



Compilation Avancée

ENSIIE – S5

Cours 1 : Structure d'un compilateur



Organisation

- Thème
 - Compilation
 - Optimisation à la compilation
 - Lien avec modèles de programmation parallèle
- Intervenants
 - Patrick Carribault (patrick.carribault@cea.fr)
 - Antoine Capra (antoine.capra@eviden.com)
 - Van Man Nguyen (van-man.nguyen.ocre@eviden.com)
- Evaluation
 - Projet (code source, rapport, soutenance)
 - Binôme



Cours 1

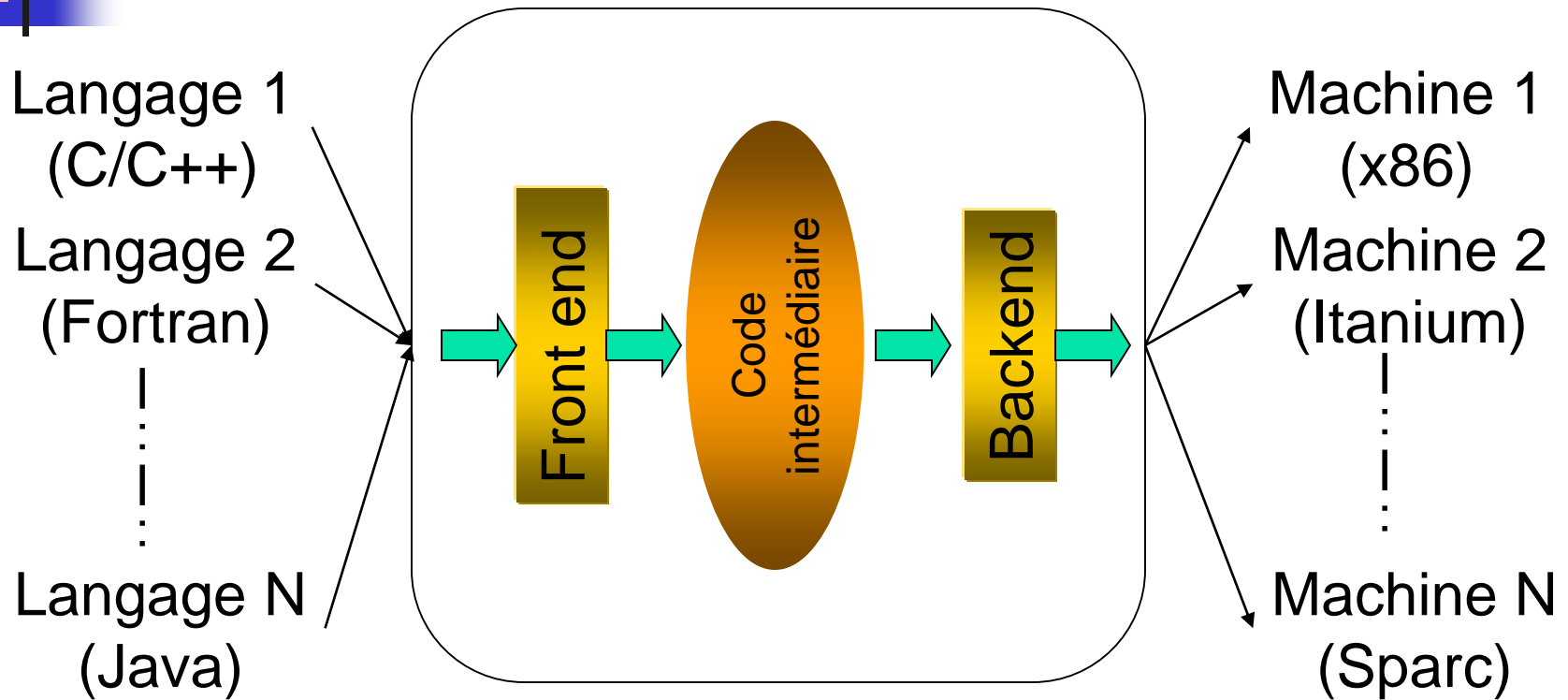
- Structure générale d'un compilateur
 - Vision d'un compilateur
 - Représentation intermédiaire
 - Notion de passes d'optimisation et de transformation
- Présentation de GCC
 - Introduction
 - Installation
- Introduction/rappel aux collectives MPI



Compilateur standard

- Définition
 - Traducteur de langages
 - Extensions : analyseur / optimiseur
- Vision boîte noire
 - Entrée : un langage de programmation
 - Sortie : un langage de programmation
- Principales parties
 1. Préprocesseur
 2. Cœur du compilateur
 3. Assembleur
 4. Linker
 5. Loader (exécution)
- Utilisation de bibliothèques/outils annexes

Compilateur standard



Vision idéaliste



Plan du cours

- Structure générale d'un compilateur
 - Vision d'un compilateur
 - Représentation intermédiaire
 - Notion de passes d'optimisation et de transformation
- Présentation de GCC
 - Introduction
 - Structure générale
 - Installation



Représentation intermédiaire

- *Code intermédiaire* ou langage intermédiaire (IR ou IL en anglais)
- Définition : réécriture d'un programme P1 d'un langage L1 vers un autre programme P2 d'un langage L2 tel que
- Contraintes
 - Conservation de la sémantique : P1 calcule la même chose que P2
 - Baisse du niveau d'abstraction : L2 est plus "près" de la machine cible (L2 est un langage "simplifié" par rapport à L1)



Sémantique

- Définition : le sens du programme, son but, son algorithme
- Changer la sémantique : c'est modifier le but ultime du programmeur
 - A tort ou à raison...
- Exemple : un code C faux

```
int *ptr = NULL ;  
*ptr = 5 ;
```
- Vis-à-vis du compilateur
 - un compilateur peut déterminer que ceci va planter
 - mais il se doit de faire ce que demande le programmeur



Sémantique

- Ce que fait le compilateur
 - Emet des avertissements lorsque le code est ambigu ou dangereux
- Exemple : un code C dangereux

```
if (x = 5)
    printf ("Hello world !") ;
```
- Vis-à-vis du compilateur
 - Rien ne dit que ceci n'est pas exactement voulu
 - Le compilateur va tout de même générer :

```
main.c :3 : warning : suggest
parentheses
```



Représentation intermédiaire et architecture

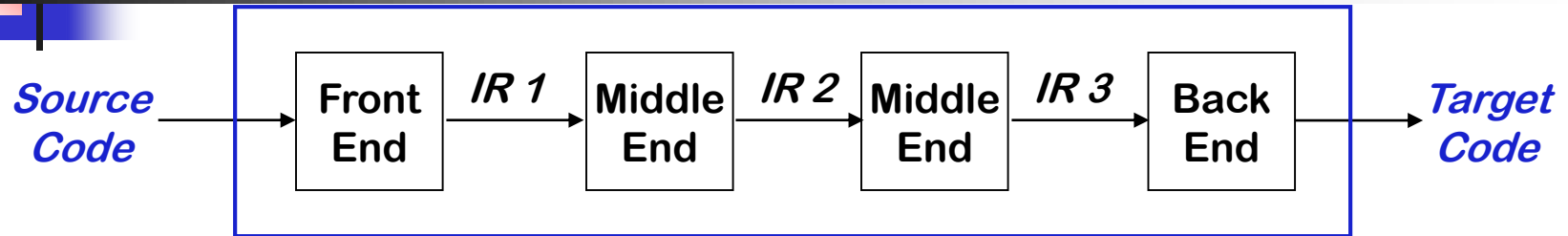
- Cas général des architectures
 - Programmes impératifs
 - Ressemblance avec le modèle von-Neumann
- Conséquences
 - Représentations intermédiaires dans ce genre de machines doivent être de langage impératif.
- Si la machine est *data flow* par ex, on aurait choisi une représentation intermédiaire data flow ou fonctionnelle.



Multiples représentations

- Complexité accrue des compilateurs
 - ➔ Plusieurs niveaux de représentations intermédiaires peuvent cohabiter
- But : procéder par étapes successives
 - Optimisation progressives
 - Plusieurs phases d'optimisations avant d'arriver au code binaire de la machine cible
- Compilateur ➔ succession de compilateurs en cascade.

multiples représentations



- Abaisser à plusieurs reprises le niveau d'abstraction de la forme intermédiaire
 - Chaque représentation intermédiaire est appropriée pour certaines optimisations
- Ex: compilateur Open64
 - Forme intermédiaire appelé WHIRL
 - Consiste en cinq représentations intermédiaires progressivement détaillées



Comment choisir ?

- Conception d'une forme intermédiaire
 - Affecte l'efficacité et la rapidité d'un compilateur
 - Affecte la qualité du programme généré
- Quelques critères de sélection
 - Facilité de génération
 - Facilité de manipulation
 - Taille de code induite
 - Liberté et puissance d'expression d'informations
 - Niveau d'abstraction
- L'importance de ces critères diffère selon les compilateurs
 - Sélectionner une forme intermédiaire pour un compilateur est une décision de conception importante !



Type de représentation intermédiaire (RI)

- On peut les classer en trois catégories majeures

- RI Linéaire (code textuel)

- Pseudo-code pour une machine abstraite
- Le niveau d'abstraction varie
- Simple et de taille plus compacte
- Facile à réécrire et manipuler

Exemples :

Code 3 adresses,
Code machine à pile

- RI structurée

- Utilise les graphes
- Beaucoup utilisée dans les traducteurs source à source
- Facilite une vision abstraite et globale d'un programme
- Nécessite une présence en mémoire qui peut être large

Exemples :

Arbres, DAGs

- Hybride

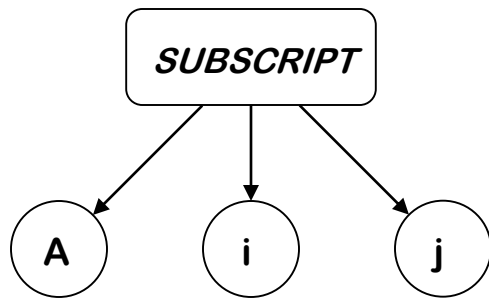
- Combinaison entre graphes et codes linéaires
- Cas le plus fréquent de nos jours

Exemple :

Graphe de flot de
contrôle

Niveau d'abstraction

- Le niveau de détails exposé dans une RI influence la profitabilité et la faisabilité de plusieurs optimisations.
- Ex : deux représentations possibles d'une référence à un élément de tableau $A[i,j]$.



