



# Message Passing Interface MPI

Module 8

INF8601 Systèmes informatiques parallèles

Michel Dagenais

École Polytechnique de Montréal  
Département de génie informatique et génie logiciel

# Sommaire

---

- 1 Introduction
- 2 Fonctions de base
- 3 Envoi de messages
- 4 Communications globales
- 5 Groupes de communication
- 6 Conclusion



# Message Passing Interface MPI

---

- 1 Introduction
- 2 Fonctions de base
- 3 Envoi de messages
- 4 Communications globales
- 5 Groupes de communication
- 6 Conclusion



## Historique

- Discuté lors d'un workshop à Williamsburg en avril 1992, planifié à Supercomputing en novembre 1992.
- Version préliminaire présentée à Supercomputing 1993.
- Version finale MPI 1.0 en juin 1994, plus de 80 personnes et 40 organisations ont été impliquées (vendeurs, universités, grands laboratoires...).
- MPI 1.1 en juin 1995
- MPI 2.0 (et MPI 1.2 comme sous-ensemble) en juillet 1997
- MPI 1.3 en mai 2008 (modifications finales).
- MPI 2.1 en juin 2008
- MPI 2.2 en septembre 2009.
- MPI 3.0 en septembre 2012, 3.1 en juin 2015 (mémoire partagée, assertions, sessions, MPI+thread/OpenCL, multi-langage, interface pour outils debug/prof).

# Objectifs

---

- API de programmation parallèle sur multi-ordinateur.
- Librairie pour gérer les communications.
- Bénéficie de l'expérience gagnée avec les systèmes antérieurs (PVM, LINDA).
- Ensemble cohérent de fonctions permettant une interface simple tout en s'assurant que l'implémentation sous-jacente puisse être efficace.
- Interopérabilité souhaitable avec OpenMP et OpenCL.
- Multiples fils d'exécution? Tolérance aux pannes?



## Public cible

---

- Toutes les grandes grappes de calcul.
- Laboratoires gouvernementaux (modélisation météorologique ou nucléaire) ou industriels (analyse de réseaux de transmission chez Hydro-Québec, Modélisation par éléments finis chez Bombardier ou Andritz).
- Versions optimisées et outils offerts par les grands fournisseurs de matériel (Intel, IBM...).



## Un premier programme

---

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char **argv) {
    MPI_Init( &argc, &argv );
    printf( "Hello world\n" );
    MPI_Finalize();
    return 0;
}
```

```
mpicc --mpilog --mpitrace -o hello hello.c
```

```
mpirun -np 2 hello
```



# Message Passing Interface MPI

---

- 1 Introduction
- 2 Fonctions de base
- 3 Envoi de messages
- 4 Communications globales
- 5 Groupes de communication
- 6 Conclusion





## Fonctions de base

---

- Initialiser la librairie.
- Utiliser le communicateur par défaut `MPI_COMM_WORLD`.
- Demander le nombre d'éléments `N` et son rang (0 à `N-1`).
- Finaliser la librairie.
- Avec `mpirun`, le même programme s'exécute sur `N` noeuds mais chaque instance fait un travail différent en se basant sur son rang qui la différencie.



## Contexte

---

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char **argv) {
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "Hello world! I'm %d of %d\n",
           rank, size );
    MPI_Finalize();
    return 0;
}
```



## Le contenu des messages

---

- Adresse, type des éléments et nombre, plutôt que adresse et nombre d'octets.
- Types définis récursivement pour représenter n'importe quelle structure de donnée.
- Permet de spécifier les données telles qu'elles sont.
- Minimise la possibilité d'erreur sur la longueur.
- Permet au système de faire les conversions si requis (e.g. petit ou gros boutien).



