Quelques notes sur MPI-IO

Anne Cadiou

Laboratoire de Mécanique des Fluides et d'Acoustique

Ateliers et séminaires pour l'informatique et le calcul scientifique
PMCS2I - LMFA
Vendredi 5 avril 2019

Introduction

Dans les systèmes HPC, les I/O sont généralement réalisés par des noeuds d'I/O, avec des liens vers les noeuds de calcul.

Obtenir des performances d'I/O proportionnelles aux ressources de calcul exploitées demande une bonne connaissance des piles matérielle et logicielle des architectures utilisées.

Actuellement, le facteur limitant dans la plupart des codes est lié au système d'I/O. Les stratégies optimales peuvent varier d'une architecture à l'autre.

Composantes matérielles et logicielles

Pile matérielle pour les I/O

- Disques de stockage
- Réseau d'accès aux disques de stockage
- Noeuds d'I/O
- Réseau d'interconnexion avec les noeuds de calcul

Pile logicielle des I/O

- Systèmes de fichier parallèle (serveur et clients)
- Gestionnaire du réseau de stockage
- Bibliothèques d'entrée/sorties (POSIX, MPI-IO, HDF5, NetCDF, etc.)

Différentes stratégies possibles pour les I/O

- Un processus se charge des I/O (I/O en série)
  - + Simple
  - + Génère peu de fichiers
  - − Intrinsèquement non parallèle (réservé aux données de petites taille, peu fréquemment accèdées)
- Chaque processus écrit indépendamment dans un fichier séparé
  - + Simple
  - + Fortement parallèle, très efficace
  - − Génère beaucoup de petits fichiers
  - − Complexifie la relecture avec un autre découpage
- Tous les processus partagent le même fichier (POSIX)
  - + Génère peu de fichiers
  - − Gestion difficile des accès concurrentiels (impacte la performance)
- Faire appel à la bibliothèque MPI-IO
  - + Génère peu de fichiers
  - + Méthode efficace (accès individuel ou collectif)

Utilisation de bibliothèques d'I/O parallèles (HDF5, NetCDF, ADIOS, etc.)

- + Intercaçage et portabilité étendue
- + Méthode efficace (HDF5 est basé sur MPI-IO)

Historique

- API standardiée depuis 1997
- Présente dans de nombreuses implémentations de MPI
- Interfaces pour le C, Fortran, Python, etc.

Fonctionnalités

- Intègre de multiples représentation de données
- I/O asynchrones
- Accès collectif ou individuel aux fichiers
- Positionnement par pointeurs individuels ou partagés
- Accès non contiguë aux données

Par comparaison, les fontions de lecture et écriture en POSIX sont bloquants, non-collectifs, réalisées par des pointeurs individuels.

Principe de MPI-IO

L'écriture et la lecture d'un fichier parallèle se fait de façon similaire à l'envoi et la réception de messages dans MPI : les processus utilisent un type standard ou dérivé pour décrire les éléments à écrire ou à lire.

Méthodes possibles :

- Accès par pas explicite (`offset`) en ayant positionné l'endroit à lire ou écrire par une recherche (`seek`)
- Décrire explicitement les éléments du fichier à lire ou écrire (`file view`)

Syntaxe pour la manipulation des fichiers

MPI a son propre type pour traiter les fichiers.

En Fortran :

```
Type(MPI_File) file
```

En C :

```
MPI_File file;
```

Ouverture et fermeture d'un fichier

Fortran

```
MPI_FILE_OPEN(comm, filename, access_mode, info, fh, ierror)

character(len=*), intent(in) :: filename
type(MPI_Info), intent(in) :: info
type(MPI_File), intent(out) :: fh
```

```
MPI_FILE_CLOSE(fh, ierror)
```

C

```
int MPI_File_open(MPI_Comm comm, char *filename, int file_access_mode,
    MPI_Info info, MPI_File *fh)
```

```
int MPI_File_close(MPI_File *fh)
```

Python

```
import mpi4py.MPI as MPI

MPI.File.Open(type cls, Intracomm comm, filename,
    int amode=MODE_RDONLY, Info info=INFO_NULL)
```

```
MPI.File.Close()
```

(méthode de la classe `MPI.File` plutôt qu'un objet du communicateur.)

