Any;
```

- La migration d'activité est obtenue par l'utilisation explicite de la primitive *migrate* du langage :

```
class Manager :  UserThread {
    ...
};

Manager *mgr;
...
mgr->migrate(machine);
```

B.17.6 Renseignements pratiques

Panda est implémenté sous la forme d'un pré-compilateur (qui peut d'ailleurs ajouter lui-même des synchronisations au niveau des accès concurrents aux objets actifs) et s'exécute en association avec un pico-kernel. Une version gratuite de Panda est disponible.

Site ftp	:	`ftp.uni-kl.de:/pub/panda`
Bibliographie	:	[ABB+93a, ABB+93b, Ass92]

B.18 pC++

pC++ est une extension parallèle objet du langage C++ développée à l'Université d'Indiana (USA).

B.18.1 Le modèle de programmation

Deux modèles de programmation sont fournis par pC++ : un modèle tâches structurées par les données et un modèle SPMD. Pour le premier modèle, le programmeur définit des "collections" d'objets, et peut par la suite appliquer de façon concurrente des fonctions sur l'ensemble des éléments de la collection. Le deuxième modèle est basé sur l'utilisation de threads.

pC++ supporte à la fois les mémoires partagées et distribuées, et offre un espace de noms partagés pour les collections distribuées.

B.18.2 Déclaration du parallélisme

Une collection est principalement constituée par une classe *Collection* qui définit la topologie de l'ensemble des éléments, et par le type de ces éléments. Lors de la définition d'une classe *Collection*, en plus des champs *private*, *protected* et *public* usuels à une classe C++, le champ *MethodOfElement* va permettre de définir des données ou fonctions agissant simultanément sur l'ensemble des éléments de la collection. L'appel de ces fonctions sur l'objet collection va s'exécuter en parallèle sur chacun des processeurs contenant un sous-ensemble des éléments de la collection. L'appel est asynchrone.

```
Collection NameOfCollectionType:  Kernel {
   ...    // private & public & protected data fields
   ...    // and method functions
MethodOfElement:
   ...    // data and functions that are added to each element
};
```

Ce parallélisme est implémenté à l'aide de threads, et on peut également utiliser le mécanisme des threads pour programmer en SPMD. Il suffit pour cela de définir une classe de threads (mot-clé *TEClass*), l'appel d'une fonction membre de cette classe se fera en parallèle sur l'ensemble des processeurs concernés. La déclaration d'une instance de classe *TEClass* se fait à l'aide de la classe *Processors* qui définit un ensemble de processeurs virtuels.

```
main() {
   Processors P;       // set of processors object thread
   C myThreadEnv(P);   // C is a TEClass
   ...
}
```

B.18.3 Utilisation du parallélisme

Que ce soit pour un objet instance d'une classe *Collection* ou *TEClass*, la syntaxe d'appel d'une fonction est identique à celle utilisée pour un objet séquentiel. On peut également restreindre l'ensemble des éléments sur lesquels la fonction doit être appelée en définissant des intervalles.

```
C<E> c;        // C is a collection class
c.f();         // f is a MethodOfElement function of C. This call
               // means :  for all e in c do e.f();
c[a:b:s].f();  // apply f on each e_i for each
               // i in [a,b] with step s
```

B.18.4 Contrôle

L'exécution d'une méthode sur un objet de classe *Collection* ou *TEClass* est vue de l'extérieur comme une exécution atomique : les différents processeurs contenant un sous ensemble de l'objet se synchronisent avant d'exécuter la fonction suivante.

B.18.5 Gestion de la distribution de l'application

Le placement des éléments d'une collection va se faire en utilisant des types d'objets prédéfinis dans le langage : le type *Template* et le type *Align* qui permettent d'établir un système de coordonnées logiques afin d'organiser les éléments entre eux, et le type *Process* qui grâce au mécanisme de distribution va permettre d'organiser les éléments sur l'ensemble des processeurs.

B.18.6 Renseignements pratiques

pC++ est un compilateur (préprocesseur) générant du code C++. Ce compilateur a été écrit à l'aide de l'outil Sage++ [BBG+].

Site www	:	`http://www.extreme.indiana.edu/sage/index.html`
Site ftp	:	`moose.cs.indiana.edu:pub/sage/`
Bibliographie	:	[BBG+93a, BBG+93b, MMB+94]

B.19 Presto

Presto, qui est développé à l'Université de Washington (USA), est un langage orienté objet utilisant des threads, écrit au dessus de C++.

B.19.1 Le modèle de programmation

Ce langage permet aux programmeurs de créer du parallélisme, et de le contrôler. Le modèle utilisé est celui de tâches concurrentes, il repose sur une architecture multi-processeurs à mémoire partagée.

B.19.2 Déclaration du parallélisme

Le langage offre une nouvelle classe : la classe *Thread*. Une instance de cette classe constitue l'unité de base d'exécution d'un programme Presto. Une thread doit tout d'abord être créée, les principaux paramètres à passer au constructeur sont : le nom de la thread, son identificateur, et l'espace mémoire requis (les derniers paramètres ayant une valeur par défaut).

