ONERA

Parallélisation d'Entrées-Sorties avec MPI-IO

Philippe d'Anfray, Thomas Varlet

(anfray@onera.fr)

6e Forum de l'ORAP   29 Janvier 1998   EDF Clamart

PLAN DE L'EXPOSE

1° Les Entrées-Sorties Parallèles

2° `MPI-IO` => `IO` dans `MPI-2`

3° Applications

- `Fortran...`
- `C++...`

4° Conclusions, perspectives

29 Janvier 1998

Entrées-Sorties Parallèles

Les entrées-sorties constituent maintenant le veritable goulot d'étranglement dea applications parallèles:

- **les données sont de plus en plus volumineuses (maillages);**
- **sorties graphiques (animations etc...);**
- **fichiers «de reprise».**

Certaines applications (typiquement en électromagnétisme) ont encore besoin de solveurs «out of core».

Entrées-Sorties Parallèles

Les solutions possibles:

```
        ...
        IF (efficace) THEN
            non_portable(...)
        ENDIF
! et de toute(s) facon(s)
        inextricable(...)
        ...
```

Contrairement à l'échange de messages, maîtrisé, normalisé, portable.

Entrées-Sorties Parallèles

Projet commun ONERA et CEA–DAM visant à la réalisation de *benchmarks* d'entrées-sorties parallèles:

- **étude des systèmes disponibles, choix d'une bibliothèque;**
- **parallélisation des entrées/sorties dans des applications existantes.**

ONERA

Entrées-Sorties Parallèles

Comparer les fonctionnalités de **`PFS`**, **`PIOFS`**, **`Vesta`**, **`Passion`**, **`Panda`**, **`Galley`**, **`ADIO`** et **`MPI_IO`**.

Le choix s'est porté sur **`MPI_IO`** pour les raisons suivantes:

- portabilité (cf *benchmarks*), standardisation (dans **`MPI-2`**);
- pas trop proche de la machine;
- grand choix de fonctions d'accès aux données.

Concrètement: **`PMPIO`** (**`NASA-AMES`**) au dessus de **`MPICH`** (**`ANL/MSU`**) sur réseau de stations de travail (**`SUN`**).

MPI-IO $=>$ IO de MPI-2

Idée de base: les entrées-sorties peuvent se modéliser comme des échanges de messages:

- **`write` et `send`;**
- **`read` et `receive`;**
- **`Contenu des fichiers` et `Contenu des messages`.**

MPI-IO $\Longrightarrow$ IO de MPI-2

Les fichiers doivent être partagés par plusieurs processus.

Souvent, chaque processus accède à une partie du fichier.

(ce n'est pas la notion de fichier `Unix`).

- **portabilité, souplesse, efficacité;**
- **répondant aux besoins des applications scientifiques;**
- **orienté «utilisateur» (et non *hardware*);**
- **reprend les concepts de `MPI`.**

MPI-IO $=>$ IO de MPI-2

Concepts de MPI, lesquels...

- «communicateurs»;
- opérations «indépendantes» ou «collectives»;
- opérations «bloquantes» ou «non-bloquantes»;
- types dérivés.

MPI-IO $=>$ IO de MPI-2

Le programmeur décrit le contenu des fichiers avec des types dérivés:

- **Soit `etype` un type «de base» ou dérivé `MPI`;**
- **un fichier est une suite de `etype`;**
- **pour chaque processus:**
  - **le `filetype` décrit un «patron» d'accès aux éléments `etype` dans le fichier.**

