Introduction to Parallel Programming with MPI

Master GENIAL (Génie Informatique en Alternance), Université de Paris

Pierre Kestener

CEA, DSSI

Université de Paris, Grands Moulins, Mai 2023

Outline

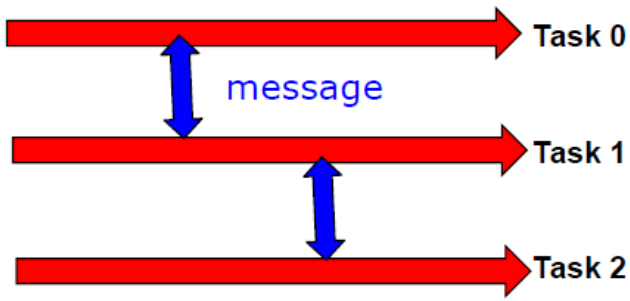
Rappel: Processus / thread

Qu'est ce qu'un thread ? un processus ?

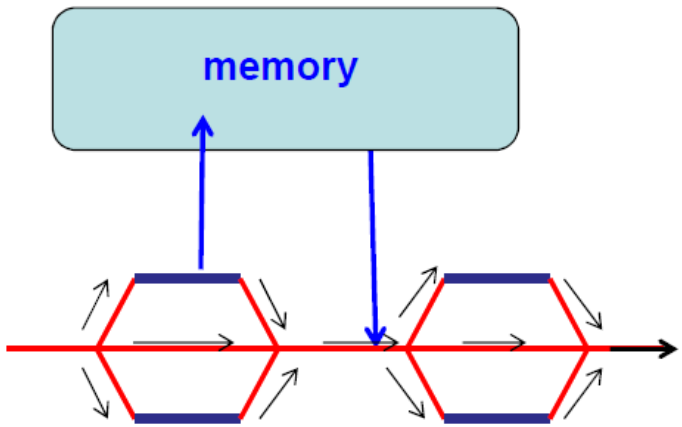
- **processus** (au sens UNIX du terme, cf *process/task* en anglais)
  - c'est un concept **logiciel** (pas matériel)
  - **une instance d'un programme, en cours d'éxecution**
  - un processus est un concept central dans le fonctionnement d'un système d'exploitation (OS)
  - un processus *possède* des ressources: copie mémoire du code à éxecuter, une pile d'appel qui enregistre le contexte de la routine en cours d'éxecution, de la mémoire, ...
  - OS multi-tâche: exécute plusieurs tâches de manière **concurrente**, entrelacée, en donnant l'*illusion* d'une exécution **parallèle**
- **thread = processus léger**
  - un processus peut-être constitué de plusieurs threads (fils d'exécution) partageant le même espace d'adresse,
  - sur un processeur mono-cœur, les threads sont exécutés de manière concurrente, par multiplexage temporel (i.e. multi-tasking)
  - sur un processeur multi-cœur, les threads peuvent être exécutés en parallèle, i.e. sur des cœurs différents.
  - **multi-threading**: une application est dite *multithreadée* lorsque le processus associé est constitué de plusieurs threads.
  - mise en œuvre: pthread, OpenMP

Types de systèmes parallèles

message passing



shared memory



Types de systèmes parallèles

- **Systèmes à mémoire partagée**
  - Les cœurs d'un CPU partagent l'accès à la mémoire
  - Programme dit *multi-threadé*, les threads partagent des variables localisées dans le même espace mémoire
  - Coordination des threads par lecture/écriture des variables partagés
- **Systèmes à mémoire distribuée**
  - Programme constitués de plusieurs tâches (au sens Unix du terme, donc espace mémoire propre pas directement accessible aux autres tâches)
  - Communication inter-tâches (coordination) via l'échange **explicit** de messages (via une interface réseau)
  - Les ressources (données en mémoire) sont **locales** (ou privées) à une tâche
  - le modèle par échange de message (*message passing*) est aussi valable sur architecture à mémoire partagée

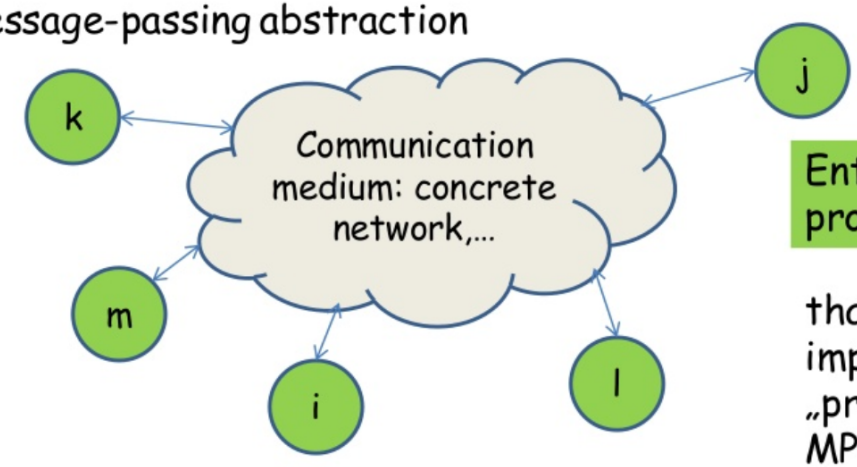
MPI - un standard / une norme

- **Historique 1992-94:** un groupe constitué de *vendeurs* et d'utilsateurs décident de créer un document décrivant les concepts / les abstractions de la norme MPI (quels types de communications, point-à-point, collectives, ..., quels modes de synchronisations) servant de base pour les **implémentations logicielles**
- **MPI forum** `http://www.mpi-forum.org/` ⇒ defines a standard set of library calls, but **does not define how to implement them**
- interface C / Fortran (et même Java); Python en 3^{rd} *-party*
- But (évident): fournir un outil **commun** pour la programmation des systèmes à mémoire distribuée; assurer la **portabilité** d'un programme d'un système à un autre

lien recommendé: le blog de Jeff Squyres sur MPI et ses slides (historique)
le manuel de OpenMPI: `https://www.open-mpi.org/doc/current/`
la norme: `http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/mpi31-report.htm`

MPI - un standard / une norme

Message-passing abstraction



Entities: MPI processes

that can be implemented as „processes" (most MPI implementations), „threads", ...

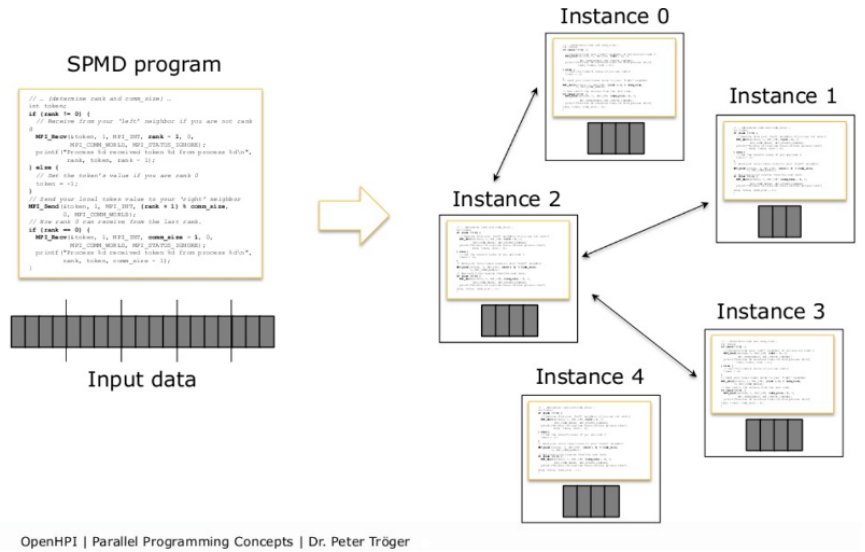
can communicate through a communication medium

nature of which is of no concern to the MPI standard:

- No explicit requirements on network structure or capabilities
- No performance model or requirements

MPI - un standard / une norme

- **MPI : Message Passing Interface**
- Plusieurs *tâches* (au sens Unix du terme) envoient et/ou recoivent des données d'autres tâches.
- **modèle SPMD (Single Program Multiple Data)**
- Toutes les tâches MPI exécutent le même programme mais utilisent des données différentes.
- **MPMD** (**Multi Program Multiple Data**) est permis par les ajouts dans version MPI-2 de la norme (en dehors du cours); utilité principale: couplage de code.



MPI - un standard / une norme

- MPI est une **librairie** (pas un language)
  - toutes les opérations MPI sont implantées sous forme de routines, interfacées dans un language comme `C` / `fortran`
- Permettre le **développement d'applications parallèles**, de bibliothèques parallèles (ex: FFTW-mpi, scalapack, PETSc, ...), *démocratiser* l'accès aux super-calculateurs
- Autre avantage: une même interface de programmation, quelque soit la technologie réseau hardware (ethernet Gigabit, infiniband, ...),
- Les différentes **implémentations** de MPI
  - OpenMPI
  - MPICH
  - MVAPICH
  - IntelMPI
  - DeinoMPI (Windows only)

MPI - un standard / une norme

Les évolutions du standard:

- MPI-2
  - New datatype constructors, language interoperability
  - new functionalities: One-side communication, MPI-IO, dynamics processes
  - Fortran90 / C++ bindings

MPI en résumé

- **une librairie** / interface de programmation, permettant à un ensemble de tâches de communiquer entre elles (point-à-point, collectivement, ...)
- **un programme d'aide à la compilation:** `mpicc` pour le `C`, `mpif90` pour le `fortran`
  - pour fortran, on remplace `gfortran` par `mpif90` pour compiler
  - `mpif90 ./helloworld_mpi.f90 -o helloword_mpi`
- **un environnement d'exécution**, la commande `mpirun` permet le lancer l'exécution de plusieurs tâches (plusieurs instances) du même programme MPI (SPMD)
  - `mpirun -np 2 ./helloworld_mpi`
  - on examinera plus tard comment on peut controller les ressources de calcul: sur quelle machine on s'exécute, combien de processus par machine, etc ...