Arbre syntaxique haut
niveau :
favorable pour une
désambiguation mémoire

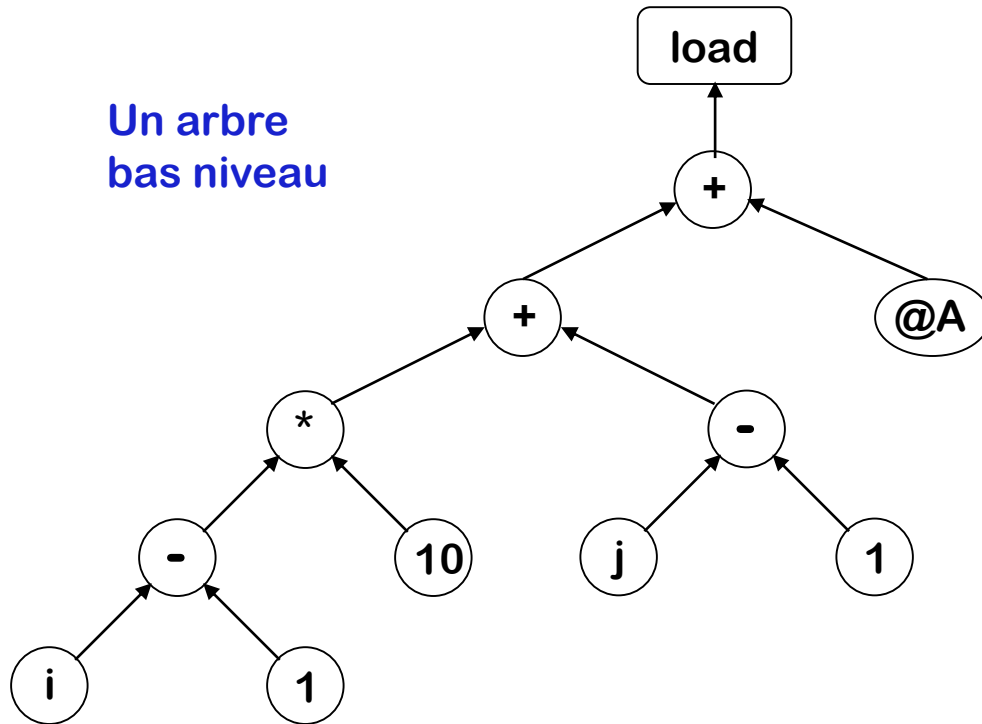
```
loadI 1      => r1
sub    rj, r1 => r2
loadI 10     => r3
mult   r2, r3 => r4
sub    ri, r1 => r5
add    r4, r5 => r6
loadI @A     => r7
Add    r7, r6 => r8
load   r8     => rAij
```

Code linéaire bas niveau :
Favorable pour le calcul d'adresse d'un élément

Niveau d'abstraction

- Une RI structurée est souvent considérée comme haut niveau
- Une RI linéaire est souvent considérée bas niveau.
- Ceci n'est pas nécessairement vrai!

Un arbre
bas niveau

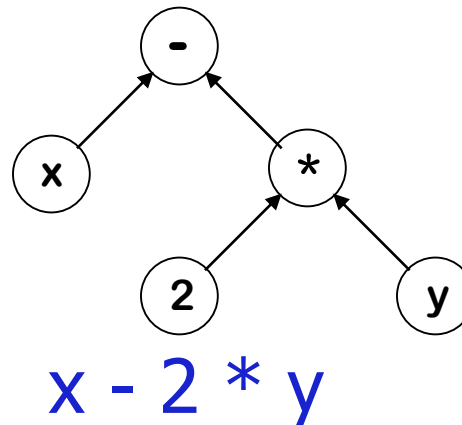


`loadArray A, i, j`

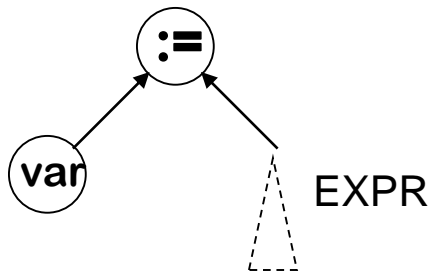
Code linéaire haut niveau

Arbre abstrait

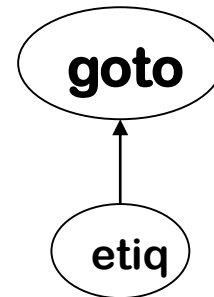
- Arbre abstrait : arbre syntaxique après avoir enlevé les nœuds des non-terminaux.
- Nœuds internes : opérateurs
- Feuilles : opérandes.



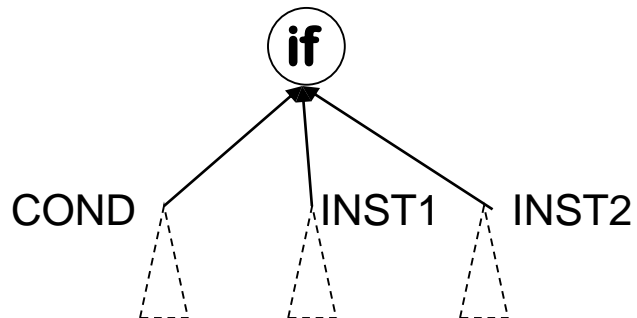
Arbre abstrait



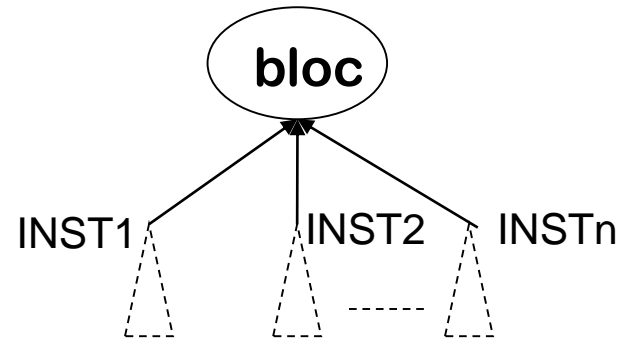
Var := EXPR



goto etiq



**if cond then INST1
else INST2**

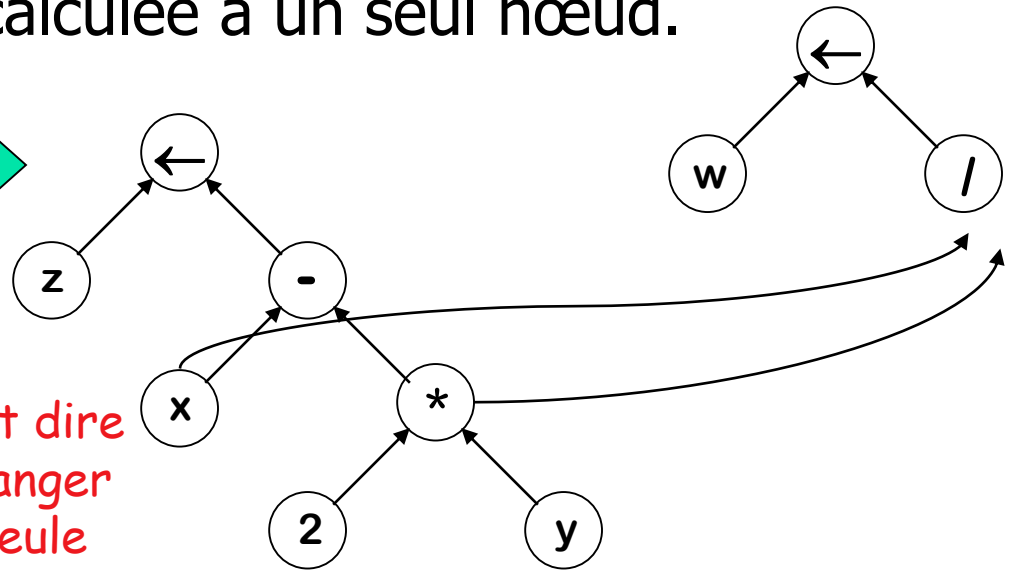
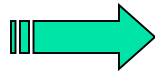


**{INST1; INST2;
...;INSTn;}**

Directed Acyclic Graph (DAG)

- C'est une forme optimisée de l'arbre abstrait.
- Chaque valeur calculée a un seul nœud.

$z \leftarrow x - 2 * y$
 $w \leftarrow x / (2 * y)$

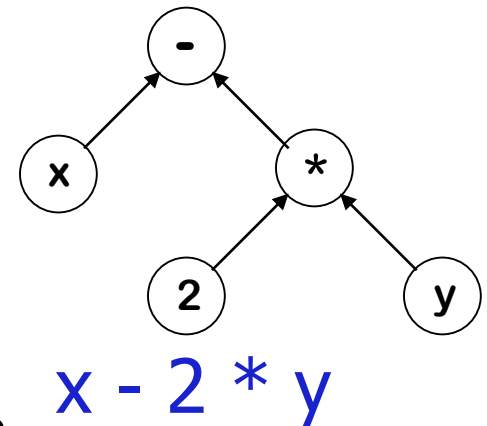


Dans un DAG, si un nœud est utilisé plusieurs fois, cela veut dire que le compilateur peut s'arranger pour qu'il ne l'évalue qu'une seule fois.

- Réutilisation des sous-expressions communes
- Encode la redondance de calcul

Formes pré et post-fixées

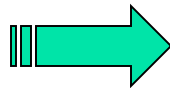
- C'est une linéarisation de l'arbre abstrait. Elles sont définies par un parcours récursif de cet arbre.
- Forme pré-fixée : visiter la racine ensuite le fils gauche suivi du fils droit
 - Ex: $- x * 2 y$
- Forme post-fixée : visiter le fils gauche suivi du fils droit, en enfin la racine
 - Ex: $x 2 y * -$



Code de machine à pile

- Utilisé à l'origine pour des machines n'ayant pas de registres, maintenant utilisé pour le bytecode java (JVM).
- Exemple :

$x - 2 * y$



```
push x  
push 2  
push y  
multiply  
subtract
```

Avantages

- Les noms utilisés sont *implicites*, et non *explicites*
- Simple à générer et à exécuter.
- Exercice : montrez comment il est facile de générer du code de machine à pile à partir d'une notation post-fixée, et vice-versa.

