Introduction aux entrées-sorties parallèles

Philippe.Wautelet@idris.fr

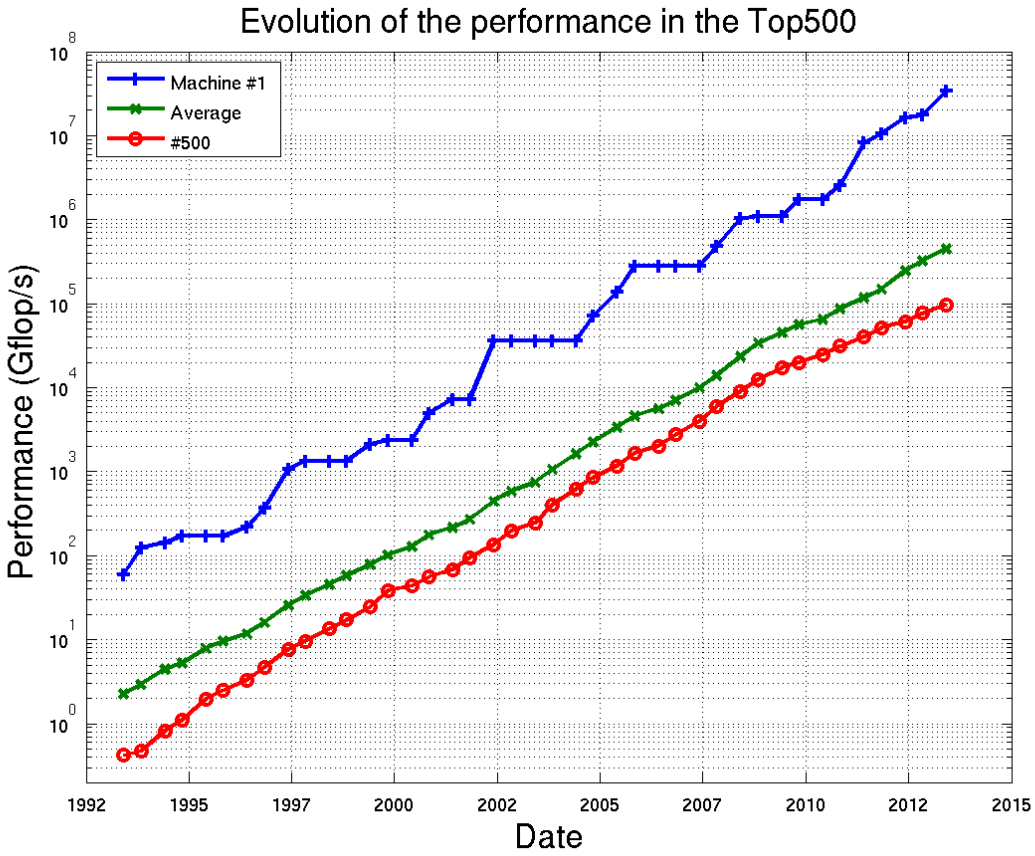
CNRS** - **IDRIS

Ecole « Optimisation »
Maison de la Simulation / 7 au 11 octobre 2013

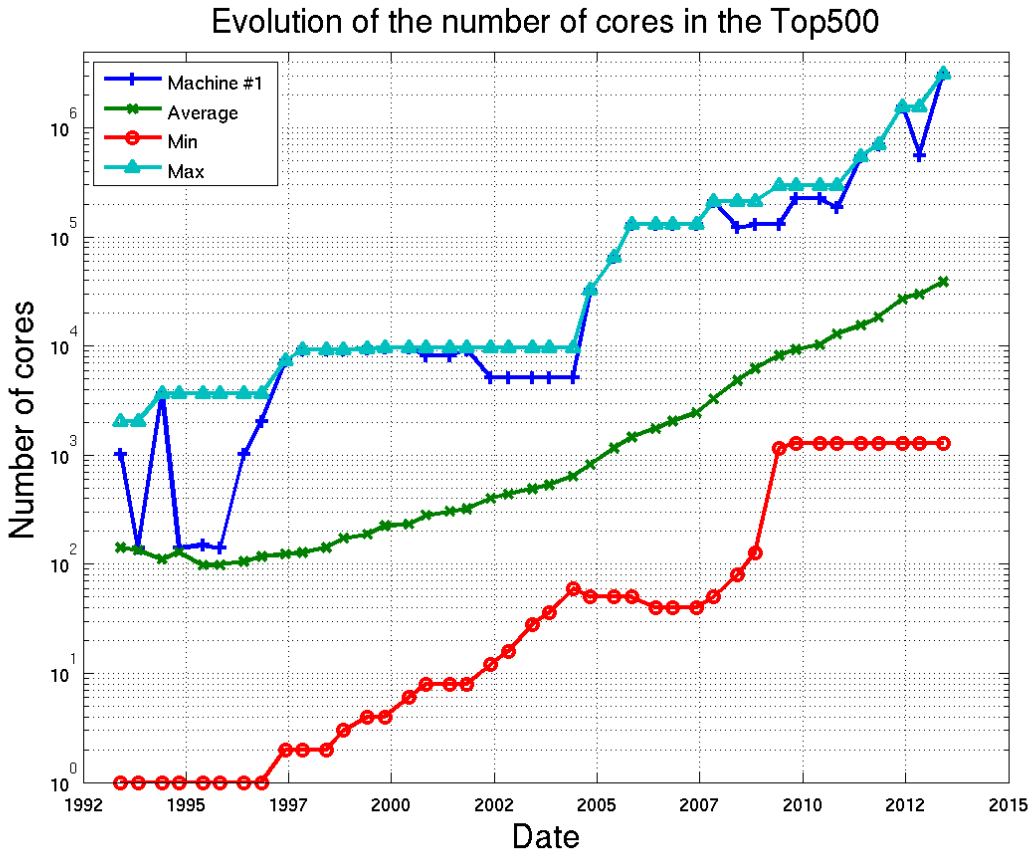
Sommaire

Problématique

Problématique



Problématique



Problématique

Evolution technique

- La puissance de calcul des supercalculateurs double tous les ans (plus rapide que la loi de Moore, aux dépens de la consommation électrique)
- Le nombre de cœurs augmente rapidement (architectures massivement parallèles et *many-cores*)
- La mémoire par cœur stagne et commence à décroître.
- La capacité de stockage augmente plus vite que la vitesse d'accès (les disques SSDs changeront peut être cela)
- Le débit vers les disques augmente moins vite que la puissance de calcul