## Les types MPI

- Primitifs: MPI\_CHAR, MPI\_UNSIGNED\_CHAR (and SHORT, INT, LONG), MPI\_FLOAT, MPI\_DOUBLE, MPI\_LONG\_DOUBLE...
- Séquences: MPI\_TYPE\_CONTIGUOUS (longueur), MPI\_TYPE\_VECTOR / MPI\_TYPE\_HVECTOR (longueur, déplacement en éléments / octets), MPI\_TYPE\_INDEXED / MPI\_TYPE\_HINDEXED (vecteur de longueurs, vecteur de déplacements en éléments / octets).
- Agrégation: MPI\_TYPE\_CREATE\_STRUCT
- Calcul de la position: MPI\_TYPE\_COMMIT
- Divers: MPI\_TYPE\_FREE, MPI\_GET\_ADDRESS, MPI\_TYPE\_CREATE\_SUBARRAY, MPI\_TYPE\_CREATE\_DARRAY, MPI\_PACK, MPI\_UNPACK



## Description de types

---

```
struct Element {
    unsigned temp;
    double force[3];
    char st;
} e_t;

MPI_Datatype ElementType;

MPI_Datatype ElementFieldTypes[3] =
{ MPI_UNSIGNED, MPI_DOUBLE, MPI_CHAR };

int ElementFieldLength[3] = {1, 3, 1};

MPI_Aint ElementFieldPosition[3] =
// { 0, sizeof(double), 4 * sizeof(double) };
{ offsetof(e_t, temp), offsetof(e_t, force), offsetof(e_t, st)};

MPI_Type_create_struct(3, ElementFieldLength,
    ElementFieldPosition, ElementFieldTypes, &ElementType);
MPI_Type_commit(&ElementType);
```



# Message Passing Interface MPI

---

- 1 Introduction
- 2 Fonctions de base
- 3 Envoi de messages**
- 4 Communications globales
- 5 Groupes de communication
- 6 Conclusion



## Envoi de messages

---

- `MPI_SEND(buf, count, datatype, dest, tag, comm)` bloque sur l'envoi d'un message à `dest`.
- `MPI_RECV(buf, count, datatype, source, tag, comm, status)` bloque sur l'attente d'un message de source avec `tag` ou `MPI_ANY_SOURCE`, `MPI_ANY_TAG`.
- `MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)` aller-retour comme pour un appel de procédure.
- `MPI_SENDRECV_REPLACE(buf, count, datatype, dest, sendtag, source, recvtag, comm, status)` échange de données.



## Envois non bloquants

---

- `MPI_ISEND(buf, count, datatype, dest, tag, comm, request)` l'envoi est demandé, `buf` ne doit plus être accédé jusqu'à ce que l'envoi soit terminé.
- `MPI_IRECV(buf, count, datatype, source, tag, comm, request)` la réception est demandée, `buf` ne peut être accédé avant que la réception ne soit terminée.
- `MPI_WAIT(request, status)`, `MPI_TEST(request, flag, status)` ou `MPI_REQUEST_GET_STATUS(request, flag, status)` et `MPI_REQUEST_FREE(request)` permettent de savoir lorsque la requête est terminée.





## Attente multiple

---

- `MPI_WAITANY(count, array_of_request, index, status)`,  
`MPI_TESTANY(count, array_of_request, index, flag, status)`  
permet d'attendre après une de plusieurs requêtes.
- `MPI_WAITALL(count, array_of_request, array_of_status)`,  
`MPI_TESTALL(count, array_of_request, flag,`  
`array_of_status)` permet de vérifier toutes les requêtes.
- `MPI_WAITSOME(incount, array_of_request, outcount,`  
`array_of_index, array_of_status)`, `MPI_TESTSOME(count,`  
`array_of_request, array_of_index, array_of_status)` permet  
de vérifier une ou plusieurs requêtes.



## Autres fonctions sur les requêtes

---

- `MPI_PROBE(source, tag, comm, status)`,  
`MPI_IProbe(source, tag, comm, flag, status)`, attend ou vérifie si un message est disponible.
- `MPI_CANCEL(request)` annule une opération,  
`MPI_TEST_CANCELLED(status, flag)` permet de le vérifier.



## Communications pré-définies

---

- `MPI_SEND_INIT(buf, count, datatype, dest, tag, comm, request)`, `MPI_RECV_INIT(buf, count, datatype, source, tag, comm, request)`, spécifie les arguments pour un envoi/réception à venir, et à répéter.
- `MPI_START(request)`, `MPI_STARTALL(count, array_of_request)`, démarre la requête.



## Autres types d'envoi

---

- `MPI_BSEND`: buffered. Le contenu est copié avant d'être envoyé.
- `MPI_SSEND`: synchronous. Ne peut se compléter avant que le receveur ait commencé à recevoir le message.
- `MPI_RSEND`: ready, le receveur doit déjà être en attente du message. Ceci peut permettre de faire l'envoi en étant assuré qu'un tampon est prêt à l'autre bout.
- `MPI_IBSEND`, `MPI_ISSEND`, `MPI_IRSEND`,  
`MPI_BSEND_INIT`, `MPI_SSEND_INIT`,  
`MPI_RSEND_INIT`.



