



Compilation Avancée

ENSIIE – S5

Cours 3 : Dominance & Directives

patrick.carribault@cea.fr



Plan

- Définition des passes dans GCC 12.2.0
- Dominance / Postdominance
- Gestion des directives



Plan

- Définition des passes dans GCC 12.2.0
- Dominance / Postdominance
- Gestion des directives



Passes de transformation

- Gestion des passes dans GCC 12
 - ➔ Principalement en C++
 - Construction basée sur une compatibilité simplifiée avec les précédentes versions en C
- Données nécessaires pour définir une passe
 - Information sur le type de passe (middle-end GIMPLE, back-end RTL ou inter-procédurale IPA)
 - Nom de la passe
 - Propriétés en entrée / sortie
 - Fonctions de décision et d'exécution



Définition d'une passe

- Object définissant une passe
 - `opt_pass` dans `tree-pass.h`
 - Répertoire include plugin
- Fonctions membres
 - Constructeur
 - Destructeur
 - Passage d'un argument à la passe
 - Décision d'exécution
 - Exécution
 - ...
- Implémentation par défaut ?

```
class opt_pass : public pass_data
{
public:
    virtual ~opt_pass () { }
    virtual opt_pass *clone ();
    virtual void set_pass_param (unsigned
int, bool);
    virtual bool gate (function *fun);
    virtual unsigned int execute (function
*fun);

protected:
    opt_pass (const pass_data&, gcc::context
*);

public:
    opt_pass *sub;
    opt_pass *next;
    int static_pass_number;

protected:
    gcc::context *m_ctxt;
};
```



Comportement par défaut

```
opt_pass *
opt_pass::clone ()
{
    internal_error ("pass %s does not support
cloning", name);
}
```

- Dans source de GCC (gcc/passes.c)
- Besoin de surcharger clone si nécessaire (notamment dans le cas d'un plugin car la passe est copiée en interne)

```
void
opt_pass::set_pass_param (unsigned int,
bool)
{
    internal_error ("pass %s needs a
set_pass_param implementation to handle
the"
                    " extra argument in
NEXT_PASS", name);
}
```

- Besoin de surcharger si la passe prend un argument en entrée

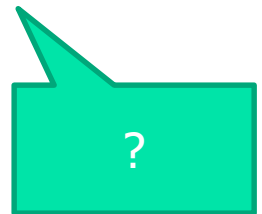
- Gate / execute par défaut :

```
bool
opt_pass::gate (function *)
{
    return true;
}

unsigned int
opt_pass::execute (function *)
{
    return 0;
}
```

- Constructeur par défaut

```
opt_pass::opt_pass (const pass_data &data,
context *ctxt)
: pass_data (data),
  sub (NULL),
  next (NULL),
  static_pass_number (0),
  m_ctxt (ctxt)
{
}
```





Données d'une passe



- Données d'une passe
 - ➔ Objet de type `pass_data`
- Informations
 - Type de passe
 - Nom de la passe
 - Différentes propriétés

```
struct pass_data
{
    enum opt_pass_type type;
    const char *name;
    unsigned int optinfo_flags;
    timevar_id_t tv_id;
    unsigned int
properties_required;
    unsigned int
properties_provided;
    unsigned int
properties_destroyed;
    unsigned int
todo_flags_start;
    unsigned int
todo_flags_finish;
};
```



Type de passe

- Définition du type de passe dans fichier tree-pass.h

```
/* Optimization pass
type.  */
enum opt_pass_type
{
    GIMPLE_PASS,
    RTL_PASS,
    SIMPLE_IPA_PASS,
    IPA_PASS
};
```

- Exemple de passe intra-procédurale du middle-end (GIMPLE)

```
/* Description of GIMPLE pass.
*/
class gimple_opt_pass : public
opt_pass
{
protected:
    gimple_opt_pass (const
pass_data& data, gcc::context
*ctxt)
        : opt_pass (data, ctxt)
    {
    }
};
```




Type de passe

- Passe intra-procédurale middle-end (GIMPLE)

```
class gimple_opt_pass : public opt_pass
```

- Passe intra-procédurale back-end (RTL)

```
class rtl_opt_pass : public opt_pass
```

- Passe inter-procédurale complexe

```
class ipa_opt_pass_d : public opt_pass
```

- Passe inter-procédurale simple

```
class simple_ipa_opt_pass : public opt_pass
```



Comment définir une passe ?

- Etapes de création d'une passe
 1. Création d'une classe qui hérite du type de classe souhaitée
 - Par exemple `gimple_opt_pass`
 2. Surcharge des méthodes nécessaires
 1. `clone` si utilisation via un plugin
 2. `gate/execute`
 3. Création d'un objet de type `pass_data`
 1. Doit contenir les informations de la passe comme le nom et le type de passe
 4. Utilisation des informations de type `pass_data` dans le constructeur dédié