Problématique

Conséquences

- La quantité de données générée augmente avec la puissance de calcul (moins vite heureusement)
- A l'IDRIS : la quantité de données archivées double tous les 2 ans et le nombre de fichiers est multiplié par 1,5 tous les 2 ans (en ralentissement)
- L'approche classique un fichier/processus génère de plus en plus de fichiers entraînant une saturation des serveurs de méta-données (trop d'accès simultanés et risques de crash)
- Moins de mémoire par cœur et plus de cœurs risque de réduire la taille moyenne des fichiers
- Trop de fichiers = saturation des systèmes de fichiers (nombre maximum de fichiers supporté et espace gaspillé à cause des tailles de bloc fixe)
- Le temps passé dans les I/O croît (surtout pour les applications massivement parallèles)
- Les étapes de pré- et post-traitement deviennent de plus en plus lourdes (nécessité de parallélisme...)

Problématique

Autres problèmes

- Difficile de gérer des millions de fichiers
- Les performances sur les I/O d'autres jobs, des effets de cache... peuvent avoir un impact très important sur les performances des entrées-sorties
- Les performances sont difficiles à mesurer (grande variabilité à cause de : impact autres jobs, effets caches/buffers système, mode dédié ou pas...)

Approches

Approches

Approches

- Fichiers séparés (1 ou plusieurs par processus)
- Processus maître collectant/distribuant les données
- Processus spécialisés
- MPI-I/O
- HDF5
- netCDF-4
- Parallel NetCDF
- ADIOS
- Autres bibliothèques

Comparaison des approches principales

Comparaison

Approche	*Facilité d'utilisation*	*Performance*	*1 fichier*	*Portabilité*	*Auto-documenté*
Fichiers séparés	++	++	Non	Non	Non
MPI-I/O	o	+	Oui	Non	Non
netCDF-4	-	?*	Oui	Oui	Oui
Parallel-NetCDF	-	o↗	Oui	Oui	Oui
HDF5	-	o↗	Oui	Oui	Oui

Approche	*Stabilité*	*Flexibilité*	*Pérennité*	*Outils*
Fichiers séparés	++	+	++	Non
MPI-I/O	+	+	++	Non
netCDF-4	?*	o	++	Oui
Parallel-NetCDF	o↗	-	+	Oui
HDF5	o↗	++	++	Oui

Remarque : cette comparaison dépend de l'expérience et du ressenti de l'auteur. Elle n'est donc pas forcément pleinement objective !

* : manque de recul de l'auteur sur netCDF-4

Fichiers séparés

Principe

La méthode la plus simple consiste à ce que tous les processus lisent et écrivent leurs données indépendamment les uns des autres dans des fichiers séparés.

Avantages

- Simplicité de l'implémentation
- Peu de changements par rapport au code séquentiel
- Parallélisation triviale
- Souvent la plus performante
- Pas de bibliothèque spécifique à installer

Fichiers séparés

Inconvénients

- Grand nombre de fichiers
- Dépendance vis-à-vis du nombre de processus
- Pré- et post-traitements
- Données non-portables (*endianness*, taille des types...)
- Format de fichier non normalisé (pérennité ?)

Usage conseillé

Cette approche n'est conseillée que pour des applications avec un niveau de parallélisme limité et pour des codes utilisés dans des projets de taille réduite.
A éviter dans les applications massivement parallèles

Processus maître

Principe

Un processus (généralement celui de rang 0) lit et écrit toutes les données. Les autres processus reçoivent toutes leurs données de celui-ci et lui envoient toutes les informations devant être écrites. Le processus peut participer aux phases de calcul.

Avantages

- Indépendant du nombre de processus (selon le format de fichier)
- Peu de fichiers
- Pré- et post-traitements simplifiés

Processus maître

Inconvénients

- Très mauvaise extensibilité (le processus maître risque de faire attendre tous les autres)
- Problèmes de consommation mémoire au niveau du processus maître
- Très gros fichiers si peu de fichiers
- Peu performant
- Non parallèle (I/O séquentielles)
- Données non portables (*endianness*, taille des types...)
- Format de fichier non normalisé (pérennité ?)

Usage conseillé

Cette approche n'est conseillée que pour des applications avec un niveau de parallélisme limité car le processus maître devient rapidement un goulet d'étranglement pour toute l'application.
A proscrire totalement dans les applications massivement parallèles

Processus spécialisés

Principe

Un ou plusieurs processus se chargent des entrées-sorties. Les autres processus réalisent les calculs et échangent toutes les données à lire et à écrire avec ce(s) processus spécialisé(s). Selon l'approche suivie, chaque processus spécialisé s'occupe des I/O d'un groupe prédéfini de processus de calcul ou peut s'occuper de n'importe quel processus de calcul.