Code trois adresses

Il y a plusieurs représentations d'un code à trois adresses

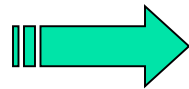
- En général, les instructions à trois adresses ont la forme :

$$x \leftarrow y \text{ op } z$$

avec un seul opérateur (op) et au plus trois opérandes (x, y, z)

Exemple:

$$z \leftarrow x - 2 * y$$



t	←	2	*	y
z	←	x	-	t

Caractéristiques :

- Ce code ressemble à celui de plusieurs machines (de moins en moins vrai).
- Introduit un ensemble de variables temporaires
- Forme compacte

Code trois adresses : Quadruplets

Représentation naïve de code trois adresses

- Table de $k*4$ entiers
- Structure simple
- Réordonnancer le code plus aisé
- Noms (et numéros de temp) explicites


```
load  r1, y
loadI r2, 2
mult  r3, r2, r1
load  r4, x
sub   r5, r4, r3
```

assembleur RISC



Le compilateur
FORTRAN d'origine
utilisait les "quads"

opérateur dest src1 src2



opérateur	dest	src1	src2
load	1	Y	
loadi	2	2	
mult	3	2	1
load	4	x	
sub	5	4	3

Quadruplets



Code trois adresses : Triplets

- L'indice de la table est utilisé pour implicitement indiquer les numéros du temporaire contenant le résultat de l'instruction.
- Economie de 25% d'espace comparé aux quads (on a éliminé une colonne)
- Plus difficile de réordonnancer le code

load	1	Y	
loadi	2	2	
mult	3	2	1
load	4	X	
sub	5	4	2

Quadruplets

(1)	load	y	
(2)	loadi	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(2)

Triplets

Code trois adresses : Triplets indirects

- Une optimisation des triplets qui utilise deux tables
 - une table contient une liste de triplets distincts
 - une table qui contient le code consistant en un ensemble de numéros de triplets.
- Ce n'est qu'une compression des triplets
- L'économie d'espace n'est pas au rendez-vous mais il est plus facile de réordonnancer le code.

(100)	load	y	
(101)	loadI	2	
(102)	mult	(100)	(101)
(103)	load	x	
(104)	sub	(103)	(102)

100	14	166	79	100	45	101	345
-----	----	-----	----	-----	----	-----	-----

Code : numéros des triplets

Table des triplets distincts



Triplets vs. Quadruplets

- Compromis entre les triplets et les quadruplet → la compacité versus la facilité de manipulation
 - Dans le passé, le temps de compilation et l'espace utilisé par le compilateur étaient des ressources critiques
 - De nos jours, la vitesse de compilation est un facteur plus important
 - Attention ! Empreinte mémoire toujours un souci pour les compilateurs optimisants
 - Les triplets et quadruplets peuvent paraître bien simples par rapport à la complexité des architectures/compilateurs actuels

Code à deux adresses

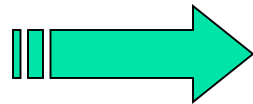
- Les instructions sont de la forme

$x \leftarrow x \text{ op } y$

Il y a un opérateur (*op*) et au plus deux opérandes (*x* et *y*). L'un d'eux est nécessairement détruit (affecté).

Exemple :

$z \leftarrow x - 2 * y$



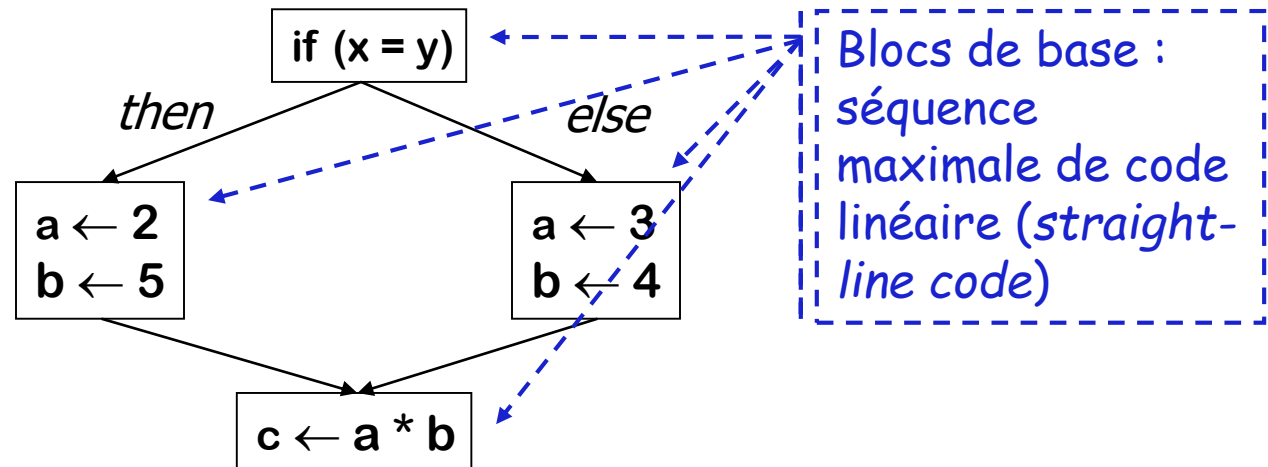
```
t1 ← 2
t2 ← load y
t2 ← t2 * t1
z ← load x
z ← z - t2
```

- Problème
- Beaucoup de machines ne se basent pas sur des opérations destructives (à effet de bord) : machines avec accumulateur
 - Les opérations destructives rendent difficile la réutilisation des temporaires (registres).

Intermédiaire hybride : Graphe de flot de contrôle

Modélise le transfert de contrôle dans un programme/fonction

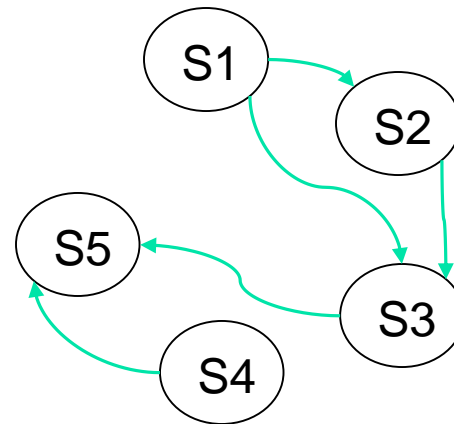
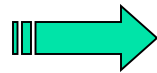
- Les nœuds représentent les blocs de base
 - Les instructions dans les blocs de bases peuvent être des quads, triplets, ou tout autre représentation intermédiaire
- Les arcs représentent le flot de contrôle (branchements, appels de fonctions, etc.)



Graphe de flot de données

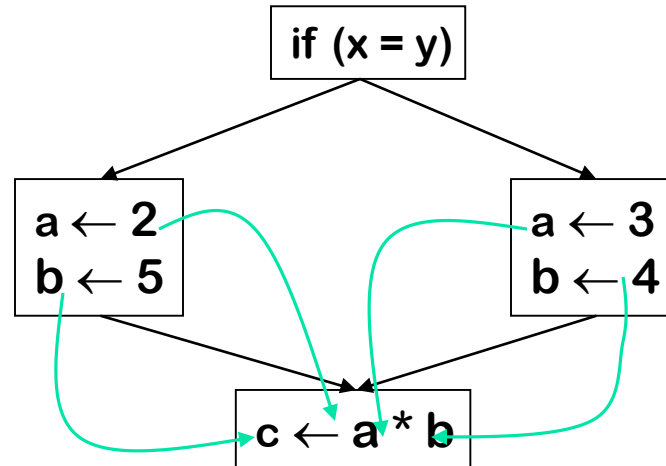
- Les nœuds sont les instructions.
- Un arc de a vers b veut dire que l'instruction a calcule un résultat lu par l'instruction b

S1:	t_1	\leftarrow	$x+y$
S2:	t_2	\leftarrow	$t_1 + 2$
S3:	t_2	\leftarrow	$t_2 * t_1$
S4:	z	\leftarrow	load x
S5:	z	\leftarrow	$z - t_2$



Intermédiaire hybride : Graphe de dépendances d'un programme

- C'est le graphe de flot de contrôle auquel on ajoute les arcs de dépendances de données.



Il évite de gérer en parallèle un graphe de contrôle et un graphe de dépendances de données



Blocs de base (*basic blocks*)

Un **bloc de base** est une séquence d'instructions *consécutives* où le flot d'exécution (contrôle) entre au début et ne sort qu'à la fin de la séquence

Seulement la dernière instruction d'un bloc de base peut être un branchement, et seulement la première instruction peut être la destination d'un branchement.



Algorithme de partition en blocs de base

1. Identifier les instructions de tête en utilisant les règles suivantes:
 - (i) La *première instruction* d'un programme est une instruction de tête
 - (ii) Toute instruction qui est une *destination d'un branchement* est une instruction de tête (dans la plupart des langages intermédiaires, ce sont des instructions avec des étiquettes)
 - (iii) Toute instruction qui suit *immédiatement* un branchement est une instruction de tête
2. Le bloc de base correspondant à une instruction de tête correspond à cette instruction, plus toutes les instructions suivantes, jusqu'à la prochaine instruction de tête exclue.



Exemple

Le code suivant calcule le produit cartésien de deux vecteurs

```
begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + a[i] * b[i]
    i = i+ 1;
  end
  while i <= 20
end
```

Code source

```
(1)  prod := 0
(2)  i := 1
(3)  t1 := 4 * i
(4)  t2 := a[t1]
(5)  t3 := 4 * i
(6)  t4 := b[t3]
(7)  t5 := t2 * t4
(8)  t6 := prod + t5
(9)  prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i <= 20 goto (3)
```

Code trois adresses



Exemple

Le code suivant calcule le produit cartésien de deux vecteurs

```
begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + a[i] * b[i]
    i = i+ 1;
  end
  while i <= 20
end
```

Code source

Règle (i)	(1) prod := 0
	(2) i := 1
	(3) t1 := 4 * i
	(4) t2 := a[t1]
	(5) t3 := 4 * i
	(6) t4 := b[t3]
	(7) t5 := t2 * t4
	(8) t6 := prod + t5
	(9) prod := t6
	(10) t7 := i + 1
	(11) i := t7
	(12) if i <= 20 goto (3)
	(13) ...