Manipulation dans une passe

- Cadre du cours : utilisation d'une passe du middle-end intra-procédurale (GIMPLE)
- Fonctions gate/execute prend en argument un objet de type function
- Fonction courante définie comme une variable globale
 - GCC assure la cohérence entre la fonction courante et la déclaration de cette fonction

```
function.h:extern GTY(()) struct function *cfun;  
tree-core.h:extern GTY(()) tree current_function_decl;
```

```
/* The function currently being compiled. */  
extern GTY(()) struct function *cfun;
```

```
/* In order to ensure that cfun is not set directly,  
   we redefine it so that it is not an lvalue.
```

```
   Rather than assign to cfun, use push_cfun or set_cfun. */  
#define cfun (cfun + 0)
```



Nom de la fonction courante

- Possibilité d'afficher le nom de la fonction courante en cours de compilation
 - Implémentation de ces fonctions dans le fichier `function.c`

```
/* Returns the name of the current function. */  
  
extern const char *fndecl_name (tree);  
  
extern const char *function_name (  
    struct function *);  
  
extern const char *current_function_name (void);
```



Manipulation de l'IR

- IR : CFG (graphe de flot de contrôle)
- Possibilité de parcourir
 - les nœuds (BB)
 - Le contenu des nœuds (statements GIMPLE)
- Correction TD2...



Plan

- Définition des passes dans GCC 12.2.0
- Dominance / Postdominance
- Gestion des directives

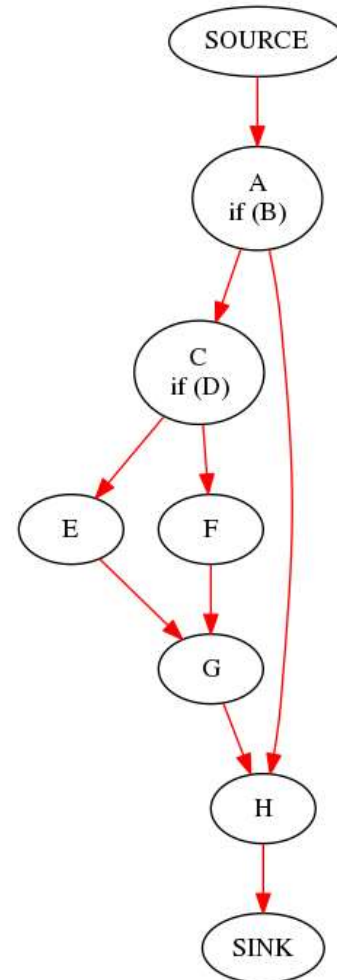


Graphe de flot de contrôle

- Etude de l'enchaînement possible des instructions dans une fonction
 - Graphe de flot de contrôle (CFG : Control-Flow Graph)
 - Notation $G = (V, E)$
- Nœud → Bloc de base (*basic block* ou BB)
 - Séquence d'instructions s'exécutant à la chaîne
 - Pas possible de commencer un BB en plein milieu
 - Pas possible de quitter avant la fin
- Arc → possibilité de continuer
- Remarques
 - CFG peut être un DAG (Cycle → Boucle dans le code)
 - Création d'un nœud unique source et puit
 - GCC a besoin de ces nœuds artificiels

Graphe de flot de contrôle

```
void f() {  
    A;  
    if (B) {  
        C;  
        if (D) {  
            E;  
        } else {  
            F;  
        }  
        G;  
    }  
    H;  
}
```



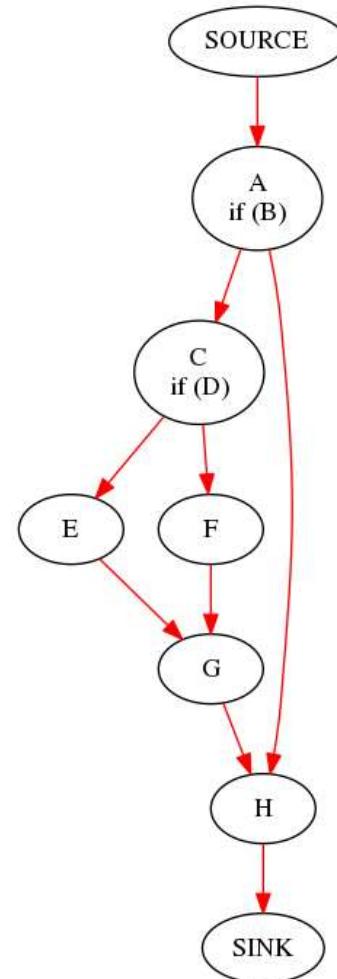


Chemin

- Notion de chemin
 - Séquence de nœuds (ordre important)
 - Les nœuds consécutifs sont reliés ensemble par un arc
- Notation
 - Premier nœud X et dernier nœud Y : $X \rightarrow^* Y$
 - Si le chemin n'est pas vide : $X \rightarrow^+ Y$

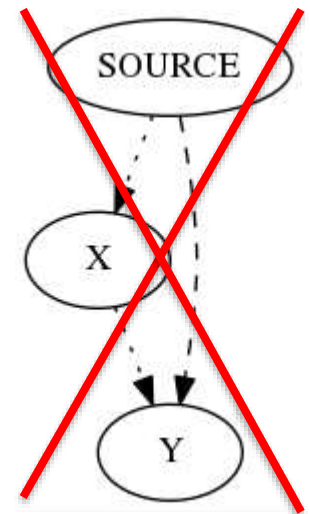
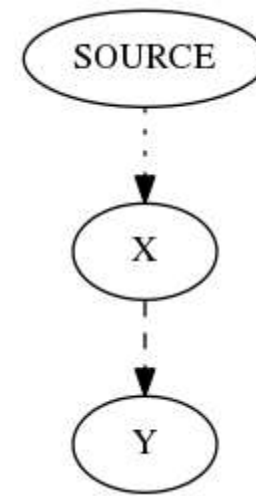
Chemin