Avantages

- Peu de fichiers (si implémenté pour)
- Indépendant du nombre de processus (délicat mais possible)
- Pré- et post-traitement simplifié si peu de fichiers, voire partiellement réalisé par ces processus
- Performant si nombre de processus spécialisés bien choisi
- Sur certains systèmes, chaque processus peut écrire sur des disques séparés et éventuellement transférer en arrière-plan les données sur un espace partagé plus lent

Processus spécialisés

Inconvénients

- Ressources de calcul inutilisées (si un processus par cœur)
- Implémentation non triviale (partage des groupes de processus, communications entre les processus d'I/O et les processus de calcul)
- Problèmes de consommation mémoire au niveau des processus d'I/O
- Dépendance du nombre de processus (sauf si implémenté pour ne pas l'être)
- Pré- et post-traitements si nombreux fichiers
- Très gros fichiers si peu de fichiers
- Données non portables (*endianness*, taille des types...)
- Format de fichier non normalisé (pérennité ?)

Usage conseillé

Cette approche peut être très utile dans certains cas particuliers (par exemple présence de nombreux disques locaux et d'un système de fichiers partagés lent). C'est également une piste très intéressante pour le massivement parallèle (par exemple la bibliothèque XIOS utilisée par la communauté du climat).

MPI-I/O

Principe

La bibliothèque MPI fournit un ensemble de sous-programmes pour réaliser des entrées-sorties parallèles à l'intérieur de toute application MPI. Les I/O peuvent se faire avec des opérations collectives ou indépendantes.

Avantages

- Peu de fichiers
- Indépendant du nombre de processus (si implémenté pour)
- Performance (théorique) élevée
- Optimisations tenant compte de l'architecture directement intégrées dans l'implémentation
- Pré- et post-traitements simplifiés
- Approche similaire au passage de messages
- Types dérivés MPI directement utilisables
- Appels non-bloquants disponibles

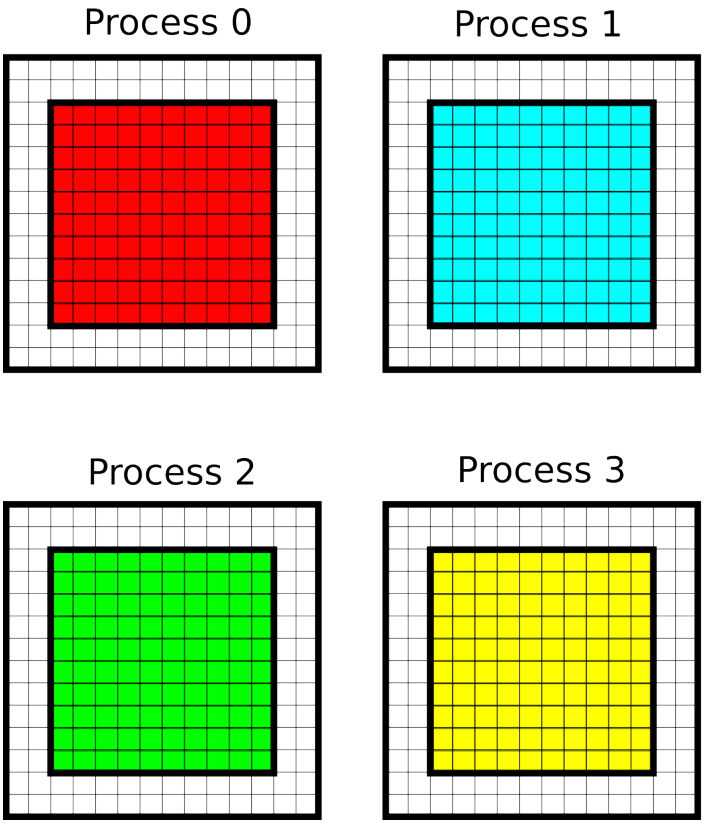
Inconvénients

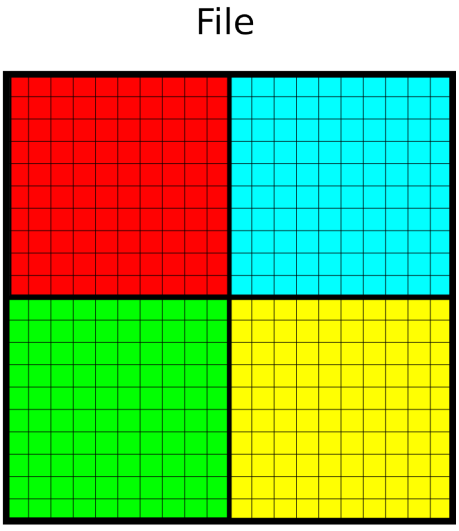
- Relecture des données dans une application non MPI (problème généralement surestimé)
- Données non portables (*endianness*, taille des types...). En théorie, portable en mode *external32*, mais rarement implémenté.
- Format de fichier non normalisé (pérennité ?)
- Très gros fichiers

Usage conseillé

MPI-I/O est conseillé pour toutes les applications parallèles. Cependant, pour des applications utilisées par une large communauté et devant être pérennes, il est conseillé de passer aux bibliothèques HDF5, netCDF-4 ou Parallel NetCDF.