## Gestion manuelle des tampons

---

- `MPI_BUFFER_ATTACH(buffer, size)` spécifier manuellement l'espace à utiliser pour les tampons et conséquemment sa taille.
- `MPI_BUFFER_DETACH(buffer, size)`.



# Multiplication de matrice

---

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define RA 62
#define CA 15
#define CB 7
#define ROOT 0
#define ROOT_A_FEUILLE 1
#define FEUILLE_A_ROOT 2

int main (int argc, char *argv[]) {
    int size, rank, src, dest, mtype;
    int rangees, moyenne, extra, offset, i, j, k, rc;
    double a[RA][CA], b[CA][CB], c[RA][CB];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```



## Multiplication de matrice

```
if (rank == ROOT) {
    moyenne = RA / (size - 1); extra = RA % (size - 1);
    offset = 0; mtype = ROOT_A_FEUILLE;
    for (dest=1; dest<=(size - 1); dest++) {
        rangees = (dest <= extra) ? moyenne+1 : moyenne;
        MPI_Send(&offset,1,MPI_INT,dest,mtype,MPI_COMM_WORLD);
        MPI_Send(&rangees,1,MPI_INT,dest,mtype,MPI_COMM_WORLD);
        MPI_Send(&a[offset][0],rangees*CA,MPI_DOUBLE,dest,
                mtype, MPI_COMM_WORLD);
        MPI_Send(&b,CA*CB,MPI_DOUBLE,dest,mtype,MPI_COMM_WORLD);
        offset = offset + rangees;
    }
    mtype = FEUILLE_A_ROOT;
    for (i=1; i<=(size - 1); i++) {
        src = i;
        MPI_Recv(&offset,1,MPI_INT,src,mtype,MPI_COMM_WORLD,&status);
        MPI_Recv(&rangees,1,MPI_INT,src,mtype,MPI_COMM_WORLD,&status);
        MPI_Recv(&c[offset][0],rangees*CB,MPI_DOUBLE,src,mtype,
                MPI_COMM_WORLD,&status);
    }
}
```



## Multiplication de matrice

```
if (rank > ROOT) {
    mtype = ROOT_A_FEUILLE;
    MPI_Recv(&offset, 1, MPI_INT, ROOT, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rangees, 1, MPI_INT, ROOT, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&a, rangees*CA, MPI_DOUBLE, ROOT, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&b, CA*CB, MPI_DOUBLE, ROOT, mtype, MPI_COMM_WORLD, &status);
    for (k=0; k<CB; k++)
        for (i=0; i<rangees; i++) {
            c[i][k] = 0.0;
            for (j=0; j<CA; j++) c[i][k]=c[i][k]+a[i][j]*b[j][k];
        }
    mtype = FEUILLE_A_ROOT;
    MPI_Send(&offset, 1, MPI_INT, ROOT, mtype, MPI_COMM_WORLD);
    MPI_Send(&rangees, 1, MPI_INT, ROOT, mtype, MPI_COMM_WORLD);
    MPI_Send(&c, rangees*CB, MPI_DOUBLE, ROOT, mtype, MPI_COMM_WORLD);
}
MPI_Finalize();
}
```





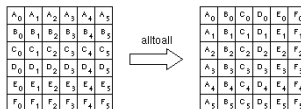
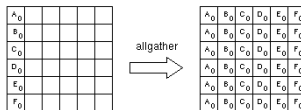
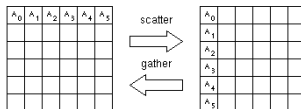
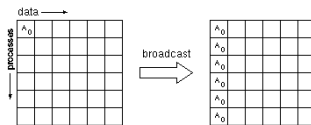
# Message Passing Interface MPI

---

- 1 Introduction
- 2 Fonctions de base
- 3 Envoi de messages
- 4 Communications globales**
- 5 Groupes de communication
- 6 Conclusion



# Communications globales



## Communications globales

---

- `MPI_BARRIER(comm)` bloque jusqu'à ce que tous les membres du groupe aient appelé cette fonction.
- `MPI_BCAST(buffer, count, datatype, root, comm)` est appelé par tous et le contenu de `buffer` sur le processus de rang `root` est copié à tous les autres.
- `MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)` est équivalent à un `send` de chaque processus (incluant `root`) et à `n` `receive` de `root` avec l'adresse de réception calculée `recvbuf + i * count`.