- Exemple précédent
 - CFG d'une fonction
- Chemins existants
 - $A \rightarrow G$
 - $SOURCE \rightarrow SINK$
- Chemins non présents
 - $F \rightarrow A$
 - $H \rightarrow C$
 - $SINK \rightarrow SOURCE$



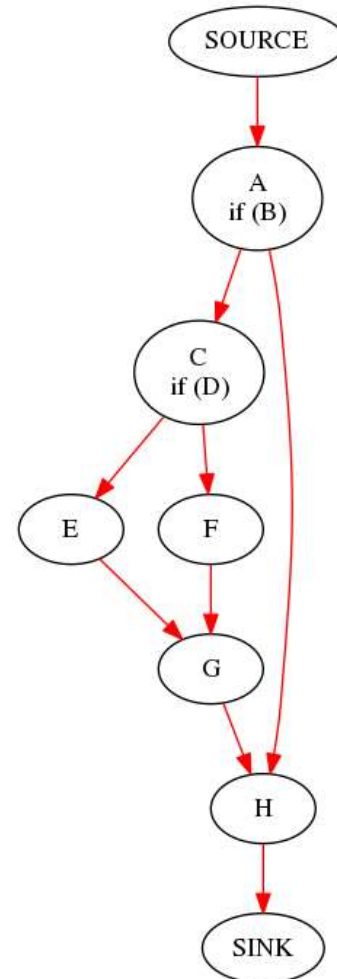
Dominance

- Définition :
 - un nœud X domine un nœud Y si tous les chemins de la SOURCE à Y passent par X
- Notation :
 - $X \text{ dom } Y$



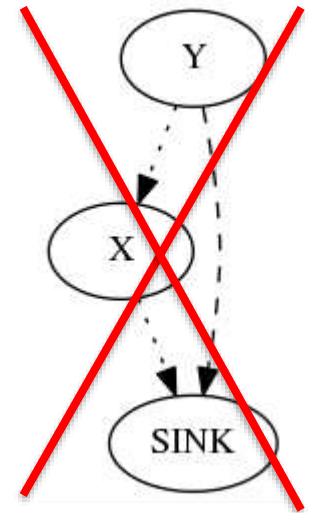
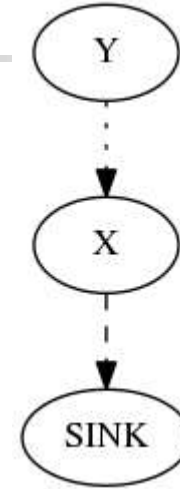
Dominance

- Exemple précédent
 - CFG d'une fonction
- Exemples de dominance
 - C dom G
 - A dom H
- Exemples faux
 - F dom G
 - C dom H



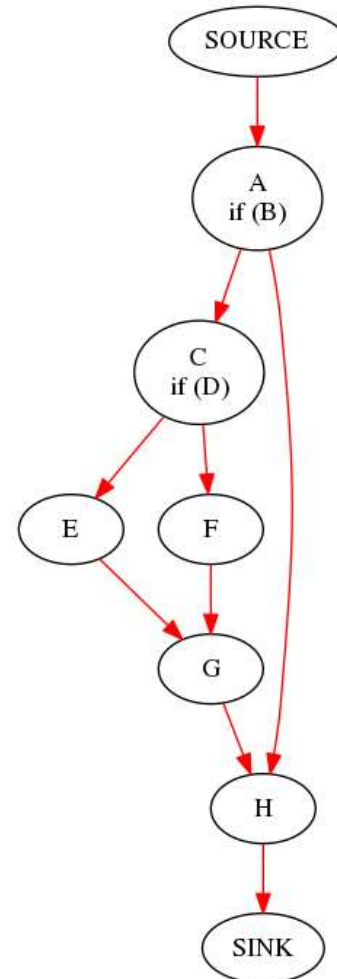
Post-Dominance

- Définition :
 - un nœud X post-domine un nœud Y si tous les chemins de Y au puits passent par X
- Notation :
 - $X \text{ pdom } Y$



Post-Dominance

- Exemple précédent
 - CFG d'une fonction
- Exemples de dominance
 - $G \text{ pdom } C$
 - $H \text{ pdom } A$
- Exemples faux
 - $F \text{ pdom } C$
 - $C \text{ pdom } A$





Relation de dominance

- Equation *dataflow* pour la domination
 - Permet de calculer la notion de dominance entre nœuds

$$\text{DOM}(n_0) = \{n_0\}$$

$$\text{DOM}(n) = \left(\bigcap_{p \in \text{preds}(n)} \text{DOM}(p) \right) \cup \{n\}$$

- Implémentation sous forme d'un algorithme itératif



Calcul de dominance

- Tableau contenant l'ensemble des dominateurs d'un nœud :