Exemple : tableau distribué





MPI-I/O : exemple C

```
1   MPI_File fh;
2   MPI_Datatype type_in, type_out;
3   MPI_Offset disp;
4
5   //Open file
6   MPI_File_open(mpi_vars.comm2d,filename,MPI_MODE_WRONLY+MPI_MODE_CREATE,
        MPI_INFO_NULL,&fh);
7
8   //Write run parameters
9   if (mpi_vars.rank==0) {
10    MPI_File_write(fh,&(H.ntx)   ,1,MPI_INTEGER,MPI_STATUS_IGNORE);
11    MPI_File_write(fh,&(H.nty)   ,1,MPI_INTEGER,MPI_STATUS_IGNORE);
12    MPI_File_get_position(fh,&disp);
13  }
14  //Broadcast offset
15  MPI_Bcast(&disp,sizeof(disp),MPI_BYTE,0,mpi_vars.comm2d);
16
17  //Create subarray to represent data to write inside each process without the
        ghost cells
18  uold_shape[0]=H.nvar ; uold_shape[1]=H.nyt ; uold_shape[2]=H.nxt ;
19  uold_noghost_shape[0]=H.nvar; uold_noghost_shape[1]=H.ny; uold_noghost_shape[2]=
        H.nx;
20  pos_noghost[0]=0 ; pos_noghost[1]=2 ; pos_noghost[2]=2 ;
21  MPI_Type_create_subarray(ndim,uold_shape,uold_noghost_shape,pos_noghost,
22                           MPI_ORDER_C,MPI_DOUBLE,&type_in);
23  MPI_Type_commit(&type_in);
```


MPI-I/O : exemple C

```
//Create subarray to represent data to write inside the global array
glob_uold_shape[0]=H.nvar; glob_uold_shape[1]=H.nty; glob_uold_shape[2]=H.ntx;
glob_pos[0]=0 ; glob_pos[1]=H.starty ; glob_pos[2]=H.startx;
MPI_Type_create_subarray(ndim,glob_uold_shape,uold_noghost_shape,glob_pos,
                         MPI_ORDER_C,MPI_DOUBLE,&type_out);
MPI_Type_commit(&type_out);

//Set file view (each process sees only its own part of uold)
MPI_File_set_view(fh,disp,MPI_DOUBLE,type_out,"native",MPI_INFO_NULL);

//Write data
MPI_File_write_all(fh,Hv->uold,1,type_in,MPI_STATUS_IGNORE);

//Free datatypes
MPI_Type_free(&type_out); MPI_Type_free(&type_in);

//Close file
MPI_File_close(&fh);
```

MPI-I/O : exemple Fortran95

```
1   integer :: fh, sz
2   integer,parameter :: ndim=3
3   integer,dimension(ndim) :: glob_pos, glob_uold_shape, pos_noghost, uold_shape,
        uold_noghost_shape
4   integer :: type_in, type_out
5   integer(kind=MPI_OFFSET_KIND)::disp
6
7   !Open file
8   call MPI_File_open(comm2d,filename,MPI_MODE_WRONLY+MPI_MODE_CREATE, &
9          MPI_INFO_NULL,fh,code)
10
11  !Write run parameters
12  if (rang==0) then
13    call MPI_File_write(fh,ntx   ,1,MPI_INTEGER,MPI_STATUS_IGNORE,code)
14    call MPI_File_write(fh,nty   ,1,MPI_INTEGER,MPI_STATUS_IGNORE,code)
15    call MPI_File_get_position(fh,disp,code)
16  end if
17
18  !Broadcast offset
19  call MPI_Sizeof(disp,sz,code)
20  call MPI_Bcast(disp,sz,MPI_BYTE,0,comm2d,code)
```


MPI-I/O : exemple Fortran95

```
1   !Create subarray to represent data to write inside each process without the
        ghost cells
2   uold_shape=shape(uold)
3   uold_noghost_shape=(/nx,ny,nvar/)
4   pos_noghost=(/2,2,0/)
5   call MPI_Type_create_subarray(ndim,uold_shape,uold_noghost_shape,pos_noghost,&
6          MPI_ORDER_FORTRAN,MPI_DOUBLE_PRECISION,type_in,code)
7   call MPI_Type_commit(type_in,code)
8
9   !Create subarray to represent data to write inside the global array
10  glob_uold_shape=(/ntx,nty,nvar/)
11  glob_pos=(/startx,starty,0/)
12  call MPI_Type_create_subarray(ndim,glob_uold_shape,uold_noghost_shape,glob_pos,&
13         MPI_ORDER_FORTRAN,MPI_DOUBLE_PRECISION,type_out,code)
14  call MPI_Type_commit(type_out,code)
15
16  !Set file view (each process sees only its own part of uold)
17  call MPI_File_set_view(fh,disp,MPI_DOUBLE_PRECISION,type_out,"native",&
18         MPI_INFO_NULL,code)
19
20  !Write data
21  call MPI_File_write_all(fh,uold,1,type_in,MPI_STATUS_IGNORE,code)
22
23  !Close file
24  call MPI_File_close(fh,code)
```

HDF5

Principe

HDF5 est une bibliothèque d'entrées-sorties parallèles utilisant MPI-I/O pour lire et écrire des fichiers. Elle travaille avec des fichiers au format HDF5.

Avantages

- Peu de fichiers
- Indépendant du nombre de processus (si implémenté pour)
- Utilise MPI-I/O $\Rightarrow$ optimisations MPI-I/O
- Pré- et post-traitements simplifiés
- Données portables (grâce au format de fichier HDF5)
- Format de fichier normalisé et auto-documenté
- Format de fichier très riche
- Nombreuses fonctionnalités

HDF5

Inconvénients

- Complexe à utiliser
- Difficile d'obtenir de bonnes performances (si fichiers complexes)
- Surcoûts par rapport à MPI-I/O (taille et performance)
- Très gros fichiers

Usage conseillé

HDF5 est conseillé pour tous les projets ayant besoin d'un format de fichier portable et pérenne. Ce format étant très riche, il est parfaitement adapté aux grosses applications complexes. Son utilisation est néanmoins réservée à des utilisateurs/communautés prêts à s'investir dans son utilisation.

