Le présent document a été établi en exécution du contrat No. 88.017.01 passé par la Direction des Recherches, Etudes et Techniques (Délégation Générale pour l'Armement)

Introduction

Le Paralléliseur Interprocédural de Programmes Scientifiques PIPS est un compilateur-restructureur *source à source* qui transforme du code Fortran séquentiel en du code Fortran parallèle. La version actuelle de PIPS ne vise pas une machine particulière. L'objectif est simplement de détecter autant de parallélisme que possible.

Cette notice s'adresse à l'utilisateur averti qui a de bonnes connaissances en programmation et en parallélisation automatique. Il convient de porter son attention directement sur les exemples, en réservant la compréhension de toutes les explications à une lecture postérieure. Elle est restreinte à la version *batch* de PIPS. La version multifenêtre de PIPS, `wpips`, n'est pas présentée.

Nous présentons tout d'abord les principes et le vocabulaire de base. Nous montrons ensuite comment initialiser l'espace de travail qui va permettre de conserver les nouvelles versions du programme et les informations qui y sont attachées.

Nous détaillons ensuite la commande *Display* qui permet d'afficher directement des résultats à l'écran. Une utilisation plus complète des fonctionalités de PIPS nécessite l'utilisation d'autres commandes comme *Select* qui permet de choisir des options de compilation ou d'analyse, *Build* qui permet de calculer des informations ou de transformer le programme sans afficher les résultats et *Perform* qui permet d'effectuer des transformations de programme explicitement.

Nous donnons ensuite la liste des analyses et des transformations de programme qui peuvent être effectuées avec PIPS.

1 Principes du fonctionnement

Pour préserver les fichiers sources, pour permettre le développement de plusieurs versions et pour regrouper en un unique endroit l'ensemble des fichiers qui sont créés au cours de la compilation d'un programme, l'analyse et les trans-

formations de programme sont réalisées dans le cadre d'un **workspace** (espace de travail) qui n'est qu'un sous-répertoire du répertoire courant.

A cet espace de travail est associée une base de données qui indique à chaque instant quelles sont les informations disponibles et quelles sont les options choisies. Chaque **phase** du compilateur va ajouter dans le *workspace* de nouveaux fichiers, appelés **ressources** dans la terminologie PIPS. Plusieurs *phases* sont généralement enchaînées pour satisfaire une seule requête de l'utilisateur.

Afin d'assurer la cohérence de cet enchaînement, toutes les requêtes de l'utilisateur sont effectuées par **Display**, **Build** ou **Perform** ainsi que **Interchange**. Ces commandes calculent l'enchaînement nécessaire à la construction cohérente de la ressource demandée ou à l'application correcte d'une règle.

A un programme source donné peuvent naturellement correspondre plusieurs espaces de travail, contenant chacun des versions parallélisées différentes, obtenues par la sélection d'options de compilation différentes ou par l'application explicite de diverses transformations de programmes.

1.1 Notion de workspace

Pour travailler sur un code source, il faut donc commencer par créer un **workspace**, composé de **modules**. Plusieurs *workspaces* peuvent être créés à partir des mêmes fichiers sources, afin d'obtenir plusieurs versions d'analyses et de transformations d'un programme. Un **module** correspond à une procédure du programme associé au workspace.

1.2 Notion de ressource et de requête

Les **ressources** sont des objets typés, qui correspondent à des structures de données. Chacune d'entre elles est stockée d'une manière permanente dans un fichier portant le même nom. Elles sont entièrement gérées par le *contrôleur de ressources*, qui permet de disposer des ressources sur disque comme en mémoire. Les ressources de chaque type sont produites lors de l'exécution d'une **phase** particulière de PIPS, et sont relatives soit au programme tout entier, soit à un module particulier, qu'il s'agisse du programme principal, d'une procédure ou d'une fonction.

Les ressources sont créées directement par une requête explicite de l'utilisateur, ou indirectement parce qu'elles s'avèrent nécessaires au calcul d'une autre

ressource demandée, elle, explicitement.

On peut considérer les requêtes essentielles comme une action sur la base de ressources. Afin de préserver la cohérence de cette base de ressources, il ne faut manipuler ces ressources qu'avec les utilitaires PIPS décrits dans cette notice: `Display`, `Build`, `Perform`, `Interchange` ainsi que `Delete`.

1.3 Manipulations du workspace. Contrôleur de ressources

Afin de minimiser le temps de calcul et de rendre PIPS interactif malgré son caractère interprocédural, les ressources ne sont construites qu'à la demande. A l'intérieur de `Display`, `Build`, `Perform` et `Interchange`, des mécanismes de type *make* enchaînent récursivement l'exécution de chaque règle (ie. chaque phase) requise au moment où les ressources dont elle a besoin ont été produites. Ces mécanismes s'appellent dans PIPS les contrôleurs de ressources (pipsmake). Cela assure aussi la réutilisation des ressources, dans la mesure où pour chaque règle les ressources produites sont postérieures aux ressources requises.