Code trois adresses



Exemple

Le code suivant calcule le produit cartésien de deux vecteurs

```
begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + a[i] * b[i]
    i = i+ 1;
  end
  while i <= 20
end
```

Code source

Règle (i)	(1) prod := 0
	(2) i := 1
Règle (ii)	(3) t1 := 4 * i
	(4) t2 := a[t1]
	(5) t3 := 4 * i
	(6) t4 := b[t3]
	(7) t5 := t2 * t4
	(8) t6 := prod + t5
	(9) prod := t6
	(10) t7 := i + 1
	(11) i := t7
	(12) if i <= 20 goto (3)
	(13) ...

Code trois adresses



Exemple

Le code suivant calcule le produit cartésien de deux vecteurs

```
begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + a[i] * b[i]
    i = i+ 1;
  end
  while i <= 20
end
```

Code source

Règle (i)	(1) prod := 0
	(2) i := 1
Règle (ii)	(3) t1 := 4 * i
	(4) t2 := a[t1]
	(5) t3 := 4 * i
	(6) t4 := b[t3]
	(7) t5 := t2 * t4
	(8) t6 := prod + t5
	(9) prod := t6
	(10) t7 := i + 1
	(11) i := t7
	(12) if i <= 20 goto (3)
Règle (iii)	(13) ...

Code trois adresses



Exemple

B1 (1) `prod := 0`
(2) `i := 1`

Blocs de base

B2 (3) `t1 := 4 * i`
(4) `t2 := a[t1]`
(5) `t3 := 4 * i`
(6) `t4 := b[t3]`
(7) `t5 := t2 * t4`
(8) `t6 := prod + t5`
(9) `prod := t6`
(10) `t7 := i + 1`
(11) `i := t7`
(12) `if i <= 20 goto (3)`

B3 (13) ...



Control Flow Graph (CFG)

Un ***graphe de flot de contrôle*** (CFG) est un multi-graphe orienté tel que :

(i) les nœuds sont les blocs de base et
(ii) les arcs représentent le flot de contrôle (ordre possible d'exécution)

- Le nœud de départ est celui qui contient la première instruction du programme
- Il peut y avoir plusieurs nœuds finaux car on peut avoir plusieurs "exits" dans le programme



Control Flow Graph (CFG)

- Il y a un arc orienté du bloc de base B1 vers le bloc de base B2 dans le CFG si :
 - (1) Il y a un branchement de la dernière instruction de B1 vers l'instruction de tête de B2, ou
 - (2) Le flot d'exécution peut passer de B1 à B2 si:
 - (i) B2 suit immédiatement B1, et
 - (ii) B1 ne finit pas avec un branchement inconditionnel

Exemple

Graphe de flot de contrôle:

Règle (2)

B1 (1) `prod := 0`
(2) `i := 1`



B2 (3) `t1 := 4 * i`
(4) `t2 := a[t1]`
(5) `t3 := 4 * i`
(6) `t4 := b[t3]`
(7) `t5 := t2 * t4`
(8) `t6 := prod + t5`
(9) `prod := t6`
(10) `t7 := i + 1`
(11) `i := t7`
(12) `if i <= 20 goto (3)`

B3 (13) ...

Exemple

Graphe de flot de contrôle:

B1 (1) `prod := 0`
(2) `i := 1`

Règle (1)

Règle (2)

B2

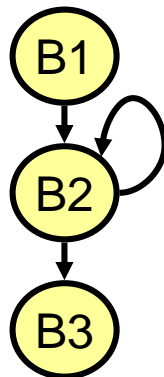
(3) `t1 := 4 * i`
(4) `t2 := a[t1]`
(5) `t3 := 4 * i`
(6) `t4 := b[t3]`
(7) `t5 := t2 * t4`
(8) `t6 := prod + t5`
(9) `prod := t6`
(10) `t7 := i + 1`
(11) `i := t7`
(12) `if i <= 20 goto (3)`

B3

(13) ...

Exemple

Graphe de flot de contrôle:



B1
(1) **prod** := 0
(2) **i** := 1

Règle (1)

Règle (2)

B2
(3) **t1** := 4 * **i**
(4) **t2** := **a**[**t1**]
(5) **t3** := 4 * **i**
(6) **t4** := **b**[**t3**]
(7) **t5** := **t2** * **t4**
(8) **t6** := **prod** + **t5**
(9) **prod** := **t6**
(10) **t7** := **i** + 1
(11) **i** := **t7**
(12) if **i** <= 20 goto (3)

Règle (2)

B3
(13) ...



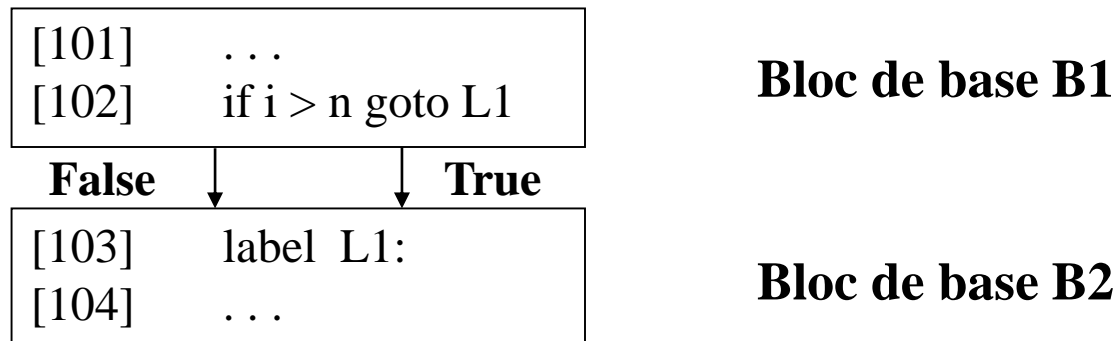
Les CFGs sont des multi-graphes

Note : il peut y avoir plusieurs arcs d'un bloc de base vers un autre dans un CFG.

Donc, en général, un CFG est un multi-graphe.

Les arcs peuvent être distingués par les étiquettes des conditions.

Un exemple trivial ci-dessous:





Graphe d'appels de fonctions

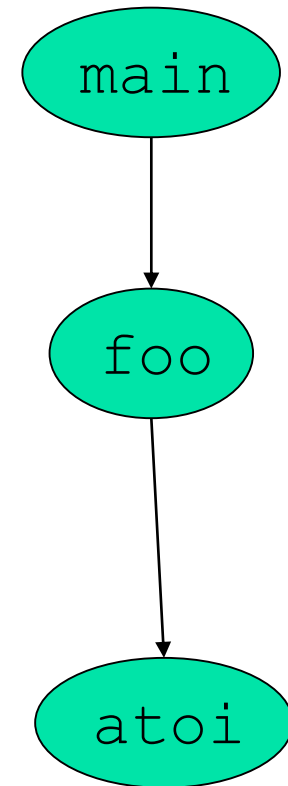
- C'est un graphe complémentaire au CFG
 - Il représente une vision de contrôle globale à l'application
 - Le CFG représente le contrôle dans une fonction
- Un nœud = une fonction
- Un arc entre f1 et f2 ssi f1 appelle f2



Graphe d'appels de fonctions

```
int main () {  
    ...  
    y=foo(5);  
    ...  
}
```

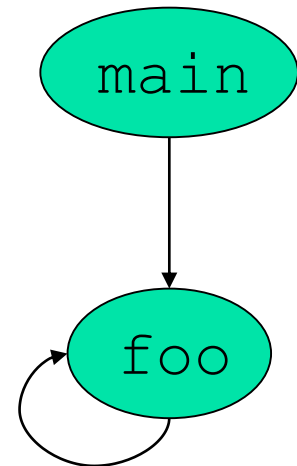
```
int foo (int y) {  
    int x=y+3;  
    if (x-5= 0)  
        x=atoi(...);  
    return x;  
}
```



Graphe d'appels de fonctions

```
int main () {  
    ...  
    y=foo(5);  
    ...  
}
```

```
int foo (int y) {  
    int x=y+3;  
    if (x-5= 0)  
        x=foo(x-1);  
    return x;  
}
```