## Communications globales

---

- `MPI_SCATTER`(`sendbuf`, `sendcount`, `sendtype`, `recvbuf`, `recvcount`, `recvtype`, `root`, `comm`) est équivalent à `root` qui fait `n` `send` de `sendbuf + i * sendcount` et chaque processus qui fait un `receive`.
- `MPI_GATHERV`, `MPI_SCATTERV`, viennent avec un vecteur de `recvcount` ou `sendcount` et un vecteur de positions afin de recevoir ou d'envoyer des messages de taille différente à une position variable.



## De tous à tous

---

- `MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)` est comme un `gather` excepté que tous les processus, il n'y a pas de `root`, reçoivent toutes les informations. `MPI_ALLGATHERV` existe aussi.
- `MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)` chaque processus reçoit une partie de ce qui est envoyé par chaque autre. Le bloc `j` envoyé par le processus `i` est placé dans le bloc `i` du processus `j`.  
`MPI_ALLTOALLV` et `MPI_ALLTOALLW` existent aussi.



## Réductions

---

- `MPI_REDUCE`(`sendbuf`, `recvbuf`, `count`, `datatype`, `op`, `root`, `comm`) les éléments de chaque processus sont combinés ensemble dans l'élément correspondant du `root` avec l'opération qui peut être `MAX`, `MIN`, `SUM`, `PROD`, `LAND`, `BAND`, `LOR`, `BOR`, `LXOR`, `BXOR`, `MAXLOC`, `MINLOC`; `MAXLOC` and `MINLOC` notent le processus qui fournit l'élément maximal ou minimal. Il existe aussi `MPI_ALLREDUCE` qui retourne le résultat à tous.



## Autres fonctions

---

- `MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcnts, datatype, op, comm)` distribue les réductions selon le vecteur d'entier `recvcnts`.
- `MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)`, `MPI_EXSCAN`, effectuent une réduction avec préfixe inclusive (0 à  $i$ ) ou exclusive (0 à  $i - 1$ ).
- `MPI_OP_CREATE(function, commute, op)` permet de définir une opération en y associant une fonction. `MPI_OP_FREE` relâche l'opération créée.



## Calcul de PI

---

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

void srandom (unsigned seed);
double dboard (int darts);
#define DARTS 50000
#define ROUNDS 10
#define ROOT 0

int main (int argc, char *argv[]) {
    double homepi, pi, avepi, pirecv, pism;
    int rank, size, src, mtype, ret, i, n;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```





## Calcul de PI

```
random (taskid);
avepi = 0;
for (i = 0; i < ROUNDS; i++) {
    homepi = dboard(DARTS);
    if (taskid != ROOT) {
        mtype = i;
        ret = MPI_Send(&homepi,1,MPI_DOUBLE, ROOT,mtype,MPI_COMM_WORLD);
    } else {
        mtype = i; pisum = 0;
        for (n = 1; n < size; n++) {
            ret = MPI_Recv(&pirecv, 1, MPI_DOUBLE, MPI_ANY_SOURCE,mtype,
                MPI_COMM_WORLD, &status);
            pisum = pisum + pirecv;
        }
        pi = (pisum + homepi)/size;
        avepi = ((avepi * i) + pi)/(i + 1);
    }
}
MPI_Finalize();
return 0;
}
```



## Calcul de PI

---

```
srandom (taskid);
avepi = 0;
for (i = 0; i < ROUNDS; i++) {
    homepi = dboard(DARTS);
    ret = MPI_Reduce(&homepi,&pisum,1,MPI_DOUBLE,MPI_SUM,ROOT,
        MPI_COMM_WORLD);
    if (taskid == ROOT) {
        pi = pisum/numtasks;
        avepi = ((avepi * i) + pi)/(i + 1);
    }
}
MPI_Finalize();
return 0;
}
```



# Message Passing Interface MPI

---

- 1 Introduction
- 2 Fonctions de base
- 3 Envoi de messages
- 4 Communications globales
- 5 Groupes de communication
- 6 Conclusion