Principaux types de mode d'accès au fichier

MPI_MODE_RDONLY	lecture seule
MPI_MODE_RDWR	en lecture et écriture
MPI_MODE_WRONLY	écriture seule
MPI_MODE_CREATE	créer le fichier seulement s'il n'exise pas
MPI_MODE_EXCL	retourne une erreur si le fichier existe déjà
MPI_MODE_DELETE_ON_CLOSE	détruire le fichier à la fermeture
MPI_MODE_UNIQUE_OPEN	pas d'ouverture concuurente
MPI_MODE_SEQUENTIAL	le fichier sera seulement en accès séquentiel
MPI_MODE_APPEND	placer les pointeurs à la fin du fichier

Fonctionnalités

positionnement	synchronisme	coordination	
		individuel	collectif
décalage explicite (offset)	bloquant	MPI_File_read_at MPI_File_write_at	MPI_File_read_at_all MPI_File_write_at_all
	non-bloquant	MPI_File_iread_at MPI_File_iwrite_at	MPI_File_iread_at_all MPI_File_iwrite_at_all
	fractionnement collectif		MPI_File_read_at_all_begin MPI_File_read_at_all_end MPI_File_write_at_all_begin MPI_File_write_at_all_end
pointeurs individuels	bloquant	MPI_File_read MPI_File_write	MPI_File_read_all MPI_File_write_all
	non-bloquant	MPI_File_iread MPI_File_iwrite	MPI_File_iread_all MPI_File_iwrite_all
	fractionnement collectif		MPI_File_read_all_begin MPI_File_read_all_end MPI_File_write_all_begin MPI_File_write_all_end
pointeurs partagés	bloquant	MPI_File_read_shared MPI_File_write_shared	MPI_File_read_ordered MPI_File_write_ordered
	non-bloquant	MPI_File_iread_shared MPI_File_iwrite_shared	
	fractionnement collectif		MPI_File_read_ordered_begin MPI_File_read_ordered_end MPI_File_write_ordered_begin MPI_File_write_ordered_end

Exemple 1D

Écriture d'un vecteur 1D composé de N réels en double précision.
Avec un seul processus, en binaire, le fichier doit occuper $8 \times N$ octets.

Si $N = 512^3$, alors le fichier écrit par 1 seul processus occupera $1,0$ G, soit 1073741824 bytes. $N = 134217728$, $1,0$ G $= 8 \times N = 1073741824$

On notera que $512^3 = 2^{27}$.

Attention, la valeur maximale d'un entier non signé est $2^{32} - 1 = 4294967295$.

Un exemple d'IO parallèle avec POSIX

```
PROGRAM main

  USE mpi
  IMPLICIT NONE

  INTEGER, PARAMETER :: N=134217728 ! 512^3 = 2^27
  INTEGER :: ierr, i, rank, nprocs
  INTEGER :: i_glob
  INTEGER :: ires
  INTEGER :: nstart, nstop, npart, ncount, nrem
  DOUBLE PRECISION, allocatable :: vec(:)
  CHARACTER :: str*80

  CALL MPI_INIT(ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)

  ncount = N/nprocs
  nrem = MOD(N,nprocs)
  IF (rank < nrem) THEN
    nstart = rank * (ncount + 1)
    nstop = nstart + ncount
  ELSE
    nstart = rank * ncount + nrem
    nstop = nstart + (ncount - 1)
  END IF
  npart = nstop - nstart + 1
```

```
  ALLOCATE(vec(npart))

  DO i=1,npart
    i_glob = nstart+(i-1)
    vec(i) = 1.0D0*(i_glob)
  END DO

  ! each process writes it own file
  ires = rank+10
  WRITE(str,'(a,i1,a)') 'res_p',rank,'.res'
  OPEN(ires,file=trim(str),access='STREAM',form='UNFORMATTED')
  WRITE(ires) (vec(i),i=1,npart)
  CLOSE(ires)

  DEALLOCATE(vec)

  CALL MPI_FINALIZE(ierr)

END PROGRAM main
```

```
$ mpirun -np 4 ./a.out
$ ls -lrth
-rw-r--r-- 1 acadiou acadiou 256M avril 4 15:17 res_p2.res
-rw-r--r-- 1 acadiou acadiou 256M avril 4 15:17 res_p0.res
-rw-r--r-- 1 acadiou acadiou 256M avril 4 15:17 res_p1.res
-rw-r--r-- 1 acadiou acadiou 256M avril 4 15:17 res_p3.res
```