1.4 Analyses et Transformations

De nombreuses phases d'analyse et de transformation du programme peuvent être appliquées successivement, sans souci de la cohérence des enchaînements, qui est prise en charge par le contrôleur de ressources. Ainsi l'utilisateur peut-il se concentrer sur le travail où sa connaissance du programme est réellement utile.

2 Création d'un workspace: Init

2.1 Environnement de PIPS

Pour utiliser PIPS, il faut pouvoir exécuter `Init`, `Build`, `Perform` et `Select`. Il convient d'ajouter à son path la directory où se trouvent ces shell-scripts et de définir quelques variables d'environnement. Pour cela, il suffit d'exécuter l'un des shell-scripts (selon votre shell usuel) fourni avec PIPS:

`. Pips/pipsrc.sh` (shell sh ou ksh) ou
`source Pips/pipsrc.csh` (shell csh ou tcsh).

Si vous utilisez PIPS plusieurs fois, cette commande devrait être placée dans votre fichier d'initialisation du shell.

Toutes ces commandes seront appelées depuis le répertoire où sont situés les fichiers sources du programme que vous voulez paralléliser.

PIPS nécessite beaucoup de place à la fois en mémoire et sur disque. Assurez-vous que vous disposez d'une capacité suffisante avant de commencer une nouvelle session...

2.2 Init

Usage: **Init workspace [file.f] ...**

Pour créer un workspace de nom `wspace` pour paralléliser les fichiers sources Fortran `src1.f` et `src2.f`, faire: `$ Init wspace src1.f src2.f`

Les structures créées sont les suivantes: *wspace.schema* qui décrit la base de ressources, *wspace.database*, répertoire qui comprendra toute la base de ressources, chaque ressource étant un fichier; elle est initialisée avec les SOURCE_FILE, correspondant au source de chacunes des procédures du programme (ou *modules*). Ces SOURCE_FILEs sont dérivés, à l'aide de l'utilitaire *fsplit*, à partir de vos propres fichiers sources, qui sont aussi vus comme des ressources, dites USER_FILE mais qui sont exceptionnellement en dehors du répertoire *wspace.database* et qui ne seront jamais modifiés. Chaque fichier SOURCE_FILE ne contient donc qu'un unique module et prend comme nom le nom de ce module.

Pour revenir à un workspace déjà existant, utiliser: `$ Init wspace`

Si vous n'êtes pas seul à travailler dans un même répertoire, il vous faut spécifier votre workspace à chaque commande (option `-w`).

2.3 Exemple

Soit le fichier matmul.f qui comprend les procédures mm et saxpy. Pour créer le programme mm1 à partir de ce fichier, placez-vous sous un répertoire qui contient matmul.f et tapez la commande:

```
$ Init mm1 matmul.f
```

Les messages qui notifient l'activité de PIPS sont nombreux afin de suivre les calculs réalisés. Remarquer simplement parmi ceux de la commande Init:

```
Spliting file    matmul.f
  Module         MM
```

```
Module          SAXPY
```

Nous savons donc que les modules de mm1 sont MM et SAXPY (qui peuvent être spécifiés en lettres minuscules dans les commandes suivantes).

Les fichiers créés sont `mm1.schema`, `mm1.database/mm.f` et `mm1.database/saxpy.f`

2.4 Destruction d'un workspace: Delete

Afin de supprimer un workspace de nom wspace, il suffit de faire: `Delete wspace`

Noter que les sources réels du programme (ressources USER_FILE) ne sont pas effacés par `Delete`.

3 Affichage des résultats: Display

3.1 Principe

La plupart des analyses et des transformations de programme produisent des résultats lisibles dans l'une des ressources PRINTED_FILE ou PARALLELPRINTED_FILE, construites avec les règles adéquates. Les transformations de code sont alors effectuées, et les analyses du code sont visibles en commentaire de ces fichiers Fortran. La fabrication des ressources et leur affichage est automatiquement réalisé par la commande `Display`, qui exécute un appel à `Build`.

3.2 Display

Usage: **Display [-w wspace] [-m module] pretty-print**

où les valeurs possibles de pretty-print sont: `para code tran prec comp prop cumu cg cgl cgc para90` , et la valeur par défaut est `para`.

Le workspace (resp. le module) courant est utilisé à moins qu'un nom ne soit spécifié par l'option `-w` (resp. `-m`).

pretty-print est une clé à laquelle sont associés des traitements PIPS qui aboutissent à la fabrication d'une ressource PRINTED_FILE ou CALLGRAPH_FILE ou encore PARALLELPRINTED_FILE; ces ressources sont des fichiers de texte qui permettent d'afficher les analyses et transformations de programme.