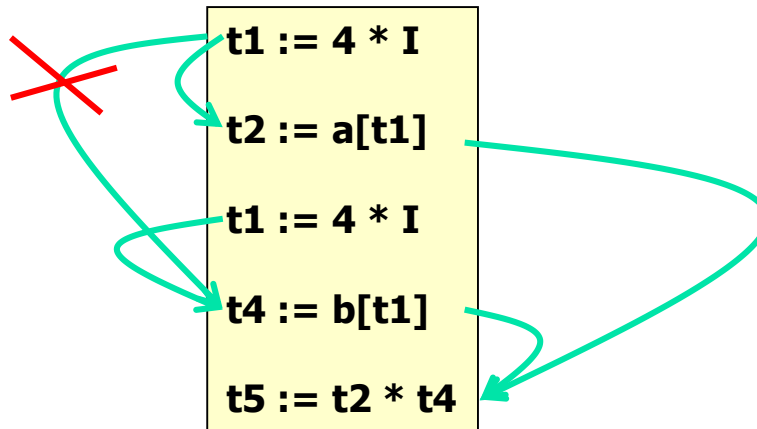




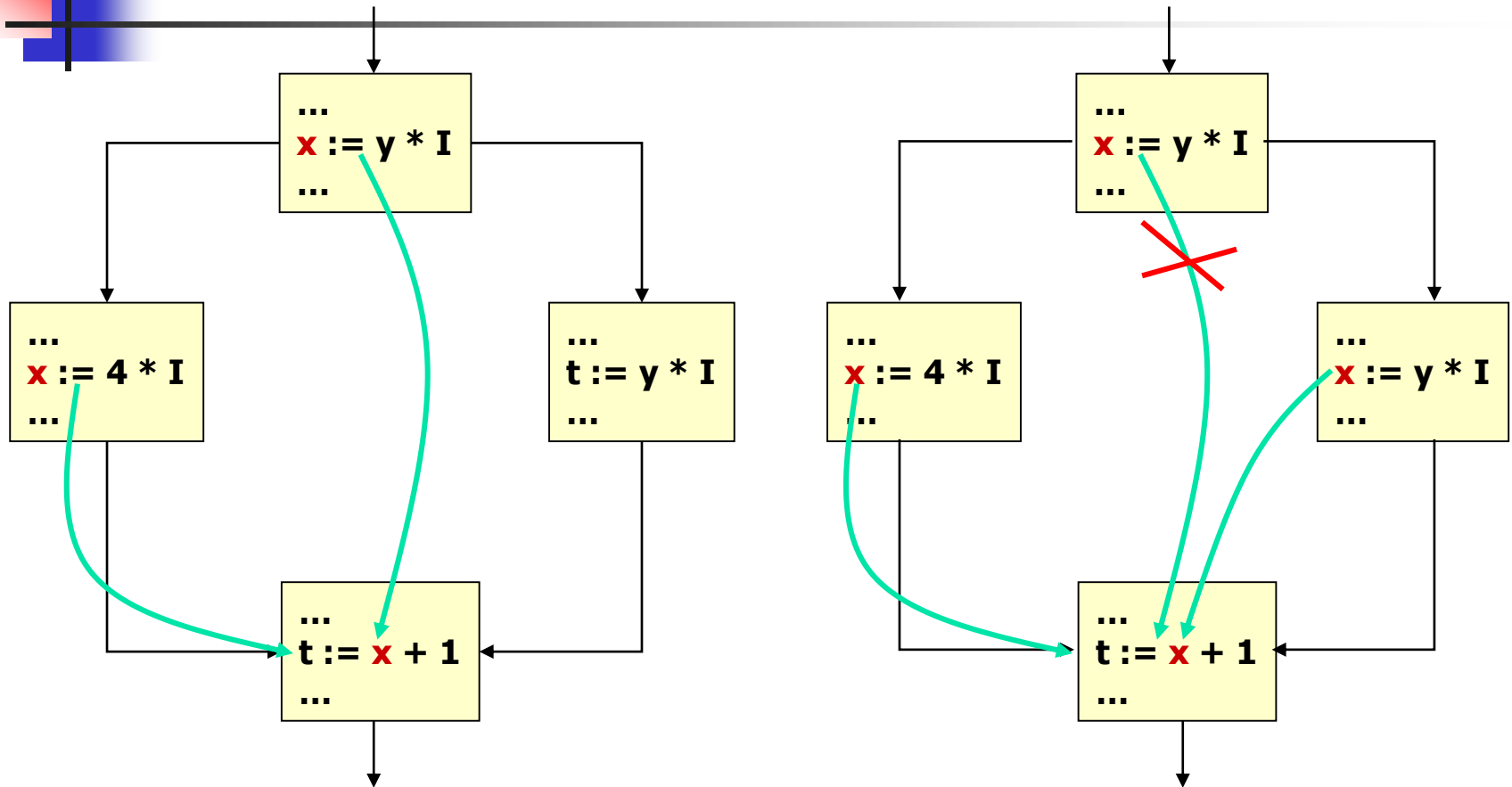
Flot de données

- On dit qu'il y a une dépendance de flot de données entre une instruction i_1 et une instruction i_2 si i_1 produit un résultat lu par i_2 .
 - Notion de dépendance de données vu dans le précédent cours
- Le problème de détection des dépendances de flot de données est indécidable dans le cas général :
 - Existence de pointeurs, de branchements, etc.

Cas simple : scalaires dans blocs de base



Cas simple : scalaires dans un CFG





Plan du cours

- Structure générale d'un compilateur
 - Vision d'un compilateur
 - Représentation intermédiaire
 - Notion de passes d'optimisation et de transformation
- Présentation de GCC
 - Introduction
 - Installation
- Introduction/rappel aux collectives MPI



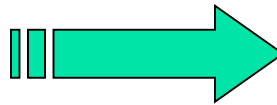
Représentation intermédiaire et passes

- Intérêt de la représentation intermédiaire
 - Permettre d'appliquer des passes sur le code
 - Passe = analyse, transformation ou optimisation
- Application d'une passe
 - Entrée : code en forme intermédiaire
 - Sortie : code transformé en même forme intermédiaire
- Analyse
 - Utilisation en lecture seule de la représentation intermédiaire
 - Mise à jour parfois de certaines informations
- Transformation
 - Modification de la structure du code permettant d'appliquer plus *facilement* une autre passe
 - Exemple : forme SSA (*Static Single Assignment*)

Forme SSA : *Static Single Assignment*

- Représentation textuelle du graphe de flot de données
- Idée principale : chaque variable est définie/écrite une seule fois
- Utiliser une fonction abstraite appelée ϕ -fonction pour restaurer le flot de données

```
Code d'origine
x ← ...
y ← ...
while (x < k)
    x ← x + 1
    y ← y + x
```



```
forme SSA
    x0 ← ...
    y0 ← ...
    if (x0 > k) goto next
loop:  x1 ←  $\phi(x_0, x_2)$ 
       y1 ←  $\phi(y_0, y_2)$ 
       x2 ← x1 + 1
       y2 ← y1 + x2
       if (x2 < k) goto loop
next:  ...
```

Caractéristiques :

- Décrit le flot de données = sémantique
- Nécessite et permet une analyse de code plus pointue
- (parfois) des algorithmes d'analyse et d'optimisation plus rapides



Passé d'optimisation

- But final de chaque optimisation
 - Minimiser ou maximiser une certaine fonction de coût
- Exemples
 - Réduction du temps d'exécution du code (*time to solution*)
 - Réduction de l'empreinte mémoire
 - Réduction de l'énergie consommée
- Modèle de coût dépendant du domaine
 - Calcul haute performance : réduction du temps d'exécution
 - Architecture embarquée (système enfoui) : réduction de l'énergie consommée
- Plusieurs modèles peuvent rentrer en jeu
 - Réduction de l'empreinte mémoire pour le HPC et l'embarqué
- Exemple
 - Réduction du nombre d'instruction
 - Sélection d'une instruction moins coûteuse
 - $x * 2 \rightarrow x + x$ ou $x \ll 1$



Passe d'optimisation

- Principe : heuristiques avant tout
- Heuristiques
 - Beaucoup d'optimisations sont basées sur des heuristiques
 - La latence mémoire par exemple n'est pas forcément connue au moment de la compilation
 - La position du code en mémoire (problème d'I-Cache)
 - Le nombre de registres potentiel pour un bout de code
- Mais comment décider ?
 - Benchmarks, benchmarks, ...
 - *If you get 1 percent better performance, commit!*
 - Restriction sur le temps de chaque passe (optimisation/transformation) : complexité en $O(N)$ avec N le nombre d'instructions



Compromis

- Compromis mémoire versus temps
 - Souvent considérer comme antagonistes puisque
 - Faire un pré-calcul et le stocker en mémoire permet de ne plus refaire le calcul
- Il faut donc décider au niveau de la compilation quelle est la priorité
- Exemple

```
for (i=0 ; i < N; i++) {  
    tab[i] = 5 * a + i ;  
}
```



Compromis

- Compromis mémoire versus temps
 - Souvent considérer comme antagonistes puisque
 - Faire un pré-calcul et le stocker en mémoire permet de ne plus refaire le calcul
- Il faut donc décider au niveau de la compilation quelle est la priorité
- Exemple

```
tmp = 5 * a ;  
for (i=0 ; i < N; i++) {  
    tab[i] = tmp + i ;  
}
```




Ordre des passes

- Mais dans quel ordre appliqué les passes ?
 - L'ordre a son importance
- Souvent une optimisation va modifier le code
- Mais l'optimisation qui va suivre n'aime pas forcément le nouveau format
- Beaucoup de recherche a été faite pour déterminer le meilleur ordre
 - Aucune solution n'est parfaite
 - Cela dépend de l'application, de la fonction, de la boucle
- Nous en revenons aux heuristiques, aux tests, à l'intuition
 - Notion de *pass manager* dans les compilateurs



Ordre des passes

- Exemple

```
a = 3 ;  
for (i=0 ; i < N; i++)  
{  
    tab[i] = 5 * a + i ;  
}
```

- Passes choisies

- Dead code elimination
- Propagation de constante

- Après dead code elimination

```
a = 3 ;  
for (i=0 ; i < N; i++)  
{  
    tab[i] = 5 * a + i ;  
}
```

- Après constant propagation

```
a = 3 ;  
for (i=0 ; i < N; i++)  
{  
    tab[i] = 15 + i ;  
}
```



Ordre des passes

- Exemple

```
a = 3 ;  
for (i=0 ; i < N; i++)  
{  
    tab[i] = 5 * a + i ;  
}
```

- Passes choisies

- Propagation de constante
- Dead code elimination

- Après constant propagation

```
a = 3 ;  
for (i=0 ; i < N; i++)  
{  
    tab[i] = 15 + i ;  
}
```

- Après dead code elimination

```
for (i=0 ; i < N; i++)  
{  
    tab[i] = 15 + i ;  
}
```



Compilateurs GCC



Plan du cours

- Structure générale d'un compilateur
 - Vision d'un compilateur
 - Représentation intermédiaire
 - Notion de passes d'optimisation et de transformation
- Présentation de GCC
 - Introduction
 - Installation
- Introduction/rappel aux collectives MPI



Chaîne de compilation GNU

- GCC : GNU Compiler Collection
 - Historiquement GNU C Compiler
- Ensemble d'outils et de bibliothèque pour la compilation
 - Plusieurs langages, plusieurs architectures
 - Générateur de compilateurs !
- Disponible sous licence GPL
 - <http://gcc.gnu.org>
- Support principal des TDs/TPs !