HDF5 : exemple C

```
hid_t fh, mode_id, prp_id;
hid_t da_id, ds_id, fs_id;
hsize_t counts[3], dims[3], starts[3];

//Create property list for parallel access
prp_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(prp_id,mpi_vars.comm2d,MPI_INFO_NULL);

//Open file
fh = H5Fcreate(filename,H5F_ACC_TRUNC,H5P_DEFAULT,prp_id);
H5Pclose(prp_id);

//Create property list for access mode (used when switching between
//independent and collective MPI-I/O modes)
mode_id = H5Pcreate(H5P_DATASET_XFER);

//Put mode_id in collective mode
H5Pset_dxpl_mpio(mode_id,H5FD_MPIO_COLLECTIVE);

//Write run parameters
H5LTset_attribute_int(fh,"/","ntx",&(H.ntx),1);
H5LTset_attribute_int(fh,"/","nty",&(H.nty),1);
```


HDF5 : exemple C

```
//Description of data in memory
counts[0] = H.nvar ; counts[1] = H.ny   ; counts[2] = H.nx ;
dims[0]   = H.nvar ; dims[1]   = H.ny+4 ; dims[2]   = H.nx+4 ;
starts[0] = 0      ; starts[1] = 2      ; starts[2]  = 2;
ds_id = H5Screate_simple(3,dims,NULL);
H5Sselect_hyperslab(ds_id,H5S_SELECT_SET,starts,NULL,counts,NULL);

//Description of data in file
counts[0] = H.nvar ; counts[1] = H.ny      ; counts[2] = H.nx ;
dims[0]   = H.nvar ; dims[1]   = H.nty     ; dims[2]   = H.ntx ;
starts[0] = 0      ; starts[1] = H.starty  ; starts[2]  = H.startx ;
fs_id = H5Screate_simple(3,dims,NULL);
da_id = H5Dcreate(fh,"u",H5T_NATIVE_DOUBLE,fs_id,H5P_DEFAULT,H5P_DEFAULT,
    H5P_DEFAULT);
H5Sselect_hyperslab(fs_id,H5S_SELECT_SET,starts,NULL,counts,NULL);

//Write data
H5Dwrite(da_id,H5T_NATIVE_DOUBLE,ds_id,fs_id,mode_id,Hv->uold);

//Free resources
H5Dclose(da_id);
H5Sclose(ds_id);
H5Sclose(fs_id);
H5Pclose(mode_id);

//Close file
H5Fclose(fh);
```

HDF5 : exemple Fortran95

```
integer(size_t) :: sz
integer(hsize_t),dimension(3) :: counts,dims,starts

call H5Pcreate_f(H5P_FILE_ACCESS_F,prp_id,status)
call H5Pset_fapl_mpio_f(prp_id,comm2d,MPI_INFO_NULL,status)
call H5Fcreate_f(filename,H5F_ACC_TRUNC_F,fh,status,access_prp=prp_id)
call H5Pclose_f(prp_id,status)

!Create property list for access mode (used when switching between
!independent and collective MPI-I/O modes
call H5Pcreate_f(H5P_DATASET_XFER_F,mode_id,status)

!Put mode_id in collective mode
call H5Pset_dxpl_mpio_f(mode_id,H5FD_MPIO_COLLECTIVE_F,status)

!Write run parameters
sz=1
call H5LTset_attribute_int_f(fh,'/','ntx',(/ntx/),sz, status)
call H5LTset_attribute_int_f(fh,'/','nty',(/nty/),sz, status)

dims(1)   = ntx ;    dims(2)   = nty ;    dims(3)   = nvar
counts(1) = nx ;     counts(2) = ny ;     counts(3) = nvar
starts(1) = startx ; starts(2) = starty ; starts(3) = 0
```


HDF5 : exemple Fortran95

```
!Description of data in memory
call H5Screate_simple_f(3,counts,ds_id,status)

!Description of data in file
call H5Screate_simple_f(3,dims,fs_id,status)
call H5Dcreate_f(fh,'u',H5T_NATIVE_DOUBLE,fs_id,da_id,status)
call H5Sselect_hyperslab_f(fs_id,H5S_SELECT_SET_F,starts,counts,status)

call H5Dwrite_f(da_id,H5T_NATIVE_DOUBLE,uold(3:nx+2,3:ny+2,1:nvar),counts,&
      status,file_space_id=fs_id,mem_space_id=ds_id,xfer_prp=mode_id)

call H5Dclose_f(da_id,status)
call H5Sclose_f(ds_id,status)
call H5Sclose_f(fs_id,status)

call H5Pclose_f(mode_id,status)
call H5Fclose_f(fh,status)
```

Parallel-NetCDF

Principe

Parallel-NetCDF est une bibliothèque d'entrées-sorties parallèles utilisant MPI-I/O pour lire et écrire des fichiers. Elle travaille avec des fichiers au format NetCDF (légèrement modifié).

Avantages

- Peu de fichiers
- Indépendant du nombre de processus (si implémenté pour)
- Utilise MPI-I/O $\Rightarrow$ optimisations MPI-I/O
- Pré- et post-traitements simplifiés
- Données portables (grâce au format de fichier NetCDF)
- Format de fichier normalisé et auto-documenté
- Format de fichier assez simple

Parallel-NetCDF

Inconvénients

- Utilisation contraignante (définition des dimensions, des variables...)
- Manque de souplesse
- Difficile d'obtenir de bonnes performances (si fichiers complexes)
- Documentation incomplète
- Légers surcoûts par rapport à MPI-I/O (taille et performance)
- Très gros fichiers

Usage conseillé

Parallel-NetCDF est conseillé pour tous les projets ayant besoin d'un format de fichier portable et pérenne. La simplicité de ce format rend son utilisation plus simple que HDF5, mais bien moins riche.

