Progra° en mémoire distribuée I — MPI : message passing interface

MPI : librairie de f°de comm° pr l'envoi/récep°de msg entre processus. SPMD.

Communicateur : regroupe les processus qui peuvent échanger des msg. MPI_COMM_WORLD : ∃ par défaut (regroupe ts les processus)

Types de communica° : pt par pt ou collectives, doivent ∈ au m̂ comm

Comm° synchrone : dmde de transmission, accepta°, envoi de données VS asynchrone : émetteur sait seult que msg envoyé

Comm° non bloquante : on vérif + tard si com° réussie. Bloquante : on attend. Synchrone : fjr bloquante.

Com° avec buffer : on fait une copie du msg, émetteur non bloqué si récepteur non dispo.

Com° collectives ↳ Barrières : pgr arrêté until ts proc arrivés à la barrière

↳ Radiation : com° 1 à tous ↳ Réduction : donnée uniq résultat de ≠ proc.

```
# include <mpi.h>          Processus triés et n°és from 0.
MPI::Init(int *argc, char **argv[]);  : 1e f° à exec, crée MPI_COMM_WORLD.   Fin : MPI::Finalize();
int MPI::Comm::Get_size() : retourne le nb de processus du communicateur Comm
int MPI::Comm::Get_rank() : retourne identificateur proc ds communicateur Comm
```

F° de comm° : send et receive. Possibilité de deadlock. Spécifier récepteur/émetteur, données(début, lea), type, id/étiq de msg (tag)

Datatype : MPI_CHAR, MPI_FLOAT...

```
MPI::Comm::Send(const void* buf, int count, MPI::Datatype& datatype, int dest, int tag)
MPI::Comm::Recv(void* buf, int count, MPI::Datatype& datatype, int source, int tag, MPI::Status& status)
```

bloquants

Obt taille msg : Status::Get_count(const Datatype& datatype)

Pr Recv : émetteur peut être MPI_ANY_SOURCE, tag peut être MPI_ANY_TAG

MPI::Status* status a comme info : MPI::Status::Get_source() : émetteur msg

int MPI::Status::Get_tag() : tag du msg reçu

int MPI::Status::Get_error() : code d'erreur

Mesurer temps : double MPI::Wtime() T1 = MPI_Wtime(); Tmp = T2 - T1;

En C++ : types MPI : MPI::[CHAR, SHORT, INT, LONG, FLOAT, DOUBLE, BOOL, COMPLEX, BYTE...]

PMD II — Comm° collective : bloquante, ts les proc du comm° doivent exec la f° collective, buf récep° doit avoir t exacte msg

Types de mvmt de données

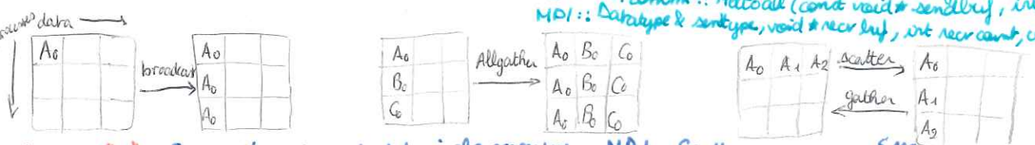
Radia°: Un procs racine send data au reste : void MPI::Comm::Bcast(void* buff, int count, const MPI::Datatype& dtt, int root)

Scatter : Distrib° data d'1 proc aux autres : void MPI::Comm::Scatter(void* buff_sendbuf, int sendcount, MPI::Dtype& sendtype, void* recvbuf, int recv count, MPI::Dtp & recvtype, int root)

Gather : Rassbl° data de ts proc do 1 void MPI::Comm::Gather(const void* sendbuf, int sendcount, const MPI::Dtype& sendtype, void* recvbuf, int recv count, const MPI::Dtype& recv type, int root)

AllGather : gather de ts à ts. void MPI::Comm::Allgather(const void* sendbuf, int sendcount, MPI::Dtype& sendtype, void* recvbuf, int recvcount, MPI::Dtype& recvType)

AllToAll : rassblmt de ts à ts les mbres d'un comm°. void MPI::comm::Alltoall(const void* sendbuf, int sendcount, const MPI::Datatype& sentype, void* recvbuf, int recvcount, const MPI::Dtype& recvtype)



Variants "v" : Envoi d'un nb != de data à chq processus. MPI_ Scatterv, Gatherv, Allgatherv, Alltoallv. #d'elts/entrayés à chq proc déf par des vecteurs de P elts, P : #de proc du communicateur.

Calcul en groupe :

Réduction : op° avec data de chq proc, rés dans root. void MPI::comm::Reduce(const void* sendbuf, void* recvbuf, int count, const MPI::Dtp& dttp, const MPI::Opk op, int root).

