Université BLAISE PASCAL
UFR Sciences Exactes et Naturelles
Département de Mathématiques et Informatique
63177 AUBIERE CEDEX

Maîtrise d'Informatique 2001-2002

Etude de cas

MPI – LAM

Nicolas DUMOULIN
Bruno GIVON
Benjamin RAVERDY
Julien VIGOUROUX

<u>Responsables :</u>
Jeanne-Marie GOURGAND
William GUYOT
Eric SANLAVILLE

SOMMAIRE

I. INTRODUCTION..4

1. PRESENTATION DE L'ETUDE DE CAS :..4
2. INTRODUCTION AU CALCUL PARALLELE : 4
3. INTRODUCTION A MPI (MESSAGE PASSING INTERFACE) :..4

II. ENVIRONNEMENT ET COMMUNICATION 6

1. GESTION DE L'ENVIRONNEMENT 6
2. FONCTIONNEMENT DE LAM..8
   2.1. Communications point à point : 8
   2.2. Communications collectives : 17

III. CONFIGURER LAM AVEC SSH..24

1. CONFIGURATION DU FICHIER .CSHRC...24
2. AUTHENTIFICATION SOUS *SSH*...25
3. CREATION D'UN FICHIER « *HOST* » 26

IV. CONCEPTION ET DEVELOPPEMENT D'UN PROGRAMME MPI : 27

1. ECHANGES DE MESSAGES ENTRE LES PROCESSUS : 27
2. PARALLELISATION D'UN PROGRAMME / PRISE EN MAIN DE MPI :..27
   2.1. Choix du langage : 27
   2.2. « Hello world » parallèle : 28
   2.3. Utilisation de MPI_Bcast() : 29
   2.4. Utilisation de MPI_Scatterv() : 30
   2.5. Utilisation d'une boucle partagée :..32
3. TECHNIQUES D'OPTIMISATION COMPLEMENTAIRES : 34
   3.1. Création de type de données personnalisées prêt à être utilisé par MPI : 34
   3.3. Utilisation des variables d'environnement MPI : 36

VI. TESTS : 37

1. TEST DE L'ENVIRONNEMENT LAM :...37
2. TEST DES PERFORMANCES DES PROGRAMMES REALISES : 39
   2.1. La boucle partagée :...39
   2.2. CROUT distribué :..42
   2.3. Programme réalisant un calcul massif :...43

SUPPLEMENT 46

V. CROUT AVEC MPI..47

1. INTRODUCTION :...47
2. QUELQUES RAPPELS SUR CROUT… 47
3. CROUT DISTRIBUE AVEC MPI...49
   3.1. Démarche de parallélisation :..49
   3.2. Listing 1 (CROUT avec MPI) :...50
   3.3. Listing 2 (CROUT avec le simulateur) :..58

VII. CONCLUSION 61

VIII. ANNEXE...62

AIDE MEMOIRE : MPI ET LE C 62

I. Introduction

1. Présentation de l'étude de cas :

L'année dernière, une étude de cas a déjà été faite sur le sujet que nous allons aborder. Cependant, en raison de problèmes techniques, elle n'avait pu aboutir. Nous allons donc la poursuivre et l'approfondir.

Le but de l'étude de cas est d'installer un logiciel capable d'implémenter des fonctions MPI puis de réaliser et d'exécuter un algorithme utilisant le parallélisme. Le but secondaire est de réaliser un comparatif entre algorithmes parallèles – linéaires et algorithmes parallèles – parallèles en fonction du nombre de machines utilisées.

2. Introduction au calcul parallèle :

Le calcul parallèle est une technique de calcul dans laquelle plusieurs actions sont menées simultanément, de telle sorte que le temps de résolution du problème se trouve réduit. Mis à part les composants matériels destinés au calcul parallèle, un support par des composants logiciels est également nécessaire afin de coordonner l'exécution simultanée de plusieurs lignes de code de calcul. Une telle dépendance est requise en raison des interdépendances existantes entre les différents codes.

3. Introduction à MPI (Message Passing Interface) :

MPI est une bibliothèque d'échanges de messages pour machines parallèles homogènes, c'est un standard créé pour le développement d'applications parallèles portables. Une application MPI est un ensemble de processus exécutant chacun son propre code (modèle SPMD) et communiquant via des appels à des sous-programmes de la bibliothèque MPI. Le modèle d'exécution d'une application MPI est le SPMD (*Single Program Multiple Data*), soit l'exécution du même programme pour tous les processus.

L'interface MPI permet donc l'exploitation des machines multiprocesseurs par passage de messages. Celle-ci doit être pratique, portable efficace et flexible. La première version finalisée date de mars 1994, le projet a principalement été mené par : le centre de recherche d'IBM T.J. Watson, le NX/2 Centre, le PARAMACS.

Les principaux objectifs de ce consortium étaient de concevoir une bibliothèque permettant les points suivants :

- communications bas et haut niveau
- portabilité quasi totale
- bonne modularité (pour le développement de bibliothèques)
- supportant l'hétérogénéité
- supportant divers groupes et topologies de processus

Remarque : la librairie MPI supporte divers langages comme le fortran 90, le C, le C++ ou encore Java.

Historique :

Version 1.0 : en juin 1994, le forum MPI (*Message Passing Interface Forum*), avec la participation d'une quarantaine d'organisations, aboutit à la définition d'un ensemble de fonctions concernant la bibliothèque d'échange de messages MPI.

Version 1.1 : apparue en juin 1995, cette version de MPI n'apporte que des changements mineurs par rapport à la version initiale.

Version 1.2 : apparue en 1997, cette version apporte essentiellement des changements mineurs pour une meilleure cohérence des dénominations de certaines fonctions.

II. Environnement et communication

1. Gestion de l'environnement

Le sous-programme ***MPI_INIT*** initialise l'environnement nécessaire et ***MPI_FINALIZE*** désactive cet environnement. Les opérations effectuées par MPI portent sur des `communicateurs`, c'est-à-dire un groupe de processus, chacun possédant un rang, et un contexte de communication, soit un objet gérant les échanges point à point et collectif à l'aide d'étiquettes (`tag`) au sein de ce groupe. Les messages sont toujours reçus dans le contexte d'où ils sont émis et les messages envoyés dans différents contextes n'interfèrent pas. Le communicateur par défaut est ***MPI_COMM_WORLD***, il comprend l'ensemble des processus actifs. Le nombre de processus gérés par un communicateur est fourni par le sous-programme ***MPI_COMM_SIZE***. De même, le sous-programme ***MPI_COMM_RANK*** retourne le rang d'un processus (valeur comprise entre 0 et celle renvoyée par ***MPI_COMM_SIZE*** -1).

Un peu de vocabulaire :

- **non bloquant** : une procédure rend la main avant que l'opération effectuée ne soit terminée et avant que l'utilisateur soit autorisé à réutiliser les ressources (comme les buffers) spécifiées dans l'appel.
- **bloquant** : lorsque la procédure rend la main, l'utilisateur peut réutiliser les ressources spécifiées dans l'appel.
- **local** : la complétion de l'opération ne dépend que du processus local. Une telle opération ne nécessite pas de communication avec un autre processus de l'utilisateur.
- **non local** : la complétion de l'opération dépend de l'exécution d'une procédure MPI avec un autre processus, par exemple une communication.
- **collectif** : tous les processus d'un groupe emploient la procédure.

Il existe différents modes de communication avec MPI :

- **standard** : MPI choisit ou non de temporiser le message en envoi (i.e. le copier dans une mémoire locale tampon). Si c'est le cas, l'envoi peut être terminé avant que la réception correspondante ne démarre. Dans le cas contraire, l'envoi ne se termine que si la réception a été postée et la donnée expédiée au destinataire. Le mode d'envoi **standard** est non local : la réussite de la communication peut dépendre de la présence de la réception correspondante.
- **synchronous** (synchronisé) : l'envoi **synchronous** peut commencer même si la réception correspondante n'est pas amorcée. Cependant, il ne sera effectué que si cette réception a été postée et l'opération de réception du message commencée. La complétion d'une opération d'envoi **synchronous** indique seulement que la réception a atteint un certain point dans son processus.
- **buffered** (bufferisé) : ce mode d'envoi peut démarrer et même finir sans que la réception ait été postée. Cette opération est alors **locale** puisqu'elle ne dépend pas de la réception correspondante.
- **ready** : un envoi qui utilise le mode de communication **ready** ne peut commencer que si la réception correspondante a déjà été postée. Dans le cas contraire, l'opération est erronée et les résultats faux.

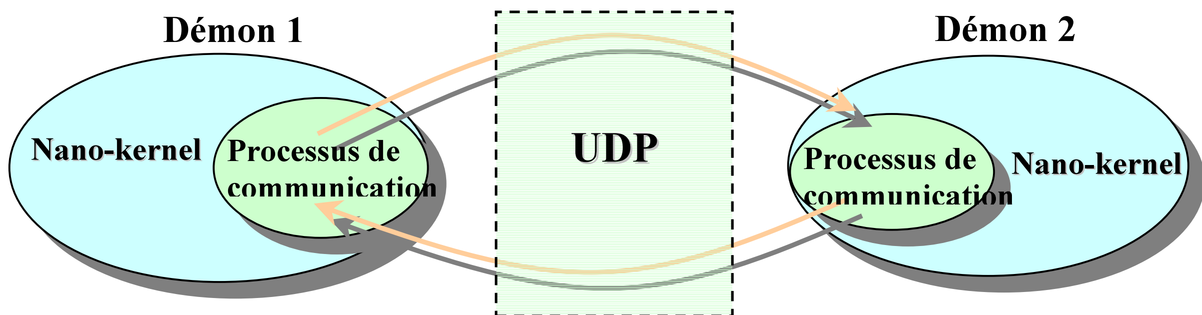
La correspondance entre les types MPI et les types Fortran et C est fournie dans les tableaux suivants :

Type MPI	**Type Fortran**
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)

Type MPI	**Type C**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

2. Fonctionnement de LAM

LAM se présente comme un simple démon présent sur chacune des machines. Chaque démon est structuré comme un micronoyau fournissant des services de passage de messages et de synchronisation. Certain processus internes au démon forment un sous-système de communication qui permet le passage de messages vers les démons des autres machines. La communication entre les différents processus se fait via UDP c'est à dire en mode non connecté et par paquets.



La communication se fait point à point : un message va d'un processus vers un autre sans nécessairement passer par le serveur. Les programmes écrits pour LAM sont compatibles avec les autres implémentations de MPI. LAM détecte les erreurs de communication dues à un « crash » d'une machine ou à une rupture de communication. LAM bloque toutes les communications vers la machine posant problème, les processus restants sont informés de la panne de manière asynchrone.

On peut donc gérer ce type d'erreurs dans les applications en supprimant les communications vers les processus ne réagissant plus et en créant de nouveaux.

2.1. Communications point à point :

Une communication point à point a lieu entre deux processus, l'un est appelé émetteur, l'autre récepteur, identifiés par leur rang. Il s'agit de la communication de base de MPI et les opérations effectuées sont un ***send*** et un ***receive***. Il existe différentes manières d'effectuer la communication. De manière générale, les paramètres nécessaires à sa construction sont :

- les rangs des processus émetteur et récepteur,
- l'étiquette du message,
- le nom du communicateur qui définit le contexte de la communication,
- le type des données échangées,
- les données.

Un code de retour fait partie des arguments, sa valeur (si différente de zéro) indique le type de problème rencontré.

a) Communications bloquantes :

Un envoi est bloquant signifie qu'il ne termine son action que lorsque les données envoyées ont été reçues et stockées par le destinataire de telle sorte que l'expéditeur peut accéder et modifier les valeurs en envoi.

La syntaxe d’un envoi bloquant ***standard*** est la suivante :

MPI_SEND (don, taille, dtype, dest, tag, comm, irc)

<dtype> don(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER taille (IN) nombre d'éléments à envoyer,
INTEGER dtype (IN) type MPI de chaque élément à envoyer,
INTEGER dest (IN) rang du processus destinataire,
INTEGER tag (IN) étiquette du message,
INTEGER comm (IN) communicateur,
INTEGER irc (OUT) code de retour.

Il s'agit de l'envoi d'un message identifié *tag*, de taille *taille*, de type *dtype*, à partir de l'adresse *on*, au processus *dest*, dans le communicateur *comm*. La longueur du message est spécifiée en nombre d'éléments plutôt qu'en octets. Seul le code de retour est modifié par l'appel (il est en sortie, soit *OUT*).

Les trois derniers arguments (sans compter le code de retour) constituent l'*enveloppe* du message. Le destinataire est *dest*, un entier dont la valeur est comprise entre 0 et le nombre de processus -1. L'étiquette *tag* est un entier compris entre 0 et 32767 (au minimum); il permet de distinguer les types de messages. L'argument *comm* indique le communicateur utilisé pour l'envoi du message.

La syntaxe d'un envoi bloquant ***synchronous*** est la suivante :

MPI_SSEND (don, taille, dtype, dest, tag, comm, irc)

<dtype> don(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER taille (IN) nombre d'éléments à envoyer,
INTEGER dtype (IN) type MPI de chaque élément à envoyer,
INTEGER dest (IN) rang du processus destinataire,
INTEGER tag (IN) étiquette du message,
INTEGER comm (IN) communicateur,
INTEGER irc (OUT) code de retour.

Un envoi bloquant en mode ***buffered*** s'écrit :

MPI_BSEND (don, taille, dtype, dest, tag, comm, irc)

<dtype> don(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER taille (IN) nombre d'éléments à envoyer,
INTEGER dtype (IN) type MPI de chaque élément à envoyer,
INTEGER dest (IN) rang du processus destinataire,
INTEGER tag (IN) étiquette du message,
INTEGER comm (IN) communicateur,
INTEGER irc (OUT) code de retour.