Survol des fonctionnalités

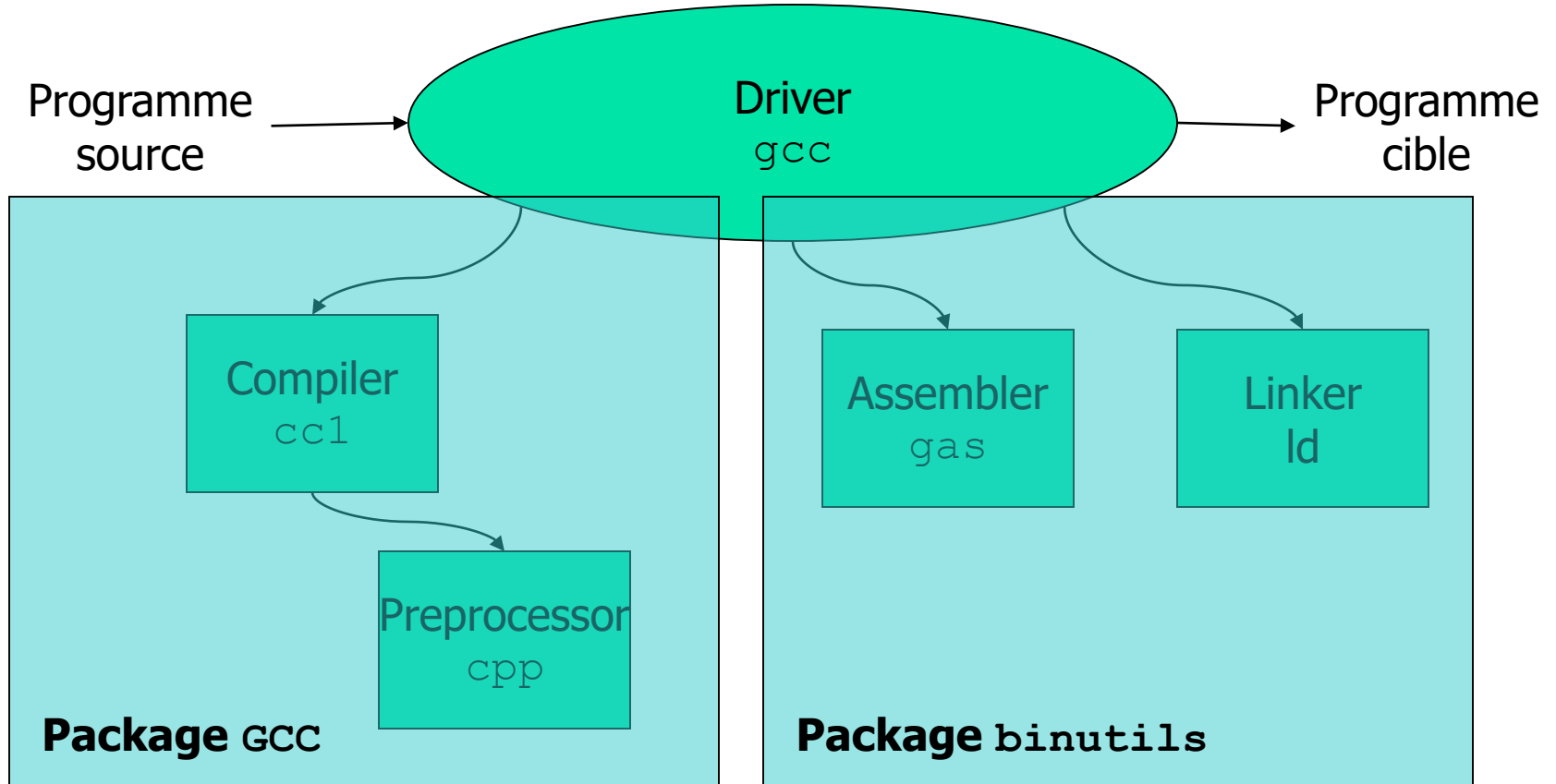
- Langages supportés
 - C, C++
 - Objective-C, Objective-C++
 - JAVA,
 - Fortran
 - ADA
- Processeurs supportés
 - ARM, IA-32 (x86), x86-64, IA-64, MIPS, SPARC, ...
- Système de *plugins* pour ajouter/modifier des passes de compilation
- Combien de lignes de code pour GCC ?



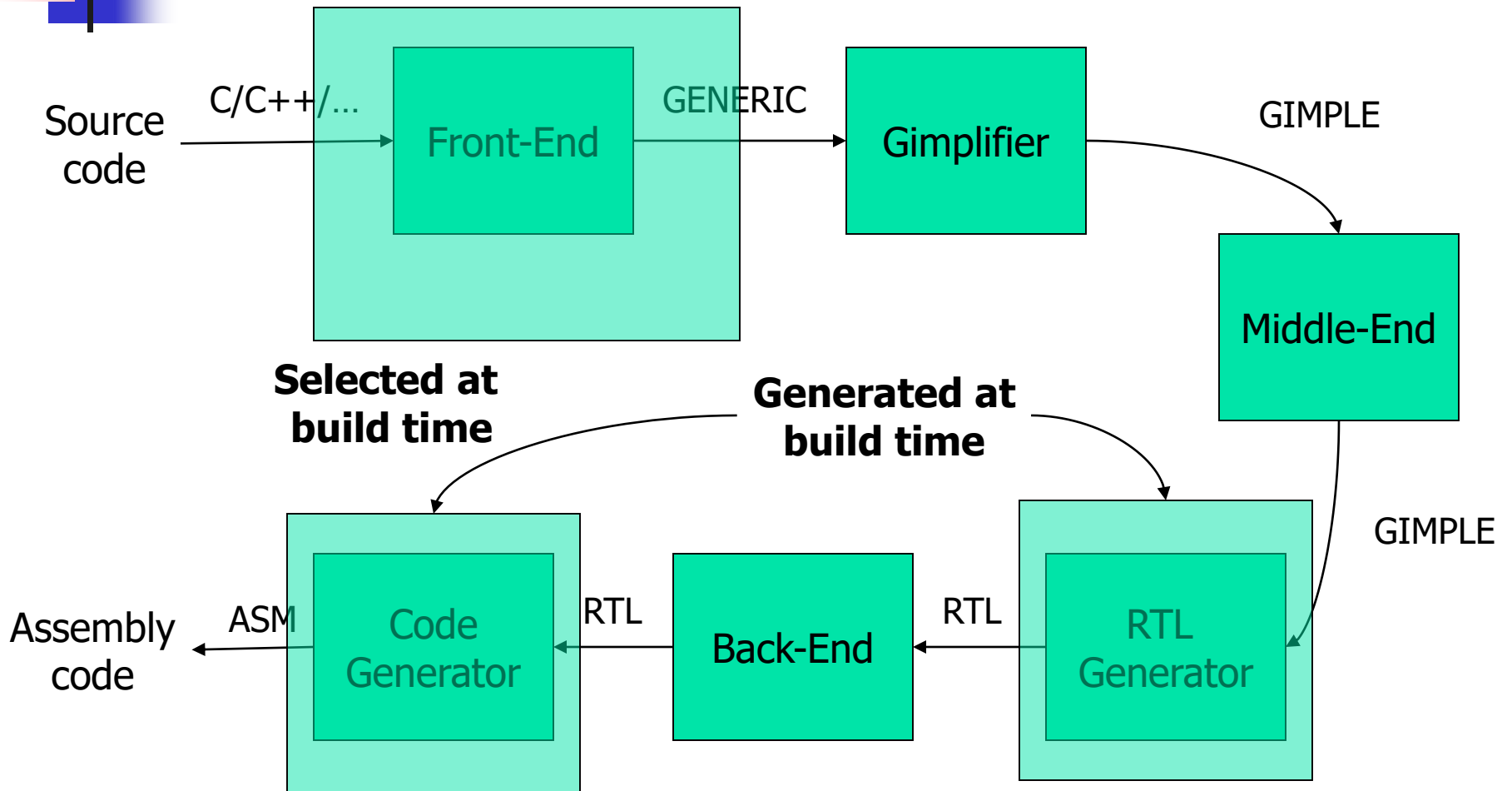
Plan du cours

- Structure générale d'un compilateur
 - Vision d'un compilateur
 - Représentation intermédiaire
 - Notion de passes d'optimisation et de transformation
- Présentation de GCC
 - Introduction
 - Structure générale
 - Installation

Architecture de GCC



Architecture de GCC





Transformations dans GCC

- GCC possède un total de 203 passes de transformations
- Le nombre total de passes effectuées lors d'une compilation est 239
 - Certaines transformations sont appelées plusieurs fois
- Pour l'enchaînement des transformations sur les représentations intermédiaires, GCC utilise un *pass manager*
 - Situé dans les fichiers `${SOURCE}/gcc/passes.c` et `${SOURCE}/gcc/passes.def`



Assembleur

- Le cœur du compilateur génère un fichier assembleur ASCII à la fin de la chaîne de compilation (sortie du back-end)
- Outil assembleur : traduction ASCII vers binaire
 - Simple traduction
- GCC utilise l'assembleur du système d'exploitation : `GAS` (package `binutils`)



Linker

- Collecte des fichiers objets pour la création de l'exécutable final
- Mise à jour des symboles pour les appels dynamiques
- Finalisation de quelques optimisations (par exemple *Thread Local Storage* ou TLS)



Plan du cours

- Structure générale d'un compilateur
 - Vision d'un compilateur
 - Représentation intermédiaire
 - Notion de passes d'optimisation et de transformation
- Présentation de GCC
 - Introduction
 - **Installation**
- Introduction/rappel aux collectives MPI



Installation de GCC

- Site web (documentation, téléchargement, ...)
 - GCC : [http ://gcc.gnu.org/](http://gcc.gnu.org/)
 - Version 12.2 actuellement (version pour TD/Projet)

- Dépendances (bibliothèques)

- GMP
 - MPFR
 - MPC

- Configuration

- Création d'un sous-répertoire travail

```
./configure --prefix=chemin-vers-travail --enable-languages=c,c++ --  
enable-plugin
```

- Compilation

- `make && make install`



Installation de GCC

- Après l'étape `make install`
 - GCC est installé dans le répertoire donné avec l'option `-prefix` lors de la configuration
- Utilisation
 - Modification du PATH

```
export PATH=chemin-vers-travail/bin:$PATH
```
 - `gcc -v` devrait vous donner la version 12.2 et la ligne de configuration que vous avez mis
- Modification du compilateur
 - On modifie ce qu'on veut et ensuite

```
make && make install
```




Documentation de GCC

- Documentation principale
 - Le code de GCC
- Important : il faut pouvoir lire le code de GCC pour comprendre comment cela fonctionne
 - Ne pas hésiter à parcourir les fichiers sources du cœur du compilateur
- Souvent la solution existe dans une autre partie de GCC
- Autre documentation de référence
 - The GCC internals
 - [http ://gcc.gnu.org/onlinedocs/gccint/Plugins.html#Plugins](http://gcc.gnu.org/onlinedocs/gccint/Plugins.html#Plugins)
- Exemple du PDF...



