Compilation

Romaric DAVID

LEM2I - Décembre 2011

Plan

Plan

- Rappels sur le rôle du compilateur.
- Utilisation des directives de compilation
- Optimisations réalisées et options du compilateur
- Les différentes familles de compilateurs (Libres GNU, Commerciaux Intel + Portland)

1 Rôle du compilateur

- Compilateur : programme chargé de la traduction du texte du programme (code source, Fortran, C, C++) ...
- ... en code binaire exécutable par la machine.

Pour ce faire, le compilateur :

- Analyse les sources du programme
- Écrit le code objet correspondant
- Peut être chargé d'assembler différents morceaux de codes objets en un exécutable (édition de liens).

Analyse des sources

- Analyse les sources du programme
  - Remplace les constantes par leurs valeurs
  - Recopie le texte des fichiers dont les nom est indiqué par les directives #include
  - Active/Désactive certaines zones de texte en fonctions des directives
- Étape de traduction source à source : précompilation

2 Précompilation

Un exemple de traduction

```
#include "toto.h"
#define MIN(x,y) (x < y ? x : y)
#define NBMAX 1000
a = MIN(NBMAX, 2*NBMAX)
```

devient

```
# 1 "tryme.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "tryme.c"

# 1 "toto.h" 1
void totofunc(float a);
# 2 "tryme.c" 2


a = (1000 < 2*1000 ? 1000 : 2*1000)
```

Pré-compilation conditionnelle

Certaines directives de pré-compilation peuvent être conditionnelles

```
#ifdef HAS_TAN
mm=tan(0.4);
#else
mm=sin(0.4)/cos(0.4);
#endif
```

Pour rentrer dans le 1er cas, il suffit que la constante HAS_TAN soit définie. Elle peut l'être dans le code source ou à la volée à la compilation. Par exemple si l'on tape `cc -DHAS_TAN mon_prog.c` on compilera le code suivant :

```
mm=tan(0.4);
```

Quelques directives de pré-compilation utiles

- `#include` (C) / `INCLUDE` (Fortran): recopie de définitions externes (ex: MPI)
- `#define (C)` : définition de constantes ou de raccourcis syntaxiques (macros)

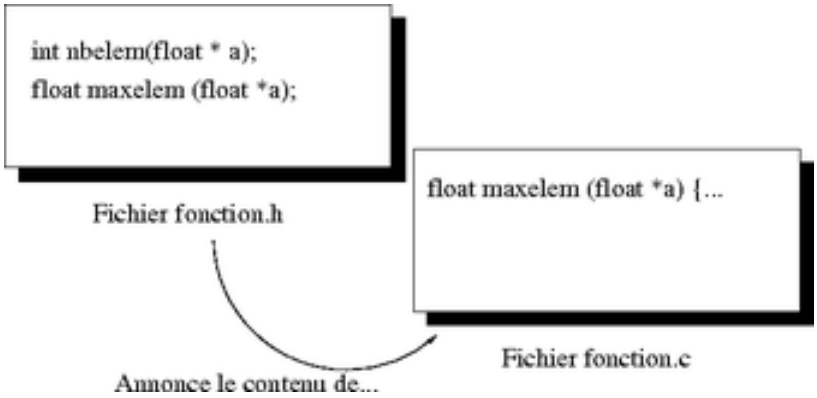
Un fichier destiné à être inclus dans un programme est souvent appelé fichier en-tête (header). A la compilation, on peut spécifier le ou les emplacement des fichiers d'en-tête par l'option `-I/chemin1 I/chemin2,...`

Le résultat de la pré-compilation d'un code peut s'obtenir (sur la sortie standard) par l'invocation du compilateur avec l'option `-E`. Il ne reste aucune directive de compilation, uniquement les constructions pures du langage. Une fois ce travail de mise en forme réalisé, la construction du code objet commence.

Les bonnes pratiques de programmation conduisent à une utilisation fréquente des directives de compilation, en particulier des `#include` et `ifdef`.

Bonnes pratiques de mise en forme du code

- Prototype des fonctions (arguments, valeurs de retour) : dans un des fichiers d'en-tête `.h` (C) ou `.inc` (Fortran)
- Code de la fonction : dans un fichier `.c` / `.F`
- Multiples inclusions d'un même fichier (voir TP) $\Rightarrow$ on utilise des directives `#ifdef` et `#define`



Le code objet est obtenu suite au parcours du code source (étendu par le pré-processeur). À chaque construction du langage de haut niveau (boucle, ...) est associé une mécanique de génération de *code intermédiaire*, la traduisant par de **nombreuses** instructions de bas-niveau.