PRINTED_FILE construit avec:

- `code` donne le pretty-print de la représentation intermédiaire du module qu'utilise PIPS;
- `tran` donne en plus l'analyse sémantique des transformers en commentaire;
- `prec` fait de même avec les préconditions;
- `comp` fait de même avec les complexités;
- `prop` fait de même avec les effets propres;
- `cumu` fait de même avec les effets cumules.

CALLGRAPH_FILE construit avec:

- `cg` affiche le graphe des appels
- `cgl` donne en plus les boucles qui comprennent des appels
- `cgc` fait de même avec les structures de contrôle.

PARALLELPRINTED_FILE construit avec:

- `para` donne le pretty-print du code parallèle;
- `para90` donne aussi le pretty-print du code parallèle, mais avec la notation Fortran 90.

La ressource construite est affichée sur le fichier de sortie standard stdout, alors que les messages informatifs sortent sur le fichier d'erreur standard stderr.

3.3 Exemple

Pour visualiser la version parallèle du module MM de mm1, taper:

```
$ Display -m mm
```

Toutes les phases doivent s'enchaîner pour construire la ressource PARALLELPRINTED_FILE du module MM. Finalement, on obtient:

```
Display of file mm1.database/MM.parf
C
C     MATRIX MULTIPLICATION - VERSION WITH CALL TO SAXPY
C
C     PARALLELIZATION OF LOOPS INCLUDING CALLS TO PROCEDURE
C
      SUBROUTINE MM(N, A, B, C)
C
      REAL*8 A(N,N), B(N,N), C(N,N), XAUX(0:127)
C
      DOALL I = 1,N,1
         PRIVATE I
         DOALL J = 1,N,1
            PRIVATE J
            C(I,J) = 0.0                                              0006
         ENDDO
      ENDDO
C
      DOALL J = 1,N,1
         PRIVATE J
         DO K = 1,N,1
            PRIVATE K
            CALL SAXPY(N, C(1,J), A(1,K), B(K,J))                     0011
         ENDDO
      ENDDO
      RETURN
      END
```

Pour conserver dans le fichier mm1.mm.prec les préconditions de ce module, faire:

```
$ Display prec > mm1.mm.prec
```

Il faut noter que malgré sa grande utilité, la commande `Display` ne permet pas de bénéficier de toute la puissance de PIPS. Aussi faut-il savoir recourir à `Select` , `Perform`, `Build` et `Interchange`.

4 Choix des options: Select

Pour initialiser un *workspace*, il faut spécifier quelles sont les règles de construction des ressources. Une phase est une exécution d'une règle, et à chaque ressource est associée au moins une règle de production. Un exemple de règle par défaut est fourni dans le fichier `$LIBDIR/pipsmake.rc`

4.1 Fichier pipsmake.rc

Au premier appel de `Build`, `Perform` ou `Select`, un fichier `pipsmake.rc` est lu pour apprendre à PIPS les règles de production des ressources: celui du répertoire courant s'il existe, celui de `$LIBDIR` à défaut. Aussi pouvez-vous créer votre propre fichier d'initialisation de PIPS en copiant `pipsmake.rc` et en le modifiant (voir la syntaxe de ce fichier en annexe *pipsmake*).

Lorsque plusieurs règles sont disponibles dans `pipsmake.rc` pour créer une même ressource, la première d'entre elles est sélectionnée. Une autre règle pourra être sélectionnée à la place de la première au moyen de la commande `Select`.

Une règle sélectionnée sera au besoin utilisée pour une phase afin de calculer une ressource requise. C'est là le principe du *make*.

Une fois que `pipsmake.rc` a été lu, il ne sera plus jamais utilisé pour ce workspace, puisqu'une représentation interne du pipsmake est conservée et tenue à jour pour chaque workspace. Elle est stockée dans le fichier `wspace.pipsmake`.

4.2 Select

Usage: **Select [-w wspace] rule [rule] ...**

Le workspace courant est utilisé à moins qu'un nom ne soit spécifié par l'option `-w`.

Les règles à sélectionner sont spécifiables soit par un nom interne à PIPS (ex. `rice_full_dependence_graph`), soit par un alias géré par le shell-script `Build` (ex. `rfulldg`). Voir l'annexe *alias*.

4.3 Exemple

Soit le programme `choles.f`. Montrons qu'il est nécessaire de sélectionner la règle

`rice_full_dependence_graph` pour le paralléliser:

`$ Init c2 choles.f`

Un seul module: CHOLES.

`$ Perform -m choles privatizer`

Ainsi sont privatisées les variables pour lesquelles cette transformation est licite.