## Groupes de communication

---

- `MPI_GROUP_SIZE(group, size)`, taille du groupe.
- `MPI_GROUP_RANK(group, rank)`, position dans le groupe.
- `MPI_GROUP_TRANSLATE_RANKS(group1, n, ranks1, group2, ranks2)`, pour chacun des  $n$  éléments du groupe 1 spécifiés dans `ranks1`, retourne le rang correspondant de `group2` dans `ranks2`, ou `MPI_UNDEFINED` s'il ne s'y trouve pas.
- `MPI_GROUP_COMPARE(group1, group2, result)`, compare les groupes pour voir si c'est le même ou s'ils ont les mêmes membres (`IDENT`, `SIMILAR`, `UNEQUAL`).



## Sous-groupes

---

- `MPI_COMM_GROUP(comm, group)` retourne le groupe du communicateur.
- `MPI_GROUP_UNION(group1, group2, newgroup)`,  
`MPI_GROUP_INTERSECTION`,  
`MPI_GROUP_DIFFERENCE` combinent deux groupes pour en faire un nouveau.
- `MPI_GROUP_INCL(group, n, ranks, newgroup)`,  
`MPI_GROUP_EXCL`, sélectionner des éléments à inclure ou exclure. `RANGE_INCL` et `RANGE_EXCL` existent aussi.



## Autres fonctions

---

- `MPI_GROUP_FREE(group)`, libère le groupe
- `MPI_COMM_COMPARE(comm1, comm2, result)`, vérifie si les communicateurs (listes de membres) sont identiques, similaires ou différents.
- `MPI_COMM_DUP(comm, newcomm)`, copie le communicateur.
- `MPI_COMM_CREATE(comm, group, newcomm)`, crée un communicateur à partir d'un groupe.
- `MPI_COMM_SPLIT(comm, color, key, newcomm)`, crée un communicateur pour chaque groupe de processus avec la même couleur.



## Attributs de Communicateurs

---

- `MPI_COMM_CREATE_KEYVAL(copy_fn, delete_fn, key, fn_data)` réserve un code, `key`, pour un nouveau type d'attribut. les fonctions `fn_copy` et `fn_delete` sont appelées avec `fn_data` lorsque l'attribut est copié ou relâché.
- `MPI_COMM_FREE_KEYVAL(key)`
- `MPI_COMM_SET_ATTR(comm, key, value)`,  
`MPI_COMM_GET_ATTR(comm, key, val, flag)`,  
`MPI_COMM_DELETE_ATTR(comm, key)`, ajoute, accède ou enlève un attribut sur un communicateur.



## Intercommunications

---

- Communicateur entre deux groupes disjoints, local (inclut le processus courant) et remote (l'autre groupe).
- `MPI_COMM_TEST_INTER(comm, flag)`,  
`MPI_COMM_SIZE/GROUP/RANK`,  
`MPI_COMM_REMOTE_SIZE/GROUP`
- `MPI_INTERCOMM_CREATE(local_comm, local_leader, bridge_comm, remote_leader, tag, newintercomm)` doit être appelé collectivement, `bridge_comm` est un groupe englobant dans lequel `remote_leader` est identifié.
- `MPI_INTERCOMM_MERGE(intercomm, high, newintracomm)`
- `MPI_Send` sur intercommunicateur utilise le rang de l'autre groupe...





## Exemple d'intercommuniqueur

```
main(int argc, char **argv) {
    MPI_Comm c, ic1, ic2;
    int mkey, rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    mkey = rank % 3; MPI_Comm_split(MPI_COMM_WORLD, mkey, rank, &c);

    /* Nous avons trois sous-groupes g0, g1 et g2 en pipeline; g0 parle
       à g1 via 1 intercommuniqueur, g1 parle à g0 et g2 et requiert 2
       intercommuniqueurs, et g2 a un intercommuniqueur vers g1. */
    if(mkey == 0) MPI_Intercomm_create(c,0,MPI_COMM_WORLD,1,1,&ic1);
    else if (mkey == 1) {
        MPI_Intercomm_create(c,0,MPI_COMM_WORLD,0,1,&ic1);
        MPI_Intercomm_create(c,0,MPI_COMM_WORLD,2,12,&ic2);
    }
    else MPI_Intercomm_create(c,0,MPI_COMM_WORLD,1,12,&ic1);
    ...
    MPI_Finalize();
}
```



## Topologies cartésiennes

---

- `MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)` crée un nouveau communicateur sur lequel il sera facile de connaître les voisins selon une grille à `ndims` dimensions (périodique, communication entre premier et dernier, ou non).
- `MPI_DIMS_CREATE` peut aider à choisir la grille.
- `MPI_CARTDIM_GET`, `MPI_CART_GET`, permettent d'interroger la structure de la grille.