Le code produit tel quel sans optimisations serait très inefficace et doit être optimisé pour produire un exécutable dont le temps d'exécution est raisonnable.

Construction du code objet

- Représentation du code source sous forme de code intermédiaire (indépendant de la plate-forme)
- Traduction vers le code objet correspondant, dépendant de la plate forme

Le code intermédiaire est une représentation très bas niveau (structures de contrôle rudimentaires (goto), opérations de la forme `résultat = valeur_1 opérande valeur_2`.

La représentation binaire finalement obtenue dépend :

- Du processeur utilisé
- Des conventions de codage utilisées par le système d'exploitation correspondant

$\Rightarrow$ optimisation nécessaire

3 Optimisation

Optimisation de code

En optimisant le code, le compilateur commence par réaliser un travail sur le code intermédiaire en :

- diminuant la taille du code (travail sur les boucles : réduction du nombre de variables et du nombres d'opérations)
- Ordonnancement des instructions en fonction des dépendances
- Utilisant au mieux les registres du processeur
- Déroulant des boucles pour profiter des unités vectorielles des processeurs

La quantité de travail du compilateur dépend du jeu d'instructions du processeur utilisé. Plus le jeu d'instructions du processeur est réduit, plus grande doit être l'intelligence du compilateur (Epic, Risc).

Optimisation de code - en pratique

Sur les compilateurs, différents niveaux d'optimisation sont mis en oeuvre (cela peut être gourmand en CPU / Mémoire) habituellement par une valeur associée à l'option "-O". En général et dans les grandes lignes :

- `-O0` : aucune optimisation. Utile dans le cas du débogage d'applications
- `-O1` : optimisations visant à accélérer le code, en particulier pour du code ne contenant pas beaucoup de boucles
- `-O2` : `O1` + déroulage de boucles
- `-O3` : `O2` + transformation des boucles, des accès mémoire

Certaines optimisations, appelées "agressives" dans les documentations, peuvent modifier la correction des résultats.

Les optimisations sur le code généré peuvent s'accompagner d'optimisations liées au jeu d'instructions de la machine cible. Par exemple, les extensions SSE (Streaming SIMD Extensions). Ces extensions proposent des instructions travaillant sur des petits vecteurs.

Auto-parallélisation

Certains compilateurs peuvent paralléliser (threads) des portions de code série. Pour cela :

- le compilateur analyse les dépendances dans les boucles
- il ajoute des instructions de création de threads. Un message de ce type est généré : `nodeps.c(7): (col. 6) remark: LOOP WAS AUTO-PARALLELIZED.`
- À l'exécution du code, l'utilisateur devra spécifier le nombre de threads qu'il souhaite utiliser ne contenant pas beaucoup de boucles.
- Ces optimisations dépendent du compilateur. Exemple de flags :
  - `-parallel` pour le compilateur Intel
  - `-Mconcur` pour le compilateur Portland.

Remarques générales

- Tester graduellement les niveaux d'optimisation et la correction des résultats
- Mesurer les temps d'exécution afin de s'assurer de l'efficacité des optimisations
- Utiliser les macro-optimisations du compilateur regroupant un ensemble de flags (ex : `-fast` sur les compilateurs Intel et Portland)

4 Les familles de compilateurs

On l'a entrevu dans les points précédents, il existe plusieurs familles de compilateurs, se différenciant par leurs fonctionnalités et leur prix.

Familles de compilateurs

Nom	Payant	Architecture
Gcc	non	toutes
Intel	oui	ia64/x86/x86_64
Portland	oui	x86/x86_64
Ibm Xlc/f	oui	PowerPC, Cell

Familles de compilateurs

Voici quelques différences de fonctionnalités :

Nom	OpenMP	Auto-parallélisation	Fortran 95
gnu	oui ($>=$ 4.2)	non	oui (gfortran)
intel	oui	oui	oui
portland	oui	oui	oui
Ibm Xlc/F	oui	oui	oui

La meilleure mesure de la qualité d'un compilateur *Comp* est le temps d'exécution du programme que vous utilisez, compilé avec *Comp*.