MPI-IO $=>$ IO de MPI-2

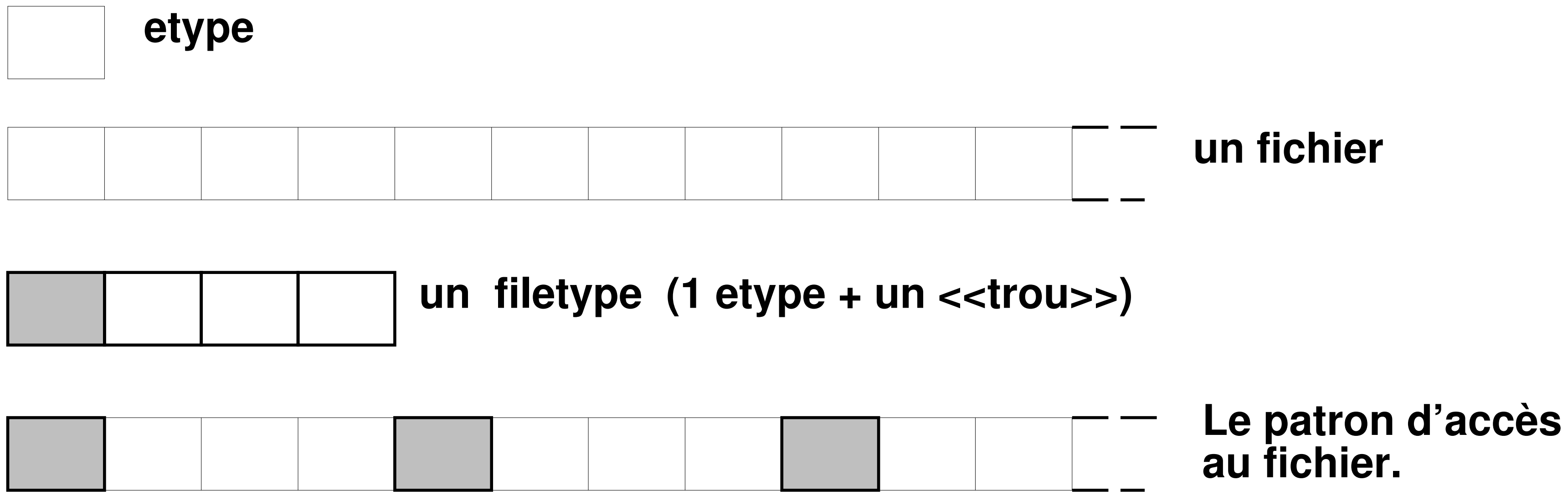

FIG. 1 – *Accès aux données*

MPI-IO => IO de MPI-2

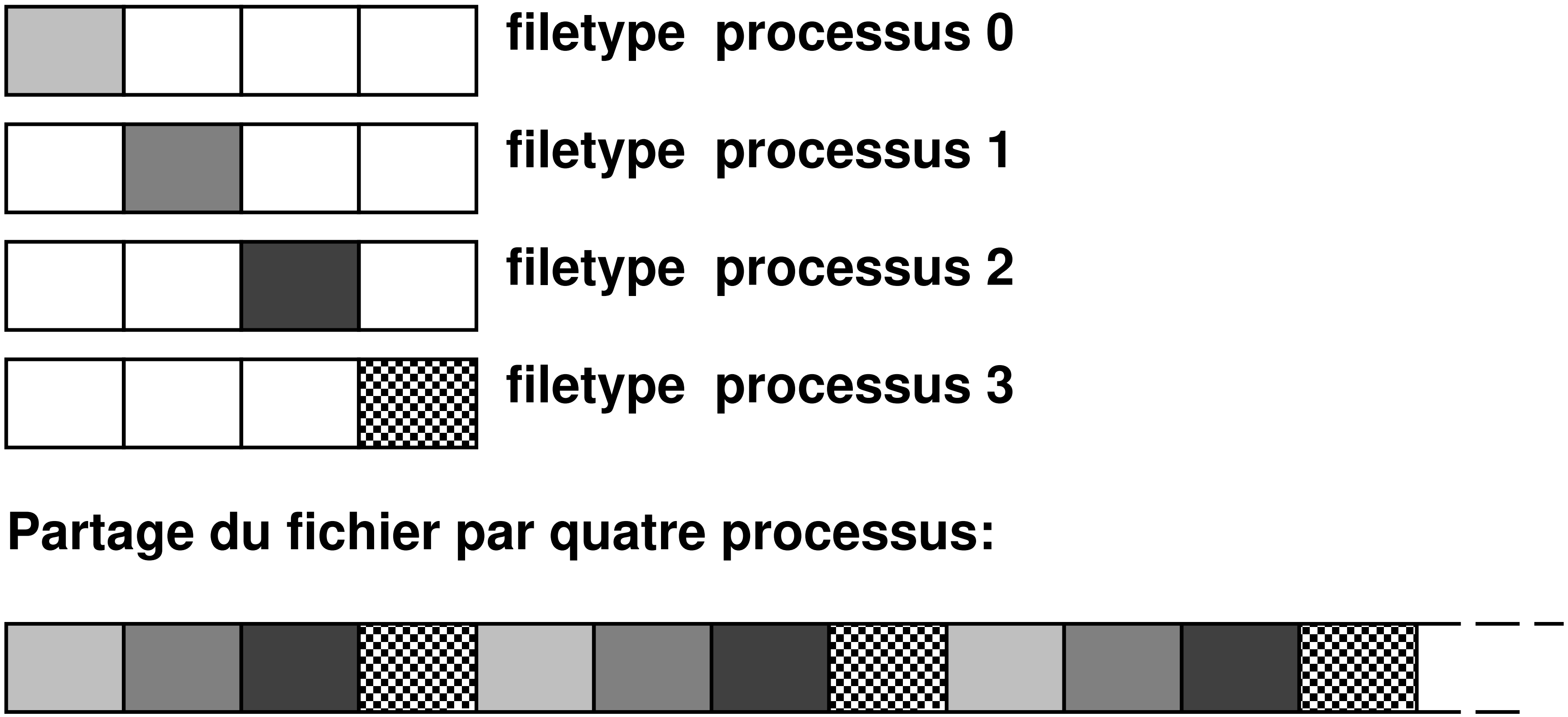

FIG. **2** – ***Partitionnement des données***

MPI-IO $=>$ IO de MPI-2

Le programmeur peut définir un «déplacement» (en octets) par