Exécuter du code sur une machine distante

- ssh permet d'ouvrir un shell [1] sur une machine distante: e.g.
  `ssh nomUtilisateur@machineDistante`
- ssh permet aussi d'exécuter du code sur une machine distante, e.g. :
  `ssh nomUtilisateur@machineDistante cat /proc/cpuinfo`
- L'environnement MPI, par l'intermédiaire de la commande
  `mpirun -np 8 ./mon_executable`
  lance 8 fois le programme `mon_executable`.
  - Par defaut, sur la machine courante
  - L'option `-machinefile` permet de passer un fichier texte contenant une liste de machines distantes, sur lesquelles `mpirun` pourra lancer une instance de `mon_executable`.
  - Pour faciliter cette mise en œuvre, il faudra que toutes les machines, participant au calcul lancé par `mpirun`, aient accès **au même système de fichiers**, et/ou que l'utilisateur ait pris soin d'installer des clés ssh pour ne pas avoir à entrer son mot de passe à chaque connexion.
  - **Sur un *véritable* calculateur / cluster, les administrateurs systèmes ont déjà configuré ces détails pour nous.**

[1] Interpréteur de commande

Le programme MPI le plus simple en C ...

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

int main(int argc, char* argv[])
{
  int nbTask;
  int myRank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nbTask);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

  printf("I am task %d out of %d\n",myRank,nbTask);

  MPI_Finalize();
  return 0;
}
```

Le programme MPI le plus simple en fortran ...

```
program helloworld_mpi

  use mpi

  implicit none

  integer    :: nbTask, myRank, ierr

  call MPI_Init(ierr)

  call MPI_COMM_SIZE(MPI_COMM_WORLD, nbTask, ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, myRank, ierr)

  write (*,*) 'I am task', myRank, 'out of',nbTask

  call MPI_Finalize(ierr)