Plan du cours

- Structure générale d'un compilateur
 - Vision d'un compilateur
 - Représentation intermédiaire
 - Notion de passes d'optimisation et de transformation
- Présentation de GCC
 - Introduction
 - Installation
- Introduction/rappel aux collectives MPI



Echange de messages

- Caractéristiques des messages
 - La source
 - La destination
 - Les données à échanger

- Vue globale du protocole
 - Deux actions permettent de réaliser l'échange du message
 - La source doit envoyer le message
 - Considérons une fonction nommée *send*
 - Le destinataire doit recevoir le message
 - Considérons une fonction nommée *recv*



Principe

- Considérons deux “travailleurs” **T0** et **T1**
 - Espaces mémoires distincts et disjoints
 - Chaque travailleur a son ensemble d’instructions à exécuter

T0

instruction1;
instruction2;

T1

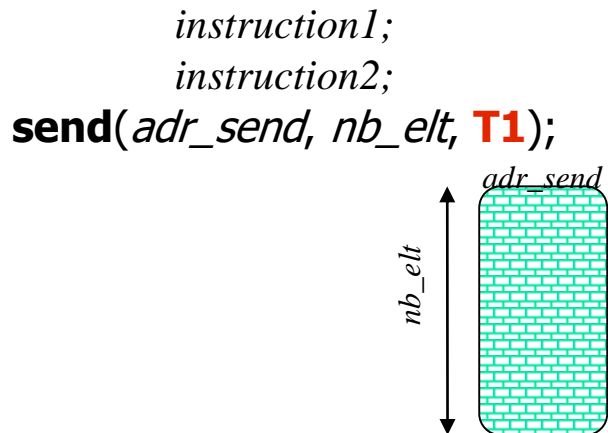
instruction1;
instruction2;

Principe

- **T1** dépend de **T0**
 - **T0** doit envoyer des données à **T1**
 - **T1** ne peut pas continuer son exécution sans les données de **T0**
 - Données situées à l'adresse *adr_send*, il y a *nb_elt* éléments

T0

T1

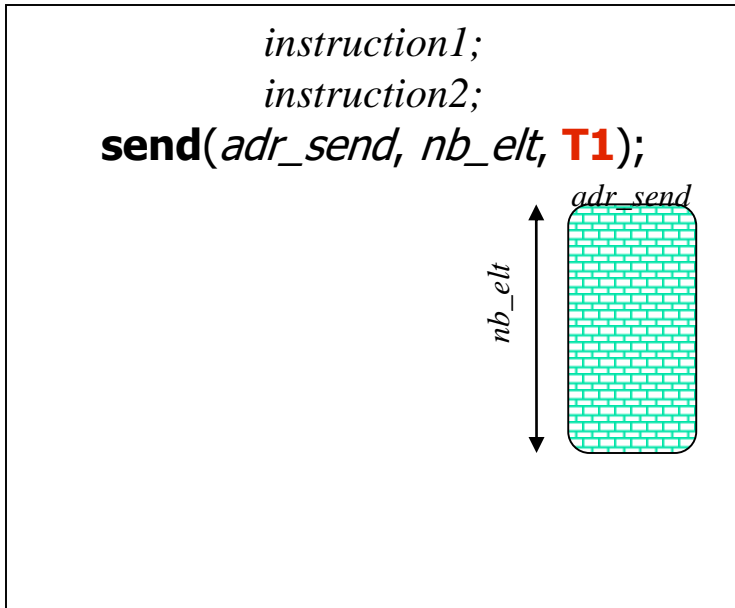


instruction1;
instruction2;

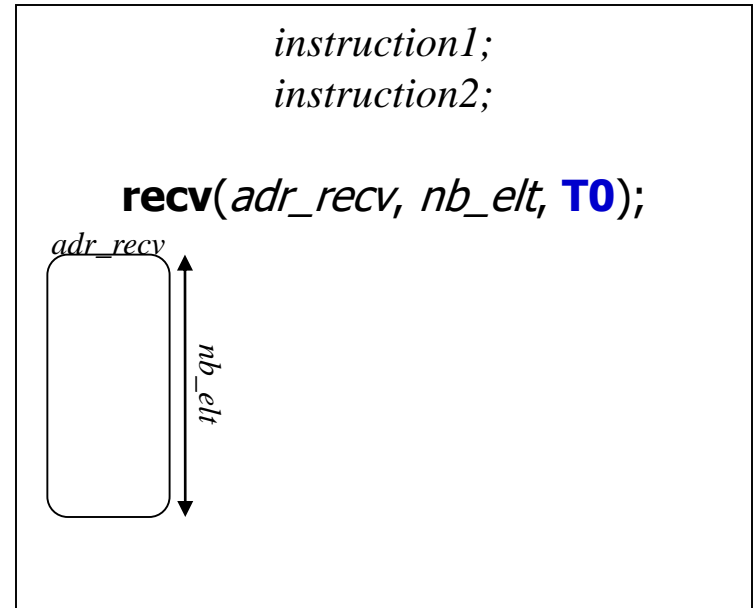
Principe

- **T1** doit recevoir les données de **T0** (*recv*)
 - La taille du message *nb_elt* doit être connue par le destinataire
 - Le destinataire peut avoir à allouer une zone mémoire pour recevoir les données (zone pointée par l'adresse *adr_recv*)

T0



T1

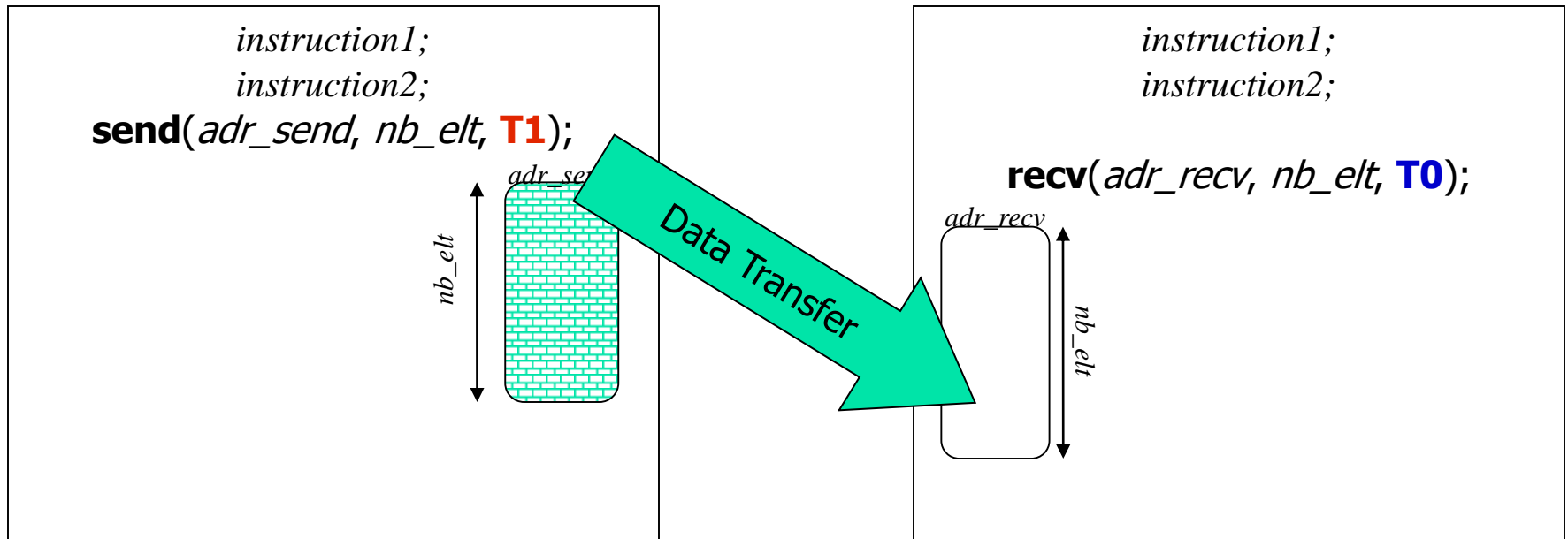


Principe

- Communication
 - *send* bloque **T0** tant que les données ne sont pas envoyées
 - *recv* bloque **T1** tant que les données ne sont pas reçues