rapport au début du fichier: `etype+filetype+déplacement=vue`

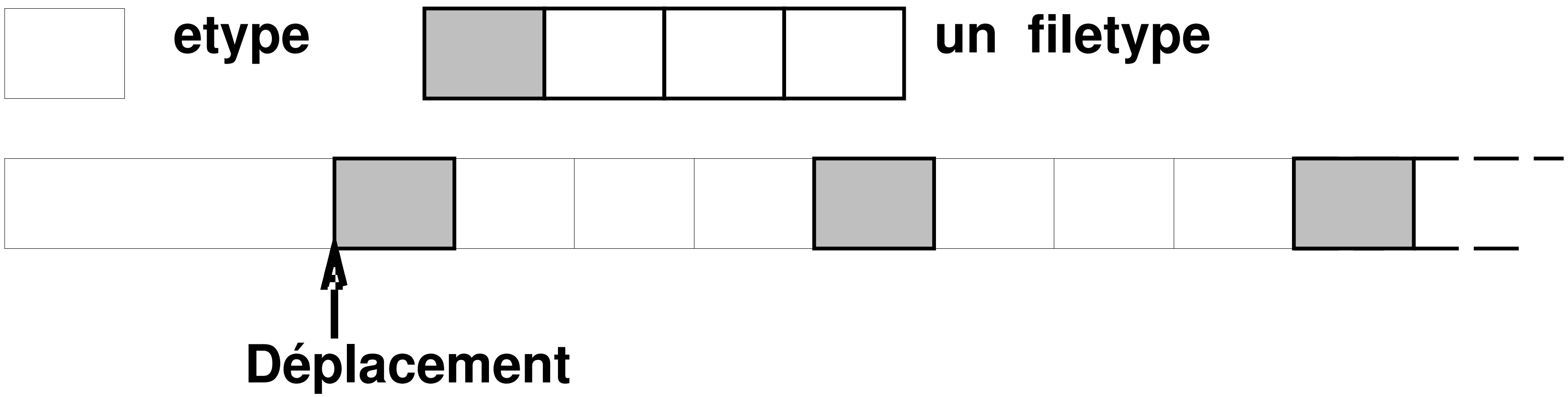

FIG. **3 –** ***Définition de la vue***

MPI-IO $=>$ IO de MPI-2

Pour un processus, les données accessibles dans le fichier sont définies par la `vue` (fonction `MPI_File_set_view(...)`).