Un exemple d'utilisation de MPI-IO

```
PROGRAM main

  USE mpi
  USE handle_errors
  IMPLICIT NONE

  INTEGER, PARAMETER :: N=134217728 ! 512^3 = 2^27
  INTEGER :: ierr, i, rank, nprocs
  INTEGER :: i_glob
  INTEGER :: ires
  INTEGER :: doublesize
  INTEGER(KIND=MPI_OFFSET_KIND) :: disp
  INTEGER :: nstart, nstop, npart, ncount, nrem
  DOUBLE PRECISION, allocatable :: vec(:)
  INTEGER, DIMENSION(MPI_STATUS_SIZE) :: fh
  CHARACTER :: str*80

  CALL MPI_INIT(ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
```

```
50  ncount = N/nprocs
    nrem = MOD(N,nprocs)
52  IF (rank < nrem) THEN
      nstart = rank * (ncount + 1)
54    nstop = nstart + ncount
    ELSE
56    nstart = rank * ncount + nrem
      nstop = nstart + (ncount - 1)
58  END IF
    npart = nstop - nstart + 1
60
    ALLOCATE(vec(npart))
62
    DO i=1,npart
64    i_glob = nstart+(i-1)
      vec(i) = 1.0D0*(i_glob)
66  END DO
```

```
  ! mpi-io (each process writes in the same file)
  ires = 14
  CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'res_mpiio.res', &
                     MPI_MODE_WRONLY + MPI_MODE_CREATE, &
                     MPI_INFO_NULL, ires, ierr)
  CALL MPI_TYPE_SIZE(MPI_DOUBLE_PRECISION, doublesize, ierr)
  disp = rank * npart * doublesize
  CALL MPI_FILE_SEEK(ires, disp, MPI_SEEK_SET, ierr)
  CALL MPI_FILE_WRITE(ires, vec, npart, MPI_DOUBLE_PRECISION, fh, ierr)
  IF (ierr.ne.MPI_SUCCESS) THEN
    CALL errmsg(rank,'FAILED TO WRITE THE FILE')
  END IF
  CALL MPI_FILE_CLOSE(ires, ierr)

  DEALLOCATE(vec)

  CALL MPI_FINALIZE(ierr)

END PROGRAM main
```

```
$ mpirun -np 4 ./a.out
$ ls -lrth
-rw-r--r-- 1 acadiou acadiou 1,0G avril 4 15:17 res_mpiio.res
```

Gestion des erreurs

```
MODULE handle_errors

  USE mpi
  IMPLICIT NONE
  PUBLIC :: errmsg

  CONTAINS

  SUBROUTINE errmsg(my_id,emess)

    USE mpi

    CHARACTER*(*), intent(in) :: emess
    INTEGER, intent(in) :: my_id
    INTEGER :: ierr

    ! routine to abort with error message

    IF (my_id.eq.0) THEN
      WRITE(*,'(a)') emess
    END IF
    CALL MPI_FINALIZE(ierr)
    STOP

  END SUBROUTINE ERRMSG

END MODULE handle_errors
```

Un exemple d'utilisation POSIX avec collecte par le process 0

```
PROGRAM main

  USE mpi
  IMPLICIT NONE

  INTEGER, PARAMETER :: N=134217728 ! 512^3 = 2^27
  INTEGER :: ierr, i, rank, nprocs
  INTEGER :: i_glob
  INTEGER :: ires
  INTEGER :: nstart, nstop, npart, ncount, nrem
  DOUBLE PRECISION, allocatable :: vec(:)
  DOUBLE PRECISION, allocatable :: buff(:)
  INTEGER, allocatable :: nblk(:), idisp(:)
  CHARACTER :: str*80

  CALL MPI_INIT(ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  CALL MPI_Comm_size(MPI_COMM_WORLD, nprocs, ierr)
```

```
20  ncount = N/nprocs
    nrem = MOD(N,nprocs)
22  IF (rank < nrem) THEN
      nstart = rank * (ncount + 1)
24    nstop = nstart + ncount
    ELSE
26    nstart = rank * ncount + nrem
      nstop = nstart + (ncount - 1)
28  END IF
    npart = nstop - nstart + 1
30
    ALLOCATE(vec(npart))
32  ALLOCATE(buff(N))
    ALLOCATE(nblk(nprocs))
34  ALLOCATE(idisp(nprocs))

36  DO i=1,nprocs
      nblk(i) = npart
38    idisp(i) = (i-1) * npart
    END DO
40  DO i=1,npart
      i_glob = nstart+(i-1)
42    vec(i) = 1.0D0*(i_glob)
    END DO
```

```
  ! master process collects everything and write in one single file
  CALL MPI_GATHERV(vec,npart,MPI_DOUBLE_PRECISION,buff,nblk,idisp, &
                   MPI_DOUBLE_PRECISION,0,MPI_COMM_WORLD,ierr)
  IF (rank.eq.0) THEN
    OPEN(ires,file='res_master.res',access='STREAM',form='UNFORMATTED')
    WRITE(ires) (buff(i),i=1,N)
    CLOSE(ires)
  END IF

  DEALLOCATE(vec)
  DEALLOCATE(buff)
  DEALLOCATE(nblk)
  DEALLOCATE(idisp)

  CALL MPI_FINALIZE(ierr)

END PROGRAM main
```

Généralités

- Choisir d'exploiter le système de fichier le plus performant
- Éviter de trop fréquentes entrées-sorties
- Ajuster les données suivant la précision souhaitée
- Écrire dans des tampons de grande taille, aux données contiguës
- Déclarer le mode ad-hoc pour traiter les fichiers
  (e.g. en lecture seule, si lecture seule)
- Utiliser des objets MPI

Références

- Parallel I/O with MPI, B. Steinbusch, Jülich SC, 2018
- Optimisation des entrées-sorties, IDRIS
- `Formations MPI, IDRIS`