`$ Display para >c2.para`

Mais ce pretty-print n'est pas parallèle:

```
$ cat c2.para
C
C     CHOLESKI METHOD - VERSION 1
C
C     PRIVATIZATION
C     DEPENDENCE COMPUTATION WITH AND WITHOUT EXECUTION CONTEXT
C
      SUBROUTINE CHOLES(A, P, N)
      REAL X, A(N,N), P(N)
C
      DO I = 1,N,1
         PRIVATE I,KK,J,K,X
         X = A(I,I)                                              0004
         DO K = 1,(I-1),1
            PRIVATE K
            X = (X-(A(I,K)*A(I,K)))                              0007
         ENDDO
         P(I) = (1.0/SQRT(X))                                    0008
         DO J = (I+1),N,1
            PRIVATE J,KK,X
            X = A(I,J)                                           0011
            DO KK = 1,(I-1),1
               PRIVATE KK
               X = (X-(A(I,J)*A(I,KK)))                          0014
            ENDDO
            A(J,I) = (X*P(I))                                    0015
         ENDDO
      ENDDO
      RETURN
```

```
      END
```

Donc il convient de choisir une règle qui implémente un algorithme qui tienne compte des préconditions:

`$ Select rfulldg`

Ainsi la règle RICE_FULL_DEPENDENCE_GRAPH est sélectionnée.

`$ Display para >c2.para`

Qui calcule une version effectivement parallèle:

```
$ cat c2.para
C
C     CHOLESKI METHOD - VERSION 1
C
C     PRIVATIZATION
C     DEPENDENCE COMPUTATION WITH AND WITHOUT EXECUTION CONTEXT
C
      SUBROUTINE CHOLES(A, P, N)
      REAL X, A(N,N), P(N)
C
      DO I = 1,N,1
         PRIVATE I,KK,J,K,X
         X = A(I,I)                                                    0004
         DO K = 1,(I-1),1
            PRIVATE K
            X = (X-(A(I,K)*A(I,K)))                                    0007
         ENDDO
         P(I) = (1.0/SQRT(X))                                          0008
         DOALL J = (I+1),N,1
            PRIVATE J,KK,X
            X = A(I,J)                                                 0011
            DO KK = 1,(I-1),1
               PRIVATE KK
               X = (X-(A(I,J)*A(I,KK)))                                0014
            ENDDO
            A(J,I) = (X*P(I))                                          0015
         ENDDO
      ENDDO
      RETURN
      END
```

4.4 Avertissement

Comme nous le constatons sur l'exemple précédent, certaines règles ne doivent pas être sélectionnées, mais appliquées. L'application consiste à demander l'exécution *immédiate* d'une règle spécifique; elle est toujours licite. Par contre, la sélection consiste à choisir la règle qui sera utilisée *ultérieurement* par défaut pour produire une ressource d'un certain type.

Toutes les règles ne peuvent pas être sélectionnées, afin de ne pas introduire de cycle dans l'enchaînement des règles. Par exemple, la règle de nom `distributer`, qui effectue la distribution de boucles, et la règle `privatizer`, qui privatise les variables scalaires dans les boucles, produisent toutes deux la ressource `CODE` à partir d'une ressource `CODE` qui doit être déjà disponible pour le même module. Il faut donc que reste sélectionnée une autre règle qui produise initialement `CODE` (`link` par défaut). Cela ne serait plus vrai si l'on sélectionnait `distributer` ou `privatizer` (cf. section *Application d'une règle: Perform*).

5 Demande d'une ressource: Build

5.1 Build

Usage: **Build [-w wspace] [-m module] [-s selected_rule[,selected_rule]...] [-p performed_rule] resource**

Le workspace (resp. le module) courant est utilisé à moins qu'un nom ne soit spécifié par l'option `-w` (resp. `-m`).

La règles et la ressource peuvent être spécifiées par leur vrai nom ou par un alias. Les alias sont généralement plus explicites mais aussi plus longs.

Les règles `selected_rule` éventuellement spécifiées avec l'option -s sont sélectionnées (cf. *Select*).

Ensuite la règle `perform_rule` éventuellement spécifiée avec l'option -p est appliquée (cf. *Perform*).

Pour finir, la ressource `resource` est calculée (build), à moins que ce ne soit une chaîne vide, auquel cas seuls le Select et le Perform sont effectués.

Il faut remarquer qu'un effet équivalent à celui d'un Select ou d'un Perform peut être obtenu en utilisant des options de Build[1].