end program helloworld_mpi
```

Le programme MPI le plus simple ...

- **Compiler une application MPI:**
  - (optionnel) `module load openmpi`
  - en C: `mpicc -o helloworld_mpi_c helloworld_mpi.c`
  - en F90: `mpif90 -o helloworld_mpi_f helloworld_mpi.f90`
  - Note: `mpicc` et `mpif90` ne sont que des *wrapper*; pour savoir ce qui se cache derrière, tapper `mpicc -showme`; en particulier `mpicc` appelle *un* compilateur C qui peut être gcc, icc, pgcc, ...
- **Exécuter:**
  - `./helloworld_mpi_c`
  - `mpirun -np 2 ./helloworld_mpi_c`
  - tapper `man mpirun`
  - sur un processeur à x cœur, on peut lancer le programme avec plus de tâches que de cœurs, elles vont être multiplexée (toute fois en HPC, cela n'est pas recommandé)

Le programme MPI le plus simple ...

- `mpirun` peut lancer un programme MPI sur un ensemble de machines simplement connectées sur le réseau
- **Mais comment lancer le programme sur plusieurs machines en même temps ?** [2]
  utiliser l'option `-hostfile`,
  Comment spécifier le nombre de tâches par proc (par *socket*), ...
  `man mpirun` (cf TP sur `odette`)
  - **manuel** [3]: avoir le même système de fichier sur tous les nœuds (e.g. avoir les répertoires `home` montés par le réseau en NFS); avoir des clefs `ssh` accessibles depuis tous les nœuds
  - **automatique**: les super-calculateurs utilisent un **gestionnaire de travaux (ou planificateurs de tâches)** [4] qui s'occupent de gérer/allouer les ressources matérielles dans un environnement multi-utilisateurs et décident quand l'application sera exécutée; exemple: SGE, LoadLeveler, SLURM, TORQUE, ...

[2] http://www.open-mpi.org/faq/?category=running
[3] http://blogs.cisco.com/performance/mpi-newbie-requirements-and-installa
[4] Job scheduler / batch system / distribute resource manager

Le programme MPI le plus simple ...

- connaître les caractéristiques de la version d'OpenMPI disponible:
  `ompi_info`
- Exemple: découvrir les paramêtres (qui peuvent être changés au runtime) concernant la couche d'interconnection:
  `ompi_info --param btl all`
- Rendre verbeux l'exécution:
  `mpirun --mca mtl_base_verbose 30 ./helloworld_mpi.exe`
- Vocabulaire OpenMPI:
  - MCA = Modular Component Architecture
  - BTL = Byte Transfert Layer

Communicateurs MPI

- **Communicateur** : structure de donnée de la norma MPI, représentant un ensemble de tâches pouvant échanger des messages.
- **Communicateur par défaut:** `MPI_COMM_WORLD` ⇒ constante définie dans `mpi.h`; désigne l'ensemble de toutes les tâches du programme MPI; communicateur global
- Chaque tâche MPI est **identifiée** de manière unique par son **rang** dans le communicateur.
- Une tâche peut appartenir à plusieurs communicateurs (et avoir un identifiant dans chacun)
- Comment créer un communicateur ? e.g. `MPI_Comm_create`
- Opérations sur les communicateur ? e.g. `MPI_Comm_split` (création de sous-groupes)

- **Start processes**
- **Send messages:**
  - point-to-point: `MPI_Send`, `MPI_Bsend`, `MPI_Isend`, `MPI_Issend`, `MPI_Ibsend`
  - collective: `MPI_Reduce`, `MPI_Scatter`, ...
- **Receive messages:**
  - point-to-point: `MPI_Recv`, `MPI_Brecv`, `MPI_Irecv`, `MPI_Isrecv`, `MPI_Ibrecv`
  - collective: `MPI_Reduce`, `MPI_Scatter`, ...
- **Synchronize:** `MPI_Probe`, `MPI_Barrier`

MPI : Point-à-Point

Ce qu'il faut connaître:

- Les 4 modes de communications: *standard, buffered, synchronous, ready*
- **Communications bloquantes** *vs* **non-bloquantes**
- Qu'est une *deadlock*; comment les éviter ?
- Les fonctions: `MPI_Wait`, `MPI_Test`
- [avancé]: les communications persistantes

Références:
https://computing.llnl.gov/tutorials/mpi/
https://www.cac.cornell.edu/VW/mpip2p/

MPI : Point-à-Point

Ce qu'il faut connaître:

- les modes de communications régissent comment le système gère l'envoi et la réception d'un message
- exemple: lorsque l'appel de fonction *MPI_Send* se termine, que peut-on dire sur la réception du message ?
  - Peut-on savoir si la réception a déjà terminée, ou à peine commencé ?
  - Peut-on ré-utiliser les ressources (tableau mémoire) du message de manière sure ?

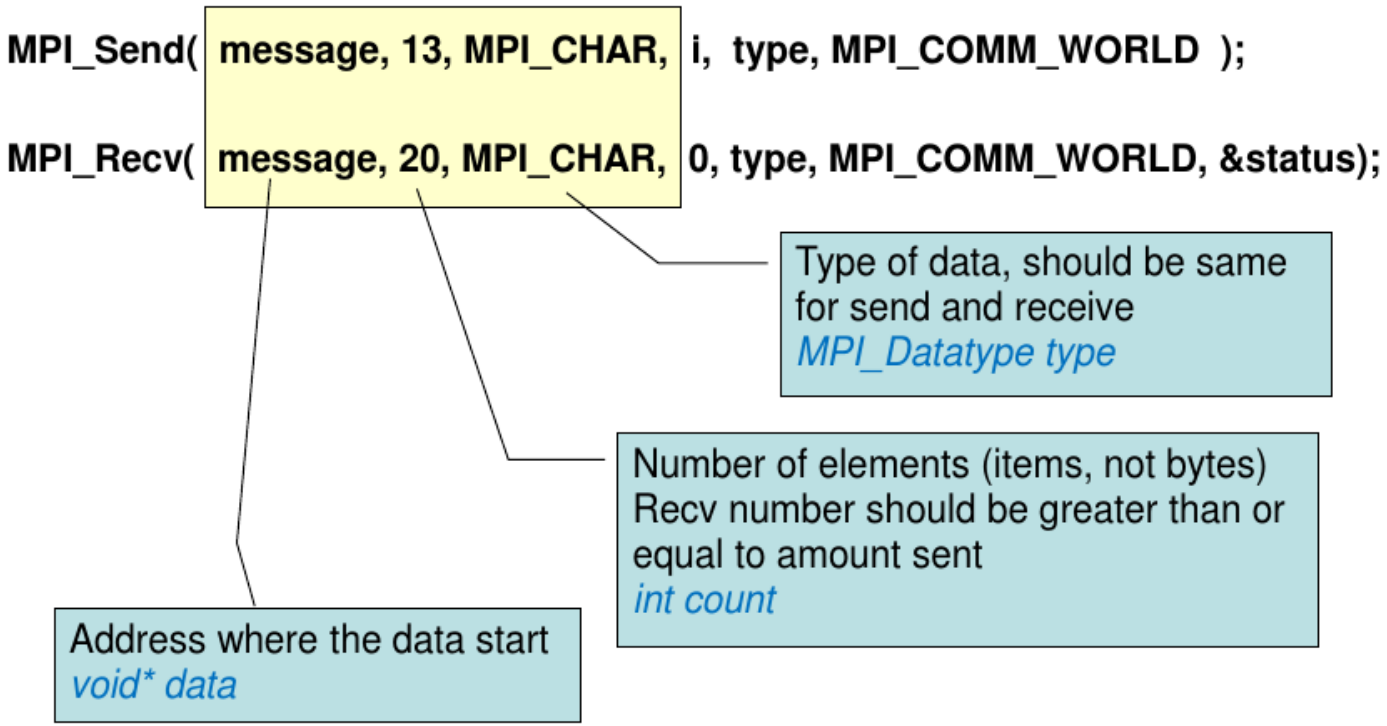
Références:
https://computing.llnl.gov/tutorials/mpi/
https://www.cac.cornell.edu/VW/mpip2p/

Ce qu'il faut connaître:

- Communications **bloquantes** *vs* **non-bloquantes**
  - Un appel à une routine de communication **bloquante** (`MPI_Send`, `MPI_Recv`) **suspend l'éxécution de la tâche MPI** appellante jusqu'à ce que le *buffer*/tableau (contenant le message) peut être ré-utilisé en toute **sécurité**.
  - Un appel à une routine de communication **non-bloquante** (`MPI_Isend`, `MPI_Irecv`) initialise le processus de communication; le **développeur** doit **vérifier** plus tard que la communication s'est bien déroulée (appel à `MPI_Wait`, `MPI_Test`), avant de pouvoir ré-utiliser le buffer.

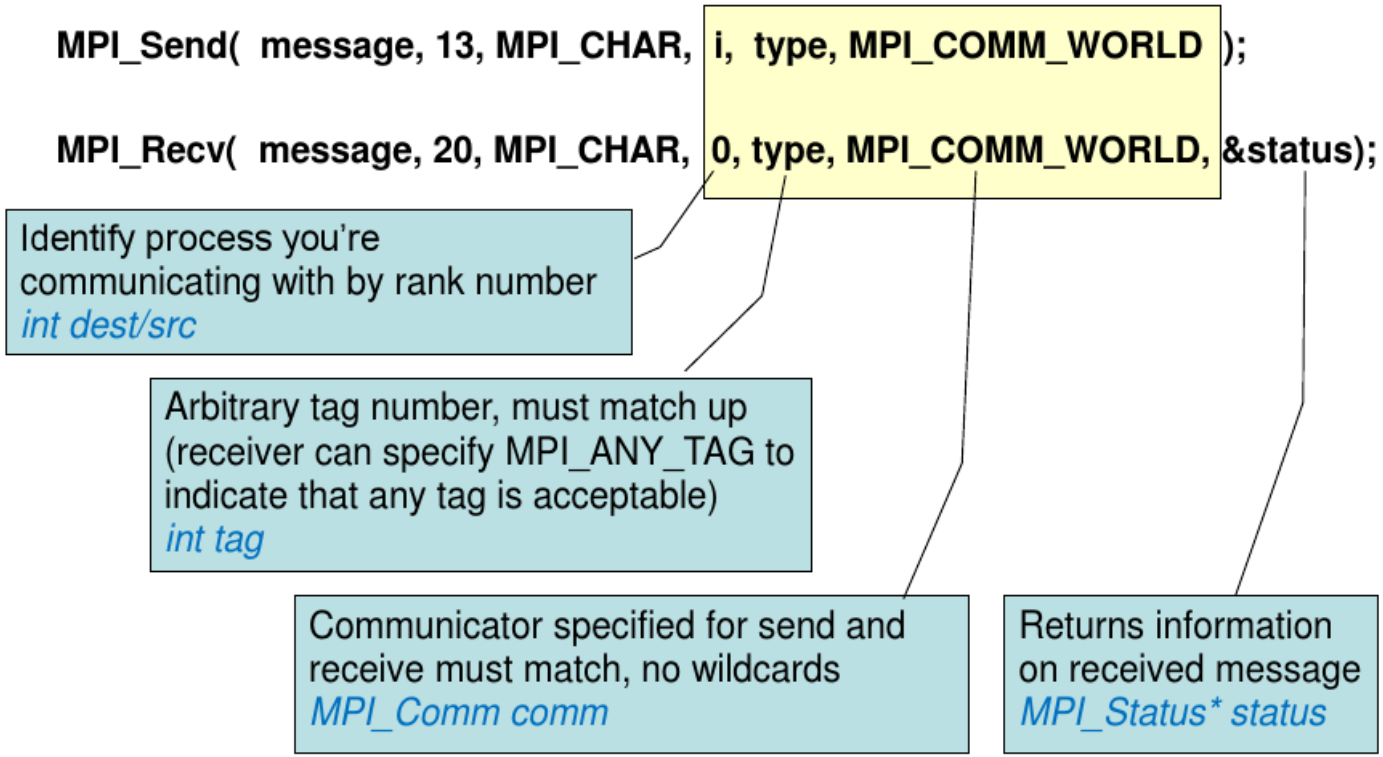
Références:
https://computing.llnl.gov/tutorials/mpi/
https://www.cac.cornell.edu/VW/mpip2p/

Point-à-Point: MPI_Send / MPI_Recv