Enfin, l'envoi bloquant en mode ***ready*** s'écrit :

MPI_RSEND (don, taille, dtype, dest, tag, comm, irc)

<dtype> don(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER taille (IN) nombre d'éléments à envoyer,

INTEGER dtype (IN) type MPI de chaque élément à envoyer,
INTEGER dest (IN) rang du processus destinataire,
INTEGER tag (IN) étiquette du message,
INTEGER comm (IN) communicateur,
INTEGER irc (OUT) code de retour.

La réception bloquante est faite de la manière suivante :

MPI_RECV (don, taille, dtype, source, tag, comm, status, irc)

<dtype> don(*) (OUT) donnée (adresse initiale) à recevoir,
INTEGER taille (IN) nombre d'éléments à recevoir,
INTEGER dtype (IN) type MPI de chaque élément à recevoir,
INTEGER source (IN) rang du processus expéditeur,
INTEGER tag (IN) étiquette du message,
INTEGER comm (IN) communicateur,
INTEGER status (MPI_STATUS_SIZE) (OUT) informations sur la réception,
INTEGER irc (OUT) code de retour.

Il s'agit de la réception d'un message identifié *tag*, de taille *taille*, de type *dtype*, placé à partir de l'adresse *don*, du processus *source*, dans le communicateur *comm*. L'argument supplémentaire *status* contient des informations sur la réception. Les constantes ***MPI_SOURCE***, ***MPI_TAG*** et ***MPI_ERROR*** sont les indices de stockage dans *status* des champs source, étiquette et code d'erreur du message reçu.

Il n'y a qu'une opération de réception qui peut être associée avec les quatre opérations d'envoi. Elle est bloquante et ne rend la main que lorsque la donnée est reçue.

Exemple :

```
      PROGRAM send_recv

      include 'mpif.h'

      INTEGER mon_rang
      INTEGER irc
      INTEGER tag
      INTEGER status (MPI_STATUS_SIZE)

      REAL valeur

C Initialisation

      CALL MPI_INIT (irc)
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, mon_rang, irc)
      tag = 100

C Envoi

      IF (mon_rang .EQ. 0) THEN
      valeur = 10.0
      CALL MPI_SEND (valeur, 1, MPI_REAL, 1, tag, MPI_COMM_WORLD, irc)

C Réception
```

```
ELSE
valeur = 0.0
CALL MPI_RECV (valeur, 1, MPI_REAL, 0, tag, MPI_COMM_WORLD,
status, irc)
WRITE (6, *) 'Processus rang ', mon_rang, ' a recu la valeur ', valeur
ENDIF

C Sortie

CALL MPI_FINALIZE (irc)

STOP
END
```

La sélection d'un message dans une opération de réception est réalisée à l'aide du contenu de l'enveloppe du message. Un message peut être reçu par une opération de réception si son enveloppe correspond aux arguments *source*, *tag* et *comm* spécifiés dans l'opération de réception. Le destinataire peut spécifier comme source ***MPI_ANY_SOURCE*** et/ou comme étiquette ***MPI_ANY_TAG*** pour indiquer que n'importe quelle source et étiquette sont acceptées. Le communicateur doit être le même que pour l'envoi. Ainsi, un envoi doit spécifier son destinataire mais une réception peut accepter des messages d'un émetteur arbitraire.

L'échange des données entre deux processus permet d'effectuer une opération d'envoi et une opération de réception en une seule fois. Les deux processus peuvent être les mêmes. Cette opération est réalisée par ***MPI_SENDRECV***. Dans le cas de ***send*** et ***receive*** bloquants, il faut faire attention à l'ordre des opérations pour éviter les *deadlock*, c'est-à-dire (par exemple) que tous les processus sont en même temps en attente de réception.

MPI_SENDRECV (senddon, sendtaille, sendtype, dest, sendtag, recvdon, recvtaille, recvtype, source, recvtag, comm, status, irc)

<sendtype> senddon(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER sendtaille (IN) nombre d'éléments à envoyer,
INTEGER sendtype (IN) type MPI de chaque élément à envoyer,
INTEGER dest (IN) rang du processus destinataire,
INTEGER sendtag (IN) étiquette de l'envoi,
<recvtype> recvdon(*) (OUT) donnée (adresse initiale) à recevoir,
INTEGER recvtaille (IN) nombre d'éléments à recevoir,
INTEGER source (IN) rang du processus expéditeur,
INTEGER recvtag (IN) étiquette de la réception,
INTEGER comm (IN) communicateur,
INTEGER status (MPI_STATUS_SIZE) (OUT) informations sur la réception,
INTEGER irc (OUT) code de retour.

Il s'agit d'un envoi et d'une réception bloquants. Ces deux opérations utilisent le même communicateur, mais des étiquettes pouvant être différentes. Les données à envoyer et recevoir doivent être disjointes mais peuvent avoir des types et des longueurs différentes. On peut aussi permuter des valeurs entre deux processus, à l'aide du ***MPI_SENDRECV_REPLACE***.

MPI_SENDRECV_REPLACE (don, taille, type, dest, sendtag, source, recvtag, comm, status, irc)

<sendtype> don(*) (INOUT) donnée (adresse initiale) à envoyer,
INTEGER taille (IN) nombre d'éléments à envoyer,
INTEGER type (IN) type MPI de chaque élément à envoyer,
INTEGER dest (IN) rang du processus destinataire,
INTEGER sendtag (IN) étiquette de l'envoi,
INTEGER source (IN) rang du processus expéditeur,
INTEGER recvtag (IN) étiquette de la réception,
INTEGER comm (IN) communicateur,
INTEGER status (MPI_STATUS_SIZE) (OUT) informations sur la réception,
INTEGER irc (OUT) code de retour.

Il s'agit d'un envoi et d'une réception bloquants. La même donnée est utilisée pour l'envoi et la réception de telle sorte que le message envoyé est remplacé par le message reçu.

b) Communications non bloquantes :

On peut augmenter les performances en faisant un recouvrement des communications par des calculs. On utilise pour cela des communications *non bloquantes*. Un ***send non bloquant*** initialise l'opération d'envoi mais ne la réalise pas. L'appel au sous-programme sera fini avant que le message ne soit parti. Une opération de vérification ultérieure est nécessaire pour s'assurer de la fin de l'envoi. Le transfert des données à partir de la mémoire de l'expéditeur peut être fait simultanément avec des calculs, après l'initialisation de l'envoi et avant sa complétion.

Il en est de même pour une réception non bloquante. Un ***receive non bloquant*** initialise l'opération de réception mais ne la réalise pas. L'appel au sous-programme sera fini avant que le message ne soit reçu. Une opération de vérification ultérieure est nécessaire pour s'assurer de la réception effective du message. Le transfert des données vers la mémoire du destinataire peut être fait simultanément avec des calculs, après que la réception soit initialisée et avant sa complétion.

Les communications non bloquantes nécessitent un argument supplémentaire appelé ***request***, qui permet d'identifier les opérations de communication et de faire correspondre l'opération qui initialise la communication avec elle qui la réalise effectivement. Un objet ***request*** contient différentes informations concernant l'opération de communication : le mode d'envoi, le contexte, l'étiquette, les arguments de destination pour un ***send***, ou d'expédition pour un ***receive***.

MPI_ISEND (don, taille, dtype, dest, tag, comm, request, irc)

<dtype> don(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER taille (IN) nombre d'éléments à envoyer,
INTEGER dtype (IN) type MPI de chaque élément à envoyer,
INTEGER dest (IN) rang du processus destinataire,
INTEGER tag (IN) étiquette du message,
INTEGER comm (IN) communicateur,
INTEGER request (OUT) identifiant de l'envoi,
INTEGER irc (OUT) code de retour.

Les envois non bloquants utilisent les mêmes quatre modes que les envois bloquants et

ont le même sens. Dans tous les cas, l'initialisation de l'envoi est locale, le sous-programme rend immédiatement la main. Enfin un envoi non bloquant peut être mis en correspondance avec une réception bloquante et vice-versa.

La syntaxe d'un envoi non bloquant ***standard*** est la suivante :

MPI_ISEND (don, taille, dtype, dest, tag, comm, request, irc)

<dtype> don(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER taille (IN) nombre d'éléments à envoyer,
INTEGER dtype (IN) type MPI de chaque élément à envoyer,
INTEGER dest (IN) rang du processus destinataire,
INTEGER tag (IN) étiquette du message,
INTEGER comm (IN) communicateur,
INTEGER request (OUT) identifiant de l'envoi,
INTEGER irc (OUT) code de retour.

La syntaxe d'un envoi non bloquant ***synchronous*** est la suivante :

MPI_ISSEND (don, taille, dtype, dest, tag, comm, request, irc)

<dtype> don(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER taille (IN) nombre d'éléments à envoyer,
INTEGER dtype (IN) type MPI de chaque élément à envoyer,
INTEGER dest (IN) rang du processus destinataire,
INTEGER tag (IN) étiquette du message,
INTEGER comm (IN) communicateur,
INTEGER request (OUT) identifiant de l'envoi,
INTEGER irc (OUT) code de retour.

Un envoi non bloquant en mode ***buffered*** s'écrit :

MPI_IBSEND (don, taille, dtype, dest, tag, comm, request, irc)

<dtype> don(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER taille (IN) nombre d'éléments à envoyer,
INTEGER dtype (IN) type MPI de chaque élément à envoyer,
INTEGER dest (IN) rang du processus destinataire,
INTEGER tag (IN) étiquette du message,
INTEGER comm (IN) communicateur,
INTEGER request (OUT) identifiant de l'envoi,
INTEGER irc (OUT) code de retour.

Enfin, l'envoi non bloquant en mode ***ready*** s'écrit :

MPI_IRSEND (don, taille, dtype, dest, tag, comm, request, irc)

<dtype> don(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER taille (IN) nombre d'éléments à envoyer,
INTEGER dtype (IN) type MPI de chaque élément à envoyer,
INTEGER dest (IN) rang du processus destinataire,
INTEGER tag (IN) étiquette du message,

INTEGER comm (IN) communicateur,
INTEGER request (OUT) identifiant de l'envoi,
INTEGER irc (OUT) code de retour.

La réception non bloquante est faite de la manière suivante :

MPI_IRECV (don, taille, dtype, source, tag, comm, request, irc)

<dtype> don(*) (OUT) donnée (adresse initiale) à recevoir,
INTEGER taille (IN) nombre d'éléments à recevoir,
INTEGER dtype (IN) type MPI de chaque élément à recevoir,
INTEGER source (IN) rang du processus expéditeur,
INTEGER tag (IN) étiquette du message,
INTEGER comm (IN) communicateur,
INTEGER request (OUT) identifiant de la réception,
INTEGER irc (OUT) code de retour.

Un appel non bloquant génère un objet associé à la communication, ***request***, qui peut être utilisé pour connaître le statut d'une communication (envoi ou réception) ou permettre d'attendre sa complétion. Pour cela on utilise respectivement les routines ***MPI_TEST*** et ***MPI_WAIT***.

MPI_TEST (request, flag, status)

INTEGER request (INOUT) objet associé à la communication
LOGICAL flag (OUT) .TRUE. si la communication est effectuée
INTEGER status (MPI_STATUS_SIZE) (OUT) informations sur la communication

Un appel à ***MPI_TEST*** renvoie ***flag = .TRUE.*** si l'opération identifiée par ***request*** est achevée. Les valeurs en sortie des arguments ***request*** et ***status*** sont alors non définies. Dans le cas contraire, l'appel renvoie ***flag = .FALSE.*** . Dans ce dernier cas, la valeur de l'objet ***status*** est non définie. Cette fonction est locale.

MPI_WAIT (request, status)

INTEGER request (INOUT) objet associé à la communication,
INTEGER status (MPI_STATUS_SIZE) (OUT) informations sur la communication.

Un appel à ***MPI_WAIT*** ne rend la main que lorsque l'opération identifiée par ***request*** est achevée. Cette fonction est non locale. Elle entraîne la désallocation de l'objet ***request***. Un objet peut aussi être libéré sans pour autant devoir attendre la complétion de la communication associée :

MPI_REQUEST_FREE (request)

INTEGER request (INOUT) objet associé à la communication.

Si la communication est en cours, l'objet ne sera désalloué qu'après complétion.

c) Communications persistantes :

Il arrive parfois que l'on effectue une même communication un grand nombre de fois avec les mêmes arguments (même si les valeurs sont différentes), par exemple dans une boucle pour faire des mises à jour en fin d'itération. Dans un tel cas, il est possible d'optimiser la communication en associant ses arguments à un objet ***request*** *persistant* (initialisation) puis en utilisant cet objet ***request*** pour effectuer les communications. Cette construction permet de réduire l'*overhead* à chaque échange. Le ***request*** associé à la communication persistante est crée à l'aide d'une des quatre routines suivantes.

Création du ***request*** pour une communication persistante en mode ***standard*** :

MPI_SEND_INIT (don, taille, dtype, dest, tag, comm, request, irc)

<dtype> don(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER taille (IN) nombre d'éléments à envoyer,
INTEGER dtype (IN) type MPI de chaque élément à envoyer,
INTEGER dest (IN) rang du processus destinataire,
INTEGER tag (IN) étiquette du message,
INTEGER comm (IN) communicateur,
INTEGER request (OUT) identifiant de la communication,
INTEGER irc (OUT) code de retour.