[1]En fait, Select et Perform font un appel à Build qui effectue les conversions et vérifications requises puis appelle les binaires nécessaires. De cette façon, les alias peuvent

5.2 Application

Vous souhaitez une ressource qui ne soit pas visualisable à l'écran comme les effets d'une procédure ou un graphe de dépendance (cf. annexe *Base de ressources*). Pour l'obtenir, vous devez alors utiliser Build. Cela est utile à des fins de debug ou bien pour récupérer des résultats partiels à l'intention d'un autre logiciel. Le graphe de dépendance pourrait ainsi être réutilisé pour faire de l'ordonnancement d'instructions.

6 Application d'une règle: Perform

6.1 Perform

Usage: **Perform [-w wspace] [-m module] rule**

Le workspace (resp. le module) courant est utilisé à moins qu'un nom ne soit spécifié par l'option `-w` (resp. `-m`).

La règle rule peut être spécifiée par son nom vrai ou par son nom d'alias.

Il est nécessaire d'appliquer explicitement les règles qui ne produisent pas directement une ressource mais qui la modifie. C'est notamment le cas de `distributer` et `privatizer` pour la production de CODE.

6.2 Exemple

Voir la section Select.

7 Analyses

Les phases décrites ci-dessous sont celles pour lesquelles il est possible de choisir un algoritme en utilisant *Select*. Il s'agit du calcul des prédicats (les transformers et les préconditions) pour l'analyse sémantique, ainsi que du calcul du graphe de dépendance. Cf. rapport EMP-CAI-I E/137 pour plus de détails.

être modifiés dans Build et valoir pour les autres shell-scripts (cf. annexe *Alias*).

7.1 Transformers

Quatre algorithmes sont disponibles et leur distinction se fait sur deux critères: Ils sont plus ou moins précis et intra ou inter-procéduraux.

Les règles associées à ces algorithmes sont: TransFormers_intra_fast (alias tf), TransFormers_intra_FULL (alias tffull), TransFormers_INTER_fast (alias tfinter) TransFormers_INTER_FULL (alias tfinterfull).

7.2 Préconditions

De manière similaire, les règles sont: PreConditions_intra (alias pc), PreConditions_INTER_fast (alias pcinter), PreConditions_INTER_FULL (alias pcinterfull).

7.3 Graphe de dépendance

Deux algorithmes sont disponibles: Rice_fast_Dependence_Graph (alias rdg)
et
Rice_FULL_Dependence_Graph (alias rfulldg), qui prend en compte les prédicats.

8 Transformations du programme

8.1 Privatisation de variables

Celle-ci est toujours réalisée pour les indices de boucle sans qu'elle soit demandée. Par contre il faut appliquer la règle `PRIVatizer` (alias `priv`) pour privatiser toutes les variables qui peuvent l'être, avant de demander la parallélisation, naturellement.

8.2 Distribution de boucles

Elle est réalisée en appliquant la règle `DISTributer` (alias `dist`).

8.3 Echange de boucles: Interchange

Cette transformation fait exception quant à son utilisation. Elle n'est pas appliquée avec Perform parce qu'il faut préciser le label de la boucle à échanger avec la boucle la plus interne du nid de boucle.

Elle est accessible de cette interface par la commande **Interchange** dont la syntaxe est:

```
$ Interchange [-w wspace] [-m module] label
```

Cette commande effectue l'équivalent de Perform d'une règle qui produirait la ressource CODE et enchaîne immédiatement un Display du pretty-print code.

8.3.1 Exemple

Soit le code initial de loop4.f:

```
      program loop4

      real t(10,10)
      real v(10,10)

      do 100 i = 1, 5
         do 100 j = 1, i
            t(j+1,i)=t(j,i) + v(j,i)
 100     continue
         end
```

Pour obtenir l'inversion des deux boucles:

```
$ Init l4 loop4.f
$ Interchange -m loop4 100 >l4.inv
$ cat l4.inv
      PROGRAM LOOP4

      REAL T(10,10)
      REAL V(10,10)

      DO 100 Ip = 1,5,1                                              0004
         DO 100 Jp = Ip,5,1                                          0006
            T((1+Ip),Jp) = (T(Ip,Jp)+V(Ip,Jp))                       0007
100         CONTINUE                                                 0008
         ENDDO
      ENDDO
      RETURN
      END
```

8.4 Parallélisation de nid de boucles

C'est la transformation de programme par défaut; elle est obtenue en demandant `Display [para]`, qui construit la ressource PARALLELPRINTED_FILE. A la différence des transformations précédentes, celle-ci est élaborée à partir de la ressource PARALLELIZED_CODE distincte de la ressource CODE à partir de laquelle elle a été construite. Elle peut donc être demandée implicitement.

Ainsi, la parallélisation interprocédurale d'un module `foo` d'un programme Fortran `bar` contenu dans le fichier `source.f` se résume à:

```
$ Init bar-v1 source.f
$ Display -m foo
```

où `bar-v1` est la zone contenant la première version parallélisée du programme `bar`. Une deuxième procédure, `foofoo`, peut être ensuite traitée par une unique commande dans le même espace de travail `bar-v1`:

```
$ Display -m foofoo
```

à condition, bien sûr, que son code se soit trouvé dans votre fichier Fortran `source.f`.