Parallel-NetCDF : exemple C

```
const int ndim = 3;
MPI_Datatype type_in;
int dims[ndim], fh, mode;
int ntx_dim_id, nty_dim_id, nvar_dim_id, int ntx_id, nty_id;
int pos_noghost[ndim], uold_shape[ndim], uold_noghost_shp[ndim];
MPI_Offset offset = 0;
MPI_Offset counts[ndim], starts[ndim], strides[ndim];

//Open file
mode = NC_64BIT_DATA;
ncmpi_create(mpi_vars.comm2d,filename,mode,MPI_INFO_NULL,&fh);

//Define dimensions
ncmpi_def_dim(fh,"ntx", H.ntx, &ntx_dim_id);
ncmpi_def_dim(fh,"nty", H.nty, &nty_dim_id);
ncmpi_def_dim(fh,"nvar",H.nvar,&nvar_dim_id);

//Define variables
dims[0]=0;
ncmpi_def_var(fh,"ntx",NC_INT,0,dims,&ntx_id);
ncmpi_def_var(fh,"nty",NC_INT,0,dims,&nty_id);

dims[0]=nvar_dim_id; dims[1]=nty_dim_id; dims[2]=ntx_dim_id;
ncmpi_def_var(fh,"u",NC_DOUBLE,3,dims,&u_id);

ncmpi_enddef(fh);
```


Parallel-NetCDF : exemple C

```
//Write run parameters
ncmpi_begin_indep_data(fh);
if (mpi_vars.rank==0) {
   ncmpi_put_var1_int(fh,ntx_id,&offset,&(H.ntx));
   ncmpi_put_var1_int(fh,nty_id,&offset,&(H.nty));
}
ncmpi_end_indep_data(fh);

//Create subarray to represent data to write inside each process without the
    ghost cells
uold_shape[0]=H.nvar ; uold_shape[1]=H.nyt ; uold_shape[2]=H.nxt ;
uold_noghost_shp[0]=H.nvar;uold_noghost_shp[1]=H.ny;uold_noghost_shp[2]=H.nx;
pos_noghost[0]=0 ; pos_noghost[1]=2 ; pos_noghost[2]=2 ;
MPI_Type_create_subarray(ndim,uold_shape,uold_noghost_shp,pos_noghost,
                         MPI_ORDER_C,MPI_DOUBLE,&type_in);
MPI_Type_commit(&type_in);

//Write data
starts[0] = 0      ; starts[1] = H.starty ; starts[2] = H.startx ;
counts[0] = H.nvar ; counts[1] = H.ny     ; counts[2] = H.nx ;
strides[0] = 1     ; strides[1] = 1       ; strides[2] = 1;
ncmpi_put_vars_all(fh,u_id,starts,counts,strides,Hv->uold,1,type_in);

//Free datatype
MPI_Type_free(&type_in);

//Close file
ncmpi_close(fh);
```

Parallel-NetCDF : exemple Fortran95

```
integer :: ntx_dim_id,nty_dim_id,nvar_dim_id
integer :: ntx_id,nty_id,u_id
integer(kind=MPI_OFFSET_KIND),dimension(3) :: counts,starts

!Create file
mode = NF_64BIT_DATA
status = nfmpi_create(comm2d,filename,mode,MPI_INFO_NULL,fh)

!Define dimensions
status = nfmpi_def_dim(fh,'ntx', int(ntx, kind=mpi_offset_kind),ntx_dim_id)
status = nfmpi_def_dim(fh,'nty', int(nty, kind=mpi_offset_kind),nty_dim_id)
status = nfmpi_def_dim(fh,'nvar',int(nvar,kind=mpi_offset_kind),nvar_dim_id)

!Define variables
status = nfmpi_def_var(fh,'ntx',NF_INT,0,(/0/),ntx_id)
status = nfmpi_def_var(fh,'nty',NF_INT,0,(/0/),nty_id)

status = nfmpi_def_var(fh,'u',NF_DOUBLE,3,(/ntx_dim_id,nty_dim_id,nvar_dim_id/),
    u_id)

!Definition of dimensions and variables finished
status = nfmpi_enddef(fh)
```


Parallel-NetCDF : exemple Fortran95

```
!Write run parameters
status = nfmpi_begin_indep_data(fh)
if (rang==0) then
   status = nfmpi_put_var1_int(fh,ntx_id,one1d,ntx)
   status = nfmpi_put_var1_int(fh,nty_id,one1d,nty)
end if

status = nfmpi_end_indep_data(fh)

starts(1) = startx+1 ; starts(2) = starty+1 ; starts(3) = 1
counts(1) = nx       ; counts(2) = ny       ; counts(3) = nvar
status = nfmpi_put_vara_double_all(fh,u_id,starts,counts,uold(3:nx+2,3:ny+2,1:
    nvar))

status = nfmpi_close(fh)
```

netCDF-4

Principe

netCDF-4 est une bibliothèque d'entrées-sorties parallèles utilisant MPI-IO pour lire et écrire des fichiers. Elle travaille avec des fichiers au format HDF5, mais peut aussi traiter des fichiers au format netCDF.