## Topologies cartésiennes

---

- `MPI_CART_RANK(comm, coords, rank)`, retourne le rank d'une coordonnée.
- `MPI_CART_COORDS(comm, rank, maxdims, coords)`, retourne les coordonnées d'un rank.
- `MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)`, calcule les voisins source et destination pour une communication d'un déplacement et direction (dimension) spécifiés.
- `MPI_CART_SUB(comm, remain_dims, newcomm)` crée un communicateur sur une topologie sur un sous-ensemble des dimensions.



## Topologies de graphes

---

- `MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)`, crée un nouveau communicateur pour lequel il est facile de savoir les voisins dans le graphe spécifié. Index dit où le trouve la dernière arête dans edges connectée au noeud correspondant.
- `MPI_GRAPHDIMS_GET`, `MPI_GRAPH_GET`, `MPI_GRAPH_NEIGHBORS_COUNT`, `MPI_GRAPH_NEIGHBORS`, permettent d'interroger la structure du graphe.
- `MPI_TOPO_TEST(comm, status)`, savoir si cartésien, graphe ou aucun.



## Récupération d'erreur

---

- `MPI_COMM_CREATE_ERRHANDLER(fn, errhandler)`,  
create a error handler from a pointer to function to call.
- `MPI_COMM_SET_ERRHANDLER(comm, errhandler)`,  
associer une fonction de rappel en cas d'erreur lors d'un appel impliquant une communication.
- `MPI_COMM_GET_ERRHANDLER`,  
`MPI_ERRHANDLER_FREE...`



# Message Passing Interface MPI

---

- 1 Introduction
- 2 Fonctions de base
- 3 Envoi de messages
- 4 Communications globales
- 5 Groupes de communication
- 6 Conclusion**



## Autres fonctions

---

- `MPI_PROCESSOR_NAME(name, resultlen)`, nom du noeud.
- `MPI_WTIME()` retourne le temps en secondes, double précision.
- `MPI_WTICK()` retourne la résolution du temps en secondes, double précision (e.g. .001s).



## Analyse de performance

---

- Pour chaque fonction MPI, par exemple MPI\_FONCTION, le vrai nom est MPI\_PFUNCTION et un symbole faible MPI\_FONCTION est ajouté (e.g. `#pragma weak MPI_SEND = PMPI_SEND`). On peut donc remplacer MPI\_FONCTION par une version qui collecte des statistiques et appelle MPI\_PFUNCTION à l'interne.
- MPI\_PCONTROL(level), choisir le niveau de profilage.





## Optimisation

---

- Utiliser tous les noeuds, tout le temps (choisir la granularité, équilibrer la charge).
- Réduire la communication (grouper les messages, recalculer, garder des copies).
- Eviter les copies de données (e.g. BSend, données non contiguës).
- RECV avant SEND, communications asynchrones en parallèle avec le calcul.
- Requêtes réutilisées, éviter la scrutation...



## Outils d'analyse de performance

---

- Tracer tous les envois de message et blocages associés.  
Obtenir un profil des temps d'exécution.
- VampirTrace est gratuit mais le visualisateur, Vampir, est commercial.
- MPE/Jumpshot (Argonne National Lab), TAU (UofOregon, Los Alamos National Lab), Paraver (Barcelona Supercomputing Center)
- Oracle Studio Performance Analyzer.



## Conclusion

---

- MPI est relativement simple (bibliothèque) et mature, les grappes de noeuds n'ont pas changé aussi vite que les multi-processeurs.
- Il n'existe plus beaucoup d'équivalent en compétition (PVM, LINDA).
- La mise à l'échelle est cruciale avec les problèmes de taille monstrueuse et des grappes de milliers de noeuds.
- Pour certaines applications, Hadoop et Spark peuvent être considérés comme des alternatives.