```
// Create a thread named ``Pusher''
Thread *t = new Thread(``Pusher'');
```

B.19.3 Utilisation du parallélisme

Une fois la thread créée, on va alors pouvoir la démarrer en lui donnant en paramètres : un objet, une méthode à appliquer sur cet objet, et les éventuels paramètres. On a alors exécution en parallèle de la thread appelante et de la thread appelée.

```
Stack *s = new Stack (100);

t->start(s, s->push, 43);    // Let t be responsible for pushing
                             // 43 onto the stack
```

A la différence d'un programme C++, le programme principal n'est pas constitué par la fonction *main()* mais par les fonctions membres *Main::init()* (initialisation), *Main::main()* (exécution) et *Main::done()* (terminaison). Ces fonctions ont des comportements par défaut que l'utilisateur pourra redéfinir à sa convenance.

B.19.4 Contrôle

Normalement, une thread est associée à un seul objet, mais il est possible d'avoir plusieurs threads concurrentes qui exécutent les fonctions d'un même objet. La synchronisation se fait de différentes manières :

- Classe *Lock*: une thread peut être interrompue par l'appel d'une méthode sur un objet bloquant (instance de la classe *Lock*). Les deux fonctions principales de cette classe (*lock()* et *unlock()*) permettent de définir des sections critiques.

```
// l is a reference to a Lock (Lock *l)
l->lock();
...            // critical code (the caller holds the lock)
l->unlock();
```

- On a également la classe *Spinlock* qui propose les mêmes primitives. Contrairement à la classe *Lock*, une thread qui essaye d'accéder à un verrou de cette classe déjà bloqué, ne va pas abandonner le processeur sur lequel elle s'exécute (attente active).
- Moniteurs et variables de condition : La classe *Monitor* permet également de définir des sections critiques de code à l'aide des fonctions *entry()* et *exit()*. Il est possible d'associer des variables de condition à un moniteur. La classe *Condition* offre deux fonctionnalités : *wait()* qui permet à une thread de se mettre en attente et *signal()* qui permet de réactiver la plus ancienne thread s'étant mise en attente.

Presto propose également un mécanisme de futurs explicites : la fonction membre *Thread::willjoin()* permet d'indiquer à une thread qu'une valeur doit être retournée. La thread va stocker la valeur de retour, et celle-ci pourra être accédée à l'aide de la fonction *Thread::join()*.