Création du ***request*** pour une communication persistante en mode ***buffered*** :

MPI_BSEND_INIT (don, taille, dtype, dest, tag, comm, request, irc)

<dtype> don(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER taille (IN) nombre d'éléments à envoyer,
INTEGER dtype (IN) type MPI de chaque élément à envoyer,
INTEGER dest (IN) rang du processus destinataire,
INTEGER tag (IN) étiquette du message,
INTEGER comm (IN) communicateur,
INTEGER request (OUT) identifiant de la communication,
INTEGER irc (OUT) code de retour.

Création du ***request*** pour une communication persistante en mode ***synchronous*** :

MPI_SSEND_INIT (don, taille, dtype, dest, tag, comm, request, irc)

<dtype> don(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER taille (IN) nombre d'éléments à envoyer,
INTEGER dtype (IN) type MPI de chaque élément à envoyer,
INTEGER dest (IN) rang du processus destinataire,
INTEGER tag (IN) étiquette du message,
INTEGER comm (IN) communicateur,
INTEGER request (OUT) identifiant de la communication,
INTEGER irc (OUT) code de retour.

Création du ***request*** pour une communication persistante en mode ***ready*** :

MPI_RSEND_INIT (don, taille, dtype, dest, tag, comm, request, irc)

<dtype> don(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER taille (IN) nombre d'éléments à envoyer,
INTEGER dtype (IN) type MPI de chaque élément à envoyer,
INTEGER dest (IN) rang du processus destinataire,
INTEGER tag (IN) étiquette du message,
INTEGER comm (IN) communicateur,
INTEGER request (OUT) identifiant de la communication,
INTEGER irc (OUT) code de retour.

La réception persistante est initialisée par :

MPI_RECV_INIT (don, taille, dtype, source, tag, comm, request, irc)

<dtype> don(*) (OUT) donnée (adresse initiale) à recevoir,
INTEGER taille (IN) nombre d'éléments à recevoir,
INTEGER dtype (IN) type MPI de chaque élément à recevoir,
INTEGER source (IN) rang du processus expéditeur,
INTEGER tag (IN) étiquette du message,
INTEGER comm (IN) communicateur,
INTEGER request (OUT) identifiant de la réception,
INTEGER irc (OUT) code de retour.

Le premier argument, *don*, est mis en sortie pour spécifier le droit d'écriture sur les données à recevoir. Ces appels n'engendrent pas de communication. Une communication persistante (envoi ou réception) qui utilise un ***request*** persistant est amorcée par la subroutine ***MPI_START*** :

MPI_START (request)

INTEGER request (INOUT) identifiant de la communication

L'argument ***request*** est construit par un des cinq appels précédents et doit être inactif (c'est-à-dire inutilisé). Il devient actif une fois l'appel effectué. Cet appel est local avec des caractéristiques analogues à celles d'une communication non bloquante. Par exemple, un appel à ***MPI_START*** avec un ***request*** crée par ***MPI_SEND_INIT*** effectue la communication de la même manière qu'un appel à ***MPI_ISEND***.

2.2. Communications collectives :

Une communication collective est une communication impliquant un groupe de processus et réaliser ainsi en une fois une série de communications point à point avec possibilité d'effectuer une opération sur les données. Ces opérations se font au sein de communicateurs et n'ont pas besoin d'étiquette (***tag***). On peut classer les communications collectives en plusieurs catégories : synchronisation, transferts de données, transferts et opérations sur les données.

a) Synchronisation :

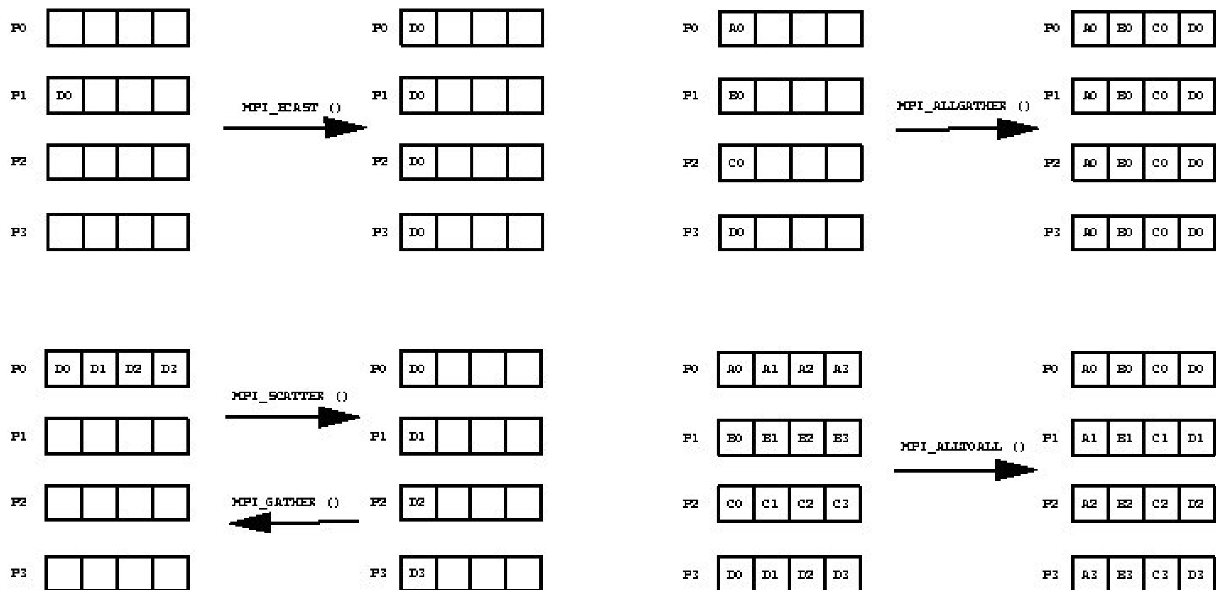
La routine ***MPI_BARRIER*** bloque le processus appelant jusqu'à ce que tous les membres du groupe aient fait de même. Ensuite il rend la main.

MPI_BARRIER (comm)

INTEGER comm (IN) communicateur

b) Transferts :

Il existe différents types de transferts, selon le nombre de processus émetteurs, récepteurs. Le diagramme suivant présente schématiquement les communications collectives :



MPI_BCAST (don, taille, dtype, source, comm, irc)

<dtype> don(*) (INOUT) donnée (adresse initiale) à envoyer
INTEGER taille (IN) nombre d'éléments à envoyer,
INTEGER dtype (IN) type MPI de chaque élément à envoyer,
INTEGER source (IN) rang du processus émetteur,
INTEGER comm (IN) communicateur,
INTEGER irc (OUT) code de retour.

MPI_BCAST diffuse un message du processus *source* vers tous les processus du groupe, lui compris. Cet appel est effectué par tous les processus utilisant les mêmes arguments pour *source* et *comm*.

MPI_GATHER (senddon, sendtaille, sendtype, recvdon, recvtaille, recvtype, source, comm, irc)

<sendtype> senddon(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER sendtaille (IN) nombre d'éléments à envoyer,
INTEGER sendtype (IN) type MPI de chaque élément à envoyer,
<recvtype> recvdon(*) (OUT) donnée (adresse initiale) à recevoir,
INTEGER recvtaille (IN) nombre d'éléments à recevoir,
INTEGER recvtype (IN) type MPI de chaque élément à recevoir,
INTEGER source (IN) rang du processus destinataire,
INTEGER comm (IN) communicateur,
INTEGER irc (OUT) code de retour.

Dans un appel à ***MPI_GATHER***, chaque processus (*source* inclus) envoie ses données au processus *source* qui les reçoit et les stocke dans l'ordre des rangs des processus. L'adresse de stockage de la réception est ignorée par tous les processus différents de *source*. Les arguments *sendtaille* et *sendtype* de tout processus émetteur doivent correspondre respectivement aux arguments *recvtaille* et *recvtype* du processus *source*. De plus les arguments *source* et *comm* doivent être les mêmes pour tous les processus.

La routine ***MPI_SCATTER*** effectue l'opération inverse de ***MPI_GATHER*** : le processus *source* distribue ses données par bloc entre les processus du groupe : le i-ème processus reçoit le i-ème bloc.

MPI_SCATTER (senddon, sendtaille, sendtype, recvdon, recvtaille, recvtype, source, comm, irc)

<sendtype> senddon(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER sendtaille (IN) nombre d'éléments à envoyer,
INTEGER sendtype (IN) type MPI de chaque élément à envoyer,
<recvtype> recvdon(*) (OUT) donnée (adresse initiale) à recevoir,
INTEGER recvtaille (IN) nombre d'éléments à recevoir,
INTEGER recvtype (IN) type MPI de chaque élément à recevoir,
INTEGER source (IN) rang du processus expéditeur,
INTEGER comm (IN) communicateur,
INTEGER irc (OUT) code de retour.

Les arguments *sendtaille* et *sendtype* du processus émetteur doivent correspondre respectivement aux arguments *recvtaille* et *recvtype* de tout processus récepteur. De plus les arguments *source* et *comm* doivent être les mêmes pour tous les processus.

MPI_ALLGATHER collecte des données réparties sur tous les processus et les leur diffusent ensuite. C'est un ***MPI_GATHER*** où tous les processus sont destinataires.

MPI_ALLGATHER (senddon, sendtaille, sendtype, recvdon, recvtaille, recvtype, comm, irc)

<sendtype> senddon(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER sendtaille (IN) nombre d'éléments à envoyer,
INTEGER sendtype (IN) type MPI de chaque élément à envoyer,
<recvtype> recvdon(*) (OUT) donnée (adresse initiale) à recevoir,
INTEGER recvtaille (IN) nombre d'éléments à recevoir,
INTEGER recvtype (IN) type MPI de chaque élément à recevoir,
INTEGER comm (IN) communicateur,
INTEGER irc (OUT) code de retour.

Les arguments *sendtaille* et *sendtype* d'un processus doivent correspondre aux arguments *recvtaille* et *recvtype* de tout autre processus.

Enfin, il est possible de faire des échanges croisés entre tous les processus à l'aide du ***MPI_ALLTOALL***. C'est en quelque sorte une extension du ***MPI_ALLGATHER*** où chaque processus envoie des données distinctes à chacun des destinataires : le j-ème bloc envoyé par le processus i est reçu par le processus j comme le i-ème bloc dans *recvdon*. Les arguments *sendtaille* et *sendtype* d'un processus doivent correspondre aux arguments *recvtaille* et *recvtype* de tout autre processus.

MPI_ALLTOALL (senddon, sendtaille, sendtype, recvdon, recvtaille, recvtype, comm, irc)

<sendtype> senddon(*) (IN) donnée (adresse initiale) à envoyer,
INTEGER sendtaille (IN) nombre d'éléments à envoyer,
INTEGER sendtype (IN) type MPI de chaque élément à envoyer,
<recvtype> recvdon(*) (OUT) donnée (adresse initiale) à recevoir,
INTEGER recvtaille (IN) nombre d'éléments à recevoir,
INTEGER recvtype (IN) type MPI de chaque élément à recevoir,
INTEGER comm (IN) communicateur,
INTEGER irc (OUT) code de retour.

Ces communications collectives ont des analogues (***MPI_GATHERV***, ***MPI_SCATTERV***, ***MPI_ALLGATHERV***, ***MPI_ALLTOALLV***) pour travailler avec des données dont le nombre n'est pas le même pour tous les processus.

c) Réduction :

Une opération de réduction consiste à appliquer une même opération à un ensemble de valeurs réparties sur les processus pour en extraire une valeur unique qui sera envoyé à un (***MPI_REDUCE***) ou tous les processus (***MPI_ALLREDUCE***). Les opérations de réduction prédéfinis sont : somme, produit, maximum, minimum, indice du maximum, indice du minimum, etc ...

MPI_REDUCE (senddon, recvdon, taille, dtype, oper, dest, comm, irc)

<sendtype> senddon (*) (IN) donnée à réduire (adresse initiale),
<recvtype> recvdon (*) (OUT) résultat (adresse initiale),

INTEGER taille (IN) nombre d'éléments à recevoir,
INTEGER dtype (IN) type MPI de chaque élément à recevoir,
INTEGER oper (IN) opération de réduction,
INTEGER dest (IN) rang du processus destinataire,
INTEGER comm (IN) communicateur,
INTEGER irc (OUT) code de retour.