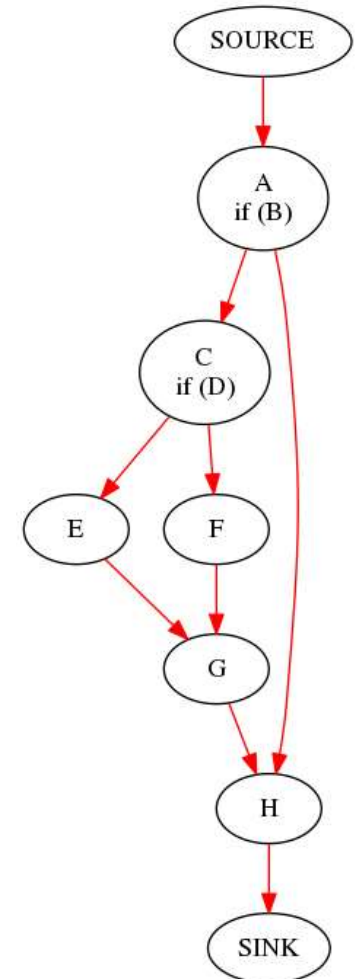
- Dom

- Processus itératif

```
for all nodes, n
  DOM[n] ← {1 ... N}
Changed ← true
while (Changed)
  Changed ← false
  for all nodes, n, in reverse postorder
    new_set ←  $\left( \bigcap_{p \in \text{preds}(n)} \text{DOM}[p] \right) \cup \{n\}$ 
    if (new_set  $\neq$  DOM[n])
      DOM[n] ← new_set
      Changed ← true
```


Calcul de dominance

- Etat initial
 - $\text{Dom}[*] = \{\text{SOURCE}, \dots, \text{SINK}\}$
- Itération 1
 - $\text{DOM}[\text{SOURCE}] = \{\text{SOURCE}\}$
 - $\text{DOM}[\text{A}] = \{\text{A}\} \cup \text{DOM}[\text{SOURCE}] = \{\text{SOURCE}, \text{A}\}$
 - ...
 - $\text{DOM}[\text{SINK}] = \{\text{SOURCE}, \dots, \text{SINK}\}$
- Itération 2 ?





Dominance dans GCC

- GCC propose déjà la notion de dominance
- Fonctions/structures accessibles dans le fichier `dominance.h`

```
enum cdi_direction
{
    CDI_DOMINATORS = 1,
    CDI_POST_DOMINATORS = 2
};
```

```
extern void
calculate_dominance_info
(enum cdi_direction);
```

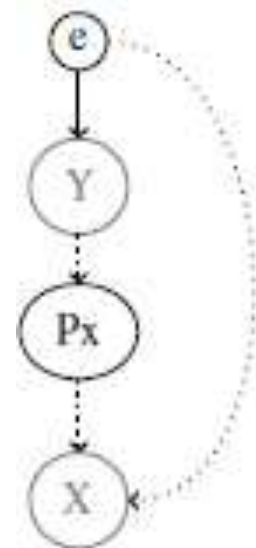


Frontière de dominance

- Au-delà de la dominance
 - Intérêt de connaître ce qu'il se passe quand un nœud n'en domine pas un autre
 - ➔ Définition de la frontière de dominance
- Idée
 - Capturer les nœuds qui sont proches des nœuds dominés
 - On dit qu'ils sont à la frontière de dominance
 - Notion de blocs « juste après »

Frontière de dominance

- Définition
- Utilisation de la dominance



$$DF(Y) = \{X | \exists Px \in preds(X), Y \underline{\gg} Px \text{ and } Y \not\gg X\}$$



Calcul de DF

- Algorithme issu de Cooper, Harvey et Kennedy
- Par du constat qu'un nœud est à la frontière de dominance s'il correspond à la jonction du flot de contrôle
 - ➔ Ce nœud a au moins 2 prédécesseurs
- Le chemin des dominants (à travers l'arbre de domination) concerne la même frontière
- Tant qu'on obtient pas un nœud qui domine le nœud de départ à traiter



Algorithme de DF

```
for all nodes, b  
  if the number of predecessors of b  $\geq 2$   
    for all predecessors, p, of b  
      runner  $\leftarrow p$   
      while runner  $\neq \text{doms}[b]$   
        add b to runner's dominance frontier set  
        runner = doms[runner]
```

Figure 5: The Dominance-Frontier Algorithm



Application à SSA

- Représentation classique en compilation → SSA
 - Static Single Assignment
- Chaque variable est écrite une seule fois
 - S'il existe une variable écrite plusieurs fois → génération de nouvelles variables suffixées avec un entier
- Besoin de mettre à jour les lectures...



Exemple SSA

■ Sans SSA

a = b + c ;

b = a + 3 ;

a = b + c ;

b = 2 ;

c = b + 4 ;

■ Avec SSA

a_1 = b + c ;

b_1 = a + 3 ;

a_2 = b + c ;

b_2 = 2 ;

c_1 = b + 4 ;



Exemple SSA

■ Sans SSA

a = b + c ;

b = a + 3 ;

a = b + c ;

b = 2 ;

c = b + 4 ;

■ Avec SSA

a_1 = **b_0** + **c_0** ;

b_1 = **a_1** + 3 ;

a_2 = **b_1** + **c_0** ;

b_2 = 2 ;

c_1 = **b_2** + 4 ;

Avec un
if/else ?