Avantages

- Peu de fichiers
- Indépendant du nombre de processus (si implémenté pour)
- Utilise Parallel-HDF5 qui repose sur MPI-I/O $\Rightarrow$ optimisations MPI-I/O
- Pré- et post-traitements simplifiés
- Données portables (grâce aux formats de fichier HDF5 et netCDF)
- Formats de fichier (HDF5 et netCDF) normalisés et auto-documentés

netCDF-4

Inconvénients

- Utilisation contraignante (définition des dimensions, des variables...)
- Manque de souplesse
- Difficile d'obtenir de bonnes performances (si fichiers complexes)
- Surcoûts par rapport à MPI-I/O (taille et performance)
- Très gros fichiers

Usage conseillé

netCDF-4 est conseillé pour tous les projets ayant besoin d'un format de fichier portable et pérenne. Son utilisation est plus simple que HDF5, mais bien moins riche.

netCDF-4 : exemple C

```
const int ndim = 3;
MPI_Datatype type_in;
int i,j,k;
int dims[ndim], fh, old_mode;
int ntx_dim_id, nty_dim_id, nvar_dim_id;
int ntx_id, nty_id, u_id;
int pos_noghost[ndim], uold_shape[ndim], uold_noghost_shape[ndim];
size_t offset = 0;
size_t counts[ndim], starts[ndim];
ptrdiff_t imap[ndim], strides[ndim];

//Create file
nc_create_par(filename,NC_MPIIO|NC_NETCDF4,mpi_vars.comm2d,MPI_INFO_NULL,&fh);

//Define dimensions
nc_def_dim(fh,"ntx", H.ntx, &ntx_dim_id);
nc_def_dim(fh,"nty", H.nty, &nty_dim_id);
nc_def_dim(fh,"nvar",H.nvar,&nvar_dim_id);

//Define variables
dims[0]=0;
nc_def_var(fh,"ntx",NC_INT,0,dims,&ntx_id);
nc_def_var(fh,"nty",NC_INT,0,dims,&nty_id);

dims[0]=nvar_dim_id ; dims[1]=nty_dim_id ; dims[2]=ntx_dim_id ;
nc_def_var(fh,"u",NC_DOUBLE,3,dims,&u_id);
```


netCDF-4 : exemple C

```
//Deactivate prefilling of variables (performance impact)
nc_set_fill(fh,NC_NOFILL,&old_mode);

//Definition of dimensions and variables finished
nc_enddef(fh);

//Write run parameters
//By default all accesses are independent
if (mpi_vars.rank==0) {
   nc_put_var1_int(fh,ntx_id,&offset,&(H.ntx));
   nc_put_var1_int(fh,nty_id,&offset,&(H.nty));
}

//Describe data structure in memory
imap[0]=(H.nx+4)*(H.ny+4); imap[1]=H.nx+4; imap[2]=1;

//Set access to collective for this variable
nc_var_par_access(fh,u_id,NC_COLLECTIVE);

//Write data
starts[0]  = 0       ; starts[1]  = H.starty ; starts[2]  = H.startx ;
counts[0]  = H.nvar  ; counts[1]  = H.ny     ; counts[2]  = H.nx ;
strides[0] = 1       ; strides[1] = 1        ; strides[2] = 1;
nc_put_varm_double(fh,u_id,starts,counts,strides,imap,&Hv->uold[2*(H.nx+4)+2]);

//Close file
nc_close(fh);
```

netCDF-4 : exemple Fortran95

```
integer :: old_mode
integer :: ntx_dim_id,nty_dim_id,nvar_dim_id
integer :: ntx_id,nty_id,u_id
integer,dimension(3) :: dims,starts

!Create file
status = nf90_create(trim(filename),IOR(NF90_NETCDF4,NF90_MPIIO),fh,comm=comm2d,
     info=MPI_INFO_NULL)

!Define dimensions
status = nf90_def_dim(fh,"ntx", ntx, ntx_dim_id)
status = nf90_def_dim(fh,"nty", nty, nty_dim_id)
status = nf90_def_dim(fh,"nvar",nvar,nvar_dim_id)

!Define variables
status = nf90_def_var(fh,"ntx",NF90_INT,ntx_id)
status = nf90_def_var(fh,"nty",NF90_INT,nty_id)

dims(1)=ntx_dim_id
dims(2)=nty_dim_id
dims(3)=nvar_dim_id
status = nf90_def_var(fh,"u",NF90_DOUBLE,dims,u_id)

!Deactivate prefilling of variables (performance impact)
status = nf90_set_fill(fh,NF90_NOFILL,old_mode)

!Definition of dimensions and variables finished
status = nf90_enddef(fh);call CHKERR(status)
```


netCDF-4 : exemple Fortran95

```
!Write run parameters
!By default all accesses are independent
if (rang==0) then
   status = nf90_put_var(fh,ntx_id,  ntx)
   status = nf90_put_var(fh,nty_id,  nty)
end if

!Set access to collective for this variable
status = nf90_var_par_access(fh,u_id, NF90_COLLECTIVE)

!Write data
starts(1) = startx+1 ; starts(2) = starty+1 ; starts(3) = 1
status = nf90_put_var(fh,u_id,uold(3:nx+2,3:ny+2,1:nvar),start=starts)

!Close file
status = nf90_close(fh)
```

ADIOS

Principe

ADIOS est une bibliothèque d'entrées-sorties parallèles. Elle travaille avec son propre format de fichier auto-documenté ou avec d'autres formats tels qu'HDF5 ou netCDF.

Avantages

- Peu de fichiers
- Indépendant du nombre de processus (si implémenté pour)
- Utilise MPI-I/O $\Rightarrow$ optimisations MPI-I/O
- Pré- et post-traitements simplifiés
- Données portables (grâce aux formats de fichier utilisés)
- Formats de fichier normalisés et auto-documentés
- Séparation de la description des données et de leurs lecture/écriture
- Souplesse lors de l'exécution : choix du format de fichier dans un fichier XML par exemple
- Evolutif
- Peut être utilisé pour de la visualisation en direct

ADIOS

Inconvénients

- 2 interfaces en lecture
  - Une similaire à celle des écritures mais ne permettant de relire qu'avec le même nombre de processus
  - Une autre très différente, mais ne relisant que les fichiers au format ADIOS
- Pas de support des sous-blocs en écriture (sous-tableau dans un tableau), ni des types dérivés MPI
- Bibliothèque encore un peu jeune et imparfaite (bugs)
- Très gros fichiers

Usage conseillé

ADIOS est encore un peu jeune. Certaines de ses fonctionnalités sont potentiellement très intéressantes, y compris pour des applications massivement parallèles.