MPI_REDUCE combine les éléments figurant dans *senddon* de chaque processus du groupe à l'aide de l'opération de réduction *oper* et met le résultat dans *recvdon* du processus *dest*. Les données à réduire sont caractérisées par les arguments *senddon*, *taille* et *dtype*. Le résultat est caractérisé par les arguments *recvdon*, *taille* et *dtype*. Les deux ont donc le même nombre d'éléments de même type. L'appel est effectué par tous les membres du groupe et ont les mêmes arguments *taille*, *dtype*, *oper*, *dest*, *comm*. Dans le cas où chaque processus fournirait plus d'un élément, l'opération est d'abord effectuée sur cet ensemble. L'opération *oper* est toujours supposée associative, les opérations prédéfinies sont aussi commutatives. Le type de données doit être compatible avec l'opération effectuée. Le développeur peut construire ses propres opérations de réduction.
Le premier tableau présente la liste des opérations de réduction prédéfinie :

Nom	**Définition**
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	somme
MPI_PROD	produit
MPI_LAND	.AND. logique
MPI_BAND	.AND. bit à bit
MPI_LOR	.OR. logique
MPI_BOR	.OR. bit à bit
MPI_LXOR	.XOR. logique
MPI_BXOR	.XOR. bit à bit
MPI_MAXLOC	maximum et indice
MPI_MINLOC	minimum et indice

Le second tableau fournit les compatibilités entre les opérations et les types de données :

oper	***dtype***
MPI_MAX, ***MPI_MIN***	entier C, entier Fortran, flottant
MPI_SUM, ***MPI_PROD***	entier C, entier Fortran, flottant, complexe
MPI_LAND, ***MPI_LOR***, ***MPI_LXOR***	entier C, booléen
MPI_BAND, ***MPI_BOR***, ***MPI_BXOR***	entier C, entier Fortran

Avec les définitions suivantes :

entier C : ***MPI_INT***, ***MPI_LONG***, ***MPI_SHORT***, ***MPI_UNSIGNED_SHORT***, ***MPI_UNSIGNED***, ***MPI_UNSIGNED_LONG***
entier Fortran : ***MPI_INTEGER***
flottant : ***MPI_FLOAT***, ***MPI_DOUBLE***, ***MPI_REAL***, ***MPI_DOUBLE_PRECISION***, ***MPI_LONG_DOUBLE***
booléen : ***MPI_LOGICAL***
complexe : ***MPI_COMPLEX***

Exemple :

```
      PROGRAM reduce

      include 'mpif.h'

      INTEGER mon_rang
      INTEGER irc
      INTEGER tag
      INTEGER status (MPI_STATUS_SIZE)

      REAL valeur, somme

C Initialisation

      CALL MPI_INIT (irc)
      CALL MPI_COMM_RANK (MPI_COMM_WORLD, mon_rang, irc)

C Valeurs

      IF (mon_rang .EQ. 0) THEN
      valeur = 1000.0
      ELSE
      valeur = REAL (mon_rang)
      ENDIF

C Réduction

      CALL MPI_REDUCE (valeur, somme, 1, MPI_REAL, MPI_SUM, 0,
      MPI_COMM_WORLD, irc)

C Sortie

      IF (mon_rang .EQ. 0) WRITE (6,*) 'Somme globale : ', somme
      CALL MPI_FINALIZE (irc)

      STOP
      END
```

Les routines ***MPI_MINLOC*** et ***MPI_MAXLOC*** sont particulières puisqu'elles retournent deux valeurs. La première (respectivement la seconde) renvoie la valeur minimale (respectivement maximale) et le rang du processus qui possède ce minimum (respectivement maximum) global. L'opération qui définit ***MPI_MINLOC*** est la suivante :
$\{u, i\} \circ \{v, j\} = \{w, k\}$ avec $w = min\ (u,v)$ et k défini par :
i , si $u < v$
$k = min\ (i,j)$, si $u = v$
j , si $u > v$
Pour ***MPI_MAXLOC*** le principe est le même :
$\{u, i\} \circ \{v, j\} = \{w, k\}$ avec $w = max\ (u,v)$ et k défini par :
i , si $u < v$
$k = min\ (i,j)$, si $u = v$
j , si $u > v$

Ces deux opérations sont commutatives et associatives. Elles s'appliquent à des données fournies sous forme de paires : la valeur et l'indice. Pour résoudre le problème de type hétérogène en Fortran, des types prédéfinis sont fournis par MPI.

Fortran	
Nom	**Définition**
MPI_2REAL	paire de *real*
MPI_2DOUBLE_PRECISION	paire de variables en *double precision*
MPI_2INTEGER	paire d'*integer*
C	
Nom	**Définition**
MPI_FLOAT_INT	couple *float* et *int*
MPI_DOUBLE_INT	couple *double* et *int*
MPI_LONG_INT	couple *long* et *int*
MPI_2INT	paire de variables *int*
MPI_SHORT_INT	couple *short* et *int*
MPI_LONG_DOUBLE_INT	couple *long double* et *int*

Pour le Fortran, le type ***MPI_2REAL*** est analogue à la définition suivante d'un type dérivé :

MPI_TYPE_CONTIGUOUS (2, MPI_REAL, MPI_2REAL)

Il en est de même pour les autres types Fortran. Pour le C, le type ***MPI_FLOAT_INT*** est défini par la suite d'instructions suivantes :

type[0] = MPI_FLOAT
type[1] = MPI_INT
disp[0] = 0
disp[1] = sizeof(float)
block[0] = 1
block[1] = 1
MPI_TYPE_STRUCT (2, block, disp, type, MPI_FLOAT_INT)

Il en est de même pour les autres types C.
MPI fournit des variantes à chaque opération de réduction qui permettent une redistribution du résultat vers tous les processus du groupe : ***MPI_ALLREDUCE***.

MPI_ALLREDUCE (senddon, recvdon, taille, dtype, oper, comm, irc)

<sendtype> senddon(*) (IN) donnée à réduire (adresse initiale),
<recvtype> recvdon(*) (OUT) résultat (adresse initiale),
INTEGER taille (IN) nombre d'éléments à recevoir,
INTEGER dtype (IN) type MPI de chaque élément à recevoir,
INTEGER oper (IN) opération de réduction,
INTEGER comm (IN) communicateur,
INTEGER irc (OUT) code de retour.

Il existe aussi des variantes qui distribuent le vecteur résultat à tous les processus du groupe :

MPI_REDUCE_SCATTER.

MPI_REDUCE_SCATTER (senddon, recvdon, recvtaille, dtype, oper, comm, irc)

<sendtype> senddon(*) (IN) donnée à réduire (adresse initiale),
<recvtype> recvdon(*) (OUT) résultat (adresse initiale),
INTEGER recvtaille(*) (IN) nombre d'éléments à envoyer, par processus,
INTEGER dtype (IN) type MPI de chaque élément à recevoir,
INTEGER oper (IN) opération de réduction,
INTEGER comm (IN) communicateur,
INTEGER irc (OUT) code de retour.

Cet appel effectue en premier une opération de réduction sur les éléments du vecteur défini par *senddon*, *total = Somme (recvtaille[i])* et *dtype*. Ensuite, le vecteur des résultats est décomposé en *n* segments disjoints, ou *n* est le nombre de processus dans le groupe. Le segment *i,* formé de *recvtaille[i]* éléments, est envoyé au processus *i* et stocké dans *recvdon*.

III. Configurer LAM avec SSH

La version de LAM utilisée est la 6.5.1.

Configurer LAM de manière à ce que les transactions se fassent via *ssh* est relativement simple, cependant il y a quelques règles à observer si l'on veut que tout fonctionne correctement.

1. Configuration du fichier .cshrc

Tout d'abord, on doit s'assurer que le script lancé par le shell lancé après *ssh* soit « propre », c'est à dire qu'il ne doit pas générer de message d'erreur. Dans le cadre de notre étude de cas, lors d'une connexion sur *sciora* ou *amipc7*, le shell lancé est *tcsh*, nous avons donc dû modifier les fichiers *.cshrc* et *.login*.

Maintenant on doit configurer LAM de manière à ce qu'il utilise ssh pour ses transactions et non rsh comme il le fait par défaut. Pour cela, on doit modifier une variable d'environnement, sous ksh, bash :

```
% set LAMRSH= « ssh -x »
% export LAMRSH
```

Sous csh ou tcsh :

```
% setenv LAMRSH « ssh -x »
```

On peut inclure cette commande dans le fichier .cshrc.

Pour pouvoir obtenir un fichier .cshrc convenable pour pouvoir lancer LAM, on peut ajouter dans le fichier des conditions de manière à ne pas avoir d’erreur :

```
if (($HOST != sciora)&& ($HOST != amipc7)) then
    # ajouter ici les lignes générant des erreurs
    # lors de la connexion sur sciora
else
    setenv LAMRSH « ssh -x »
endif
```

2. Authentification sous *ssh*

Ensuite on peut configurer *ssh* de manière à ce qu'on n'ait pas à retaper le mot de passe à chaque connexion. En effet, à chaque fois que l'on « boote » LAM, un mot de passe est demandé pour chaque machine, ce qui s'avère fastidieux dès que le nombre de machines connecter est important.

Tout d'abord on doit générer une clé RSA pour *ssh*, pour cela on utilise la commande *ssh-keygen* :

```
% ssh-keygen
```

Un emplacement et un nom sont demandés pour la création du fichier, il est néanmoins recommandé de conserver l'emplacement par défaut (*~/.ssh/identity*)

On doit ensuite copier et renommer le fichier *~/.ssh/identity.pub* généré par *ssh-keygen* en *authorized_keys* :

```
% cd ~/.ssh
% cp identity.pub authorized_keys
```

On doit ensuite vérifier les droits d'accès à ces fichiers, les fichiers du répertoire *~/.ssh* doivent avoir les droits suivants :

```
-rw-r--r--  authorized_keys
-rw-------  identity
-rw-r--r--  identity.pub
-rw-------  known_hosts
```

Vérifier que les répertoires *$HOME* et *$HOME/.ssh* ont les droits correspondants à 755 de manière à ce qu'ils soient accessibles partout sur le réseau.

Maintenant, on doit lancer le programme *ssh-agent*, c'est ce programme qui est responsable du fait que l'on n'aura pas à retaper un mot de passe à chaque connexion. Pour le lancer, il faut taper la commande suivante :

```
% eval `ssh-agent`
```

Pour ne pas avoir à taper cette commande à chaque fois que l'on se connecte :

```
% ssh-add
```

Voilà, maintenant si tout se passe bien ssh est configuré.

3. Création d'un fichier « *host* »

Reste ensuite à créer un fichier lamhost où l'on spécifie les machines à employer pour l'exécution du programme, par exemple :

lamhost :

```
sciora
amipc7
```

Pour plus de sécurité on peut spécifier le nom complet des machines :

lamhost :

```
sciora.univ-bpclermont.fr
amipc7.univ-bpclermont.fr
```

Néanmoins cette solution permet d'accéder aux machines par un chemin extérieur au réseau, ces accès peuvent être interdits par l'administrateur.

Il est possible de spécifier pour une machine le fait qu'on fasse tourner plusieurs processus :

lamhost :

```
sciora.univ-bpclermont.fr CPU=2
amipc7.univ-bpclermont.fr
```

Sur cet exemple, on demandera à ce que 2 processus tournent sur sciora. Cette possibilité est très utile pour effectuer des tests lorsque l'on ne dispose que de peu de machines. On « bootera » LAM par la commande suivante :

```
% lamboot -v lamhost
```

Où *lamhost* est le fichier contenant le nom des machines du réseau.

IV. Conception et développement d'un programme MPI :

Nous allons maintenant nous intéresser à la démarche à adopter pour concevoir un programme parallèle.

Tout d'abord, rappelons que le but est d'améliorer les performances d'un algorithme, notamment en terme de temps. Il faudra donc tout particulièrement s'attacher à produire un code optimisé et le plus rapide possible en exécution. En effet, comme nous allons le développer par la suite, un programme MPI nécessite d'utiliser plusieurs variables tampons pour préparer les données à envoyer, ou préparer la mémoire pour recevoir les données envoyées.

Il faudra également déterminer comment répartir les tâches à traiter sur l'ensemble des processeurs.

Pour compiler en C un programme développé avec MPI :

% hcc –o programme programme.c -lmpi

1. Echanges de messages entre les processus :

Dans le cas du simulateur utilisé jusqu'à maintenant en TP, la mémoire est partagée. Dans le cas de MPI, nous utilisons un parc de machines, ce qui fait que nous avons une mémoire distribuée.

Les programmes MPI étant basés sur des échanges de messages, il va falloir essayer de diminuer au maximum ce nombre d'échanges, afin de ne pas ralentir l'exécution du programme.

Pour cela, nous allons faire une sélection des données à envoyer à chaque processus, puis nous allons copier ces données dans un tableau afin de l'envoyer. De la même manière il faudra préparer un tableau pour pouvoir accueillir ces données. En effet si un processus est chargé d'effectuer des calculs sur une partie des éléments d'un tableau (ou d'une matrice), il n'a pas besoin des autres éléments ; on ne lui envoie donc que le strict nécessaire.

2. Parallélisation d'un programme / prise en main de MPI :

Dans cette partie, nous allons décrire quelques petits programmes pour se familiariser avec l'environnement MPI, et quelques fonctions très utiles.

2.1. Choix du langage :

Nous avons choisi de développer en C. Dans un premier temps, il nous avait été demandé de développer en Fortran. Le problème qui s'est posé est que nous n'avions à notre disposition qu'un compilateur Fortran77. Hors, d'après nos connaissances, le Fortran77 ne permet pas l'allocation dynamique, ce qui est indispensable pour faire des programmes efficaces. Nous nous sommes donc tourné vers le C. Ce qui, à nos yeux ne pose pas de problème, car la différence entre le C et le Fortran90 est uniquement d'ordre syntaxique, le reste étant assez similaire.

2.2. « Hello world » parallèle :

Voici un premier programme qui permet de prendre contact avec l'environnement MPI.

```
////////////////////////////////////////
//      hello.c
//      Hello world parallèle
////////////////////////////////////////

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

int main(int argc, char** argv)
{
        int myrank, size;

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

        printf("Hello world, je suis le processus %d.\n",myrank);

        MPI_Finalize();

        exit(EXIT_SUCCESS);
}
```

Après compilation : *%hcc –o hello hello.c* et exécution : *%mpirun –np 2 hello*, on obtient le résultat suivant :
Hello world, je suis le processus 0.
Hello world, je suis le processus 1.

<u>Examinons le listing :</u>

Tout d'abord les inclusions de fichiers : nous avons besoin de mpi.h qui est le seul fichier à inclure pour utiliser MPI. Puis on remarque que MPI_Init() exige qu'on lui donne les arguments de la ligne de commande, il faut donc les récupérer même si on ne s'en sert pas. On récupère le nombre de processeurs actifs sur ce programme à l'aide de MPI_Comm_size(), et notre rang qui est compris entre 0 et le nombre de processeurs –1. Puis on demande un affichage, on quitte MPI, et on renvoi un message de succès.