Exemple SSA

■ Sans SSA

```
a = b + c ;  
if (a) {  
    b = a + 3 ;  
    a = b + c ;  
}  
c = b + a ;
```

■ Avec SSA

```
a_1 = b_0 + c_0 ;  
if (a_1) {  
    b_1 = a_1 + 3 ;  
    a_2 = b_1 + c_0 ;  
}  
c_0 = b + a ;
```



Fonctions Phi

- Les variables peuvent avoir un contenu différent en fonction du chemin parcouru
- Besoin de capturer cette information
 - → jonction du flot de contrôle
- Ajout de fonction phi pour modéliser cette action



Exemple SSA

■ Sans SSA

```
a = b + c ;  
if (a) {  
    b = a + 3 ;  
    a = b + c ;  
}  
c = b + a ;
```

■ Avec SSA

```
a_1 = b_0 + c_0 ;  
if (a_1) {  
    b_1 = a_1 + 3 ;  
    a_2 = b_1 + c_0 ;  
}  
b_2 = Phi(b_0, b_1) ;  
a_3 = Phi(a_1, a_2) ;  
c_0 = b_2 + a_3 ;
```

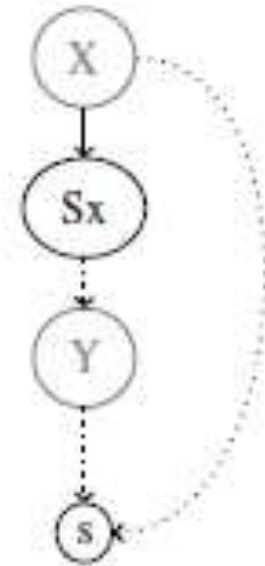


Fonctions Phi

- Comment ajouter ces fonction phi ?
- Utilisation de la frontière de dominance pour connaître les endroits
- SSA
 - Simplification de certaines transformations
 - Passe de De-SSA pour supprimer les fonctions Phi (avant le passer dans le back-end du compilateur)

Frontière de post-dominance

- Définition
- Utilisation de la post-dominance



$$PDF(Y) = \{X \mid \exists Sx \in succs(X), Y \underline{\gg}_p Sx \text{ and } Y \not\gg_p X\}$$



DF dans GCC

- Fonction disponible dans le fichier `cfganal.h`
 - ```
extern void
compute_dominance_frontiers
(struct_bitmap_head *);
```
- Application de l'algorithme précédent
- Pas de calcul de la frontière de post-dominance (PDF) !



# PDF et Projet

---

- Application de la PDF au projet
  - Lien avec MPI
- Besoin d'étendre sur 2 aspects
  - PDF d'un ensemble
  - Frontière itérée
- ...





# Plan

---

- Définition des passes dans GCC 12.2.0
- Dominance / Postdominance
- Gestion des directives



# Plan

---

- Définition des passes dans GCC 12.2.0
- Dominance / Postdominance
- Gestion des directives
  - Introduction à OpenMP
  - Transformation manuelle
  - Implémentation dans le compilateur GCC



# Introduction à OpenMP

---

- Modèle de programmation parallèle
  - Exploitation du parallélisme de données
  - Exploitation du parallélisme de tâches (depuis OpenMP 3.0)
- Mémoire partagée
  - Exécution sur un nœud de calcul
  - Possibilité d'extension à de la mémoire distribuée (DSM – Distributed Shared Memory)
    - Par exemple Intel Cluster OpenMP
- Basé sur des threads
- Exemple de coopération entre un compilateur et une bibliothèque



# Historique d'OpenMP

---

- Gestion par l'OpenMP ARB (*Architecture Review Board*)
- OpenMP 1.0 pour Fortran
  - Octobre 1997
- OpenMP 1.0 pour C/C++
  - Octobre 1998
- OpenMP 2.0 pour Fortran → 2000
- OpenMP 2.0 pour C/C++ → 2002
- OpenMP 2.5
  - Unification de la norme pour Fortran et C/C++
  - Sortie en Mai 2005
- OpenMP 3.0
  - Ajout du support du parallélisme de tâches
  - Sortie en Mai 2008 (remplacée par OpenMP 3.1 depuis 2011)
- OpenMP 4.0
  - Gestion des accélérateurs
- OpenMP 5.0 (2018)
  - Gestion de multiples mémoires
  - Interface avec outils (OMPT)



# Exemple de programme OpenMP

- Exemple de code séquentiel
- Addition de 2 vecteurs
  - Fonction `vecAdd`
- Parallélisme disponible dans cette fonction ?
- OUI, sur la boucle, mais
  - Indépendance des zones d'allocation de `a`, `b` et `c`

```
void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 int i;
 for (i=0 ; i<N ; i++) {
 c[i] = a[i] + b[i];
 }
}
```