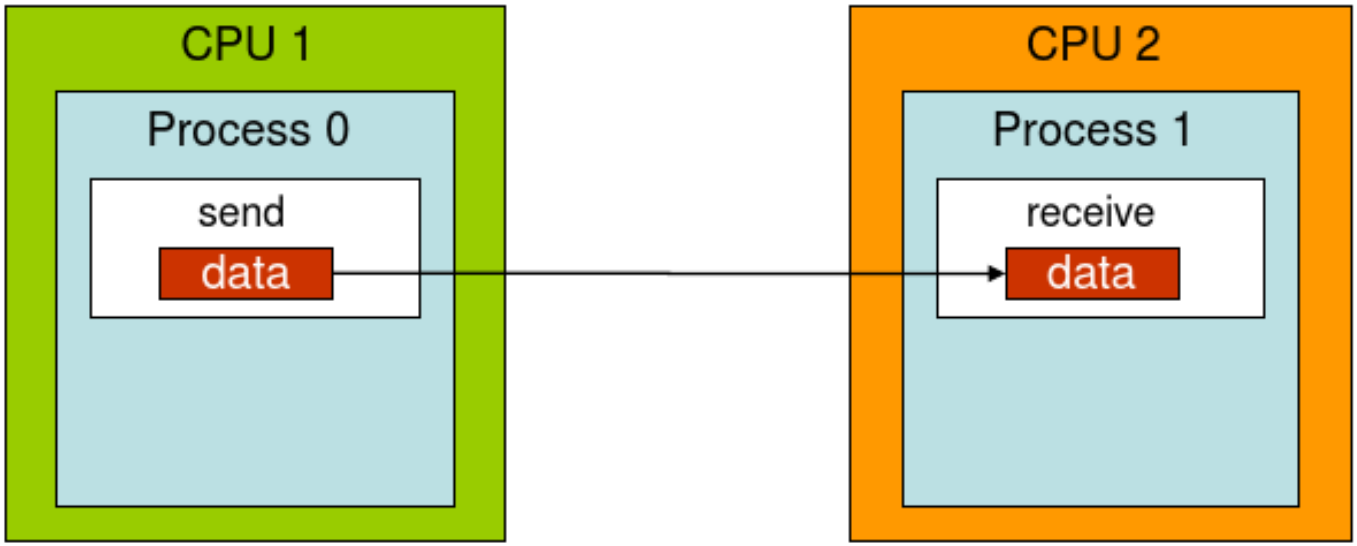
- Les 3 paramêtres qui décrivent les données / le message
- Les 3 paramêtres qui décrivent le *routage*
- *src*, *tag* peuvent prendre des valeurs *génériques*; noter les différences C/Fortran (passage par références et retour d'erreur)

Point-à-Point: MPI_Send / MPI_Recv



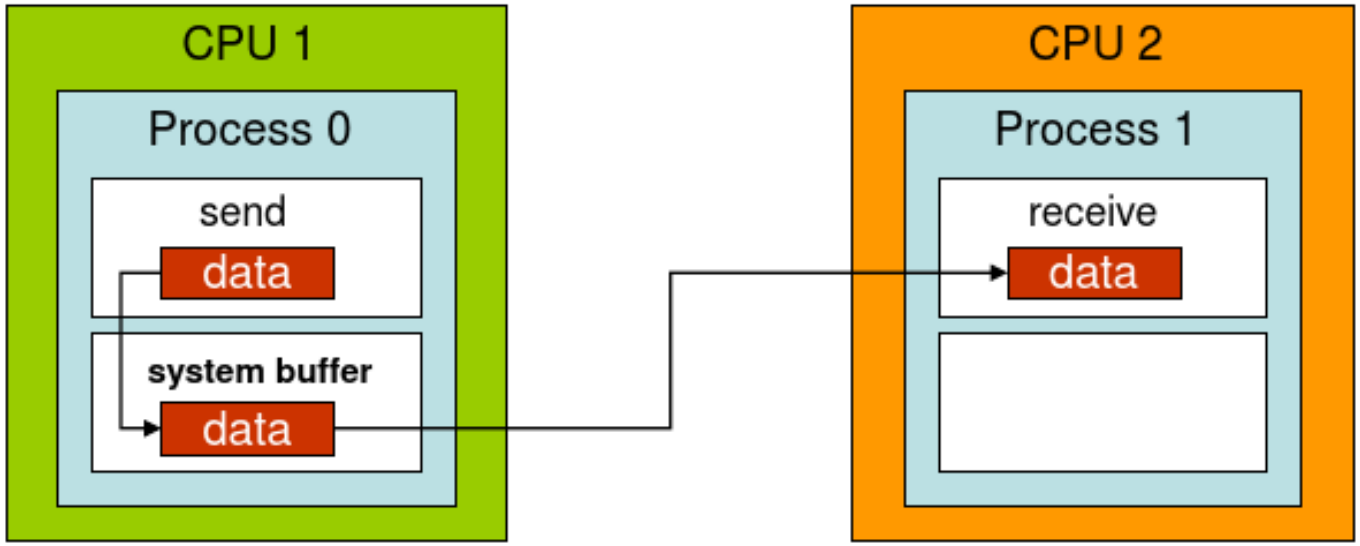
- Les 3 paramêtres qui décrivent les données / le message
- Les 3 paramêtres qui décrivent le *routage*
- *src*, *tag* peuvent prendre des valeurs *génériques*; noter les différences C/Fortran (passage par références et retour d'erreur)

Point-à-Point: `MPI_Send` / `MPI_Recv`



- Communication point-à-point; nécessite un envoyeur et un récepteur
- Communication *bufferisée*: `MPI_Bsend`, `MPI_Buffer_attach`

Point-à-Point: `MPI_Send` / `MPI_Recv`



- Communication point-à-point; nécessite un envoyeur et un récepteur
- Communication *bufferisée*: `MPI_Bsend`, `MPI_Buffer_attach`

Point-à-Point: `MPI_Send` / `MPI_Recv`

Communication Mode	Blocking Routines	Non-Blocking Routines
Synchronous	MPI_Ssend	MPI_Issend
Ready	MPI_Rsend	MPI_Irsend
Buffered	MPI_Bsend	MPI_Ibsend
Standard	MPI_Send	MPI_Isend
	MPI_Recv	MPI_Irecv
	MPI_Sendrecv	
	MPI_Sendrecv_replace	

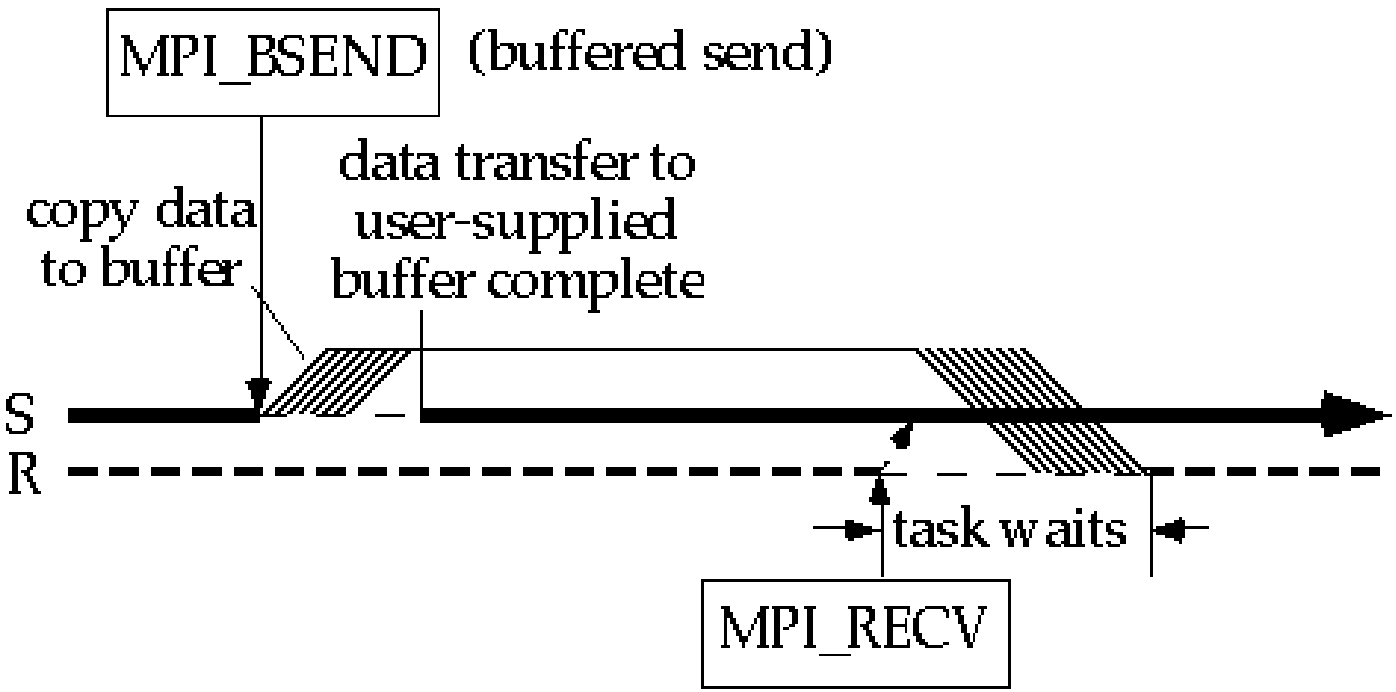
- Communication point-à-point; nécessite un envoyeur et un récepteur
- Communication *bufferisée*: `MPI_Bsend`, `MPI_Buffer_attach`

MPI : Point-à-Point

- **System overhead**
  Cost of transferring data from the sender's message buffer onto the network, then from the network into the receiver's message buffer.
  Good network connections improve system overhead.
  - Buffered send has more system overhead due to the extra buffer copy.
- **Synchronization overhead**
  Time spent waiting for an event to occur on another task.
  - In certain modes, the sender must wait for the receive to be executed and for the handshake to arrive before the message can be transferred.
  - Synchronous send has no extra copying but requires more waiting; a receive must be executed and a handshake must arrive before sending.
- `MPI_Send`
  Standard mode tries to trade off between the types of overhead.
  - Large messages use the "rendezvous protocol" to avoid extra copying: a handshake procedure establishes direct communication.
    http://www.cac.cornell.edu/ranger/mpip2p/rendezvous.html
  - Small messages use the "eager protocol" to avoid synchronization cost: the message is quickly copied to a small system buffer on the receiver.

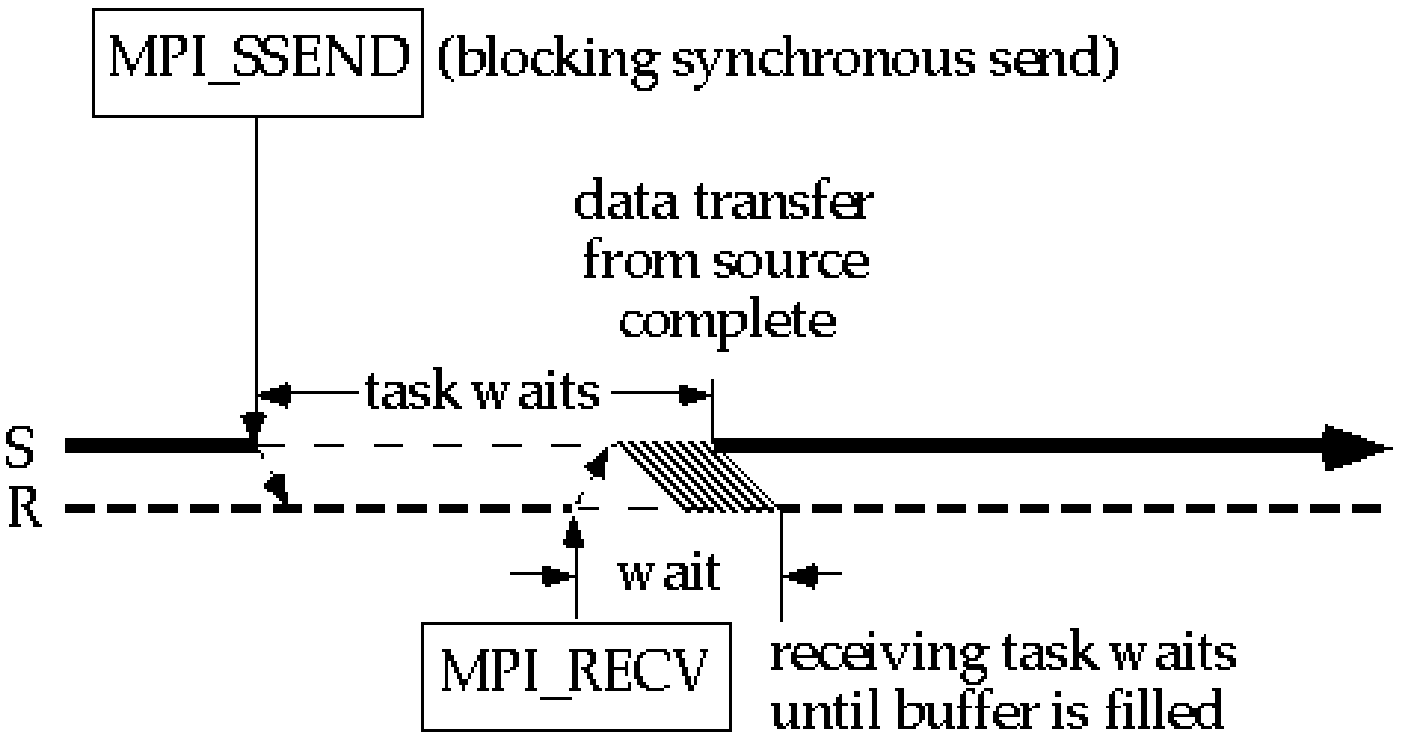
MPI : modes de communications

- **mode bufferisé**
  - le message est d'abord copié dans un buffer intermédiaire fourni par l'utilisateur (`MPI_Buffer_attach` / `MPI_Buffer_detach`)
  - les données (zone mémoire / tableau) en argument de `MPI_Bsend` peut être ré-utilisé
  - *system overhead* ajouté (coût en mémoire)
  - bonne synchro (les appels `MPI_Bsend` sont potentiellement *courts*)



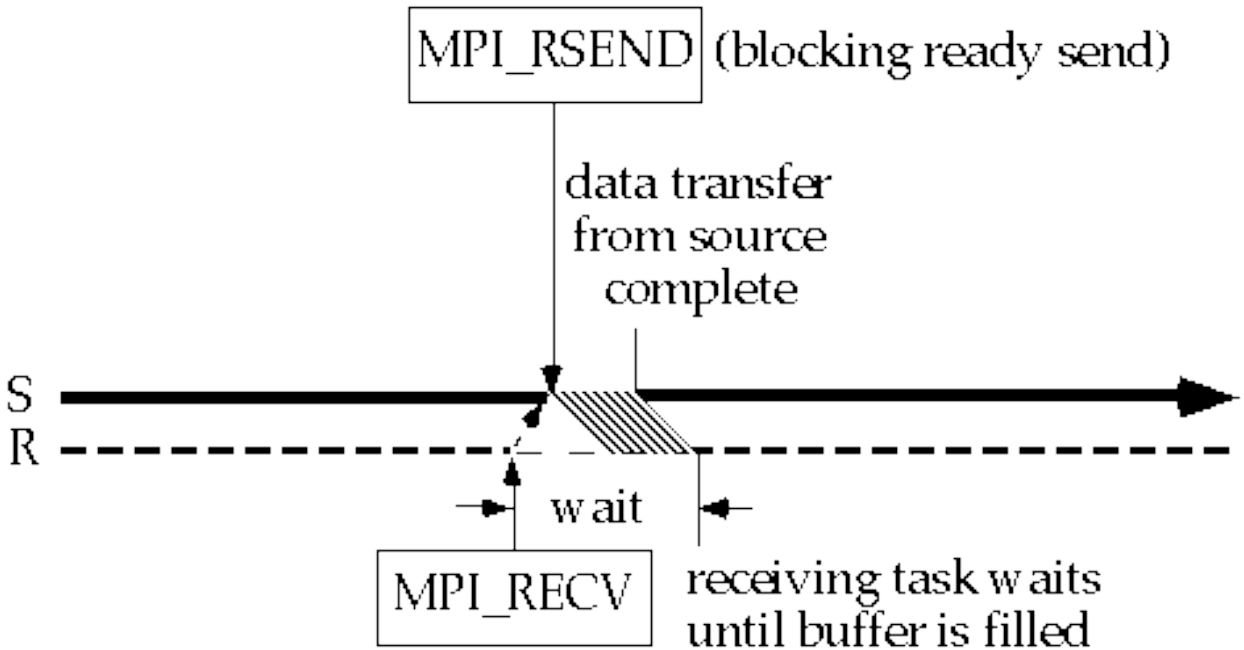
MPI : modes de communications

- **mode synchrone**
  - c'est le mode le plus *sûr*
  - l'envoi effectif du message sur le réseau ne commence que lorsque la procédure de *hand-shake* est terminée
  - l'envoyeur est bloqué jusqu'à ce que le récepteur soit prêt.
  - *synchronisation overhead* potentiellement important (appel à `MPI_Ssend` sont potentiellement *longs*)
  - pas d'*overhead system* (pas de tableau temporaire à allouer)



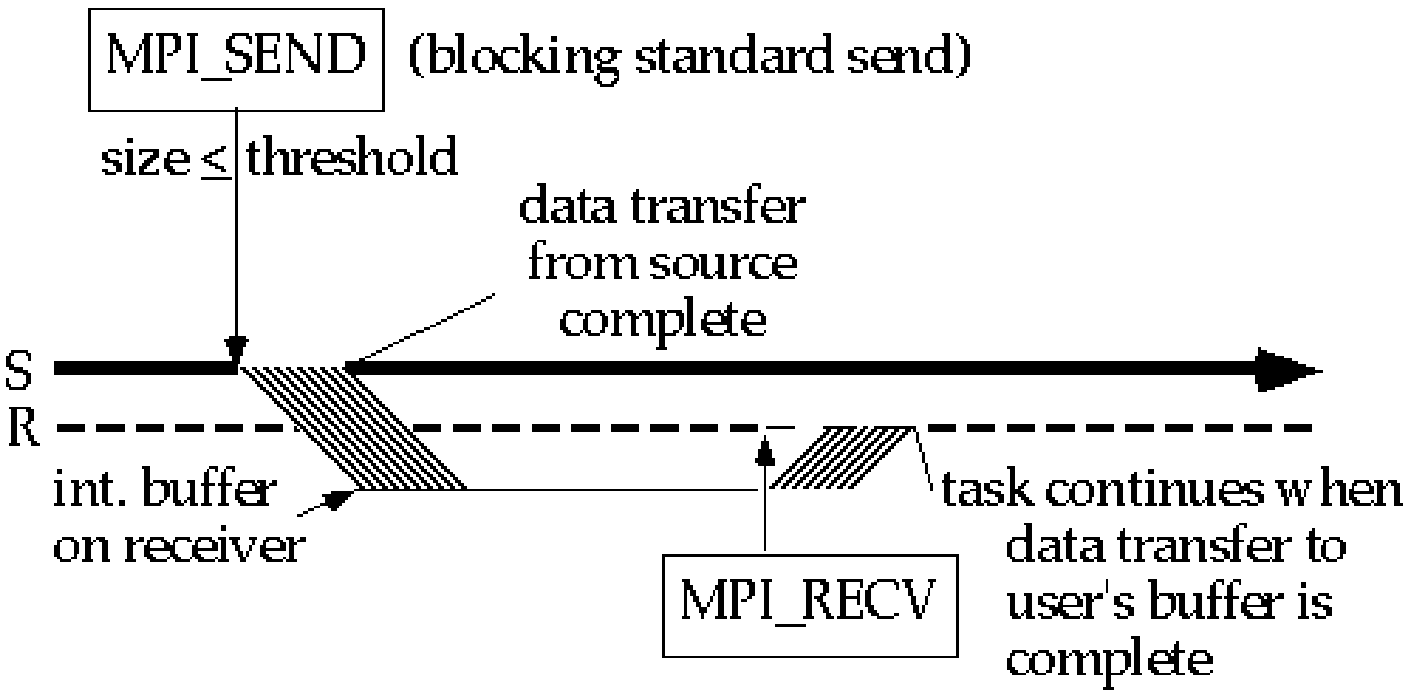
MPI : modes de communications

- **mode ready**
  - Réduire à la fois les délais systèmes et de synchronisation (beurre et l'argent du beurre)
  - **Hypothèse forte** le récepteur est déjà prêt à recevoir; si ce n'est pas le cas une erreur est générée coté récepteur (mais pas envoyeur) ⇒ difficile à gérer
  - coté envoyeur c'est le cas optimal; coté récepteur, il y a un coût potentiel à la synchro (`MPI_Recv` peut avoir démarré bien avant)
  - **Mode peu fréquent à l'utilisation**; l'algorithme doit garantir que les récepteurs sont toujours prêts pour éviter les erreurs.



MPI : modes de communications

- **mode standard / normal** (**compromis entre buffered et synchrone**)
  - plus difficile à définir: 2 stratégies
  - les messages courts sont bufferisés sur le **récepteur** (low synchro overhead), aussi appelé *eager protocol*
  - les messages longs on utilise le mode synchone (avec rendez-vous), mais pas de bufferisation
  - le seuil message court/long dépend de l'implémentation (voir l'exercise helloworld); parfois configurable au runtime par une variable d'environnement



MPI - Communications non-bloquantes

1. The letter `I` (think of `initiate`) appears in the name of the call, immediately following the first underscore: e.g., `MPI_Irecv`.
2. The final argument is a handle to an opaque (or hidden) request object that holds detailed information about the transaction. The request handle can be used for subsequent Wait and Test calls.

MPI - Communications non-bloquantes

- Blocking send, non-blocking recv