```
t->willjoin();            // preserve the return value
t->start(x, foo, args);   // go
...                       // stuff here
ObjAny tval = t->join();  // wait here
```

Au démarrage d'un programme Presto, une thread est allouée pour l'exécution du programme principal, qui pourra alors créer des objets threads afin de gérer le parallélisme. Les différentes threads sont ordonnancées à l'aide d'un "objet scheduler". Cet objet et l'objet thread pourront être personnalisés par l'utilisateur en vue de besoins spécifiques. Le scheduler fonctionne selon le modèle préemptif, mais une thread peut décider elle-même de son caractère préemptif ou non-préemptif.

B.19.5 Gestion de la distribution de l'application

Presto supportant un modèle mémoire globale, aucune restriction ne s'impose pour la gestion des données (mise à part la gestion des accès concurrents).

B.19.6 Renseignements pratiques

Site www	:	`http://www.cs.washington.edu/research/projects /parsw/Benchmarks/Presto/www/index.html`
Site ftp	:	`cs.washington.edu:/pub/presto1.0.tar.Z` `softserv.zdv.uni-tuebingen.de:/pub/parallel/PRESTO`
Bibliographie	:	[BLL88, FL90, TE94]

B.20 QPC++

Le langage QPC++ est développé à l'Université Carl-von-Ossietsky (Oldenburg, Allemagne). Il étend C++ par l'introduction de quelques nouveaux mots-clé et de types spécialisés (*process*, *set*, ...). Le modèle permet une programmation explicite du parallélisme. Les processus (objets actifs) communiquent de manière aussi bien synchrone qu'asynchrone. D'autres types spécialisés permettent aussi des regroupements de processus.

B.20.1 Le modèle de programmation

Le modèle de programmation comporte des processus (objets actifs) ayant une vie propre, et des objets C++ normaux (objets passifs). Les classes processus (*process class*) relèvent du modèle d'objets actifs communicants, tandis que les ensembles de processus (*process set*) sont des collections dont le parallélisme est structuré par leurs données.

B.20.2 Déclaration du parallélisme

Un objet actif est instancié sur une classe spécialisée (distinguée par le mot clé *process*[7]); cette classe possède une fonction membre dédiée à la description du contrôle (son "body", noté $nom_de_la_classe, de manière analogue aux destructeurs), qui est démarrée automatiquement après création et intialisation du processus par son constructeur. Lors de l'héritage d'une classe processus, le body est également hérité (à moins d'être purement et simplement redéfini).

```
process class Writer {  // active object : process class
   Writer() { ... };    // standard constructor
   $Writer() { ... };   // body
                        // this member function is called automatically
                        // after the creation and initialization
   ...
};
```

Les ensembles de processus, quant à eux, sont déclarés par un mécanisme de templates autorisant l'instanciation des éléments de la collection sur la ou les classes données en paramètre. On peut avoir deux types de groupes : statiques, on définit explicitement les instances qui en feront partie; dynamiques, toutes les instances de la classe passée en paramètre au template (voire des descendants) sont concernées.

```
process class Graphic { ... };                   // refresh() method
process class Box : public Graphic { ... };
process class Line : public Graphic { ... };
...
Box b1, b2;
Line l1, *l2;

process set<Box,Line> s1(b1,l1);                 // static process set
dyn process set<Graphic&> d;                     // dynamic process set
```

[7] il faut indiquer "process class" même pour une classe héritant d'un autre processus.

B.20.3 Utilisation du parallélisme

La communication, unifiée à l'appel de méthode, est synchrone par défaut, mais peut être rendue asynchrone en préfixant l'appel par le mot clé *asynchronous*:

```
process class ActiveObj {
public:
    void set_mean(int,int);
};
...
ActiveObj ao;

ao.set_mean(3,12);              // synchronous call :  waiting
asynchronous ao.set_mean(4,5);  // asynchronous call
...  // process is not stopped
```

B.20.4 Contrôle

On peut recenser deux types de synchronisations :

(1) L'envoi de message entre objets. L'appel est synchrone ou asynchrone : dans le dernier cas, on dispose d'un mécanisme de futur implicite avec attente par nécessité. La réception est bloquante : le message permet de sortir d'une attente volontaire (accept). On indique à un point d'acceptation les fonctions membres qui peuvent être appelées et par quels processus elles peuvent l'être. La sélection entre plusieurs requêtes peut être rendue indéterministe (*select*). On dispose d'un mécanisme de réponse anticipée, qui permet de libérer l'appelant avant la fin de l'exécution de la fonction membre.

(2) L'envoi de message (synchrone ou asynchrone) à un groupe de processus : l'appel d'une méthode sur le groupe provoque un envoi multiple aux membres du groupe.