Chaque processeur lancé va exécuter exactement le même code, linéairement. Mais le résultat différera en fonction du rang, ici seul l'affichage change, mais ensuite on va pouvoir de la même manière affecter des tâches différentes à chaque processeur.

Après cette première prise de contact, nous allons pouvoir tester quelques fonctions très pratiques.

2.3. Utilisation de MPI_Bcast() :

Rappel : Cette fonction va permettre de diffuser un message à l'ensemble du groupe de processeurs. Voici le source du programme :

```
//////////////////////////////////////////
//	MPI_Bcast.c
//	Test de la fonction de diffusion MPI_Bcast()
//////////////////////////////////////////

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

int main(int argc, char** argv)
{
	int myrank, size, valeur;

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

	if (myrank==0) {
	  printf("Entrez la valeur a diffuser : ");
	  scanf("%d",&valeur);
	}

	MPI_Bcast(&valeur,1,MPI_INT,0,MPI_COMM_WORLD);
	printf("Moi processus %d, j'ai reçu la valeur %d\n",myrank,valeur);

	MPI_Finalize();

	exit(EXIT_SUCCESS);
}
```

Le résultat de l'exécution de ce programme sur trois processeurs est :
Entrez la valeur a diffuser : 5
Moi processus 0, j'ai reçu la valeur 5
Moi processus 2, j'ai reçu la valeur 5
Moi processus 1, j'ai reçu la valeur 5

On voit ici, que seulement le processus 0 s'occupe de la saisie de la variable.

Ce qu'il y a d'étonnant, c'est que tous les processus appellent la même fonction mais en fonction de leur rang, la variable *valeur* servira soit de variable à émettre, soit de variable de réception, c'est le quatrième argument de la fonction MPI_Bcast() qui précise qui est l'initiateur de la diffusion.

On notera également que la fonction MPI_Bcast(), comme beaucoup de fonction de communication servent à synchroniser les processus entre eux.

2.4. Utilisation de MPI_Scatterv() :

Maintenant que nous avons vu comment fonctionnait un programme MPI, nous allons pouvoir étudier une des fonctions les plus intéressantes.

Cette fonction est dérivée de la fonction MPI_Scatter(), qui sert à disperser une variable. Par exemple, elle permet de diviser un tableau en 4 parties égales et d'envoyer ces parties aux 4 processus. La fonction complémentaire associée est MPI_Gather(), qui fait l'inverse. Ces fonctions sont très pratiques en algorithmique parallèle. En effet cela permet aux processus de travailler sur une partie réduite d'une variable. Seul petit problème, c’est que la dimension de la variable n'est pas toujours proportionnelle au nombre de processus.

Par exemple, si l’on veut distribuer un tableau de 15 éléments à 4 processus, MPI_Scatter permettra seulement d'envoyer 4 tableaux de 3 éléments, et il restera 3 éléments à la charge du processus maître. On pourrait ensuite redistribuer les éléments restants un par un, mais le tableau perdrait probablement son intégrité (le processus 0 aurait les 3 premiers éléments et le troisième élément en partant de la fin). C'est là qu'intervient MPI_Scatterv(), qui permet d'envoyer des parties de taille variable pour chaque processus, elle permet également de ne pas prendre tous les éléments (de laisser des "trous" entre les parties prélevées) et d'ajuster le découpage de la variable à disperser.

Le programme suivant permet d'illustrer, en partie, l'utilité de cette fonction. Le maître alloue un tableau égal à 10*(nombre de processus), puis le distribue aux processus, en envoyant au processus i, (10-i) éléments du tableau, sans laisser de "trous".

Par exemple, si on lance le programme sur 4 processus, on obtient :
processus 0 : [0 ; 1 ; 2 ; 3 ; 4 ; 5 ; 6 ; 7 ; 8 ; 9]
processus 1: [10 ; 11 ; 12 ; 13 ; 14 ; 15 ; 16 ; 17 ; 18 ;]
processus 2 : [19 ; 20 ; 21 ; 22 ; 23 ; 24 ; 25 ; 26]
processus 3 : [27 ; 28 ; 29 ; 30 ; 31 ; 32 ; 33]

Voici le listing de ce programme :

```
/////////////////////////////////////////////
//    MPI_Scatterv.c
//  test de la fonction MPI_Scatterv
/////////////////////////////////////////////

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

#define DIMENSION 10

int main(int argc, char** argv)
{
  int myrank, i, n_procs;
  int * sendbuf;       //buffer à disperser
  int * tab_indice;    //indice de début de chaque subdivision
  int * tab_taille;    //nombre d'éléments à envoyer pour chaque processus
  int * rbuf;          //buffer de reception
  int taille;          //taille de la partie reçue

  //Initialisation de MPI
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&n_procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
```

```
  if (myrank==0) {
    //allocation mémoire
    sendbuf=(int *)malloc(n_procs*DIMENSION*sizeof(int));
    tab_indice=(int *)malloc(n_procs*sizeof(int));
    tab_taille=(int *)malloc(n_procs*sizeof(int));
    //remplissage du buffer à disperser
    for (i=0;i<n_procs*DIMENSION;i++) sendbuf[i]=i;
    //initialisation des subdivisions
    for (i=0;i<n_procs;i++){
      //nombre d'éléments à envoyer
      tab_taille[i] = DIMENSION-i;
      //indice de début du processus i = celui de i-1 + nombre d'éléments
      // envoyés par i-1
      if (i!=0) tab_indice[i]=tab_indice[i-1]+tab_taille[i-1];
      else tab_indice[i]=0;
    }
  }

  //communication de la taille de la partie reçue à chaque processus
  MPI_Scatter(tab_taille,1,MPI_INT,&taille,1,MPI_INT,0,MPI_COMM_WORLD);
  //allocation du buffer de reception
  rbuf=(int *)malloc(taille*sizeof(int));

  //dispersion
  MPI_Scatterv(sendbuf,tab_taille,tab_indice,MPI_INT,rbuf,DIMENSION,MPI_INT
,0,MPI_COMM_WORLD);

  //affichage
  printf("processus %d : [ ",myrank);
  for(i=0;i<taille;i++) printf("%d ",rbuf[i]);
  printf("]\n");

  //desallocation mémoire
  free(rbuf);
  if (myrank==0) {
    free(sendbuf);
    free(tab_indice);
    free(tab_taille);
  }

  MPI_Finalize();
  exit(EXIT_SUCCESS);
}
```

En observant la partie déclarative, on peut voir l'essentiel de ce qu'il faut retenir de l'usage de cette fonction. Il faut bien sûr une variable contenant les variables à envoyer (*sendbuf*) et une autre destinée à recevoir les données que le processus maître va envoyer. Mais aussi un tableau contenant la taille de la partie attribuée à chaque processeur, et un autre contenant l'indice de début de la zone attribuée à chaque processus dans la variable à disperser.

Il faut maintenant se pencher sur un point important : la propriété des variables. En effet, toutes les variables déclarées ne seront pas utilisées par tous les processus. Par exemple, tous les processus, sauf le processus maître (0), n'ont pas besoin de la variable *sendbuf*, puisqu'on ne désire pas envoyer l'intégralité de cette variable à tous les processus. Il en est de même pour *tab_index* et *tab_taille*. Il faudra donc, selon le rang de chaque processus, allouer ou pas une variable, si on l'alloue, le faire en utilisant les bonnes dimensions.

Une fois que le processus maître a initialisé correctement ses variables. C'est-à-dire que sa variable *sendbuf* est initialisée et prête à envoyer, et qu'il a calculé les indices et tailles des parties à envoyer pour chaque processus. Il faut maintenant qu'il communique la taille des parties envoyées à chaque processus, pour que ceux-ci allouent leur variable de réception *rbuf*, en conséquence.

Ceci étant fait, MPI_Scatterv() s'occupe du reste, et on récupère dans *rbuf*, la partie de la variable initiale *sendbuf* désirée.

2.5. Utilisation d'une boucle partagée :

Nous avons maintenant à notre disposition les outils nécessaires pour effectuer un premier calcul reparti. Nous allons utiliser dans l'exemple suivant la méthode de la boucle partagée pour calculer la somme des éléments d'un vecteur. Cette méthode consiste simplement à diviser une boucle en tranches. Voici le listing de ce programme :

```
/////////////////////////////////////////////
//    somme_partagee.c
//
//    Boucle partagee pour calculer la somme
//   des éléments d'un vecteur
/////////////////////////////////////////////

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

//nombre d'éléments du tableau
#define DIMENSION 30

int main(int argc, char** argv)
{
  int * tab;         //tableau de départ
  int * sous_tab;    //partie du tableau que possèdent les processus
  int * tab_index;   //indexation des parties à envoyer
  int * tab_tailles; //taille des parties à envoyer

  int taille;        //taille de la partie reçue
  int myrank, i, j, n_procs, resultat;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&n_procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

  if (myrank==0) {
    //allocation mémoire
    tab_index=(int *)malloc(n_procs*sizeof(int));
    tab_tailles=(int *)malloc(n_procs*sizeof(int));
    tab=(int *)malloc(DIMENSION*sizeof(int));
    //remplissage du tableau
    for (i=0;i<DIMENSION;i++) tab[i]=i+1;
    //initialisation des tailles
    for (i=0;i<n_procs;i++) {
      tab_tailles[i]=DIMENSION/n_procs;
      //On distribue aux premiers processus le reste du tableau,
```

```
      //  lorsque  sa  taille  n'est  pas  proportionnelle  au  nombre  de
processus.
      if (i < (DIMENSION%n_procs)) tab_tailles[i]++;
    }
    //calcul des index
    resultat=0; // variable temporaire
    for (i=0;i<n_procs;i++) {
      tab_index[i]=resultat;
      resultat+=tab_tailles[i];
    }
  }

  //communication de la taille de la partie reçue à chaque processus
  MPI_Scatter(tab_tailles,1,MPI_INT,&taille,1,MPI_INT,0,MPI_COMM_WORLD);
  //allocation mémoire du buffer de reception
  sous_tab=(int *)malloc(taille*sizeof(int));

  //dispersion du tableau de départ

MPI_Scatterv(tab,tab_tailles,tab_index,MPI_INT,sous_tab,taille,MPI_INT,0,MP
I_COMM_WORLD);

  //calcul (somme des éléments)
  for (i=1;i<taille;i++) sous_tab[0] += sous_tab[i];

  //le processus récolte tous les résultats en faisant la somme
  MPI_Reduce(sous_tab,&resultat,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
  if (myrank==0) printf("  somme = %d\n",resultat);

  MPI_Finalize();

  exit(EXIT_SUCCESS);
}
```

Lorsqu'on lance ce programme, le résultat fourni est :
somme = 465

Et ce, quel que soit le nombre de processeurs. Le résultat est bien celui attendu, en effet le tableau contient les entiers de 1 à 30, donc la somme fait bien 30*31/2=465. On peut remarquer l'utilisation de la fonction MPI_Reduce(), qui permet de récolter les résultats de l'ensemble des processus en appliquant une fonction de réduction (ici, une somme). En fait, ce programme ne gagne pas à être parallélisé, car le calcul au sein même de la boucle est réduit à une somme qui est une opération élémentaire, et donc les échanges de messages font perdre plus de temps que l'on en gagne en utilisant plusieurs processeurs. Pour plus de détails sur les résultats obtenus, référez-vous à la partie consacrée au « benchmarking ».

Nous savons donc maintenant utiliser la bibliothèque MPI pour réaliser un calcul en parallèle.

3. Techniques d'optimisation complémentaires :

Nous allons, dans cette partie, donner quelques moyens supplémentaires pour mieux utiliser l'environnement MPI. Nous n'avons pas testé ces méthodes, mais elles nous paraissent assez intéressantes pour figurer dans ce rapport.

3.1. Création de type de données personnalisées prêt à être utilisé par MPI :

Dans les communications MPI, les données transmises sont typées. On dispose des types prédéfinis par MPI comme ***MPI_INTEGER, MPI_DOUBLE_PRECISION, MPI_SHORT_INT, etc,***

On peut construire des structures de données plus complexes : homogènes (toutes les données sont du même type comme les éléments d'un tableau) ou hétérogènes (comme les structures en C ou les types dérivés en Fortran) pour effectuer les communications point à point.

✓ *Types homogènes :*

Ce sont des ensembles de données élémentaires de même type prédéfini mais leur distribution au sein du tableau peut être quelconque.

Définir de tels types sera plus particulièrement utile lorsqu'on veut communiquer des éléments d'une matrice. Dans les programmes que nous avons réalisés, nous utilisons des tableaux temporaires, que nous remplissons avec les valeurs à envoyer, et que nous envoyons.

On peut ainsi définir des types colonnes, lignes, ou encore des types ayant pour cible des éléments non contigus dans une matrice, par exemple des diagonales, des triangles.

✓ *Types hétérogènes :*

Ils seront eux très utiles si l'on veut communiquer des types dérivés (ou struct en C).

3.2. Optimisation des fonctions de diffusion :

La fonction fournit est MPI_Bcast(), qui permet à un processus de diffuser un message à l'ensemble des processus de son groupe. Le principe utilisé est que le processus initiateur envoie successivement à tous les autres processus son message (figure 1.).