Conclusion

PIPS est un paralléliseur interprocédural expérimental. La version décrite dans ce rapport est la première qui soit mise entre les mains d'utilisateurs extérieurs à l'Ecole des Mines. Il est vraisemblable que de nombreuses erreurs subsistent et nous vous sommes d'avance reconnaissant de nous les signaler (e-mail: `<pipsgroup@ensmp.fr>`). Néanmoins nous espérons que PIPS vous permettra d'obtenir des résultats intéressants en parallélisation interprocédurale.

Annexe 1: Installation de PIPS

La version initiale de PIPS est fournie pour stations et serveurs SUN4, exploités sous SUNOS 4.0.3. PIPS fonctionne avec 8 Mo de mémoire physique et 14 Mo d'espace *swap* pour de petits programmes, mais il faut disposer de plus d'espace de swap pour des programmes de plusieurs milliers de lignes, et si possible de plus de mémoire. La configuration utilisée à l'Ecole des Mines consiste en un SUN 4/260, 32 Mo de mémoire et 50 Mo d'espace de swap.

Il faut:

1. choisir un répertoire, un propriétaire et un groupe pour la sous-arborescence PIPS; nous avons choisi de définir au CRI un compte *pips* qui se trouve dans le groupe *staff* comme les chercheurs;

2. effectuer un `tar x Pips` dans le répertoire où Pips doit être installé; ce répertoire est au CRI */home/users/pips*

3. mettre à jour le propriétaire, le groupe et les droits pour cette sous-arborescence; généralement `chmod` a une option récursive; sinon, il faut utiliser find;

4. mettre à jour les variables shell PIPSDIR, LINEARDIR, NEWGENDIR, TMPDIR et éventuellement OPENWINHOME dans le fichier `Pips/pipsrc.sh` en fonction du répertoire qui a été choisi pour l'installation; il est préférable de donner des noms absolus à ces variables; au CRI, `PIPSDIR=/home/users/pips/P` Le fichier `Pips/pipsrc.csh` doit alors être mis à jour en executant la commande `Pips/make-pipsrc.csh`.

5. exécuter directement celui de ces deux fichiers qui correspond à l'analyseur de commande retenu (cf. section *Environnement de PIPS*); avec le Bourne Shell ou le Korn Shell:

   ```
   . pipsrc.sh
   ```

   avec le C Shell ou le TC Shell:

   ```
   source pipsrc.csh
   ```

6. pour continuer l'installation suivre les instructions du fichier `$UTILDIR/install-pips-src`.

7. Pour installer le `man` (dans `/usr/man/manl`:
   `cd $DOCDIR`make man+

Annexe 2: Fortran PIPS

Le compilateur PIPS n'accepte pas l'ensemble du langage Fortran tel qu'il est défini dans la norme Fortran-77. Les restrictions et extensions qui lui ont été apportées sont définies et justifiées en détail dans le rapport EMP-CAI-I E/103.

Les restrictions essentielles au langage sont les suivantes:

1. ENTRY
2. BLOCKDATA
3. ASSIGN et GOTO assigné
4. RETURN multiple
5. GOTO calculé
6. opérateur substring ":"
7. initialisation de chaînes de caractères de type Hollerith
   (par exemple `DATA A /8HOPERATIN,8HG POINT /` doit être transformé en `DATA A /'OPERATIN','G POINT '/`; l'utilitaire `ISTJS` de Toolpack effectue cette conversion);
8. le caractère *double quote* est prohibé comme dans la norme bien que de nombreux compilateurs acceptent un jeu de caractères plus grand que celui qui est spécifié dans la norme;
9. les fonctions formules ne sont pas traitées;
10. les constantes complexes doivent être remplacées par un appel à `CMPLX`;
11. les déclarations de COMMONs doivent apparaître après toutes les déclarations de type.
12. les déclarations d'un même COMMON doivent avoir la même taille.

Ces restrictions peuvent être contournées en modifiant *syntaxiquement* le programme (voir le rapport EMP-CAI-I E/103). Les BLOCKDATAs devraient être inclus dans le programme principal. Les mécanismes fondamentaux de Fortran sont bien tous traités par PIPS.