```
asynchronous s1.refresh();  // asynchronous refresh() sent to b1 and l1
d.refresh();                // synchronous refresh() sent to b1, b2, l1 and *l2
```

B.20.5 Gestion de la distribution de l'application

Les variables qui appartiennent aux portées de deux processus sont partagées entre eux. Les objets actifs sont aussi globalement accessibles. Au niveau des variables partagées, l'accès doit être protégé par un mécanisme de sémaphores.

B.20.6 Renseignements pratiques

Un prototype a été implémenté sur stations de travail Sun, au-dessus de UNIX. Un système de run-time (écrit en C++) fournit des mécanismes pour la création des processus, les ordonnance, et assure la communication inter-procédurale. Les processus utilisés sont des processus légers, qui partagent un processus UNIX.

Bibliographie : [Bol93]

B.21 UC++

UC++ est développé à l'University College de Londres au LPAC (Angleterre), en collaboration avec la société ICL.

B.21.1 Le modèle de programmation

UC++ consiste en un modèle d'objets actifs communicants. Chaque objet actif n'exécute simultanément qu'une seule activité. On identifie les objets actifs à des processus et les appels fonctionnels à des communications inter-processus.

B.21.2 Déclaration du parallélisme

Il n'y a pas de déclaration d'entité parallèle; l'activité d'un objet dépend de sa création : s'il est instancié grâce à un nouveau mot-clé, *activenew* (à la place du *new* courant), il aura un comportement propre :

```
class X;
...
X *p = activenew X(3);
```

B.21.3 Utilisation du parallélisme

Les appels de méthodes synchrones sont unifiés aux requêtes synchrones. Les requêtes asynchrones sont distinguées des appels fonctionnels normaux par la notation (@ ... @) :

```
Obj *p = activenew Obj;
...
r=p->f(i,j,k);            // synchronous call
p->g(@ p,q,r @);          // asynchronous call
```

Tous les arguments sont passés par copie, pour des raisons de cohérence; les objets passifs sont gérés localement par l'objet actif qui les référence.

B.21.4 Contrôle

Le traitement des messages reçus est sérialisé, et ceux-ci sont exécutés selon une politique FIFO (premier arrivé, premier traité). Cet ordonnancement des services est figé et ne peut être modifié.

Seul un appel synchrone permet de réaliser un appel bloquant. Au point de vue réutilisation, tous les appels d'un programme séquentiel seront transformés en appels synchrones.

B.21.5 Gestion de la distribution de l'application

Le placement peut être effectué par la clause *on* qui indique un processeur (réel) particulier. Un objet ne peut être migré après sa création (qu'il soit actif ou non).

B.21.6 Renseignements pratiques

UC++ est implémenté sous la forme d'un pré-processeur écrit en flex, et est disponible gratuitement par ftp.

Site www	:	`http://www.lpac.ac.uk:80/SEL-HPC/Articles/GeneratedHtml/europa.ucpp.html`
Site ftp	:	`ftp://ftp.lpac.ac.uk/pub/UC++/`
Bibliographie	:	[ORW+95]

B.22 μC++ (Micro C++)

μC++ est développé à l'Université de Waterloo (Canada) et permet un mode de programmation explicite du parallélisme.

B.22.1 Le modèle de programmation

μC++ supporte un modèle mémoire globale, seuls les environnements mono-processeur (la concurrence est obtenue en entrelaçant l'exécution des tâches) et multi-processeurs à mémoire commune sont supportés.

B.22.2 Déclaration du parallélisme

Le parallélisme est introduit en μC++ par l'apport de 2 abstractions qui sont aussi des types : les coroutines et les tâches.

- Une coroutine est un objet dont l'exécution peut être suspendue et réactivée ultérieurement. Une coroutine ne possède pas son propre flot d'exécution. Un type coroutine (*uCoroutine*) est similaire à une classe, il possède en plus une fonction membre privée *main()* qui est le corps de la coroutine. La coroutine ne peut être activée qu'à travers une de ses fonctions membres qui activera ou ré-activera à son tour la fonction *main()*.

```
uCoroutine Consumer {
   void main() {                // 1st resume starts here
      ...
      uSuspend;                 // restart Consumer::delivery
   }  // main()
public:
   int delivery(int p1) {
      uResume;                  // restart Consumer::main
   }  // delivery()
}  // class Consumer

