RenPar'7,
Actes des 7^{es} Rencontres
Francophones du Parallélisme,
Mons, Belgique, 30 mai–2 juin 1995.

Parallélisme et Distribution en C++

Nathalie FURMENTO, Yves ROUDIER [a] {furmento,roudier}@unice.fr
Günther SIEGEL [b] gsiegel@sophia.inria.fr

[a]I3S - CNRS - UNSA, bât. ESSI, route des Colles, BP 145, 06903 Sophia Antipolis Cedex (FRANCE)

[b]I.N.R.I.A. - Centre de Sophia Antipolis : 2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (FRANCE)

Résumé

Nous étudions dans cet article 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++.

Cette étude a fait l'objet d'un travail plus détaillé, présenté dans [11]

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 ces systèmes. Le langage C++ [10] 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 distingue 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 les extensions étudiées. Une classification 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. Nous analysons ensuite le mode d'introduction du parallélisme, en analysant notamment l'aspect déclaration et l'aspect utilisation. Pour finir, nous détaillons les abstractions de contrôles proposés ainsi que les mécanismes liés à la distribution des objets.

Les extensions de C++ qui font l'objet de cette synthèse sont : Arjuna [20], Avalon/C++ [9], C** [17], Charm++ [15], Compositional C++ (CC++) [8], Concurrent C++ [12], Data Parallel C++ (DPC++) [13], Distributed C++ (DC++) [7], Dome [1], DoPVM [14], Mentat [21], Mercury [16], pC++ [4], Presto [3], QPC++ [5], UC++ [18], μC++ [6] et l'extension C++ du LITP [19]. Ces langages sont analysés et présentés de manière plus détaillée dans [11].

1. Modèle de programmation

Selon l'extension utilisée, le programmeur devra structurer son application selon trois modèles :

- le plus basique consiste à structurer l'exécution du programme en un ensemble de tâches concurrentes manipulées de manière explicite. Le concept de tâche doit être rapproché de celui de flot d'exécution, et est indépendant de la notion d'objet (CC++, Concurrent C++, DoPVM, DC++, Presto et Mercury).

- un autre modèle consiste à organiser le programme en un ensemble de tâches structurées par les données manipulées. Dans le système Mentat, 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. Les langages Dome, C^{**} , DPC++, pC++ et QPC++ permettent de définir des *collections* d'objets dotées d'opérations globales qui s'exécuteront en parallèle sur la totalité des objets.

- 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. A chacun des objets peut être associée une activité unique, l'objet est alors unifié à une activité (Avalon/C++, QPC++, UC++, Charm++) ; d'autres extensions proposent aussi du parallélisme intra-objet, unifiant ainsi méthode et activité ([19], Arjuna, μC++).

2. Introduction du parallélisme

2.1. Déclaration

Dans le langage UC++, l'introduction du parallélisme ne nécessite aucune déclaration, mais plutôt une initialisation spécifique (*activenew*). Dans la plupart des extensions, au contraire, le parallélisme est déclaré explicitement, soit par l'introduction de nouveaux mots-clés ou constructions syntaxiques ([...] pour déclarer un agrégat de C^{**}), soit au travers du système de typage.

Le parallélisme par typage est probablement l'approche la plus élégante et la plus intéressante dans la mesure où elle est conforme à l'approche objet, masquant ainsi la gestion des activités parallèles. Nous avons rencontré trois approches pour la déclaration d'entités parallèles faisant appel au système de typage :

- ajout d'une abstraction parallèle similaire à la notion de classe (*Collection* de pC++, *uTask* de μC++).

- existence de classes de base intégrées au compilateur (*parallel_int* de DPC++), ou création de classes spécifiques par préfixage à la déclaration (*chare* de CHARM++, les mot-clés *persistent*, *sequential* et *regular* de Mentat).

- dérivation de classes connues du compilateur (*shared* de DoPVM) ou de bibliothèques (Mercury).

2.2. Utilisation

L'utilisation du parallélisme peut être entièrement implicite du fait de sa déclaration, ou au contraire explicite et nécessiter soit des directives de compilation, soit des invocations de primitives (*start* des threads de Presto), ou encore des constructions syntaxiques particulières.

Parmi les constructions syntaxiques, nous avons rencontré l'ajout de nouvelles structures de contrôle (le *parfor*{} et le *par*{} de CC++), le préfixage des appels fonctionnels (mot-clé *asynchronous* de QPC++, le *spawn* de CC++).

3. Contrôle

3.1. Synchronisation dirigée par les activités

La synchronisation peut intervenir au sein d'un groupe d'objets, à l'interface d'un objet, ou par le biais d'entités externes à l'objet.

Dans certains langages, la notion de groupe d'objets recouvre celle de localité de données. 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 : dans pC++ et C^{**}, l'exécution d'une opération en parallèle sur l'ensemble des objets de la collection est vue extérieurement comme une exécution atomique.

La synchronisation à l'interface d'un objet consiste à ordonnancer ses méthodes par la mise en place de politiques de service. Nous avons recensé différentes approches :

- ordonnancement fixé (sérialisation automatique en UC++).