- **vision «utilisateur» du fichier;**
- **plusieurs «vues» possibles pour un même fichier;**
- **varier `déplacement` et `filetype` permet des patrons d'accès complexes.**

ONERA

MPI-IO $=>$ IO de MPI-2

Une opération de lecture/écriture consiste à lire un certain nombre de `datatype` («multiple» de `etype`) en utilisant la `vue` définie par le `filetype` et le `déplacement`..

- **les opérations collectives, n'impliquent pas *a priori* de synchronisation entre les processus.**
- **gestion d'erreur, *à la* `MPI` `(call MPI_File...(..., STATUS))`**

MPI-IO $=>$ IO de MPI-2

Comment se positionner dans un fichier, notion de décalage (*offset*):

- **les décalages sont exprimés en nombre de `etypes` par rapport au déplacement initial;**

Le décalage s'entend par rapport à la vue du processus (les `etypes` accessibles à un moment donné).

N.B. La fin de fichier pour un processus dépend du dernier `etype` accessible.

MPI-IO $=>$ IO de MPI-2

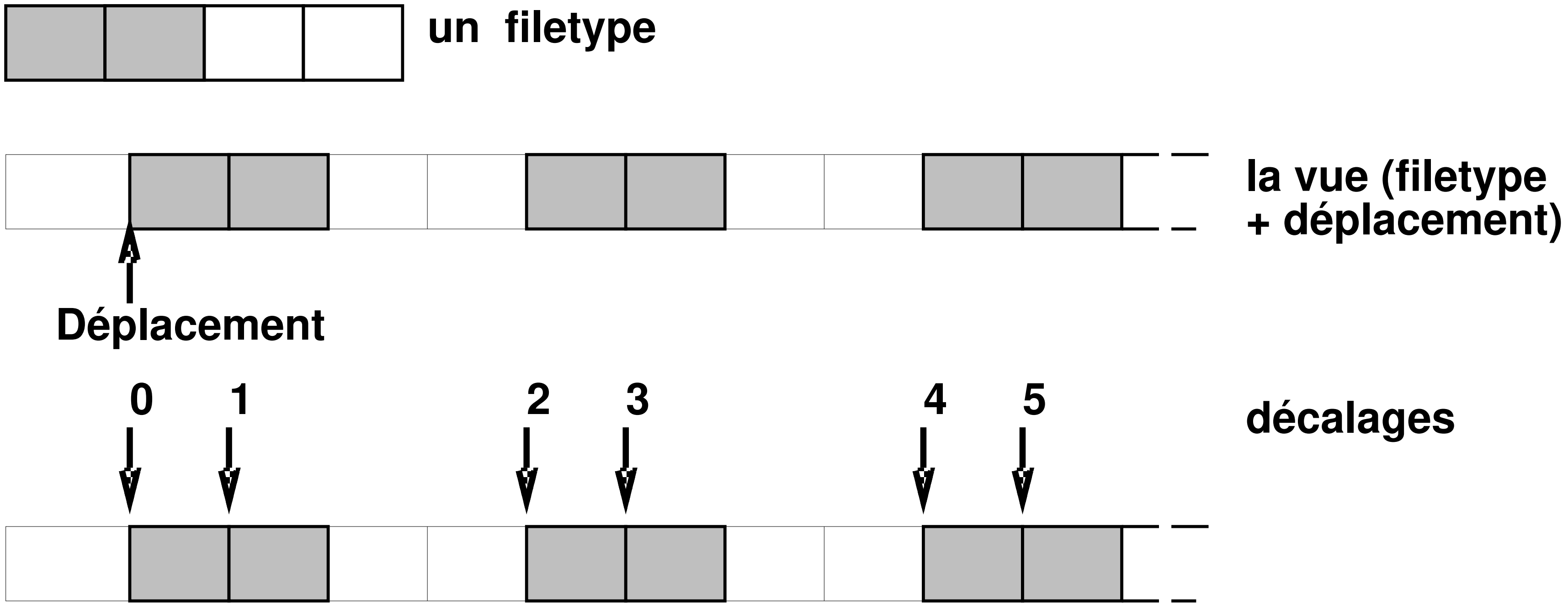

FIG. 4 – ***Notion de décalage***

MPI-IO => IO de MPI-2

En pratique, on peut utiliser:

- **des décalages explicites;**
- **des décalages implicites (« les pointeurs de fichier » gérés par le système) et dans ce cas:**
  - **soit un pointeur global (tous les processus ont le même `filetype`);**
  - **soit des pointeurs locaux (un par processus dans le groupe).**

Les pointeurs (global ou locaux) peuvent être positionnés par des appel à des fonctions spécifiques (`...seek`).

MPI-IO $=>$ IO de MPI-2

Fonctions d'accès aux données, décalage explicite:

Synchronisation	Indépendant `MPI_File_...`	Collectif `MPI_File_...`
bloquant	`Read_explicit`,	`Read_explicit_all`
non bloquant	`Iread_explicit`	`Read_explicit_begin` `Read_explicit_end`

(idem pour `write`).

ONERA

MPI-IO $=>$ IO de MPI-2

Fonctions d'accès aux données, pointeurs individuels:

Synchronisation	Indépendant `MPI_File_...`	Collectif `MPI_File_...`
bloquant	`Read`,	`Read_all`
non bloquant	`Iread`	`Read_begin` `Read_end`

(idem pour `write`).

ONERA

MPI-IO $=>$ IO de MPI-2

Fonctions d'accès aux données, pointeur partagé::

Synchronisation	Indépendant `MPI_File...`	Collectif `MPI_File...`
bloquant	`Read_shared`,	`Read_ordered`
non bloquant ou 2 étapes	`Iread_shared`	`Read_ordered_begin` `Read_ordered_end`

(idem pour `write`).

MPI-IO $=>$ IO de MPI-2

Pointeur global, tous les processus utilisent la même `vue`.

Si plusieurs processus émettent des `Read_shared`:

- **ils sont «séquentialisés»;**
- **le pointeur global «avance» à chaque `lecture`;**
- **l'ordre n'est pas déterministe.**

Les opérations collectives (`ordered`) sont séquentialisées (rangs des processus dans le groupe).

ONERA

MPI-IO $=>$ IO de MPI-2

Fonctions non bloquantes individuelles (cf échange de messages):

```
      call MPI_File_IWrite (...)
      ...
      ...
      ...
      call MPI_Wait(...)
c .......... ou MPI_Test(...) ou etc...
      ...
```

MPI-IO $=>$ IO de MPI-2

Fonctions (non bloquantes) collectives, en deux parties:

```
call MPI_File_read_begin (...)
...
...
...
call MPI_File_read_end (fh, buf, STATUS)
...
```


ONERA

MPI-IO $=>$ IO de MPI-2

Tout l'art consiste à définir habilement les `filetypes`, et il existe des fonctions «toutes faites» pour les plus fréquents:

- **parties de tableaux (`MPI_Type_create_subarray (..)`);**
- **lien avec `HPF`, distribution par blocs ou cyclique de tableaux, (`MPI_Type_create_darray (..)`);**
- ***scatter/gather* fixes et variables, (`MPI_Type_create_hvector (..)` et `..._hindexed (..)`);**

Application 1 (`Fortran`)

Base: code Navier-Stokes parallèle 3D (`MPI`) développé à l'`ONERA`.

On y trouve:

- **un «petit fichier» de données (lu par tous les processeurs);**
- **un fichier résultat (champ de vitesse 3D) (données réparties);**
- **fichier(s) «de reprise» (champ de vitesse 3D+ pression) données réparties à lire ou à écrire.**

ONERA

Application 1 (`Fortran`)

État initial:

- **«petit fichier» lecture par 0 et diffusion:**
  pas trop pénalisant car peu de valeurs à lire;
- **le fichier résultat écrit par 0, boucle «réception-écriture»: hum. . . coûteux;**
- **fichier(s) «de reprise» un fichier par nœud (+ celui des données),**
  catastrophe si on «repart» sur un nombre de nœuds différents.