main(){
   Consumer * cons=new Consumer;
   cons->delivery(2);
}
```

- Une tâche est un objet possédant son propre flot de contrôle, et dont les membres sont accédés en exclusion mutuelle. Un type tâche (*uTask*) est similaire à une classe, il possède en plus une fonction membre privée *main()* qui est le corps de la tâche. La fonction *main()* contrôle explicitement et dynamiquement le prochain appel à un de ses membres qu'elle va servir.

B.22.3 Utilisation du parallélisme

Le parallélisme intervient entre les objets instances du type tâche. A la création, une tâche exécute automatiquement sa fonction *main()*. μC++ introduit la notion de processeur logique (classe *uProcessor*), correspondant à un processus Unix, sur lesquels

s'exécutent les tâches. Ces processeurs logiques sont regroupés en clusters (classe *uCluster*). Un cluster contient les variables d'environnement (taille de la pile, etc.) utilisées par les tâches s'exécutant sur ses processeurs.
Les tâches peuvent migrer (primitive *uMigrate()*) entre clusters.

B.22.4 Contrôle

Pour une tâche, le programmeur peut explicitement programmer dans la fonction *main()* le service des appels aux fonctions membres. Il dispose pour cela :

- De la primitive d'acceptation directe *uAccept(member)*.
- Des structures gardées :