Il est aussi à noter que les restrictions 1 (M.V.13), 2 (M.III.4, M.V.17), 3 (M.III.6), 4 (M.V.12, M.V.16) et 7 (M.VI.15))font partie des constructions qu'il est conseillé d'éviter dans:

> FORTRAN 77 - Guide pour l'écriture de programmes portables
> Françoise Ficheux-Vapné
> (annexe B)

D'autres conseils donnés dans cet ouvrage permettront d'améliorer les résultats donnés par PIPS:

- utilisation de `END` sans `STOP` ni `RETURN` pour terminer les unités de programme (M.IV.3, M.V.1, M.V.8, M.V.14);
- définition uniforme des `COMMONs` dans tout un programme (M.V.21)

Il faut s'assurer que tous les modules sont explicitement nommés par une instruction `PROGRAM`, `SUBROUTINE` ou `FUNCTION`. Cela veut dire qu'il faut ajouter l'instruction:

```
PROGRAM MAIN
```

aux programmes principaux qui ne comprennent pas d'instructions `PROGRAM`. Un nom plus évocateur que `MAIN` peut bien sûr être choisi.

L'objectif n'étant pas d'écrire un compilateur commercial, les messages d'erreurs ne sont pas forcément très explicites. Il faut donc s'assurer au préalable que le programme soumis à PIPS est accepté par un compilateur conventionnel. Les options de compilation choisies devraient être aussi sévères que possible de manière à éliminer au maximum les extensions par rapport à la norme Fortran-77.

Un des problèmes rencontrés est la restriction des instructions aux colonnes 7 a 72. Avec les editeurs pleine page, il est parfois difficile de se rendre compte qu'on a depassé cette limite. Les messages fournis par PIPS sont alors imprévisibles et incompréhensibles par celui qui regarde le source.

Pour minimiser les problèmes, les déclarations devraient avoir la structure suivante:

1. déclaration des arguments;
2. ordre `IMPLICIT` optionnel;

3. déclaration des types et dimensions des variables;

4. déclaration des externes, l'ordre EXTERNAL précédant la déclaration de type;

5. déclaration des `COMMON`s.

Les boucles DO ne devraient pas porter d'étiquettes. Pour en assurer la parallélisation, il faut les réécrire comme un CONTINUE portant l'étiquette suivi d'un DO sans étiquette:

```
100 DO 200 I = 1, N
```

doit être mis sous la forme:

```
100 CONTINUE
    DO 200 I = 1, N
```

Enfin, les ordres DATA portant sur des variables statiques ne sont correctement traités que s'ils se trouvent dans le programme principal ou, tout au moins, dans le module le plus haut dans l'arbre des appels. Cela est dû à des hypothèses approximatives sur la manière dont les ordres DATA sont implémentés pour les variables dynamiques...

Annexe 3: Messages d'erreur

De nombreux messages d'erreurs ou d'avertissement peuvent être émis par PIPS (fichier non existant, erreur de syntaxe, option inconnue, etc...). Les erreurs propres au compilateur provoquent volontairement un message *core dumped* de manière à pouvoir analyser ce qui s'est passé.

Le non-fonctionnement de `pips_make`, qui se traduit par des recalculs inutiles ou par des absences de calculs utiles, est généralement dû au fonctionnement de NFS. Il faut vérifier que la machine sur laquelle tourne PIPS a exactement la même heure que la machine qui gère les fichiers. Le test peut se faire avec une commande du genre:

```
date; echo bonjour >foo; ls -l foo;rm foo
```

Les stations peuvent être resynchronisées sur le serveur NFS en utilisant la commande `rdate`.

Annexe 4: Erreurs connues

L'algorithme de calcul des *use-def chains* crée de fausses dépendances sur des indices de boucle. Elles sont dues à l'utilisation d'un indice identique dans un DO implicite ou dans une expression de bornes d'autres boucles. Ce problème est facilement identifiable parce que l'indice de la boucle non parallélisée n'est pas déclaré `PRIVATE`. Le programme résultant n'est pas optimal mais est correct.

Les commentaires portant sur les instructions `GOTO` et `RETURN` sont perdus. Ceci est dû au traitement des `RETURN` comme des `GOTO` vers la fin de la procédure courante et à la conversion des `GOTO` en arcs du graphe de contrôle. Il n'a pas été prévu de faire porter des commentaires par ces arcs.

L'impression des formats longs de plus d'une ligne se fait en un seul enregistrement. Les lignes suites qui devraient être crées ne le sont pas. Le fichier résultant n'est pas compilable.

Les fonctions formelles sont reconnues par le parser mais elles ne sont pas traitées correctement par les phases d'analyse. Le parser emet un message de type `user_error`.

Dans les déclarations, les informations de type et de dimension doivent être données avant les déclarations de commons. En cas d'inversion, les

adresses des variables dans les commons peuvent être fausses ainsi que les calculs de dépendance qui porteront sur elles.

Les STOPs apparaissant dans des sous-programmes et les exceptions en général ne sont pas correctement traités.