# Exemple de programme OpenMP

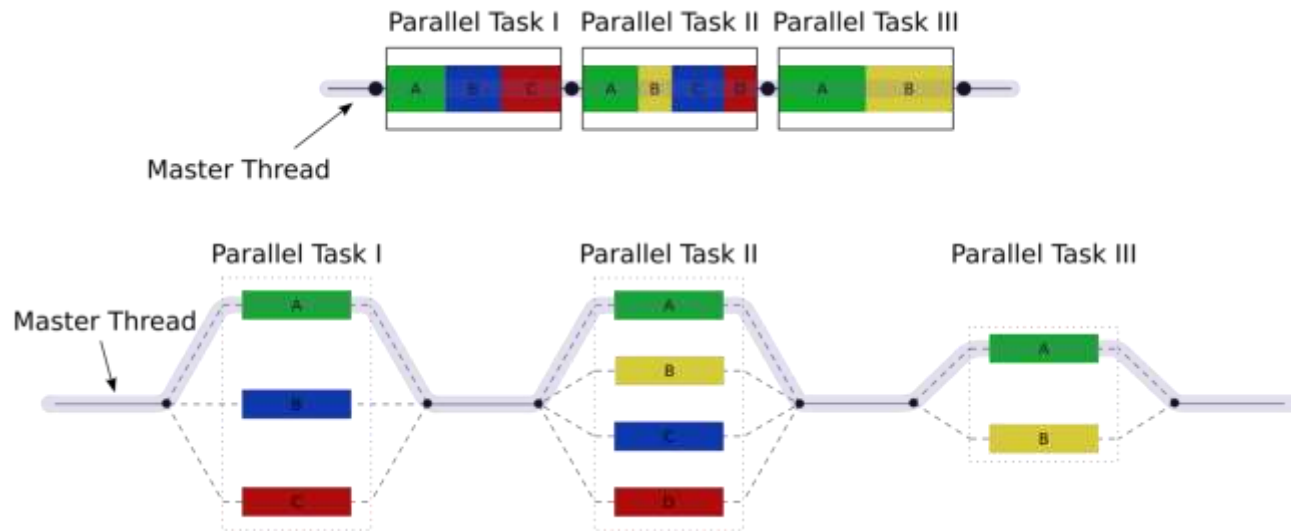
---

- Exploitation du parallélisme de données dans la boucle
- Ajout d'une directive pour la parallélisation
- Effets de cette directive
  - Création d'une région parallèle
  - Partage du domaine d'itérations de la boucle entre les threads participant à la région parallèle
  - Chaque thread a une copie privée de la variable  $i$
  - La fin de la région parallèle implique une barrière
- Concrètement
  - Avec  $T$  threads
  - Le thread 0 va exécuter les  $N/T$  premières itérations
  - Le thread 1 va exécuter les  $N/T$  itérations suivantes
  - ...

```
void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 #pragma omp parallel for\
 private(i)
 for (i=0; i<N; i++) {
 c[i] = a[i] + b[i] ;
 }
}
```

# Modèle d'exécution d'OpenMP

- Modèle d'exécution fork/join
  - Entrée dans une région parallèle → fork
  - Sortie de région parallèle → join (barrière)
  - A l'intérieur : coopération entre les threads grâce à des constructions dédiées (comme `#pragma omp for`)





# Plan

---

- Définition des passes dans GCC 12.2.0
- Dominance / Postdominance
- Gestion des directives
  - Introduction à OpenMP
  - Transformation manuelle
  - Implémentation dans le compilateur GCC





# Transformation manuelle

---

- Comment transformer le code source avec directives OpenMP en code qui s'exécute en parallèle ?
- Participation du compilateur et d'une bibliothèque externe
- Dans GCC :
  - Le compilateur transforme le code et génère les appels à la bibliothèque GOMP (GCC OpenMP)
  - GOMP est en charge de la création/maintenance des threads et des synchronisations/communications entre eux



# Transformation manuelle - Etapes

---

- Etude du flot de données
  - Quelles variables sont en vie dans la région parallèle ?
- Création d'une structure intermédiaire
  - On copiera les variables nécessaires au flot de données de la région parallèle
- Extraction de la région parallèle dans une nouvelle fonction (*outlining*)
- Restauration du flot de données
  - Copie des données de la structure intermédiaire
- Appel au *runtime* pour le démarrage de la région parallèle
  - Appel à une fonction de la bibliothèque OpenMP avec un pointeur sur la nouvelle fonction extraite
- Tout ceci est fait en pratique par le compilateur GCC
- Application manuelle sur notre code d'exemple !