```
uWhen (condition)
   uAccept(member)
uOr uWhen (condition)
      uAccept(member)
...
uElse
   uAccept(member)
```

Pour la synchronisation entre un ensemble de tâches, μC++ introduit les variables de condition (classe *uCondition*) et les objets mutex (instances des classes préfixées par le qualifieur *uMutex*).
Une variable de condition contient une file d'attente et est dotée des primitives classiques permettant de se mettre en attente (*uWait()*), de savoir s'il existe des tâches en attente (*uEmpty()*), et de réactiver la tâche de tête (*uSignal()*).
L'accès aux membres d'un objet mutex (activation de fonctions membres ou manipulation de membres de données) se fait en exclusion mutuelle, on définit ainsi un moniteur. Le programmeur a la possibilité de spécifier qu'un membre peut être accédé en parallèle en le qualifiant par *uNoMutex*. Un objet mutex peut, comme les tâches, contrôler le prochain membre qui pourra être accédé.

Le type *uCoMonitor* permet de définir un objet coroutine qui sera accédé en exclusion mutuelle.

B.22.5 Gestion de la distribution de l'application

μC++ supportant un modèle mémoire globale, aucune restriction ne s'impose pour la gestion des données (mise à part la gestion des accès concurrents).

B.22.6 Renseignements pratiques

μC++ est implémenté sous la forme d'un pré-processeur. Une version gratuite de μC++ est disponible.

site ftp	:	`watmsg.uwaterloo.ca:pub/uSystem/`
		`plg.uwaterloo.ca:pub/uSystem/`
Bibliographie	:	[BS95, BDS$^+$92, BS90]

Références

[ABB+93a] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, et R. Schwarz. PANDA - Supporting Distributed Programming in C++. Dans *Proceedings of the 7th European Conference on Object Oriented Programming (ECOOP'93)*, LNCS 707, Springer 1993, pages 361–383, 1993.

[ABB+93b] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, et R. Schwarz. The PANDA System Architecture - A Pico-Kernel Approach. Dans Los Alamitos IEEE Computer Society Press, éditeur, *Proceedings of the Fourth Workshop on Future Trends of Distributed Computing Systems (FTDCS'93)*, pages 470–476, Lisbonne, Portugal, 1993.

[ABG+86] M. Accetta, R. Baron, D. Golub, R. Rashid, et A. Tevanian. Mach: a New Kernel Foundation for Unix Development. Dans *Summer USENIX Conference*, pages 93–112, Juillet 1986.

[Acc95] Accredited Standards Committee (ANSI), X3, INFORMATION PROCESSING SYSTEMS. *Working Paper for Draft Proposed International Standard for Information Systems - Programming Language C++*, Mai 1995. `http://lomond.mathcs.warwick.ac.uk:80/c%2b%2b/pub/wp` - `ftp://research.att.com:/dist/c++std/WP`.

[Ass92] H. Assenmacher. Panda Runtime Package, The Panda-System Runtime Environment. Rapport technique SFB124-27/92, University of Kaiserslautern, Computer Science Department, 1992.

[BBG+] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Naranaya, S. Srinivas, et B. Winnicka. Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. Rapport technique, Department of Computer Science, Indiana University.

[BBG+93a] F. Bodin, P. Beckman, D. Gannon, S. Naranaya, et S. X. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. *Scientific Programming*, 2(3), 1993.

[BBG+93b] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, et B. Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. Dans *Proceedings of the Supercomputing'93 Conference*, Portland, Oregon, Novembre 1993.

[BDS+92] P. Buhr, G. Ditchfield, R. Stroobosscher, B. Younger, et C. Zarnke. μC++: Concurrency in the Object-Oriented Language C++. *Software - Practice and Experience*, 22(2):137–172, Février 1992.

[Ber94] L. Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, 1994.

[BFB90] J. Bahsoun, L. Feraud, et C. Betourne. A Two Degrees of Freedom Approach for Parallel Programming. Dans *Proceedings IEEE International Conference on Computer Languages*, pages 261–270, New Orleans, Mars 1990.

[BFPS94] J. Bahsoun, L. Feraud, J. Pouilly, et J. Sakdavong. Programmation Orientée Objet Distribuée : les Perspectives du Modèle Hector. Dans *Journées du GDR Programmation - Lille*, Septembre 1994.

[BLL88] B. Bershad, E. Lazowska, et H. Levy. Presto: A System for Object-Oriented Parallel Programming. *Software - Practice and Experience*, 18(8):713–732, Août 1988.

[BMa94] J.C. Bermond, P. Mussi, et al. SLOOP : Simulation, Langages à Objets et Parallélisme. `ftp:ftp.unice.fr:/pub/sloop`, Décembre 1994.

[Bol93] D. Boles. Parallel Object-Oriented Programming with QPC++. *Structured Programming*, 14:158–172, 1993.

[BS90] P. Buhr et R. Stroobosscher. The μC++: Providing Light-Weight Concurrency on Shared-Memory Multiprocessor Computers Running Unix. *Software - Practice and Experience*, 20(9):929–963, Septembre 1990.