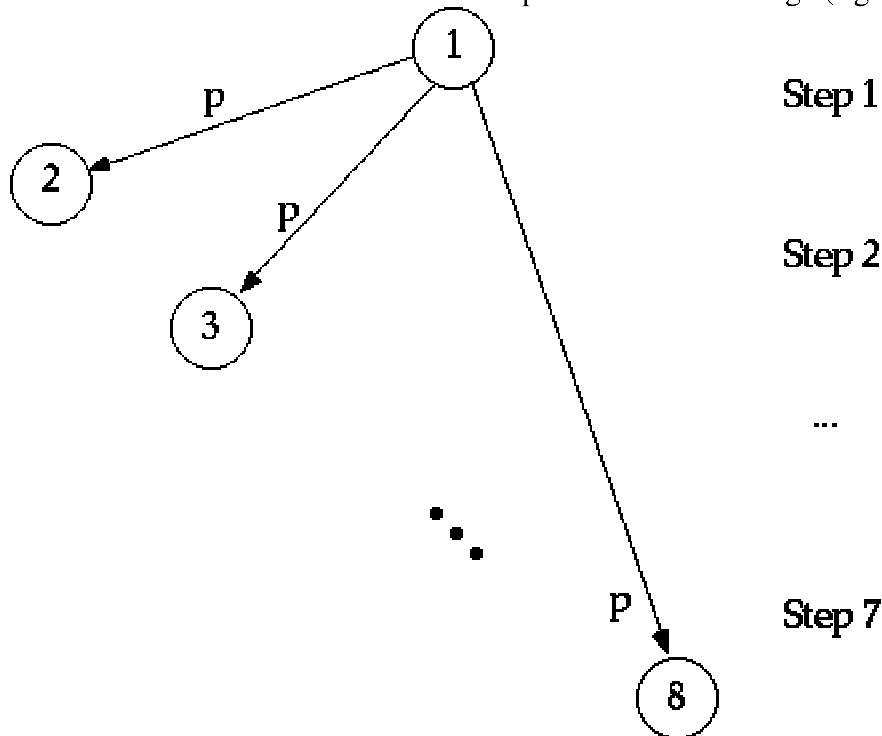


Figure 1 - principe de la fonction MPI_Bcast()

On pourrait cependant implémenter une alternative à cette fonction, en utilisant le principe suivant : la diffusion suit un schéma isomorphe à un arbre binaire, c'est-à-dire que le processus initiateur envoie son message à un autre processus, à cette étape deux processus ont maintenant le message et l'envoie maintenant à un autre processus chacun, et ainsi de suite, comme illustré sur le schéma suivant (figure 2.) :

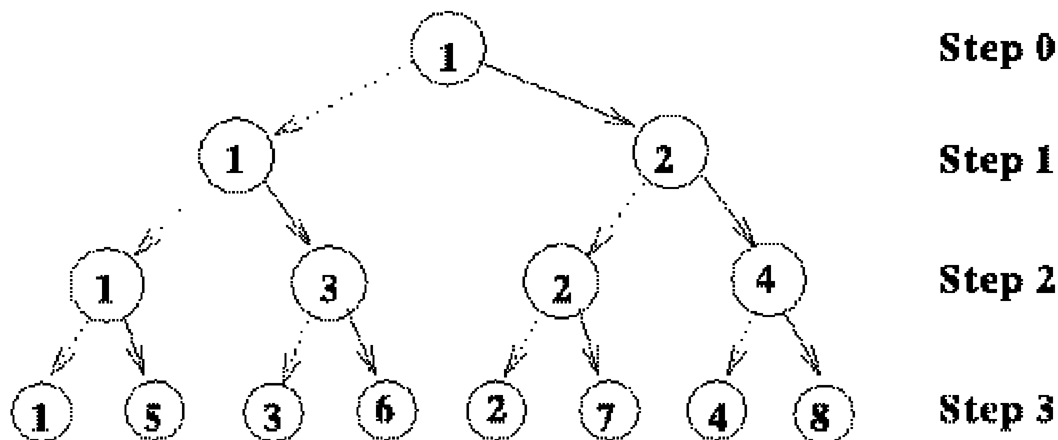


Figure 2 - Méthode plus intelligente pour la diffusion de messages

La quantité de données est la même dans les deux cas. En revanche le nombre d’étape est de l’ordre de log(n) dans la deuxième approche, contre (N-1)*p dans la seconde. (N étant le nombre de processeurs et p, la taille du message envoyé)

Cette technique peut également être appliquée aux autres fonctions utilisant des procédés similaires, comme MPI_Scatter.

3.3. Utilisation des variables d’environnement MPI :

Les variables d'environnement que l'on peut utiliser sont les suivantes :

setenv MPC_GANG OFF
setenv MPI_DSM_MUSTRUN
setenv MPI_DSM_PPM 1

- ✓ **MPI_DSM_PPM** indique le nombre de processus que l'on met par nœud (dans le cas d’utilisation de machines bi-processeur).
- ✓ **MPI_DSM_MUSTRUN** indique au système que le processus doit rester sur le même processeur pour toute.
- ✓ **MPC_GANG** permet d'activer ou désactiver le gang scheduling, c’est-à-dire le fait que tous les processus doivent tourner en même temps.

VI. Tests :

1. Test de l’environnement LAM :

Le premier test que nous avons réalisé visait à répondre à la question :
Est-ce que lorsque l’on lance plusieurs sessions LAM sur des même machines, celles-ci n’interagissent pas ?

Sur le site officiel de LAM (www.mpi-lam.org), il nous est assuré qu’une session LAM est totalement isolé des autres sessions LAM qui pourraient éventuellement tourner sur une même machine. Nous avons tout de même préféré faire un test pour nous en convaincre.

Sur cette idée, nous avons réalisé un programme qui comporte deux processus, un maître et un esclave. L'application demande un entier `i` à l'utilisateur. Puis l'application fait en boucle :

> *Le maître envoie l'entier `i` à l'esclave, l'esclave vérifie que la valeur est juste (si elle est erronée, elle affiche un message d'erreur), puis le renvoie au maître, qui vérifie à nouveau.*

Voici le code de ce programme :

```
/////////////////////////////////////////////
//      test_multi_lam.c
//
//      Utile pour tester si il y a des
//   Interférences entre deux session
//   lam qui tournent en même temps
/////////////////////////////////////////////


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

#define MSG_DATA  100
#define MSG_RESULT 101
#define MASTER 0
#define SLAVE 1

void master (int emis)
{
        int resultat;
        MPI_Status status;
        MPI_Send(&emis,1,MPI_INT,SLAVE,MSG_DATA,MPI_COMM_WORLD);
        MPI_Recv(&resultat,1,MPI_INT,SLAVE,MSG_RESULT,MPI_COMM_WORLD,&statu
s);
        if (resultat!=emis) printf("erreur : maitre : Valeur reçue
differente de la valeur envoyee!\nenvoye : %d *** recu :
%d\n",emis,resultat);
}

void slave(int emis)
{
        int resultat;
        MPI_Status status;
```

```
        MPI_Recv(&resultat,1,MPI_INT,MASTER,MSG_DATA,MPI_COMM_WORLD,&status
);

        if (resultat!=emis) printf("erreur : esclave : Valeur reçue
differente de la valeur envoyee par le maitre!\nenvoye par le maitre : %d
*** recu : %d\n",emis,resultat);

        MPI_Send(&resultat,1,MPI_INT,MASTER,MSG_RESULT,MPI_COMM_WORLD);
}

int main(int argc, char** argv)
{
        int myrank, size, emis, i;
        MPI_Status status;

        emis=0;

        MPI_Init(&argc,&argv);

        MPI_Comm_size(MPI_COMM_WORLD,&size);

        if (size!=2)
        {
                fprintf(stderr,"Error : Deux programmes doivent
tourner!\n");
                MPI_Finalize();
                exit(EXIT_FAILURE);
        }

        MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

        //saisie de la valeur à envoyer en boucle
        if (myrank== MASTER)
                { printf("Entrez la valeur a envoyer en
                boucle:");
                scanf("%d",&emis);
                MPI_Send(&emis,1,MPI_INT,SLAVE,MSG_DATA,MPI_COMM_WORLD);
        } else

        MPI_Recv(&emis,1,MPI_INT,MASTER,MSG_DATA,MPI_COMM_WORLD,&status);

        MPI_Barrier(MPI_COMM_WORLD);
        //boucle for(i=0;i<1000000;i++) {
                if (myrank==MASTER) master(emis);
                else slave(emis);
        }

        MPI_Barrier(MPI_COMM_WORLD);

        if (myrank==MASTER) printf("\n     program complete\n\n");
        MPI_Finalize();

        exit(EXIT_SUCCESS);
}
```

Dans un premier temps nous avons lancé ce programme sur deux sessions LAM ouvertes sur une même machine. Puis nous l'avons lancé sur deux sessions tournant sur deux machines (sciora et amipc7), la première session ayant été lancé depuis sciora et la seconde

sur amipc7, les deux utilisant les deux machines. Dans les deux cas, le résultat fut concluant : aucune interaction entre les deux sessions ouvertes.

2. Test des performances des programmes réalisés :

Nous avons ensuite testé le programme décrit un peu avant dans ce rapport qui consiste à sommer les éléments d'un vecteur en utilisant une boucle éclatée. Comme nous l'avons déjà expliqué, nous ne nous attendons pas à obtenir un gain pour l'exécution sur plusieurs machines.

Pour réaliser ce test, et tous ceux qui vont suivre, nous avons utilisé les machines amipc6 et amipc7. Le serveur sciora était aussi configuré pour exécuter des programmes MPI, en revanche sa puissance de calcul était beaucoup plus importante que les deux autres. Cette différence posait un problème pour les tests, car lorsque l'on effectuait un test sur sciora seul, le résultat en temps d'exécution était plutôt bon, et dès qu'l'on rajoutait une des deux machines (amipc6 ou amipc7), cela faisait chuter les performances. Nous avons retenu amipc6 et amipc7 qui ont la même configuration matérielle (Pentium 133 MHz).

En effet, l'échange de messages prend alors plus de temps que les calculs en eux-mêmes.

2.1. La boucle partagée :

Nous avons d'abord modifié le programme pour qu'il affiche les temps d'échange de messages et de calcul. Pour cela, nous avons utilisé des barrières comme vous pourrez le constater sur le listing ci-après.

Listing :

```
/////////////////////////////////////////////
//
//    somme_partagee.c
//
//    Boucle partagee pour calculer la somme
//   des éléments d'un vecteur
//
/////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

#define DIMENSION 300

int main(int argc, char** argv)
{
  double * tab;         //tableau de départ
  double * sous_tab;    //partie du tableau que possèdent les processus
  int * tab_index;   //indexation des parties à envoyer
  int * tab_tailles; //taille des parties à envoyer

  double time1,time2,timetotal,MPI_WTIME;  // mesure du temps
```

```
  int taille;        //taille de la partie reçue
  int myrank, i, j, n_procs;
  double resultat;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&n_procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

  if (myrank==0) {
    //allocation mémoire
    tab_index=(int *)malloc(n_procs*sizeof(int));
    tab_tailles=(int *)malloc(n_procs*sizeof(int));
    tab=(double *)malloc(DIMENSION*sizeof(double));
    //remplissage du tableau
    for (i=0;i<DIMENSION;i++) tab[i]=i+1;
    //initialisation des tailles
    for (i=0;i<n_procs;i++) {
      tab_tailles[i]=DIMENSION/n_procs;
      //On distribue aux premiers processus le reste du tableau,
      // lorsque sa taille n'est pas proportionnelle au nombre de
processus.
      if (i < (DIMENSION%n_procs)) tab_tailles[i]++;
    }
    //calcul des index
    resultat=0; // variable temporaire
    for (i=0;i<n_procs;i++) {
      tab_index[i]=resultat;
      resultat+=tab_tailles[i];
    }
  }

  //synchronisation pour mesurer le temps entre l'émission et la reception
  MPI_Barrier(MPI_COMM_WORLD);
  if (myrank==0) time1=MPI_Wtime();

  //communication de la taille de la partie reçue à chaque processus
  MPI_Scatter(tab_tailles,1,MPI_INT,&taille,1,MPI_INT,0,MPI_COMM_WORLD);
  //allocation mémoire du buffer de reception
  sous_tab=(double *)malloc(taille*sizeof(double));

  //dispersion du tableau de départ

MPI_Scatterv(tab,tab_tailles,tab_index,MPI_DOUBLE,sous_tab,taille,MPI_DOUBL
E,0,MPI_COMM_WORLD);

  //synchronisation pour mesurer le temps entre l'émission et la reception
  MPI_Barrier(MPI_COMM_WORLD);
  if (myrank==0) {
    time2=MPI_Wtime();
    timetotal=time2-time1;
    printf("  echange de message effectue en .... %lf sec.\n",timetotal);
  }

  //synchronisation pour mesurer le temps de calcul
  MPI_Barrier(MPI_COMM_WORLD);
  if (myrank==0)  time1=MPI_Wtime();

  //calcul (somme des éléments)
  for (i=1;i<taille;i++) {
    sous_tab[0] += sous_tab[i];
  }
```

```
  //synchronisation pour mesurer le temps de calcul
  MPI_Barrier(MPI_COMM_WORLD);
  if (myrank==0) {
    time2=MPI_Wtime();
    timetotal=time2-time1;
    printf("  calcul effectue localement en ..... %lf sec.\n",timetotal);
  }

  //synchronisation pour mesurer le temps entre l'émission et la reception
  MPI_Barrier(MPI_COMM_WORLD);
  if (myrank==0) time1=MPI_Wtime();

  //le processus récolte tous les résultats en faisant la somme
  MPI_Reduce(sous_tab,&resultat,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

  //synchronisation pour mesurer le temps entre l'émission et la reception
  MPI_Barrier(MPI_COMM_WORLD);
  if (myrank==0) {
    time2=MPI_Wtime();
    timetotal=time2-time1;
    printf("  resultat (=%lf) recolte en %lf sec.\n",resultat,timetotal);
  }

  MPI_Finalize();

  exit(EXIT_SUCCESS);
}
```

Résultat :

```
amipc7 maitrise/lam> mpirun -np 2 somme_partagee
  echange de message effectue en .... 0.001922 sec.
  calcul effectue localement en ..... 0.000563 sec.
  resultat (=45150.000000) recolte en 0.000800 sec.
```

Le tableau utilisé est de dimension 300. On voit ici aisément que le programme prend plus de temps à échanger des messages entre les processus qu'à effectuer les calculs, ce qui était prévisible étant donné les calculs demandés (somme).