T0

T1

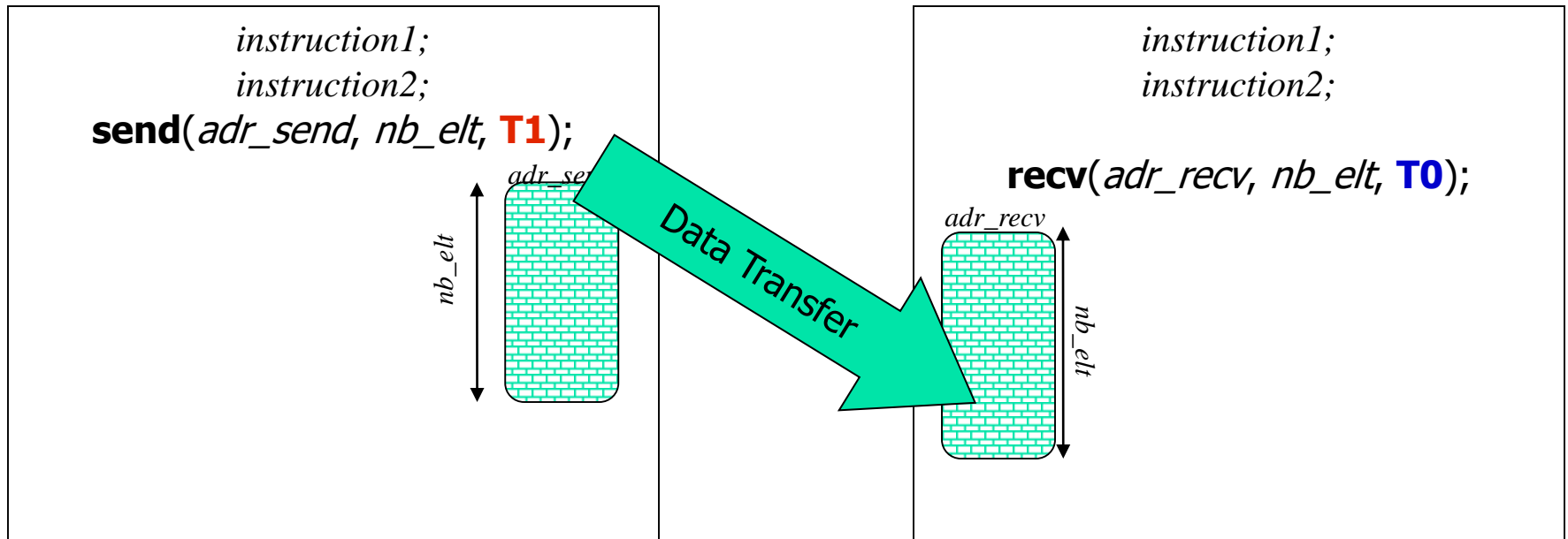


Principe

- Communication
 - *send* bloque **T0** tant que les données ne sont pas envoyées
 - *recv* bloque **T1** tant que les données ne sont pas reçues

T0

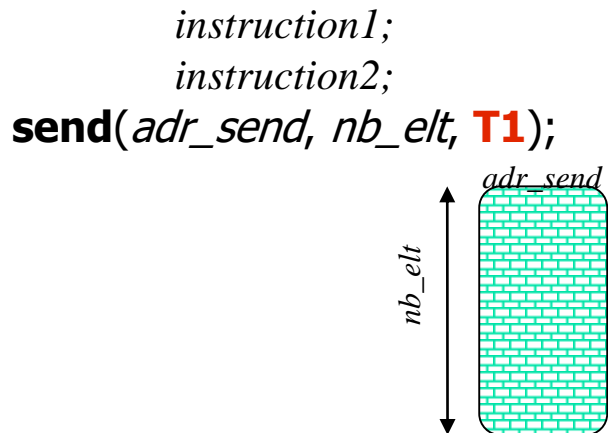
T1



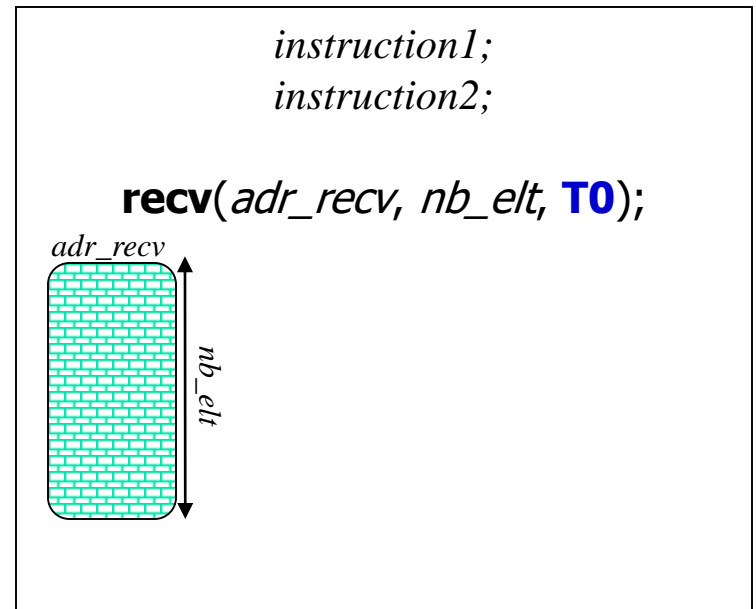
Principe

- **T1** possède une copie complète des données envoyées par **T0**

T0



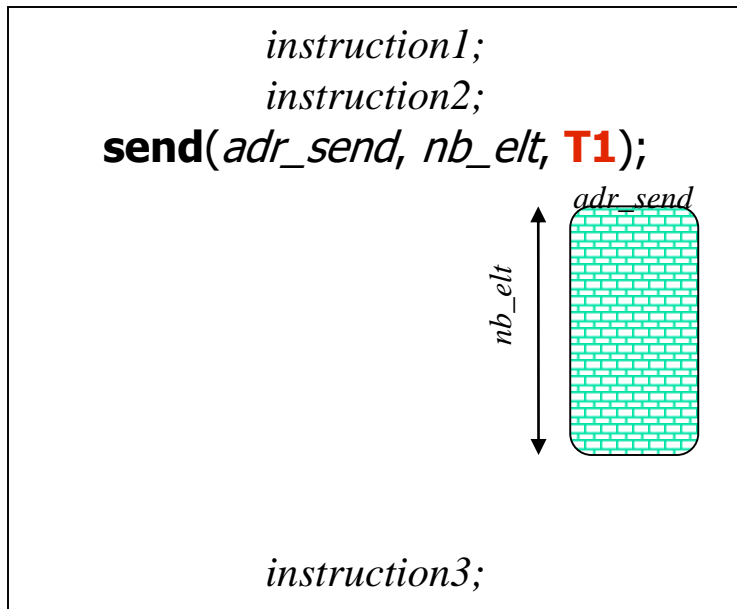
T1



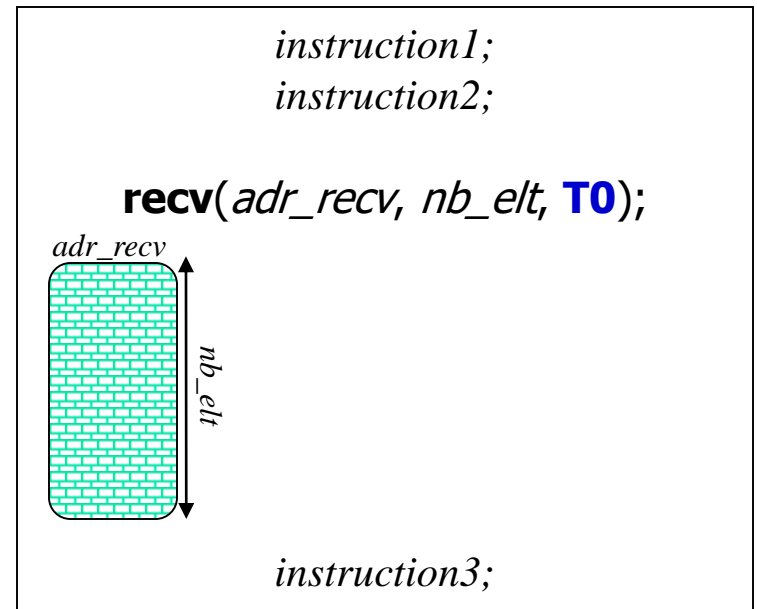
Principe

- Les travailleurs **T0** et **T1** peuvent continuer leur exécution
- Les instruction suivantes de **T1** peuvent accéder aux données stockées à l'adresse *adr_recv*

T0



T1





Communication MPI

- MPI est un modèle de programmation parallèles à mémoire distribuée
 - Chaque *processus MPI* accède à son propre espace mémoire
 - Basé sur le passage de message
- Quel est l'interface principal pour les échanges de données dans MPI?
 - Un rang (ou processus MPI) envoie un message avec la fonction `MPI_Send`
 - Un rang (ou processus MPI) reçoit un message avec la fonction `MPI_Recv`



Remarques sur l'envoi de messages

- `MPI_Send` est une fonction bloquante
 - Au retour de `MPI_Send`, le processus MPI peut manipuler (i.e. écrire) le buffer de données contenant le message
 - `/!\` Cela ne veut pas forcément dire que
 - Le message a été reçu
 - Le message a été envoyé!
- Si le message n'est pas trop volumineux, une copie locale est réalisée par le runtime pour libérer le buffer utilisateur plus rapidement
 - Le message sera traité en interne et envoyé dès que possible par le runtime, sans avoir à bloquer le buffer utilisateur



Collectives

- Une communication collective est une communication entre plusieurs processus MPI (généralement plus que 2).
- Chaque opération collective requiert un communicateur
 - Dans MPI, un communicateur définit un ensemble de processus MPI
- Tous les processus appartenant au groupe identifié par le communicateur doivent appeler la fonction de communication collective
- Au retour d'un appel à une opération collective sur un processus MPI ne veut pas dire que la collective est finie
 - Cela veut juste dire que le buffer est libre



Collectives

- Si le processus MPI courant ne requiert pas le résultat de la collective mais ne fait que envoyer des données, il se comporte comme MPI_Send
 - Après l'appel à la collective, l'utilisateur peut utiliser le buffer
 - Cela ne veut pas dire que le message a été envoyé
- Si tous les processus MPI fournissent des données et requièrent le résultat, alors la collective est synchronisante
 - /!\ La barrière (MPI_Barrier) est la seule opération collective synchronisante officielle dans l'API MPI



Règles pour les collectives

- Que se passe-t-il si deux rangs appellent des collectives différentes?
- Si les rangs sont dans le même communicateur, cela est illégal
 - Au sein du communicateur spécifié dans l'appel à la collective, tous les processus MPI doivent appeler la fonction collective
 - Au sein d'un communicateur, tous les processus MPI doivent appeler la même séquence de communication collective (même nombre, même ordre, avec des arguments compatibles)
- Par contre, entre deux rangs dans des communicateurs différents, il n'y a pas de restrictions



Conclusion



Conclusion

- Structure générale d'un compilateur
 - 3 parties : Front End, Middle End et Back End
 - **Front end** : dépend du langage en entrée et génère un code en une représentation intermédiaire
 - **Middle-end** : application de passes (analyses/transformations/optimisations) et génération de multiples autres représentations intermédiaires
 - **Back-end** : génération de code (langage cible)
- Ordre des passes
 - Problème connu
 - Dépend du but du compilateur
 - Dépend de l'application
- Présentations succinctes de GCC et des communications collectives MPI pour le projet