[BS95] P. Buhr et R. Stroobosscher. μC++ Annotated Reference Manual. Rapport technique V 4.2, University of Waterloo, Janvier 1995. `ftp:watmsg.uwaterloo.ca:pub/uSystem`.

[BSS94] A. Beguelin, E. Seligman, et M. Starkey. Dome: Distributed Object Migration Environment. Rapport technique CMU-CS-94-153, School of Computer Science, Carnegie Mellon University, Mai 1994.

[Car93a] H. Carr. Distributed C++ Runtime Library Reference Guide. Rapport technique, University of Utah, 1993.

[Car93b] H. Carr. The DC++ and Concurrent Scheme Kernel. Rapport technique, University of Utah, 1993.

[Car94] H. Carr. *Distributed C++*. PhD thesis, University of Utah, Juin 1994. `ftp:cs.utah.edu:pub/dc++`.

[CCK93] M. Chandy, P. Carlin, et C. Kesselman. The Compositional C++ Language Definition. Rapport technique CS-TR-92-02, California Institute of Technology, Septembre 1993. `http://www.compbio.caltech.edu/`.

[Chi93] S. Chiba. Open C++ Programmer's Guide. Rapport technique 93-3, Dept. of Information Science, University of Tokyo, 1993.

[CK92a] M. Chandy et C. Kesselman. CC++: A Declarative Concurrent Object Oriented Programming Notation. Rapport technique, California Institute of Technology, Septembre 1992. `http://www.compbio.caltech.edu/`.

[CK92b] M. Chandy et C. Kesselman. Compositional C++: Compositional Parallel Programming. Rapport technique CS-TR-92-13, California Institute of Technology, 1992. `http://www.compbio.caltech.edu/`.

[CKS93a] H. Carr, R. Kessler, et M. Swanson. Compiling Distributed C++. Rapport technique, University of Utah, 1993. `ftp:cs.utah.edu:pub/dc++`.

[CKS93b] H. Carr, R. Kessler, et M. Swanson. Distributed C++. *ACM SIGPLAN Notices*, 28(1), Janvier 1993.

[CKS93c] H. Carr, R. Kessler, et M. Swanson. Distributed Object-Oriented Programming with C++. Rapport technique, University of Utah, Septembre 1993. `ftp:cs.utah.edu:pub/dc++`.

[CM92] S. Chiba et T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. Dans O. Lehrmann Madsen, éditeur, *Proceedings of the ECOOP'92*, volume 615 de *Lecture Notes in Computer Science*, pages 482–501, Utrecht, the Netherlands, Juillet 1992. European Conference on Object-Oriented Programming, Springer-Verlag.

[CP94] M. Carroll et L. Pollock. Composites: Trees for Data Parallel Programming. Dans *Proceedings of the 1994 International Conference on Computer Languages*, pages 43–54, Toulouse, France, Mai 1994. IEEE Computer Society Press. `gopher://gopher.eecis.udel.edu/h0/CIS/Tech_reports/Tech94/Tech_Rep-94_14.ps`.

[DHW88] D. Detlefs, M. Herlihy, et J. Wing. Inheritance of Synchronization and Recovery Properties in Avalon/C++. *IEEE Computer*, pages 57–69, Décembre 1988.

[DN93] Drakos et Nikos. Summary of Workshop on Parallel Programming in C++. `http://cs.caltech.edu/comp/papers/cppworkshop/tableofcontents3_1.html`, Septembre 93.

[DPS91] G. Dixon, G. Parrington, et S. Shrivastava. An Overview of the Arjuna Distributed Programming System. *IEEE Software*, pages 66–73, Janvier 1991.

[ES94] M. A. Ellis et B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Avril 1994.

[FGT94] I. Foster, J. Garnett, et S. Tuecke. Nexus User's Guide. Rapport technique, California Institute of Technology, Août 1994. `http://www.compbio.caltech.edu/`.

[FKOT94] I. Foster, C. Kesselman, R. Olson, et S. Tuecke. Nexus: An Interoperability Layer for Parallel and Distributed Computer Systems. Rapport technique, California Institute of Technology, Août 1994. `http://www.compbio.caltech.edu/`.

[FKT94a] I. Foster, C. Kesselman, et S. Tuecke. Nexus: Runtime Support for Task-Parallel Programming Languages. Rapport technique, California Institute of Technology, 1994. `http://www.compbio.caltech.edu/`.

[FKT94b] I. Foster, C. Kesselman, et S. Tuecke. Portable Mechanisms for Multithreaded Distributed Computations. Rapport technique, California Institute of Technology, 1994. `http://www.compbio.caltech.edu/`.

[FL90] J. E. Faust et H. M. Levy. The Performance of an Object-Oriented Threads Package. Dans *ECOOP/OOPSLA'90 Proceedings*, pages 278–289, Octobre 1990.

[FRS95] N. Furmento, Y. Roudier, et G. Siegel. Parallélisme et Distribution en C++. Dans *Septièmes Rencontres Francophones du Parallélisme - RenPar'7*, Faculté Polytechnique de Mons, Belgique, Juin 1995.

[GCLR92] R. Guerraoui, R. Capobianchi, A. Lanusse, et P. Roux. Nesting Actions through Asynchronous Message Passing: the ACS Protocol. Dans O. Lehrmann Madsen, éditeur, *Proceedings of the ECOOP'92*, volume 615 de *Lecture Notes in Computer Science*, pages 170–184, Utrecht, the Netherlands, Juillet 1992. European Conference on Object-Oriented Programming, Springer-Verlag.