```
IF (rank==0) THEN
     ! Do my work, then send to rank 1
  CALL MPI_SEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
ELSEIF (rank==1) THEN
  CALL MPI_IRECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req,ie)
     ! Do stuff that doesn't yet need recvbuf from rank 0
  CALL MPI_WAIT (req,status,ie)
     ! Do stuff with recvbuf
ENDIF
```

- Non-blocking send, non-blocking recv

```
IF (rank==0) THEN
    ! Get sendbuf ready as soon as possible
  CALL MPI_ISEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req,ie)
    ! Do other stuff that doesn't involve sendbuf
ELSEIF (rank==1) THEN
  CALL MPI_IRECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req,ie)
ENDIF
CALL MPI_WAIT (req,status,ie)
```

MPI - Deadlock

- ***deadlock* (ou inter-blocage)**: en programmation concurrente, désigne une situation où 2 processus concurrent s'attendent mutuellement, et restent dans cet état d'attente définitivement ! ☹
- Programme MPI: situation typique où au moins 2 tâches MPI veulent échanger des messages, mais toutes les 2 veulent envoyer leur message respectif et ne sont pas prêtes à recevoir.

MPI - Deadlock - exemples

- **Deadlock 1**

```
IF (rank==0) THEN
  CALL MPI_RECV (recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
  CALL MPI_SEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
ELSEIF (rank==1) THEN
  CALL MPI_RECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
  CALL MPI_SEND (sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
ENDIF
```

- **Deadlock 2**

```
IF (rank==0) THEN
  CALL MPI_SSEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
  CALL MPI_RECV (recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
ELSEIF (rank==1) THEN
  CALL MPI_SSEND (sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
  CALL MPI_RECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
ENDIF
```

reference: Steve Lantz, CAC, Cornell University,
Workshop on Parallel Computing

MPI - Deadlock - exemples

- Solution 1

```
IF (rank==0) THEN
  CALL MPI_SEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
  CALL MPI_RECV (recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
ELSEIF (rank==1) THEN
  CALL MPI_RECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
  CALL MPI_SEND (sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
ENDIF
```

- Solution 2

```
IF (rank==0) THEN
  CALL MPI_SENDRECV (sendbuf,count,MPI_REAL,1,tag, &
                     recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
ELSEIF (rank==1) THEN
  CALL MPI_SENDRECV (sendbuf,count,MPI_REAL,0,tag, &
                     recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
ENDIF
```

reference: Steve Lantz, CAC, Cornell University,
Workshop on Parallel Computing

MPI - Deadlock - exemples

- Solution 3

```
IF (rank==0) THEN
  CALL MPI_IRECV (recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req,ie)
  CALL MPI_SEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
ELSEIF (rank==1) THEN
  CALL MPI_IRECV (recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req,ie)
  CALL MPI_SEND (sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
ENDIF
CALL MPI_WAIT (req,status)
```

- Solution 4