# Etude du flôt de données

---

- Etude du flot de données dans notre fonction `vecAdd`
- Les variables `a`, `b`, `c`, `N` et `i` sont utilisées de la région parallèle
  - La variable `i` est déclarée privée dans la région parallèle
  - Chaque thread va avoir sa propre copie
- Elles sont *en vie* avant et après la région parallèle

```
void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 int i ;

 #pragma omp parallel for\
 private(i)
 for (i=0; i<N; i++) {
 c[i] = a[i] + b[i] ;
 }
}
```



# Création d'une structure

- Création d'une structure
  - Un champ par variable en entrée de la région parallèle
  - Les variables *private* ne sont pas concernées
- Cette structure est ensuite remplie avec les bonnes données
- Dans notre exemple
  - Transfert de *a*
  - Transfert de *b*
  - Transfert de *c*
  - Transfert de *N*

```
struct _s {
 double * _a ;
 double * _b ;
 double * _c ;
 int _N ;
};
```

```
void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 int i;
 struct _s s ;
 s._a = a ;
 s._b = b ;
 s._c = c ;
 s._N = N ;
 #pragma omp parallel for\ private(i)
 for (i=0; i<N; i++) {
 c[i] = a[i] + b[i] ;
 }
}
```

# Extraction de la région parallèle



- L'étape 3 consiste en l'extraction de la région parallèle en une nouvelle fonction
  - *Outlining* de fonction
  - Par opposition à l'*inlining*
- Déroulement
  - Sélection du bloc délimitant la région parallèle
  - Dans notre cas, il s'agit juste de la boucle `for`
  - Création d'une nouvelle fonction nommée `omp1` (importance de l'unicité du symbole)
  - Passage en argument d'une structure de donnée intermédiaire pour rétablir le flôt de données



# Extraction de la région parallèle

---

```
struct _s {
 double * _a ;
 double * _b ;
 double * _c ;
 int _N ;
} ;

void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 int i;
 struct _s s ;
 s._a = a ;
 s._b = b ;
 s._c = c ;
 s._N = N ;
 GOMP_start_parallel(omp1, &s);
}
```

```
void omp1(struct _s * s) {
 int i;
 double * a ;
 double * b ;
 double * c ;
 int N ;
 int min ;
 int max ;

 // Calcul des bornes
 // de l'espace d'itérations
 min = ... ;
 max = ... ;

 for (i=min; i<max; i++) {
 c[i] = a[i] + b[i] ;
 }
}
```



# Restauration du flot de données

---

- Appel d'une fonction interne à la bibliothèque OpenMP pour démarrer une région parallèle
- En pratique, lance plusieurs threads ou réveille certains threads dormant
- Parmi les paramètres d'entrée : notre structure
- Ajout d'affectations au début de la nouvelle fonction pour mettre à jour les variables



# Restoration du flot de données

---

```
void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 int i;
 struct {
 double * _a ;
 double * _b ;
 double * _c ;
 int _N ;
 } s ;
 s._a = a ;
 s._b = b ;
 s._c = c ;
 s._N = N ;
 GOMP_start_parallel(omp1, &s);
}
```

```
void omp1(struct _s * s) {
 int i;
 double * a ; double * b ;
 double * c ; int N ;
 int min ;
 int max ;

 // Calcul des bornes
 // de l'espace d'itérations
 min = ... ;
 max = ... ;

 a = s->_a ;
 b = s->_b ;
 c = s->_c ;
 N = s->_N ;

 for (i=min; i<max; i++) {
 c[i] = a[i] + b[i] ;
 }
}
```





# Plan

---

- Définition des passes dans GCC 12.2.0
- Dominance / Postdominance
- Gestion des directives
  - Introduction à OpenMP
  - Transformation manuelle
  - Implémentation dans le compilateur GCC



# Implémentation dans GCC

---

- Toutes les étapes de cette transformation sont assurées par le compilateur
- Cahier des charges
  - Gestion des pragmas (syntaxe, sémantique, ...)
  - Création de nouvelles fonctions (gestion des symboles)
  - Analyse du flot de données
  - Création de nouvelles structures
  - Ajout d'appels de fonctions
  - Support d'une bibliothèque OpenMP
- Détails dans les différentes parties de GCC



# Gestion des pragmas

---

- Enregistrement dans le préprocesseur
- Concerne les langages C et C++
  - GCC supporte également OpenMP dans les applications FORTRAN
  - Le support des directives est alors un peu différent (commentaires spéciaux dans le code source)
- Permet de laisser passer les pragmas
- Gestion réelle par le front-end



# Gestion des pragmas

---

- Fichier `gcc/c-family/c-pragma.c`
- Structure contenant les informations sur les directives

```
struct const omp_pragma_def omp_pragmas[] = {
 { "parallel", PRAGMA_OMP_PARALLEL},
 ...
}
```
- Enregistrement des directives dans le préprocesor

```
if (flag_omp)
 for (i = 0 ; i < n_omp_pragmas ; i++)
 cpp_register_deferred_pragma(parse_in, "omp", omp_pragmas[i].name,
 omp_pragmas[i].id, true, true) ;
```
- Note : dépend des options de compilation
  - La variable `flag_omp` est automatiquement générée pendant la compilation de GCC et est mise à 1 lorsque l'option `-fopenmp` est activée pendant la compilation d'un fichier



# Front-end

---

- Transforme les directives en nœuds de la RI
  - Représentation GENERIC
- Pragmas viennent du préprocesseur
- Certaines constructions OpenMP sont directement transformées en appels de fonction
  - Exemple : `#pragma omp barrier`
  - Devient lors du front-end : `GOMP_Barrier()` ;
- Création de nœud (*tree code*) spéciaux
  - OMP\_PARALLEL pour les région parallèles