Application 1 (`Fortran`)

Après utilisation de `MPI_IO`:

- **«petit fichier» `MPI_File_read_all` peut faire l'affaire: compliqué car «binaire»;**
- **le résultat, un fichier par composante de la vitesse (pb. de taille): chaque processus écrit sa partie du tableau global (`MPI_File_write_all`);**
- **fichier(s) «de reprise»: un fichier par grandeur physique (composantes, pression) chaque processeur écrit sa partie du tableau global (`MPI_File_write_all`).**

Application 1 (`Fortran`)

Stratégie:

- **le fichier est l'«image» d'un tableau partagé.;**
- **le `etype` est simplement `MPI_DOUBLE_PRECISION`;**
- **le `filetype` est construit avec `MPI_Type_create_subarray(.);`**
- **chaque processus appelle `MPI_File_write_all(...);`**

Application 1 (`Fortran`)

Pour les fichiers «de reprise» en choisissant habilement l'ordre des écritures, on pourrait recouvrir les écritures par des calculs (fonctions collectives en deux parties).

Cette approche est totalement indépendante du nombre de processeurs.

ex) fichiers de reprise écrits avec 100 processus, relus avec 99. . . seule la «vue» a changée.

Le même type de travail a été effectué sur un code «neutronique» du CEA.

Application 2 (`Fortran`)

Base: un code parallèle (`MPI`) qui résout l'équation de Boltzmann (gaz raréfiés)

On y trouve:

- **un «petit fichier» de données (lu par tous les processeurs);**
- **un fichier résultat (données réparties) (idem cas -1);**
- **un très volumineux fichier de données: interactions précalculées entre particules.**

Application 2 (`Fortran`)

Le problème vient de ces données, accédées aléatoirement au cours des itérations, et trop volumineuses pour être dupliquées.

État initial:

- **on «éclate» le fichier en autant de parties qu'il y a de nœuds. Le fichier est structuré (répartition cyclique des blocs). Tout ceci est séquentiel, long et coûteux (en temps et en place !!)**

Application 2 (Fortran)

structure du fichier <<interactions>>

exemple sur 3 processeurs, l'en–tête est dupliquée.

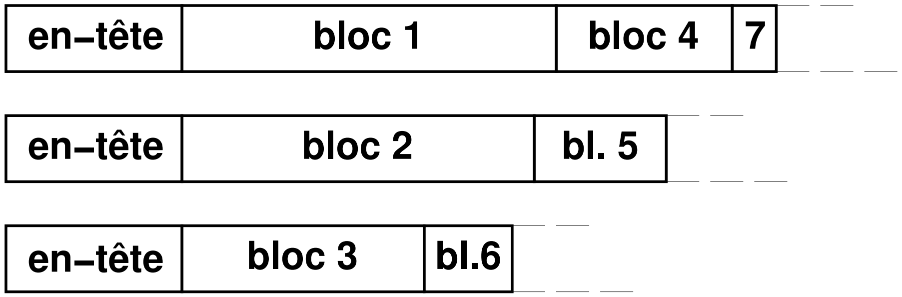

FIG. 5 – ***Éclatement des fichiers Gamma***

Application 2 (`Fortran`)

Après utilisation de `MPI_IO`:

1° **Tous les processus doivent lire l'en-tête: `filetype` adapté (`MPI_Type_create_struct`) et `MPI_File_read_all`.**

2° **Le reste est un « *scatter* » variable:**

- **redéfinition du déplacement (après l'en-tête) et nouveau `filetype` (avec `MPI_Type_create_hindexed(.)`);**
- **puis lecture collectice (`MPI_File_read_all(.)`).**

Stratégie indépendante du nombre de nœuds !

Application 3 (C++)

Action en cours. Il s'agit du code POOPOV développé à l'ONERA. C'est un code PIC (*particles in cell*) parallèle, développé en C++.