```
IF (rank==0) THEN
  CALL MPI_BSEND (sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ie)
  CALL MPI_RECV (recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ie)
ELSEIF (rank==1) THEN
  CALL MPI_BSEND (sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ie)
  CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ie)
ENDIF
```

reference: Steve Lantz, CAC, Cornell University,
Workshop on Parallel Computing

MPI - Deadlock - exemples

	CPU 0	CPU 1
Deadlock 1	Recv/Send	Recv/Send
Deadlock 2	Send/Recv	Send/Recv
Solution 1	Send/Recv	Recv/Send
Solution 2	Sendrecv	Sendrecv
Solution 3	Irecv/Send, Wait	Irecv/Send, Wait
Solution 4	Bsend/Recv	Bsend/Recv

reference: Steve Lantz, CAC, Cornell University,
Workshop on Parallel Computing

MPI Deadlock

Expliquer pourquoi le programme suivant n'est par sûr:

```
/* example to demonstrate the order of receive operations */
MPI_Comm_rank (comm, &myRank);
if (myRank == 0) {
  MPI_Send(sendbuf1, count, MPI_INT, 2, tag, comm);
  MPI_Send(sendbuf2, count, MPI_INT, 1, tag, comm);
 } else if (myRank == 1) {
  MPI_Recv(recvbuf1, count, MPI_INT, 0, tag, comm, &status);
  MPI_Send(recvbuf1, count, MPI_INT, 2, tag, comm);
 } else if (myRank == 2) {
  MPI_Recv(recvbuf1, count, MPI_INT, MPI_ANY_SOURCE, tag, comm, &status)
      ;
  MPI_Recv(recvbuf2, count, MPI_INT, MPI_ANY_SOURCE, tag, comm, &status)
      ;
 }
```

../code/c/solution/deadlock.c

MPI Performance

https://computing.llnl.gov/tutorials/mpi_performance/

- MPE (MPI Parallel Environment): Outil de logging/tracing pour MPI
- Visualisation de fichier trace (format CLOG2) avec `jumpshot` (distribué avec MPE)
- (septembre 2013): seule la version git de MPE est compatible avec MPICH, la version release suffit pour OpenMPI

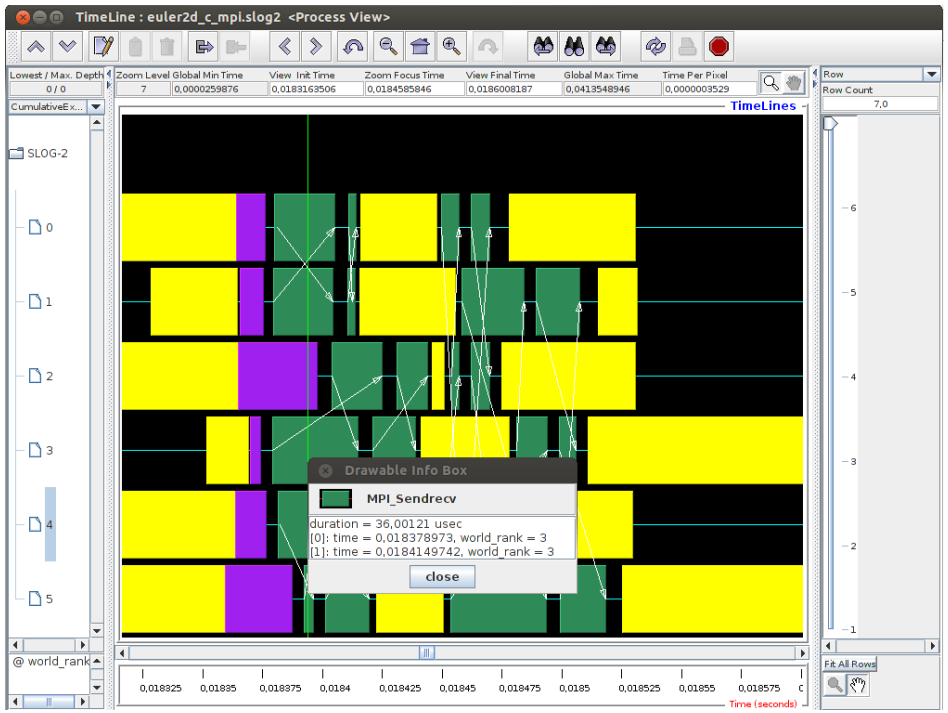


Figure: Capture d'écran de `jumpshot` montrant un exemple de trace d'éxecution d'un programme MPI.

Implementation d'OpenMPI

OpenMPI internal's:
http://www.open-mpi.org/video/?category=internals
En savoir plus sur l'implémentation bas-niveau:
slides de Brian Barret

Installer MPI / Executer un programme MPI

Quelques conseils pratiques pour utiliser MPI sur des postes de travail usuels (pas un super-calculateur, pas de gestionnaires de ressources/*job*):

- Installer MPI (e.g. OpenMPI) localement sur toutes les machines (ou sur un système de fichier partagé, par exemple montage NFS)
- Configuration SSH: il faut mettre en place des clés ssh (sans passphrase)
  - Créer la paire clé privée / clef publique: `ssh-keygen -t rsa`
  - Copier la clé publique sur les machines distantes: `ssh-copy-id -i /.ssh/id_rsa.pub monlogin@machine_distante`
  - (optionnel) S'assurer que *.bashrc* contient tout ce qui est nécessaire pour exécution en mode non-interactif [2].
- On pourra ensuite faire un petit test avec 1 seul processeur MPI, pour vérifier que l'exécution distante fonctionne:

```
ssh monlogin@machine_distante mpirun -np 1
$HOME/helloworld_mpi
```

[2] Le fichier `.bashrc` par defaut d'Ubuntu par ex, ne fait rien en mode non-interactif. Au besoin, commenter la ligne qui teste si la variable `PS1` n'existe pas.

Installer MPI / Executer un programme MPI

Quelques conseils pratiques pour utiliser MPI sur des postes de travail usuels (pas un super-calculateur, pas de gestionnaires de ressources/*job*):

- On peut ensuite faire un teste d'un programme MPI sur plusieurs nœuds:
  - Créer un fichier `machinefile.txt` contenant la liste des noms de machines sur le réseau local sur lesquelles on vient d'installer MPI et configurer ssh.
  - `mpirun -np 4 -machinefile machinefile.txt $HOME/helloworld_mpi`

MPI - Communications collectives

```c
/* Naive broadcast */
if (my_rank == 0) {
  for (rank=1; rank<nranks; rank++)
    MPI_Send( (void*)a, /* target= */ rank, ... );
} else {
  MPI_Recv( (void*)a, 0, ... );
}
```

../code/c/solution/naive_bcast.c

- *broadcast*: une tâche envoie un message à toutes les autres
- implémentation naïve est très (trop) lente; messages sérialisés dans la tâche émettrice
- optimisation en implementant de meilleurs algorithmes

MPI - Communications collectives