ADIOS : exemple C

```
int i, j, k;
int64_t fh;
uint64_t group_size, total_size;
double *work;

//Open file
adios_open(&fh,"output",filename,"w",mpi_vars.comm2d);

//Provide group_size to ADIOS
group_size = (H.nx*H.ny*H.nvar)*sizeof(double)+6*sizeof(int);
adios_group_size(fh,group_size,&total_size);

//Write run parameters
adios_write(fh,"ntx",  &(H.ntx));
adios_write(fh,"nty",  &(H.nty));

//Copy data into a contiguous region
work = malloc(sizeof(double)*H.nx*H.ny*H.nvar);
for (k=0;k<H.nvar;k++)
  for (j=0;j<H.ny;j++)
    for (i=0;i<H.nx;i++)
      work[(k*H.ny+j)*H.nx+i]=Hv->uold[(k*(H.ny+4)+j+2)*(H.nx+4)+i+2];
```


ADIOS : exemple C

```
//Write data structure (necessary to provide to ADIOS all necessary information
    to write u)
adios_write(fh,"nx",&(H.nx));
adios_write(fh,"ny",&(H.ny));
adios_write(fh,"startx",&(H.startx));
adios_write(fh,"starty",&(H.starty));

//Write data
adios_write(fh,"u",work);

//Free work array
free(work);

//Close file
adios_close(fh);
```

ADIOS : exemple Fortran95

```
integer :: sz_double, sz_int
integer(kind=8) :: fh, group_size, total_size

!Create file
call adios_open(fh,"output",filename,"w",comm2d,ierr)

!Provide group_size to ADIOS
call MPI_Type_size(MPI_INTEGER,sz_int,ierr)
call MPI_Type_size(MPI_DOUBLE_PRECISION,sz_double,ierr)
group_size = (nx*ny*nvar)*int(sz_double,kind=8)+6*int(sz_int,kind=8)
call adios_group_size(fh,group_size,total_size,ierr)

!Write run parameters
call adios_write(fh,"ntx",  ntx,ierr)
call adios_write(fh,"nty",  nty,ierr)

!Write data structure (necessary to provide to ADIOS all necessary information
     to write u)
call adios_write(fh,"nx",nx,ierr)
call adios_write(fh,"ny",ny,ierr)
call adios_write(fh,"startx",startx,ierr)
call adios_write(fh,"starty",starty,ierr)

!Write data
call adios_write(fh,"u",uold(3:nx+2,3:ny+2,1:nvar),ierr)

!Close file
call adios_close(fh,ierr)
```


ADIOS : exemple fichier XML

```
<?xml version="1.0"?>
<adios-config>
  <adios-group name="output" coordination-communicator="comm">
    <var name="t" type="double" />
    <var name="gamma" type="double" />
    <var name="ntx" type="integer" />
    <var name="nty" type="integer" />
    <var name="nvar" type="integer" />
    <var name="nstep" type="integer" />
    <var name="nx" type="integer" />
    <var name="ny" type="integer" />
    <var name="startx" type="integer" />
    <var name="starty" type="integer" />
    <global-bounds dimensions="nvar,nty,ntx" offsets="0,starty,startx">
      <var name="u" type="double" dimensions="nvar,ny,nx" />
    </global-bounds>
  </adios-group>

  <method group="output" method="PHDF5" />

  <buffer size-MB="16" allocate-time="now" />
</adios-config>
```

Autres bibliothèques

- VTK (plutôt axé visualisation, mais support du parallélisme)
- SIONlib (rassemble de façon transparente dans un ou quelques fichiers l'ensemble des données des différents processus)

Conseils

Conseils

Réduire les quantités de données

- N'écrire que ce qui est nécessaire
- Ecrire le moins souvent possible (regrouper les petites écritures)
- Réduire la précision (écrire en simple précision par exemple)
- Recalculer quand c'est plus rapide

Conseils généraux

- Ne pas ouvrir et fermer des fichiers trop fréquemment car cela implique de nombreuses opérations sur le système. La meilleure façon de procéder est d'ouvrir le fichier au début et de ne le fermer que lorsque son usage n'est pas nécessaire pendant un laps de temps suffisamment long.
- Limiter le nombre de fichiers ouverts simultanément car pour chaque fichier ouvert, le système doit réserver et maintenir certaines ressources.

Conseils

Conseils généraux

- Ouvrir les fichiers dans le bon mode. Si un fichier n'est destiné qu'à être lu, il faut l'ouvrir en mode read-only car le choix du bon mode permet au système d'appliquer certaines optimisations et de n'allouer que les ressources nécessaires.
- Ne faire des flushs (vidange des buffers) que si cela est nécessaire. Les flushs sont des opérations coûteuses.
- Ecrire/lire les tableaux/structures de données en une fois plutôt qu'élément par élément. Ne pas respecter cette règle aura un impact négatif très important sur les performances des I/O.
- Séparer les procédures faisant les I/O du reste du code source pour une meilleure lisibilité et maintenabilité du code.
- Séparer les métadonnées des données. Les métadonnées sont tout ce qui décrit les données. Il s'agit généralement des paramètres des calculs effectués, les tailles des tableaux... Il est souvent plus simple de séparer le contenu des fichiers en une première partie contenant les métadonnées suivie par les données proprement dites.
- Créer des fichiers indépendants du nombre de processus. Cela rendra beaucoup plus simple les post-traitements ainsi que les redémarrages (restarts) avec un nombre différent de processus.