# Front-end

---

- Fichier `gcc/c/c-parser.c`
- Fonction `c_parser_pragma`
  - Général pour tous les pragmas du langage C
- Sélection sur les directives OpenMP à transformer directement
  - Certaines directives sont traitées directement (exemple : `PRAGMA_OMP_BARRIER`)
  - Dans ce cas : appel à une fonction qui termine la transformation (ex : appel à `c_parser_omp_barrier`)
- Concernant les autres directives :
  - Appel à fonction `c_parser_omp_construct`
  - Sélection sur le nom des directives



# Front-end – Barrière OpenMP

---

- Fichier `gcc/c-family/c-omp.c`
- Fonction `c_finish_omp_barrier()`

```
void
c_finish_omp_barrier(location_t loc) {
 tree x;
 x = built_in_decls[BUILT_IN_GOMP_BARRIER];
 x = build_call_expr_loc(loc, x, 0)
 add_stmt (x);
}
```
- Effets
  - Création d'un appel de fonction
  - Cette fonction est *built-in* (définie dans le fichier `omp-builtins.def`) → `GOMP_barrier()`
  - Ajout de cet appel aux *statements* courants



# Front-end – Région parallèle

---

- Fichier gcc/c/c-typeck.c
- Fonction `c_finish_omp_parallel`
  - Après avoir parsé les clauses

```
void
c_finish_omp_parallel(location_t loc, tree clauses, tree block) {
 tree stmt;
 block = c_end_compound_stmt (loc, block, true);
 stmt = make_node (OMP_PARALLEL);
 TREE_TYPE (stmt) = void_type_node;
 OMP_PARALLEL_CLAUSES (stmt) = clauses;
 OMP_PARALLEL_BODY (stmt) = block;
 SET_EXPR_LOCATION (stmt, loc);
 return add_stmt (stmt);
}
```

- Effets :
  - Création d'un noeud de *code* OMP\_PARALLEL
  - Le type est considéré comme *void*
  - L'emplacement dans le fichier source est lié grâce à la structure *location\_t*





# Front-end → Middle-end

---

- Une fois les fonctions parsées, la représentation intermédiaire évolue
  - Passage de GENERIC à GIMPLE
  - Nom de code : *gimplification*
- Fichier `gimplify.c`
  - Fonction `gimplify_expr()`
  - Sélection sur la valeur du *tree code*
  - Dans le cas de `OMP_PARALLEL`
    - Appel à `gimplify_omp_parallel()`
    - Création du noeud `GIMPLE_OMP_PARALLEL`



# Middle-end

---

- Bilan en entrée du middle-end
  - Les directives ont été parsées
  - Certaines ont déjà été transformées en appels de fonction à la bibliothèque GOMP
  - Les autres ont été intégrées à la représentation GIMPLE
- Le coeur principal de la gestion OpenMP se situe dans le middle-end
- Gestion en 2 transformations (fichier `omp-low.c`)
  - `omp-low`
  - `omp-exp`
- Place dans le pass manager
  - Fichier `passes.c`
  - Fonction `init_optimizations_passes()`
  - `NEXT_PASS(pass_lower_omp)`
  - `NEXT_PASS(pass_expand_omp)`



# Middle-end - Exemple

- Revenons à notre exemple
  - Addition de 2 vecteurs
  - Fonction `vecAdd`
  - Fichier `test.c.008t.omplower`

```
void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 int i ;
 #pragma omp parallel for\
 private(i)
 for (i=0; i<N; i++) {
 c[i] = a[i] + b[i] ;
 }
}
```

```
void vecAdd(/* ... */) {
 struct .omp_data_s.0 .omp_data_o.1
 ;
 int i;
 .omp_data_o.1.a = a;
 .omp_data_o.1.b = b;
 .omp_data_o.1.c = c;
 .omp_data_o.1.N = N;
 // ...
 a = .omp_data_o.1.a ;
 b = .omp_data_o.1.b ;
 c = .omp_data_o.1.c ;
 N = .omp_data_o.1.N ;
}

void
vectAdd.omp_fn.0(.omp_data_o.1
) {
 .omp_data_i = &.omp_data_o.1
 D.3455 = .omp_data_i->N ;
}
```



# Middle-end - Exemple

- Revenons à notre exemple
  - Addition de 2 vecteurs
  - Fonction `vecAdd`
  - Ficheir `test.c.022t.ompexp`

```
void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 int i ;
 #pragma omp parallel for\
 private(i)
 for (i=0; i<N; i++) {
 c[i] = a[i] + b[i] ;
 }
}
```

```
void vecAdd(/* ... */) {
 struct .omp_data_s.0
 .omp_data_o.1 ;
 int i;
 .omp_data_o.1.a = a;
 .omp_data_o.1.b = b;
 .omp_data_o.1.c = c;
 .omp_data_o.1.N = N;
 // ...
 __builtin_GOMP_parallel_start
 (vectAdd.omp_fn.0,
 &.omp_data_o.1, 0);
 vectAdd.omp_fn.0(
 &.omp_data_o.1);
 __builtin_GOMP_parallel_end()
 ;
 // ...
}
```



# Directive et plugin

---

- Possibilité de déclarer de nouvelles directives dans un plugin
- Utilisation d'un évènement dédié
- Démo...