Le caractère *double quote* n'est pas accepté dans les chaînes de caractères (N.B. il ne fait pas partie du jeu de caractères standard de Fortran). Mais il ne peut pas servir comme délimiteur de constante chaîne de caractères.

Si un ordre ENTRY est utilisé, PIPS déclare qu'il manque le source du module de nom sorrespondant si un CALL est rencontré avant la définition de l'ENTRY. La définition de l'ENTRY provoque une erreur normale quand elle est rencontrée.

Les PARAMETERs entiers doivent être initialisés par des expressions intégralement entières. Aucun casting n'est effectué.

Annexe 5: Base de ressources

Les calculs effectués, la ressource sera disponible dans le fichier `wspace.database/MODULE.extension`. extension est en lettres majuscules s'il s'agit d'une ressource structurée pour PIPS (et correspond au nom de la ressource), et en lettres minuscules s'il s'agit d'une ressource construite par un pretty-printer (qui génère du code plus des commentaires).

Les ressources structurées pour PIPS ne sont pas lisibles, et ne peuvent servir qu'à de futurs calculs dans la base.

Parmi les pretty-prints, on aura les extensions `f` pour le Fortran du SOURCE_FILE, `pref` pour le pretty-print du Fortran séquentiel, `parf` pour le pretty-print du Fortran parallèle, `pred` pour le pretty-print des prédicats. Pourtant, il est préférable d'utiliser la commande Display qui fournit une version toujours mise à jour du pretty-print requis.

Les manipulation directes du workshpace sont à prohiber parce qu'elles risquent d'engendrer des incohérences entre la base et son schéma.

Annexe 6: Contrôleur de ressources pipsmake

La syntaxe d'une règle du fichier pipsmake.rc est la suivante:

```
rule            > OWNER.resource
                [> OWNER.resource]...
        [< OWNER.resource]...
```

Ces règles ne doivent pas être modifiées, mais il est possible d'en changer l'ordre. Cf. rapport EMP-CAI-I E/133 pour plus de détails.

La manipulation directe du fichier `pipsmake.rc` est à éviter. Il faut commencer par en faire une copie dans la directory courante. Cette copie n'est prise en compte que pour les nouveaux workspaces.

Tous les résultats souhaitables doivent pouvoir être obtenus à l'aide de la commande *Select*, sauf la modification des options par défaut. Par exemple, le test de dépendance rapide peut être remplacé par le test de dépendance précis en échangeant les règles `rice_fast_dependence_graph` et `rice_full_dependence_graph`.

Annexe 7: Alias

Les alias sont utilisables pour les règles et les ressources, dans chacun des arguments correspondants de Build, Perform et Select. Pourtant leur usage n'est jamais nécessaire (les noms de règles ou les ressources sont aussi valides).

Les alias sont implémentés dans le shell-script de Pips (présent dans `$UTILDIR/Pips`), au moyen de la commande `sed` de la fonction `rename()`. Une ligne correspond à un alias, et la syntaxe est la suivante:

`s/alias_name/true_name/;`

Lorsque l'alias_name est une sous-chaîne d'un autre alias, il faut qu'il soit positionné après lui.

Chaque utilisateur peut donc redéfinir les alias, qui sont initialisés ainsi:

```
# resources
        s/PPF/ParallelPrinted_File/; \
        s/CGF/CallGraph_File/; \
        s/PF/Printed_File/; \
# rules for
    #dg
        s/rfulldg/Rice_FULL_Dependence_Graph/; \
        s/rsdg/Rice_Semantics_Dependence_Graph/; \
        s/rdg/Rice_fast_Dependence_Graph/; \
        s/prsdg/Print_Rice_Semantics_Dependence_Graph/; \
    #transformers
        s/tffull/TransFormers_intra_FULL/; \
        s/tfinterfull/TransFormers_INTER_FULL/; \
        s/tfinter/TransFormers_INTER_fast/; \
        s/tf/TransFormers_intra_fast/; \
    #preconditions
        s/pcinterfull/PreConditions_INTER_FULL/; \
        s/pcinter/PreConditions_INTER_fast/; \
        s/pc/PreConditions_intra/; \
    #callgraph_file
        s/prcgl/PRint_Call_Graph_with_Loops/; \
        s/prcgc/PRint_Call_Graph_with_Control/; \
        s/prcg/PRint_Call_Graph/; \
    #printed_file
        s/prct/PRint_Code_Transformers/; \
```

```
        s/prcpe/PRint_Code_Proper_Effects/; \
        s/prcce/PRint_Code_Cumulated_Effects/; \
        s/prcp/PRint_Code_Preconditions/; \
        s/prcc/PRint_Code_Complexities/; \
        s/prc/PRint_Code/; \
#Transformations to perform:
        s/dist/DISTributer/; \
        s/priv/PRIVatizer/; \
```