2.2. CROUT distribué :

Deuxième programme aux bancs d'essais : la résolution d'un système linéaire en utilisant la méthode de CROUT. Ce programme est expliqué dans le chapitre suivant.

Voici les résultats des tests que nous avons effectués :

Nous avons testé le programme sur amipc6 et amipc7 et nous avons constaté que pour les matrices de dimension supérieure à 40 (soit 1 600 éléments), le programme plante, alors que sur sciora le programme reste stable. Nous en avons donc déduit que cela venait du matériel.

amipc6 et amipc7 :

```
amipc7 maitrise/lam> mpirun -np 2 lu
Entrez la dimension : 40
initialisation faite en : 0.001252 sec.

temps ecoule : 0.138544 sec.

amipc7 maitrise/lam> mpirun -np 2 lu
Entrez la dimension : 50
initialisation faite en : 0.001671 sec.

*MPI process rank 1 (n1, p6655) caught a SIGSEGV.
-----------------------------------------------------------------------------

One of the processes started by mpirun has exited with a
nonzero exit
code.  This typically indicates that the process finished in
error.
If your process did not finish in error, be sure to include a
"return
0" or "exit(0)" in your C code before exiting the application.

PID 6655 failed on node n1 with exit status 1.
-----------------------------------------------------------------------------
```

amipc7 :

```
amipc7 maitrise/lam> mpirun -np 1 lu
Entrez la dimension : 40
initialisation faite en : 0.000854 sec.

temps ecoule : 0.008321 sec.
```

sciora :

```
sciora maitrise/lam> mpirun -np 1 lu
Entrez la dimension : 40
initialisation faite en : 0.000293 sec.

temps ecoule : 0.003251 sec.
```

On peut remarquer que pour une matrice de dimension 40, l'exécution sur deux machines n'apporte rien en terme de rapidité, d'autant plus que sur une machine, le test est effectué avec le programme MPI, alors que nous pourrions également faire le test avec un programme 100% linéaire. Mais ce résultat nous suffit. On voit également que sciora effectue le calcul encore plus rapidement.

A partir de là, nous pouvons émettre trois hypothèses :

- La dimension n'est pas assez importante pour pouvoir apprécier le gain apporté par l'exécution réparti.
- L'algorithme de CROUT ne se prête pas à une exécution réparti : nous n'avons pas trouvé d'informations complémentaires à ce sujet, mais cela nous paraîtrait étrange.
- Notre programme n'est pas assez optimisé : il faut savoir rester modeste.

2.3. Programme réalisant un calcul massif :

Etant donné ces résultats, nous avons décidé d'imposer un calcul massif au sein de la boucle partagée, la taille de cette boucle dépendant de la taille des données à traiter. Puis nous affichons les temps de calculs respectifs pour chacun des processus.

Nous travaillons sur un tableau de 30 éléments, et chaque processus réalise une affectation en boucle. Le programme est donc le même que celui qui calcule la somme des éléments d'un vecteur, sauf que la dimension a été divisé par 10, la mesure du temps a été modifiée, et la boucle du calcul devient :

```
  //timing pour le calcul
  time1=MPI_Wtime();

  //calcul de masse
  for (i=1;i<taille;i++) {
     for (j=1;j<taille*100000;j++) resultat=taille;
     sous_tab[0] += sous_tab[i];
  }

  //mesure du temps de calcul
  //  MPI_Barrier(MPI_COMM_WORLD);
  time2=MPI_Wtime();
  timetotal=time2-time1;
  printf("     processus     %d     -     calcul     effectue     en     %lf
sec.\n",myrank,timetotal);
```

Nous avons modifié la mesure du temps car nous nous sommes rendu compte que l'exécution était toujours plus rapide sur une seule machine. Nous avons donc voulu nous intéresser au temps de calcul de chaque machine (processus). Et voici les résultats obtenus :

```
amipc7 maitrise/lam> mpirun -np 2 calcul_masse
  echange de message effectue en .... 0.000942 sec.
  processus 0 - calcul effectue en 1.080855 sec.
  processus 1 - calcul effectue en 4.725035 sec.
  resultat (=465.000000) recolte en 0.000925 sec.

amipc7 maitrise/lam> mpirun -np 1 calcul_masse
  echange de message effectue en .... 0.000016 sec.
  processus 0 - calcul effectue en 4.485034 sec.
  resultat (=465.000000) recolte en 0.000011 sec.
```

Nous avons donc été surpris de constater que la deuxième machine (amipc6) mettait beaucoup plus de temps que la première (amipc7) alors qu'elle sont supposées être identiques et que nous étions les seuls à les utiliser à ce moment. Cela vient sûrement de la gestion des variables de MPI.

En revanche, en ce qui concerne la première machine, nous avons pu constater un gain significatif lors de l'exécution sur deux machines. Nous avons voulu alors affiner ce résultat en exécutant cette même boucle mais dans un programme classique.

Voici le listing de ce programme :

```
/////////////////////////////////////////////
//
//    calcul_masse_mono.c
//
//    Réalisation d'un calcul de masse au sein d'une
//   boucle sur une seule machine.
//
/////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <sys/timeb.h>

#define DIMENSION 30

int main(int argc, char** argv)
{
  double * tab;         //tableau de départ

  struct timeb time1,time2;  // mesure du temps
  unsigned long int timetotal;

  int i, j;
  double resultat,taille=30;

  //allocation mémoire
  tab=(double *)malloc(DIMENSION*sizeof(double));
  //remplissage du tableau
  for (i=0;i<DIMENSION;i++) tab[i]=i+1;

  //timing pour le calcul
```

```
  ftime(&time1);

  //calcul de masse
  for (i=1;i<DIMENSION;i++) {
      for (j=1;j<DIMENSION*100000;j++) resultat=taille;

      tab[0] += tab[i];
  }

  //mesure du temps de calcul
  ftime(&time2);
  timetotal=(time2.time-time1.time)*1000+time2.millitm-time1.millitm;
  printf("  calcul effectue en %d millisec.\n",timetotal);

  free(tab);
}
```

Et voici le résultat en l'exécutant sur amipc7 :

```
amipc7 maitrise/lam> ./calcul_masse_mono
  calcul effectue en 5217 millisec.
```

Ce test est donc très concluant car il met en évidence le gain linéaire/reparti : 5.217 secondes contre 1.080855 secondes.

Certes ce calcul ne présente aucun intérêt du point de vue algorithmique et de l'utilité du résultat, mais il met en évidence l'utilité de l'utilisation de la bibliothèque MPI pour l'exécution d'algorithme répartis.

SUPPLEMENT

Suite à la rédaction de ce rapport, nous avons pu affiner nos tests grâce à une nouvelle machine amipc5, qui s'avère être de puissance équivalente à amipc6, si l'on en juge par ces résultats :

```
amipc7 maitrise/lam> mpirun -np 1 calcul_masse
  echange de message effectue en .... 0.000016 sec.
  processus 0 - calcul effectue en 4.485034 sec.
  resultat (=465.000000) recolte en 0.000011 sec.
[dumouni@amipc6 lam]$ mpirun -np 1 calcul_masse
  echange de message effectue en .... 0.000043 sec.
  processus 0 - calcul effectue en 16.967383 sec.
  resultat (=465.000000) recolte en 0.000030 sec.
[dumouni@amipc5 lam]$ mpirun -np 1 calcul_masse
  echange de message effectue en .... 0.000050 sec.
  processus 0 - calcul effectue en 16.980645 sec.
  resultat (=465.000000) recolte en 0.000037 sec.
```

Armés de cette nouvelle machine, nous avons alors renouvelé le test pour obtenir le résultat suivant :

Sur amipc5 et amipc6 :

```
dumouni@amipc6 lam]$ mpirun -np 2 calcul_masse
  echange de message effectue en .... 0.001599 sec.
  processus 1 - calcul effectue en 4.727118 sec.
  processus 0 - calcul effectue en 4.796282 sec.
  resultat (=465.000000) recolte en 0.002215 sec.
```

Sur amipc6 seule :

```
[dumouni@amipc6 lam]$ ./calcul_masse_mono
  calcul effectue en 6525 millisec.
```

A la suite de ce test, nous avons donc pu enfin constater un gain **sur le temps de calcul. On voit donc ainsi, que tous les algorithmes n'ont pas intérêt à être portés sur une architecture répartie (par exemple les algorithme de fouille de données qui nécessitent beaucoup de communications). En revanche les algorithmes qui nécessitent de gros calculs sur un petit ensemble de données peuvent y trouver leur intérêt, on retrouve par exemple dans cette catégorie les algorithmes de rendu d'images à partir de modèles tridimensionnels.**

V. CROUT avec MPI

1. Introduction :

Les TP d'architecture parallèle se font actuellement à l'aide d'un simulateur qui ne permet que de tester la correction d'algorithmes parallèles et non l'intérêt du parallélisme et les conditions proches d'un cas réel.

C'est pour cela que nous sommes chargés d'installer MPI sur des machines de l'université et de pouvoir expliquer les méthodes à suivre pour son bon fonctionnement.

L'une des applications majeures des TP portant sur les machines parallèles est le programme utilisant la méthode de CROUT, bien connu maintenant des maîtrises d'informatique de l'Université Blaise Pascal.

Nous avons donc adapté le programme et testé celui-ci afin de vérifier son bon fonctionnement et de faire des études sur le gain de temps de la parallélisation sur un cas concret.

2. Quelques rappels sur CROUT…

Commençons par rappeler ce qu'est la méthode de CROUT.

Afin de résoudre un système du type AX=B avec A, X et B des matrices carrées de même taille n, X étant la matrice inconnue dont on cherche les composantes, on remarque que la résolution est beaucoup plus simple si l'on peut décomposer la matrice A en produit de deux matrices triangulaires, L (triangulaire inférieure (Lower)) et U (triangulaire supérieure (Upper)).

L'algorithme de CROUT consiste en cette décomposition.

Il est ensuite suivi de deux sous programmes qui résolvent à eux deux LUX=B.

L'algorithme de CROUT repose sur des formules mathématiques adaptées à l'informatique qui permettent dans la majorité des cas le passage de A en LU.

Voici ces formules :

$$L_{ik} = A_{ik} - \sum_{j=1}^{k-1} L_{ij} * U_{jk}$$

Pour i allant de k à n

$$U_{kj} = \frac{A_{kj} - \sum_{i=1}^{k-1} L_{ki} * U_{ij}}{L_{kk}}$$

Pour j allant de k+1 à n

$$U_{ii} = 1$$

Pour tout 1<=i<=n

Notons au passage que nous donnons les formules « type fortran » c'est-à-dire avec des indices commençant à 1 et terminant à n, et non de 0 à n-1 comme nous l'avons fait en C, ceci pour plus de clarté. Cela dit ce changement d'indices pose quelques problèmes lors de boucles imbriquées, puisqu'il ne faut pas retrancher un si le point de départ de la boucle en tien déjà compte ; par exemple, la boucle :

Pour i allant de 1 à n faire
    Pour j allant de i à n faire
        Calcule de L_{ij}

Deviendra

Pour i allant de 0 à n-1 faire
    Pour j allant de i à n-1 faire
        Calcule de L_{ij}

La méthode utilisée consiste à calculer alternativement une colonne de la matrice L et une ligne de U, puisque dans cet ordre on peut calculer tous les indices de L et de U en ayant toujours les coefficients nécessaires.

Ensuite on résout le système obtenu grâce à deux sous programmes simples nommés « descente » et « remontée ». Pour ces phases de résolution nous passons par une étape LY=B, ou Y est l'inconnue, puis nous résolvons UX=Y, ce qui revient bien au même avec la résolution de AX=B.

Nous avons utilisé les formules suivantes :

« descente »

$$Y_{ij} = \frac{B_{ij} - \sum_{k=1}^{i-1} L_{ik} * Y_{kj}}{L_{ii}}$$

Pour i allant de 1 à n et j allant de 1 à n

« remontée »

$$X_{ij} = Y_{ij} - \sum_{k=i+1}^{n} U_{ik} * X_{kj}$$

Pour i allant de n à 1 par pas –1
Et pour j allant de 1 à n

Remarquons que l'ordre « pour i allant de… pour j allant de… » est équivalant si on inverse l'imbrication des boucles, c'est-à-dire en faisant « pour j allant de… pour i allant de… », ce que nous ferons pour pouvoir travailler par groupe de colonnes.

3. CROUT distribué avec MPI

3.1. Démarche de parallélisation :

Nous avions le choix entre deux méthodes :

- décomposer les sommes rencontrées lors de chaque calcul des composantes de L et de U, comme on le fait pour le calcul de la somme des éléments d'un tableau par une boucle partagée (cette méthode consiste à envoyer à chaque processus un bout du tableau et celui-ci en renvoie la somme, le processus père n'ayant ensuite plus qu'à sommer les résultats reçus).
- Faire calculer par chaque processus une partie des composantes de L lors du calcul de chaque colonne de L et de la même manière partager les composantes de la ligne de U à calculer pour chaque ligne de U.