```c
/* Naive broadcast */
if (my_rank == 0) {
  for (rank=1; rank<nranks; rank++)
    MPI_Send( (void*)a, /* target= */ rank, ... );
} else {
  MPI_Recv( (void*)a, 0, ... );
}
```

../code/c/solution/naive_bcast.c

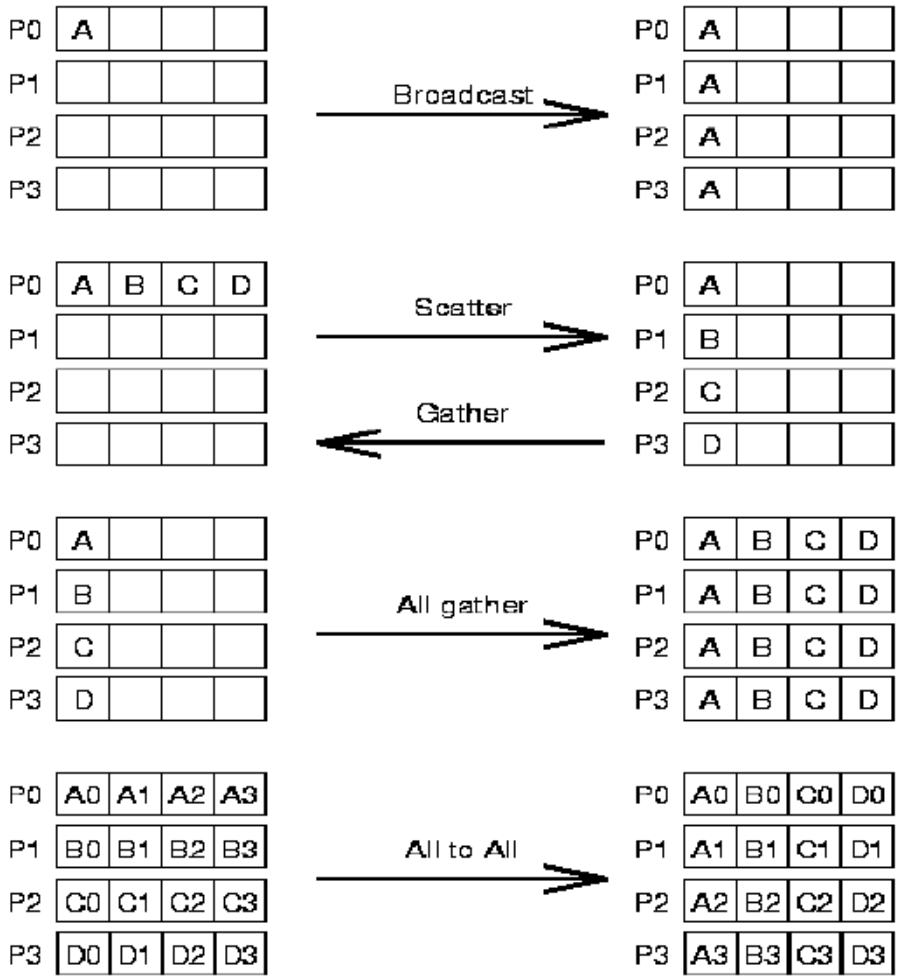
- OpenMPI par exemple utilise 6 algorithmes différents pour implanter l'opération *broadcast* (`MPI_Bcast`) (le choix est fait en fonction de la taille des données à diffuser et le nombre de récepteurs):
  - binary tree
  - binomial tree
  - splitted binary tree
  - pipeline
  - ...
- Qu'est ce qu'une communication collective ?
  - toutes les tâches d'un communicateur sont impliquées
  - Remplace une séquence complexe d'opérations P2P

MPI collective communication can be divided into three subsets:

- **Synchronization**
  - Barrier synchronization
- **Data Movement**
  - Broadcast from one member to all other members
  - Gather data from an array spread across processes into one array
  - Scatter data from one member to all members
  - All-to-all exchange of data
- **Global Computation**
  - Global reduction (e.g., sum, min of distributed data elements)
  - Scan across all members of a communicator

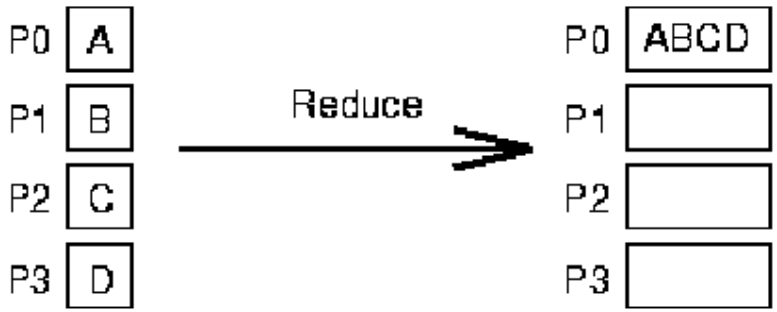
MPI - Communications collectives

- Broadcast
- Scatter/Gather
- Allgather
- Alltoall

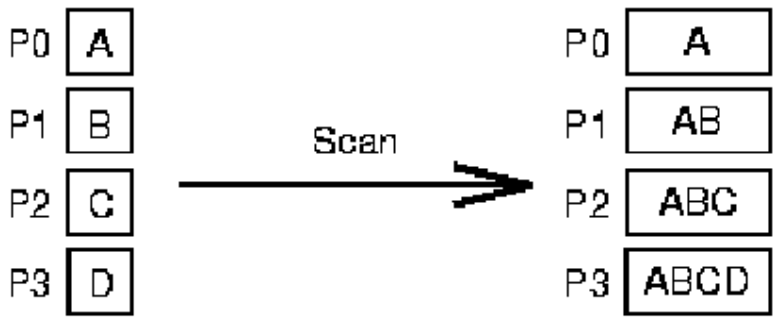


MPI - Communications collectives

- Reduce



- Scan



Exercice1: pour démarrer...

1. `Helloworld_mpi`
   - Compiler avec `mpicc` sur `odette`
   - Exécuter sur la frontale avec un petit nombre de tâches, puis augmenter. Jusqu'à combien de processeur pouvez aller ?
   - Que remarquez-vous sur l'ordre d'écriture des messages ?
   - Exécuter sur les noeuds de calcul avec le script de soumission
   - Changer le nombre de tâches MPI par nœud, nombre de nœuds, ..
   - Modifier le code en ajoutant un appel à `MPI_Get_processor_name`
   - Modifier le code pour que la tâche 0 affiche la version du standard MPI supporté par OpenMPI (utiliser `MPI_Get_version`)
   - Modifier le code pour que la tâche 0 affiche la version de l'implémentation MPI utilisée (utiliser la fonction `MPI_Get_library_version`)
   - Vérifier le *mapping* des tâches sur les nœuds en exécutant sur `gin` (`mpirun` avec l'option `-report-bindings`)

Exercice1: pour démarrer...

1. `Helloworld_mpi2` avec `MPI_Send` / `MPI_Recv` La tâche 0 envoit un *message* à la tâche 1.
   - Compiler et exécuter l'exemple.
   - Visualiser les traces avec l'outil `jumpshot` [2].
   - Que se passe-t-il si le paramêtre `source` de `MPI_Recv` est hors des possibilités (négatif ou supérieur à la taille du communicateur) ?

[2] Re-compiler l'exemple avec mpecc / mpefc et avec l'option `-mpilog`

Exercice1: pour démarrer...

1. `Helloworld_mpi3`
   - modifier `Helloworld_mpi2` pour que la tâche 0 envoie un message à la tâche 1 et réciproquement
     - Echanger les *tags* des messages. Que constatez-vous ?
     - Que se passe-t-il si l'ordre des opérations send/receive est inversé dans la tâche 1 ?
   - Modifier le code pour les messages envoyés contiennent un tableau.
     - Reprendre les questions précédentes (changer l'ordre de `MPI_Send` / `MPI_Recv`) et faire varier la taille des tableaux échangés. Vous devez constater un blocage pour une certaine taille de tableau. Laquelle ? Comment expliquer ce blocage en fonction des modes de communication ?
     - Que se passe-t-il si le tableau de réception n'est pas assez grand pour contenir le message ?
     - Que se passe-t-il si les paramêtres `count` de `MPI_Send` / `MPI_Recv` ne correspondent pas (trop grand ou trop petit) ?
   - Ré-écrire le code de cet exemple en utilisant `MPI_ISend` puis `MPI_Sendrecv`
   - Visualiser les traces avec jumpshot des différents programmes (avec communications bloquantes et non-bloquantes).

Helloworld revisité

- Peut-on forcer l'ordre dans lequel les messages sont affichés ?
  Utiliser `MPI_Barrier`
- Modifier `Helloworld_mpi` pour afficher les noms des noeuds sur lesquels s'exécutent le programme.
  Utiliser `MPI_Get_processor_name`

Exercice 2: latence / BW des communications MPI point à point

1. **Pourquoi est-ce important de connaitre la bande passante (B) et la latence (L)?**
   On peut grossièrement dire que le temps d'une communication est modélisée par $t_{com} = L + \frac{S}{B}$ où S est la taille en octets du message.
2. **Bande passante:** Utiliser le fichier `mpi/code/latency/mpi_bandwidth.c` pour mesurer la bande passante associée à une communication entre 2 tâches MPI.
3. **Latence:** Utiliser le fichier `mpi/code/latency/mpi_latency.c` pour mesurer la latence associée à une communication entre 2 tâches MPI.
4. **Mesurer la latence dans les 3 cas suivants:**
   - entre 2 tâches sur 2 cœurs d'un même *socket*
   - entre 2 tâches sur 2 cœurs de *sockets* différents
   - entre 2 tâches sur 2 nœuds différents

Parallelisation MPI

- Calcul de π
  utiliser `mpi/code/c/compute_pi.c`
- Intégration numérique par la méthode des trapèzes
  utiliser `mpi/code/c/trapeze.c`

MPI Exercice - anneau logique

- anneau: écrire le code d'un programme MPI, où on considère que les tâches MPI constitue un anneau et on souhaite que les données de la tâche i soit envoyée à la tâche $(i+1)\%p$
- assurez-vous que le code est fonctionnel quelque soit le nombre de tâches, et qu'il n'y a pas de blocage.

MPI Exercice - anneau logique

```c
void gather_ring(float *x, int blocksize, float *y)
{
  int i, p, my_rank, succ, pred;
  int send_offset, recv_offset;
  MPI_Status status;