Effectue beaucoup d'entrées-sorties: notamment fichiers de reprises et fichiers permettant de visualiser (animation) le phénomène modélisé (ici trajectoires de particules).

Application 3 (C++)

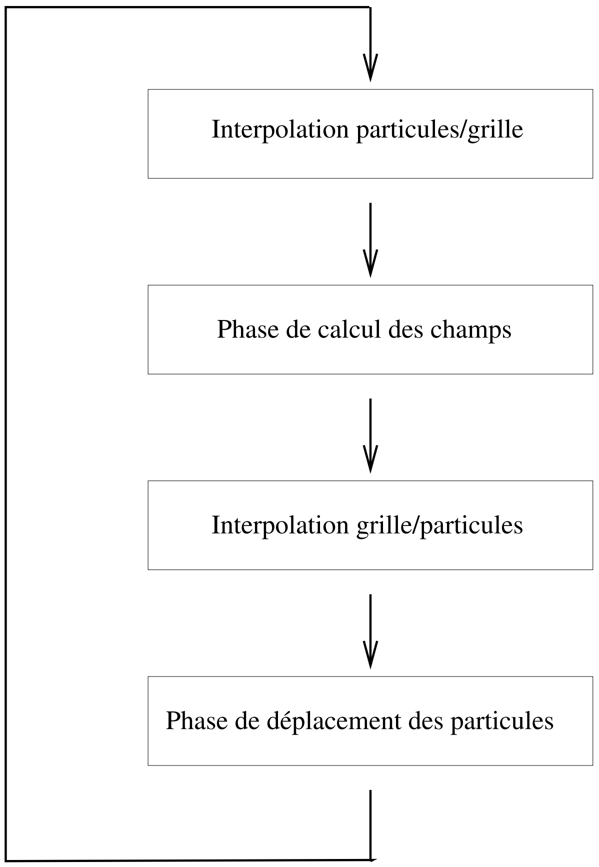

FIG. 6 – ***Structure générale d'un code « particules-grille »***

ONERA

Application 3 (C++)

La parallélisation est effectuée:

- **par échanges de messages (`MPI` «encapsulé» ou `Para++...`);**
- **en utilisant une mémoire virtuelle partagée (`LPARX`);**

Application 3 (C++)

Approche objet et calcul haute performance: modèle SPMD.

«optimisation» «composant logiciel»	vectoriel superscalaire	«micro-tâches»	Éch. de messages Mém. virtuelle
Types «de base»	OUI	OUI	NON
Classes fonctionnelles	OK	OUI	OUI
Applications	OK	OK	OUI

Les mouvements de données sont «au-dessus» des objets définis par le programmeur.

Application 3 (C++)

Dans ce contexte, il est facile

- **d'associer au «contenu» d'un objet, un type dérivé `MPI`;**
- **de redéfinir «dynamiquement» les `vues` en fonction de ce qui vient d'être traité par un processus;**
- **donc d'«encapsuler» `MPI_IO` pour l'utiliser dans les classes fonctionnelles et les applications.**

Conclusions

- facilité de programmation évidente (prolongement de **`MPI-1`**);
- portabilité «intrinsèque» (non lié à une architecture);
- performances?????
  - pas très significatives dans le contexte actuel (**`PMPIO+MPICH+`** réseau de stations) mais néanmoins meilleures par rapport aux anciennes versions !
  - **`MPI-2`** permet aussi de préciser certaines caractéristiques du système de fichiers de la machine.

Espoir: standard $=>$ mises en œuvre efficaces !

Conclusions

`MPI_IO` répond aux problèmes standards rencontrés dans les codes de simulation numérique.

- **accès simultanés à un fichier;**
- **manipulations «externes» de fichiers (*file knitting*).**

`MPI_IO` très adapté aux langages où l'on peut définir des types structurés:

- **C, `Fortran 90`, et `HPF` (avec en plus les «distributions»);**
- **les langages permettant une approche objet: C++ . . .**

`MPI−2` possède une «interface» C++.

ONERA