- ordonnancement programmable : il peut être très rudimentaire et/ou manipulé explicitement (manipulation des files d'attente de DC++) ou plus sophistiqué (points d'entrée de QPC++, conditions d'attente de [19]) .

La synchronisation peut aussi être introduite par le biais d'entités spécialisées, n'ayant aucun traitement associé et dont le seul rôle est de suspendre une activité (classe *lock* de Presto).

3.2. Synchronisation dirigée par les messages

Les systèmes distribués sont généralement implémentés à l'aide d'envois de messages. Les langages ayant un modèle de programmation objet communicant (à l'exception de Charm++, qui définit explicitement un type *message*) unifient les concepts d'appel de méthode et d'envoi de message. Certains langages proposent par défaut une activation de méthode synchrone ([19]), d'autres une activation asynchrone (UC++) ; quelques-uns offrent les deux modes, en explicitant soit l'un, soit l'autre (*asynchronous* en QPC++).

Certains langages à appels de fonctions asynchrones implémentent un mécanisme de futurs implicites (QPC++) ou explicites (variables *sync* de CC++), la synchronisation étant alors dirigée par les données.

4. Nature distribuée de l'application

Les langages permettant de développer sur des environnements à mémoire distribuée et offrant une vision "machine virtuelle" à espace d'adressage unique imposent des restrictions 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).

Pratiquement aucun des langages étudiés n'offre de mécanismes 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 fournit à l'utilisateur des primitives de migration d'activités (*migrate* sur les threads).

En effet, le problème est complexe 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.

5. La standardisation

Plusieurs consortiums travaillent actuellement de manière indépendante à la réalisation d'extensions de C++. Nous pouvons notamment citer le consortium européen Europa[1] et le projet HPC++[2].

6. Conclusion

Dans le cadre du projet Sloop [2], nous sommes actuellement en train de définir une extension parallèle de C++ (C++//). Le projet vise notamment les applications non régulières, et plus particulièrement la réalisation d'un environnement de simulation distribuée.

L'objectif de cette étude était de recenser un échantillon représentatif des extensions parallèles de C++ afin de les comparer à notre modèle. Nous avons ainsi pu identifier des mécanismes originaux compatibles avec ceux

[1] `http://www.lpac.ac.uk:80/europa/`

[2] `http://www.cica.indiana.edu/extreme/hpc++/arpa/hpccBB.html`

que nous proposons et qui pourraient être intégrés à C++//.

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 étudiées.

Bibliographie

1. 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.
2. J.C. Bermond, P. Mussi, et al. Sloop : Simulation, Langages à Objets et Parallélisme. `ftp.unice.fr:/pub/sloop`, Décembre 1994.
3. 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.
4. 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.
5. D. Boles. Parallel object-oriented programming with QPC++. *Structured Programming*, (14):158–172, 1993.
6. P. Buhr, G. Ditchfield, R. Stroobosscher, B. Younger, et C. Zarnke. μ-C++: Concurrency in the Object-Oriented Language C++. *Software Practice and Experience*, 22(2):137–172, Février 1992.
7. H. Carr. *Distributed C++*. PhD thesis, University of Utah, Juin 1994. `ftp:cs.utah.edu:pub/dc++`.
8. 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/`.
9. D. Detlefs, M. Herlihy, et J. Wing. Inheritance of Synchronization and Recovery Properties in Avalon/C++. *IEEE Computer*, pages 57–69, Décembre 1988.
10. M. A. Ellis et B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Avril 1994.
11. N. Furmento, Y. Roudier, et G. Siegel. Une revue des langages C++ parallèles et distribués. Rapport technique RR 95-02, I3S - CNRS - UNSA, BP 145, 06903 Sophia Antipolis Cedex - France, Février 1995. A paraître.
12. N. H. Gehani et W. D. Roome. Concurrent C++ : Concurrent Programming with Class(es). *Software - Practice and Experience*, 18(12):1157–1177, Décembre 1988.
13. S. Gilbert. A MasPar implementation of Data Parallel C++. Rapport technique, London Parallel Application Centre (LPAC), Septembre 1992.
14. C. L. Hartley et V. S. Sunderam. Concurrent Programming with Shared Objects in Networked Environments. Dans *International Parallel Processing Symposium*, Avril 1993.
15. L. Kale et S. Krishnan. CHARM++ : A Portable Concurrent Object Oriented System Based On C++. Rapport technique, University of Illinois, 1994.
16. L. Kontothanassis. *The Mercury users's manual*. University of Rochester, Computer Science Department, Août 1993.
17. 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.
18. 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 Centre (LPAC), Janvier 1995.
19. 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.
20. S. K. Shrivastava, G. N. Dixon, et G. D. Parrington. An overview of the Arjuna Distributed Programming System. *IEEE Software*, Janvier 1991.
21. The Mentat Research Group. Mentat 2.7 Tutorial. Rapport technique, University of Virginia, Juin 1994. `ftp:uvacs.cs.virginia.edu`, `http://uvacs.cs.virginia.edu/~mentat/`.