  MPI_Comm_size (MPI_COMM_WORLD, &p);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  for (i=0; i<blocksize; i++)
    y[i+my_rank*blocksize] = x[i];
  succ = (my_rank+1)   % p;
  pred = (my_rank-1+p) % p;
  for (i=0; i<p-1; i++) {
    send_offset = ((my_rank-i+p)   %p ) * blocksize;
    recv_offset = ((my_rank-i-1+p) %p ) * blocksize;
    MPI_Send (y+send_offset, blocksize, MPI_FLOAT, succ, 0,
     MPI_COMM_WORLD);
    MPI_Recv (y+recv_offset, blocksize, MPI_FLOAT, pred, 0,
     MPI_COMM_WORLD, &status);
  }
}
```

../code/c/solution/gather_ring.c

MPI Exercice - anneau logique

- Le code de la page précédente permet de rassembler des blocs de données distribués; faire un schéma pour illustrer les accès mémoire
- Comment modifier le code si chaque tâche à un `blocksize` différent ?

MPI Exercice - multiplication de matrices

A faire pour la semaine prochaine:

- Implanter une multiplication matrice-matrice ($C = A \times B$) parallèle avec MPI.
- On suppose qu'au démarrage de l'application chaque tâche MPI *possède* un sous ensemble des lignes de A et des colonnes de B.
- Etablir le schéma de communication et proposer une implantation en C ou Fortran.

MPI - Exercice communicateur

- **A quoi peut servir un communicateur MPI ?**
  - Regrouper des processus MPI suivant une règle définie par le programme (cf `MPI_Comm_split`)
  - Regrouper des processus MPI pour former une topologie virtuelle (cf `MPI_Cart_create`) ce qui simplifie l'implantation d'une décomposition de domaine en géométrie cartésienne
  - Regrouper des processus MPI par applications; par exemple lorsque l'on souhaite faire du couplage de code, il est souvent quasi-impératif que chaque code travaille chacun avec un sous-communicateur de `MPI_COMM_WORLD`
  - Conception (génie logiciel) de librairie scientifique réutilisable: l'interface de programmation doit être générique, doit accepter de recevoir en entrée un communicateur pour permettre l'utilsation de la librairie avec un sous-ensemble de tâches MPI.
  - ...

MPI - Exercice communicateur

Ecrire un programme MPI basé sur `MPI_Comm_split` tel que:

- on créé un nouveau communicateur qui sépare en 2 groupes les tâches de rang pair / impair
- chaque tâche initialise une variable `data` à -1
- la tâche de rang 0 du group *pair* met `data` à 0 et celle du groupe *impair* met `data` à 1
- on effectue un *broacast* de `data` à l'intérieur de chaque groupe
- on affiche à l'écran la valeur de `data` pour vérification

MPI - Exercice équilibrage de charge

On suppose que chaque tâche MPI possède un tableau de taille différente. On souhaite faire un équilibrage de charge, i.e. faire un sorte que chaque tâche MPI après équilibrage possède un nouveau tableau ayant la même taille partout (sauf peut-être la dernière tâche). On suppose dans un premier temps que les données sont trop volumineuses pour être rassemblées dans la mémoire d'un seul processeur.
Proposer un algorithme, qui réalise l'équilibrage. Ecrire un tel programme à l'aide de MPI [2]. On pourra choisir la taille initiale du tableau aléatoire dans l'intervalle 100 - 200 par exemple.

1. Commencer par déterminer la taille du nouveau tableau.
2. Ensuite implanter les communications nécessaires pour redistribuer les données.

Optionnel:
Proposer une autre variante dans le cas où on lève la contrainte sur la taille des données [3].

[2] Indication: `MPI_Reduce`, `MPI_Scan`
[3] Essayer d'utiliser `MPI_Gatherv`

La machine odette de l'UFR

Le nœud de calcul

- **1 frontale interactive**
  `odette.informatique.univ-paris-diderot.fr` équipée de :
  - 2 processeurs -24 cœurs - AMD EPYC 7352
  - 1 To de mémoire vive
  - lancer la commande `lstopo`
  - lancer la commande `cat /proc/cpuinfo`
  - lancer la commande `cat /proc/meminfo`

Autres caractéristiques d'un nœud de calcul HPC:

- un système de fichiers parallèle (Lustre, GPFS, ...)
- un réseau rapide, fibre optique, très faible latence, très haut débit: infiniband, Fibre Channel
- Gestionnaire des ressources / tâches (*jobs*): SLURM, PBS, GridEngine (SGE), ..

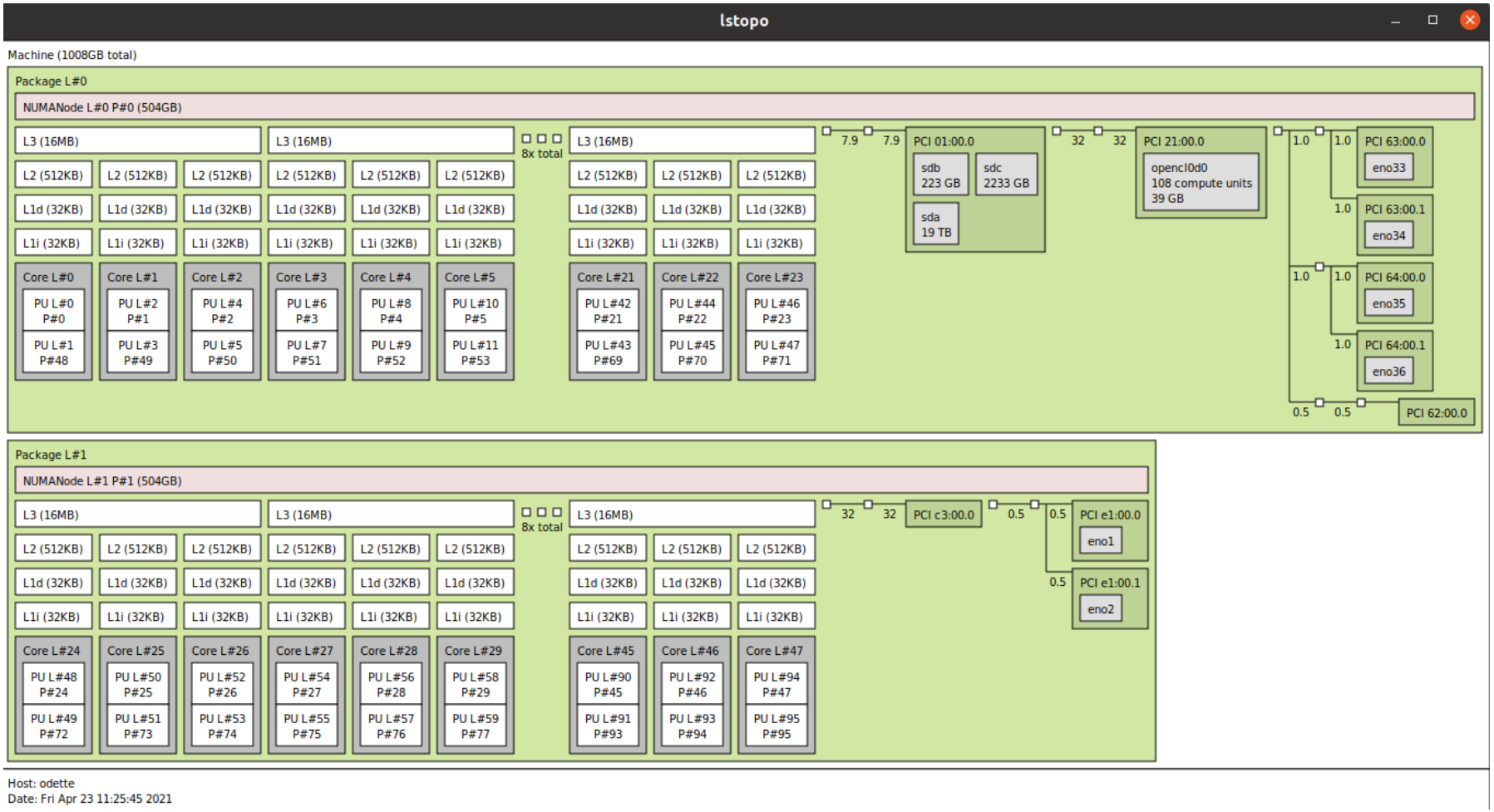
Environnement module (http://modules.sourceforge.net/):

La gestion de l'environnement est faite via l'utilisation de `module` :

- Ajout de dépôt: `module use /opt/modulefiles` (à mettre dans son .bashrc)
- Liste les softs disponibles :
  `module avail`
- Liste les softs intégrés /chargés dans l'environnement courant :
  `module list`
- Intègre/charge le logiciel `tool` dans l'environnement courant :
  `module load tool/version`
- Supprime le soft tool de l'environnement courant :
  `module unload tool/version`
- Remplace la version *old-version* du soft `tool` par la version *new-version* dans l'environnement courant :
  `module switch tool/old-version tool/new-version`
- Affiche les variables d'environnement modifiées/restaurées quand on charge/décharge un module :
  `module show tool/version`

Topologie matérielle sur `odette`

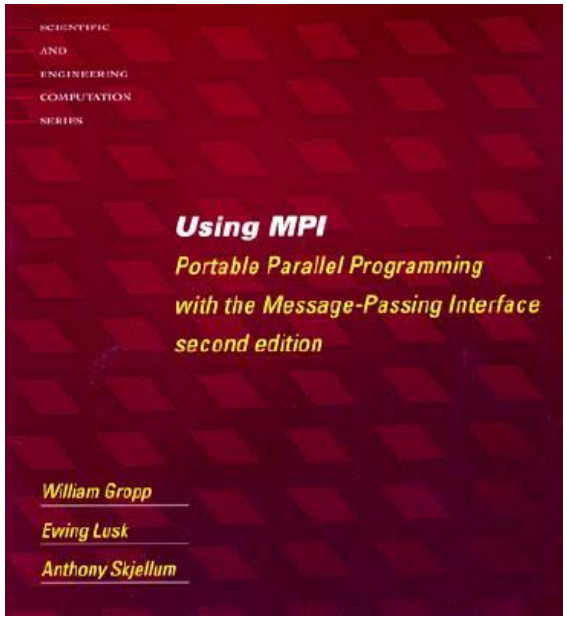
- use command `lstopo` to explore hardware on `odette`

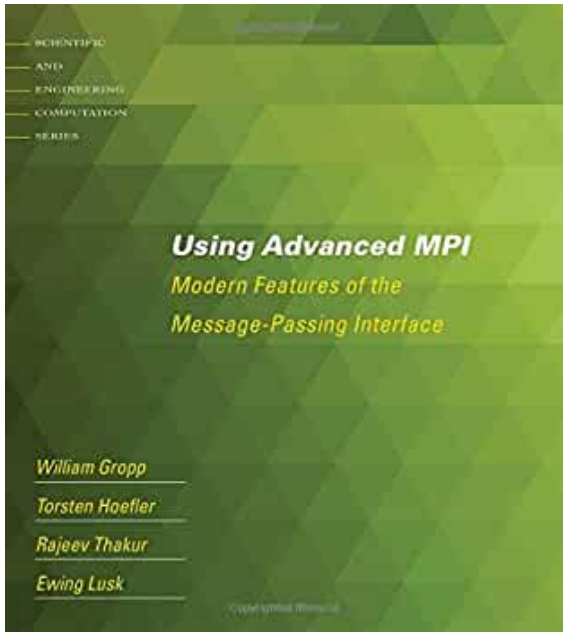


MPI-IO

- **Comment lire/écrire efficacement de (gros) fichiers en parallèle ?** i.e. comment utiliser au mieux la bande passante disponible d'accès aux *disques durs* ?mpi-io
- **At hardware level :** a distributed cluster of IO dedicated nodes + a parallel filesystem: Lustre, GPFS, etc...
- **At software level :** MPI-IO is a low-level programing interface to handle parallel file reading/writing
- examples : `https://github.com/pkestene/mpi-io-examples` ⇒ have a look at `mpi_file_set_view.c` to understand how MPI processes can collectively write into the same file efficiently
- **other high-level libraries** (that rely on MPI-IO to provide structured file format) used in scientific computing: HDF5, parallel-NetCDF

Livres sur MPI





- *Using MPI*, par W. Gropp, E. Lusk et A. Skjellum, MIT Press
- *Using advanced MPI*, par W. Gropp, T. Hoefler, R. Thakur et E. Lusk, MIT Press