Op° de réduc°prédefinies : MPI_[MAX, MIN, SUM, PROD, LAND, LOR, BAND, BOR, LXOR, BXOR, MAXLOC, MINLOC]. L: Logic, B: Bitwise

Synchro° : Barrière : arrêt until ts arrivés : int MPI_Barrier(MPI_Comm comm)

PMD III — Comm° en anneau : int MPI_sendrecv(...) contient params de MPI_Send/Recv ensemble, évite deadlock

Send/Rcv non bloquant : int MPI_Isend/Irecv(..., MPI_Request * request) copie msg ds buffer et retourne tds

Attend finalisa° comm° : int MPI_Wait(MPI_Request * Request, MPI_Status * status)

int MPI_Test(..., request, int * flag, ..., status) : flag = 0 si comm non complete

Savoir si/ces comm° complétées : MPI_[Test/Wait][all/any/some] ttes / ∀ / ensemble spécifié

Tester si msg arrivent sans ds lire, connaître émetteur, l'étiq, taille : MPI_Probe & MPI_Iprobe (avt Recv ou IRcv)

Avec buffer : MPI_Bsend (m̂ args q Send). À faire : MPI_Buffer_[attach/detach] (void * buffer, int size)

Récepteur prêt immédiatemt : MPI_Rsend (ready)

Envoi synchrone : MPI_Ssend (retourne seult qd msg reçu et commence à lire)

Versions non bloquantes : MPI_[Isend, Irecv, Ibsend, Irsend, Issend]

Comm° persistante : m̂ com° qd nb de x, m̂ args (vals <>)

↳ Création du contexte : int MPI_[Send/Recv]*_init(..., MPI_Request * request)

↳ Envoi du message : int MPI_Start(MPI_Request * request)

* Bsend, Irecv, Ssend

≡ Astuces ≡

Isend : besoin de comm° non bloquantes

Ssend : la + habituelle

Ssend : best result car pas buffers

Bsend et Rsend : cas très particuliers

PROGRA //

include <omp.h>

HPC : High performance computing (algos, archi, progra //)

SPMD : Single Program Multiple Data → mémoire partagée : OpenMP (MultiProcessing) pr C, C++, Fortran

→ mémoire distribuée : MPI

Progra° en mémoire partagée I : La coordina° et coop° st faites en usant R/W de vars partagées + vars de synchro.

→ Bas niveau : barrières, locks, sec°critiq, sémaphores...

→ Haut niveau : directives de compila° (OpenMP).

Constru° de régions // parallel

```
# pragma omp parallel {...}
```

Def # threads : (void) omp_set_num_threads(8);

```
# pragma omp parallel num_threads(8){...}
int main(int argc, char *argv[]) {... int a = atoi(argv[1]);
```

(nb params)

int id = omp_get_thread_num();

Nb threads en exec° : int omp_get_num_threads()

Clauses dans une région //:

```
# pragma omp parallel if (a > 6) {...}
# pragma omp parallel private (a) shared (b, c) : vars non init
# pragma omp parallel firstprivate (a) : var init mais sont pas
# pragma omp parallel reduction (+ : b) : res final de + * max min and or
```

PMP II — Partage du travail : for, sections, single

```
# pragma omp (//) for : barrière implicite à la fin de la bcle à - que nowait
```

↳ private ↳ 1stprivate ↳ reduction ↳ lastprivate : val de la dernière it°/thread

```
# pragma omp for schedule (type [, taille bloc]) : politiq de partage des it°d'1 bcle parmi threads
```

↳ static : it° q% en bloc de taille chunk assignés en round-robin, default = num_it/num_threads

↳ dynamic : 1th fini, 1* served. default = 1. ↳ runtime : var d'env° OMP_SCHEDULE

↳ guided : taille chunk * n it/n tR et dcs exp° ↳ auto : politiq laissée au compilateur ou au runtime

Astuce : static si it° équilibrées static, n pr déséquilibre léger dynamic : ⊕ si charges <>, mais - localité data

guided : - coût q dyna mais ⚠ Qd 1e it° les + chères

```
# pragma omp for collapse (val-pos) : it° groupées pr former 1 espace d'it° réparti entre threads
# pragma omp [//] sections [clauses] { # pragma omp sec° et bloc code et #p. omp sec° et bloc ... }
```

↳ private ↳ 1st private ↳ lastprivate ↳ red° ↳ nowait

```
# pragma omp single [clauses] {...} : exec par 1 thread
```

↳ private ↳ 1st private ↳ nowait ↳ copyprivate : pas avec nowait. val (var privée) copiée aux autres threads à la fin.

PMP III — Synchronisation : master, critical, atomic, barrier, ordered

```
# pragma omp master
    bloc code
```

Seul le th master exec bloc

les autres sautent

Pas de barrières implicites

```
# pragma omp barrier
```

on attend que ts les threads soient arrivés à cette ligne

```
#pragma omp critical [(nom)]{
    bloc code }
```

Ts threads exec bloc, mais 1! th a accès au code en m̂ tmps

```
# pragma omp ordered
    bloc
```

bloc exec ds le m̂ ordre qu'en séquentiel

```
# pragma omp atomic
    sentence code
```

- couteux que critical, ne s'applique qu'à la sentence code immédiatemt suiv

Protège màj de var partagée

x++ ++x
x-- --x
x binop = expr
binop : + * - / & | ^ << >>

Gestion de tâches : task, taskwait

```
# pragma omp task [clauses] : crée une new tâche (unité de travail) qui sera exec par le th l'ayant créée, add à l'équipe de th (différée)
    bloc
```

↳ firstprivate ↳ private ↳ shared ↳ default (pri* pv | shrd | none) ↳ untied : tâche plus liée à th donné

↳ if (expr) : si True, th exec tâche, sinon différé

```
# pragma omp taskwait : on attend la terminaison de ttes les tâches
```

{ static, dynamic, guided, runtime, auto }