[Gil92] S. Gilbert. A MasPar implementation of Data Parallel C++. Rapport technique, London Parallel Applications Center (LPAC), Septembre 1992.

[GR86] N. H. Gehani et W. D. Roome. Concurrent C. *Software - Practice and Experience*, 16(9):821–844, Septembre 1986.

[GR88] N. H. Gehani et W. D. Roome. Concurrent C++: Concurrent Programming with Class(es). *Software - Practice and Experience*, 18(12):1157–1177, Décembre 1988.

[HS93] C. L. Hartley et V. S. Sunderam. Concurrent Programming with Shared Objects in Networked Environments. Dans *International Parallel Processing Symposium*, Avril 1993.

[HW93] A. Hugues et M. Wallas. Synthèse et Classification de Langages Parallèles. *Génie Logiciel et Système Expert*, 32:6–22, Septembre 1993.

[IMT90] Y. Ishikawa, C. Mercer, et H. Tokuda. Object Oriented Real-Time Language Design: Constructs for Timing Constraints. Dans *ECOOP/OOPSLA*, pages 289–299, 1990.

[Joh95] D. Johnston. C++ Parallel Systems. *Engineering Computing Journal, DRAL - Daresbury Laboratory and Rutherford Appleton Laboratory*, 1995. `email:david@inf.rl.ac.uk`.

[Ket95] N. Kettani. Le Point sur la Normalisation de C++. *L'OBJET : Logiciel, Bases de Données, Réseaux*, 1(1):16–18, 1995.

[KK93] L. Kale et S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based On C++. Rapport technique UIUCDCS-R-93-1796, University of Illinois, Mars 1993. Also in ACM Sigplan Notes, Vol 28 No. 10, pp 91-108. Proceedings of OOPSLA-93, September 26-October 1, 1993, Washington DC.

[KLM93] D. Kafura, K. Lee, et M. Mukherji. ACT++: A Class Library for Concurrent Programming in C++ Using Actors. *Journal of Object Oriented Programming*, pages 47–55, Octobre 1993.

[Kon93] L. Kontothanassis. *The Mercury User's Manual.* University of Rochester, Août 1993. ftp://ftp.cs.rochester.edu/pub/papers/systems/93.tr465.Mercury_users_manual.ps.Z.

[KS90] R. Kessler et M. Swanson. Concurrent Scheme. *Parallel Lisp: Languages and Systems*, pages 200–234, 1990.

[LRV92] J. R. Larus, B. Richards, et G. Viswanathan. C**: a Large-Grain, Object-Oriented, Data-Parallel Programming Language. Rapport technique 1126, Computer Science Department, University of Wisconsin-Madison, Novembre 1992. ftp://ftp.cs.wisc.edu:tech-reports/tech-reports/reports/92/tr1126.ps.Z.

[MMB+94] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, et F. Bodin. Performance Analysis of pC++: A Portable Data-Parallel Programming System for Scalable Parallel Computers. Dans *Proceedings of the 8th International Parallel Processing Symbosium (IPPS)*, Cancún, Mexico, Avril 1994.

[MY93] S. Matsuoka et A. Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. Dans G. Agha, P. Wegner, et A. Yonezawa, éditeurs, *Research directions in concurrent object-oriented programming*, chapitre 1, pages 107–150. MIT Press, 1993.

[ORW+95] T. O'Brien, G. Roberts, M. Wei, R. Winder, D. Titcombe, M. Plaxton, A. McEwan, et J. Poole. *UC++ V1.3 Language and Compiler Documentation.* London Parallel Applications Center (LPAC), Janvier 1995.

[Par] G. Parrington. Reliable Distributed Programming in C++: The Arjuna Approach. http://arjuna.ncl.ac.uk/.

[PN91] M. Papathomas et O. Nierstrasz. Supporting Software Reuse in Concurrent Object-Oriented Languages: Exploring the Language Design Space. *Object Composition, Université de Genève, edited by D. Tsichritzis*, Juin 1991.

[PS88] F. Panzieri et K. Shrivastava. Rajdoor; A Remote Procedure Call Mechanism Supporting Orphan Detection and Killink. *IEEE Transactions on Software Engineering*, 14(1)30-37, Janvier 1988.

[SG91] H. Saleh et P. Gautron. A Concurrency Control Mechanism for C++ objects. Dans *ECOOP'91 / Workshop on object-based concurrent computing*, volume 612 de *LNCS*, pages 195–210, 1991.

[Stu94] D. C. Sturman. *The Broadway User's Manual – Broadway v. 1.0.* University of Illinois, Urbana-Champaign, 1994.

[Sun90] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4), Décembre 1990.

[TE94] R. Thekkath et S. J. Eggers. The PRESTO Application Suite. Rapport technique 94-09-01, Department of Computer Science and Engineering, University of Washington, Septembre 1994.

[TMRG94a] The Mentat Research Group. Mentat 2.8 Programming Language Reference Manual. Rapport technique, University of Virginia, Juin 1994. `ftp:uvacs.cs.virginia.edu`, `http://uvacs.cs.virginia.edu/~mentat/`.

[TMRG94b] The Mentat Research Group. Mentat 2.8 Tutorial. Rapport technique, University of Virginia, Juin 1994. `ftp:uvacs.cs.virginia.edu`, `http://uvacs.cs.virginia.edu/~mentat/`.

[TT92] K. Tazunori et M. Tokoro. DROL: An Object Oriented Programming Language for Distributed Real-Time Systems. Dans *OOPSLA*, 1992.