Nous avons bien entendu opté pour la deuxième possibilité qui permet aux processus de travailler sur de plus vastes données et d'être ainsi plus actifs.

Il faut tout d'abord que le processus père détermine en fonction du nombre de processus et de la taille de la matrice les affectations des parties à effectuer par chaque processus, ainsi que des données nécessaires qu'il doit leur envoyer.

Bien entendu ces affectations varieront et le processus père refait ces opérations à chaque nouvelle colonne et ligne.

Une fois les données reçues, chaque processus fils ainsi que le père lui-même se charge de calculer la partie qui lui est affectée, puis renvoie les résultats au père.

Le père stocke les données reçues, puis reprends à l'étape suivante.

Afin de calculer la partie qui lui est attribuée, un processus doit avoir à chaque étape un bloc des données de L et de U.

Soit le processus père pouvait lui envoyer lors de chaque calcul, soit le père envoyait à tous ses fils les colonnes de L et les lignes de U nouvellement établies.

Nous avons opté pour la deuxième possibilité car les deux ont le même coût en terme d'envoie de message, et les processus fils auront besoins par la suite (lors de la résolution) des matrices L et U.

Afin que le stockage de ces matrices chez les fils soit restreint, nous avons stocké les valeurs de L et de U dans une même matrice *LUcompact*, les valeurs de la diagonale de U étant toujours 1, il n'est pas utile de les stocker.

Pour la phase de résolution, nous avons vu juste à la suite des formules, que nous pouvons faire les calculs par colonnes et non par lignes.

Chaque processus peut donc se permettre de calculer un ensemble de colonnes de Y, puis de manière similaire un ensemble de colonnes de X.

Nous avons pu nous rendre compte lors de l'élaboration de ce programme qu'il s'agit d'une importante part de travail qui à notre jugement pourrait être lourd à faire si on y consacre moins de 3 TP. Nous pensons aussi que cela conviendrait sûrement plus dans le cadre d'un projet en architecture parallèle, mais dispenserait alors les TP de ceux-ci….

Notons par ailleurs que notre programme, largement commenté, permet de bien comprendre chacune de ses étapes.

Vous trouverez ci-après le listing du programme réalisant la décomposition LU par la méthode de CROUT et la résolution en utilisant MPI, et le second qui utilisait le simulateur. On remarque aisément que le premier demande beaucoup plus de travail du fait que l'on n'a pas de mémoire partagée mais distribuée, et qu'il faut minimiser la quantité d'information échangée.

3.2. Listing 1 (CROUT avec MPI) :

```
    ////////////////////////////////////////////////
    //                                            //
    //      crout.c                               //
    //                                            //
    //      Résolution d'un système linéaire      //
    //       en utilisant la méthode de CROUT     //
    ////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

#define EPS 10E-6
//#define DEBUG
//#define DEBUG_L
//#define DEBUG_U
#define K_TEST 2
//#define AFF_MAT
#define RESULTAT


////////////////////////////////////////////
// fonction de résolution du systeme LY=B
////////////////////////////////////////////
void descente(float ** L,float ** Y,float ** B,int N, int n_procs){
        int I,J,K;
        float S;
        float ** Buff;
        int tab_tailles[n_procs],index[n_procs];// pour le père
        int dep, tail; // pour chaque processus
// Buff sert de buffer pour un processus fils pour recevoir les colonnes de B qui lui sont
nécessaires
        // et revoyer apres calcul les colonnes de Y au père
// chaque processus a une partie des colonnes de Y à calculer :
        if (myrank == 0) {
                // calcul des tailles :
                for (i=0;i<N%n_procs;i++) tab_tailles(i)=N/n_procs+1;
                for (i=N%n_procs+1;i<N;i++) tab_taille(i)=N/n_procs;
                // calcul de l'index :
                index(0)=0;
                for (i=1;i<N;i++) index(i)=index(i-1)+tab_taille(i);
        }
        //MPI_le pere envoie à ses fils(taille(i)-> ils le mettent dans tail)...
         MPI_Scatter(tab_taille,1,MPI_INT,tail,1,MPI_INT,0,MPI_COMM_WORLD);

        //MPI_le pere envoie à ses fils(index(i) -> ils le mettent dans dep)...
         MPI_Scatter(index,1,MPI_INT,dep,1,MPI_INT,0,MPI_COMM_WORLD);

        // allocation des buffers pour la réception :
        Buff=(float**) malloc (tail*sizeof(float*));
        for (i=0;i<tail;i++) Buff[i]=(float*) malloc (tail*sizeof(float));

        // le pere distribue aux processus les colonnes de B dont
        // ils auront besoins (reçus dans le buffer "Buff") :
        //  MPI_j'envoie à mes fils(pour le processus i : colonnes de B de index(i) à
index(i)+taille(i))...
         //  pour ttes les lignes :
        for(i=0;i<n;i++)
MPI_Scatterv(B[i],tab_tailles,index,MPI_DOUBLE,Buff[i],tail,MPI_DOUBLE,0,MPI_COMM_WORLD);

        // chacun traite sa partie
```

```
        for (J=dep;dep+tail-1;J++)
                for (I=0;N-1;I++){
                  S=0;
                  for (K=0;I-1;K++) S=S+L(I,K)*Y(K,J);
                  Buff(I,J) /* càd Y(I,J) */=(Buff(I,J)/* càd B(I,J)*/ -S)/L(I,I);
                }

        // envoie du buffer résultant au père qui met le tout dans Y
         //  pour ttes les lignes :
        for(i=0;i<n;i++)
MPI_Gather(Y[i],tab_tailles,index,MPI_DOUBLE,Buff[i],tail,MPI_DOUBLE,0,MPI_COMM_WORLD);

        //libération de la mémoire :
        for (i=0;i<taill;i++) free(Buff(i));
        free(Buff);
}


///////////////////////////////////////////
// fonction de résolution du systeme UX=Y
///////////////////////////////////////////
void remontee(float ** U,float ** X,float ** Y, int N, int n_procs){
        int I,J,K;
        float S;
        float ** Buff;
        int tab_tailles[n_procs],index[n_procs];// pour le père
        int dep, tail; // pour chaque processus
// Buff sert de buffer pour un processus fils pour recevoir les colonnes de Y qui lui sont
nécessaires
        // et revoyer apres calcul les colonnes de X au père
// chaque processus a une partie des colonnes de X à calculer :
        if (myrank == 0) {
                // calcul des tailles :
                for (i=0;i<N%n_procs;i++) tab_tailles(i)=N/n_procs+1;
                for (i=N%n_procs+1;i<N;i++) tab_taille(i)=N/n_procs;
                // calcul de l'index :
                index(0)=0;
                for (i=1;i<N;i++) index(i)=index(i-1)+tab_taille(i);
        }
        //MPI_le pere envoie à ses fils(taille(i)-> ils le mettent dans tail)...
         MPI_Scatter(tab_taille,1,MPI_INT,tail,1,MPI_INT,0,MPI_COMM_WORLD);

        //MPI_le pere envoie à ses fils(index(i) -> ils le mettent dans dep)...
         MPI_Scatter(index,1,MPI_INT,dep,1,MPI_INT,0,MPI_COMM_WORLD);

        // allocation des buffers pour la réception :
        Buff=(float**) malloc (tail*sizeof(float*));
        for (i=0;i<tail;i++) Buff[i]=(float*) malloc (tail*sizeof(float));

        // le pere distribue aux processus les colonnes de Y dont
        // ils auront besoins (reçus dans le buffer "Buff") :
        //  MPI_j'envoie à mes fils(pour le processus i : colonnes de Y de index(i) à
index(i)+taille(i))...
         //  pour ttes les lignes :
        for(i=0;i<n;i++)
MPI_Scatterv(Y[i],tab_tailles,index,MPI_DOUBLE,Buff[i],tail,MPI_DOUBLE,0,MPI_COMM_WORLD);

        // chacun traite sa partie
        for (I=N-1;0;I--)
                for (J=0;N-1;J++){
                        S=0;
                        for (K=I+1;N-1;K++) S=S+U(I,K)*X(K,J);
                        X(I,J)=(Y(I,J)-S);
                }

        // envoie du buffer résultant au père qui met le tout dans X
         //  pour ttes les lignes :
        for(i=0;i<n;i++)
MPI_Gather(X[i],tab_tailles,index,MPI_DOUBLE,Buff[i],tail,MPI_DOUBLE,0,MPI_COMM_WORLD);

        //libération de la mémoire :
        for (i=0;i<taill;i++) free(Buff(i));
        free(Buff);

}

///////////////////////////////////////////////////
```

```
// MAIN
/////////////////////////////////////////////////////

int main(int argc, char** argv)
{

  double time1,time2,timetotal,MPI_WTIME;  // mesure du temps
  // EXTERN MPI_WTIME;

  double ** A,**L,**U;        // matrices pour la décomposition
  double **B,**Y,**X;         // matrices
  double ** LUcompact;        // sert à stocker les valeurs de L et U déjà calculées
  double * Aik_f;             // valeurs reçues par les processus fils pour les calculs de L
  double * Akj_f;             // -------------------------------------------------------- U
  double * Aik_p;             // valeurs à envoyer par le pere aux processus fils pour les
calculs de L
  double * Akj_p;             // -------------------------------------------------------------
------- U
  double * NewColL;           // buffer d'émission pour mettre à jour les copies locales des
valeurs de L déjà calculées
  double * NewRowU;         // buffer d'émission pour mettre à jour les copies locales des
valeurs de U déjà calculées
  double S;                   // variable temporaire pour faire une somme ...
  int * index,index_i;        // indexation des parties à envoyer, et index de sa partie à soi
même
  int * tab_tailles,taille;   // taille des parties à envoyer (pour le processus père), et
taille de sa partie à soi même
  int n;                        // n est la dimension de la matrice A
  int n_procs, myrank,i,j,k,erreur;  //

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&n_procs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);


  if (myrank==0) { // c'est le pere
    printf("Entrez la dimension : ");
    scanf("%d", &n);
    // Timing
    time1=MPI_Wtime();
  }

  //partage de la dimension
  MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
  //////////////////////
  // initialisation du pere et des fils
  //////////////////////
  if (myrank==0) { // c'est le pere
    // allocation des matrices :
    A = (double **) malloc (n*sizeof(double*));
    B = (double **) malloc (n*sizeof(double*));
    Y = (double **) malloc (n*sizeof(double*));
    X = (double **) malloc (n*sizeof(double*));
    U = (double **) malloc (n*sizeof(double*));
    L = (double **) malloc (n*sizeof(double*));
    for (i=0;i<n;i++) {
      A[i]= (double*) malloc (n*sizeof(double));
      B[i]= (double*) malloc (n*sizeof(double));
      Y[i]= (double*) malloc (n*sizeof(double));
      X[i]= (double*) malloc (n*sizeof(double));
      U[i]= (double*) malloc (n*sizeof(double));
      L[i]= (double*) malloc (n*sizeof(double));
    }
    // initialisation des matrices :
    for (i=0;i<n;i++){
      for (j=0;j<n;j++) {
       A[i][j]=i+j+2;
       B[i][j]=i+j+2;
      }
      for (j=i+1;j<n;j++) U[i][j]=0;
      U[i][i]=1;
      for (j=0;j<i;j++) L[i][j]=0;
    }
#ifdef AFF_MAT
    printf("0 - matrices initialisees\n");
    for (i=0;i<n;i++) {
      for (j=0;j<n;j++) printf(" %lf *",A[i][j]);
```

```
      printf("\n");
    }
#endif
    // calcul des indexs et longueurs de tranches
    index=(int*)malloc(n_procs*sizeof(int));
    tab_tailles=(int *)malloc(n_procs*sizeof(int));
    // initialisation des tailles
    for (i=0;i<n_procs;i++) {
      tab_tailles[i]=(n-1)/n_procs;
      // On distribue aux premiers processus le reste du calcul de la somme,
      // lorsque sa taille n'est pas proportionnelle au nombre de processus.
      if (i < ((n-1)%n_procs)) tab_tailles[i]++;
    }
    //calcul des index
    index[0]=0;
    for (i=1;i<n_procs;i++) index[i]=index[i-1]+tab_tailles[i];
  }

  // on alloue LUcompact, qui permettra de stocker les valeurs de L et U déjà calculées
  //  (un peu comme la méthode de CROUT compact)
  LUcompact = (double **) malloc (n*sizeof(double*));
  for (i=0;i<n;i++) LUcompact[i]= (double*) malloc (n*sizeof(double));

  MPI_Barrier(MPI_COMM_WORLD);
  //distribution des tailles et des index, pour que chaque processus connaissent la taille et
l'indice de début de sa partie
  MPI_Scatter(tab_tailles,1,MPI_INT,&taille,1,MPI_INT,0,MPI_COMM_WORLD);
  MPI_Scatter(index,1,MPI_INT,&index_i,1,MPI_INT,0,MPI_COMM_WORLD);

  //timing
  if (myrank==0) {
    time2=MPI_Wtime();
    timetotal=time2-time1;
    printf("initialisation faite en : %lf sec.\n\n",timetotal);
    time1=MPI_Wtime();
  }

  //test de solvabilité
  if (myrank==0) erreur = (fabs(A[0][0])<EPS);
  MPI_Bcast(&erreur,1,MPI_INT,0,MPI_COMM_WORLD);
  if (erreur) {
    if (myrank==0) printf("La methode de crout ne permet pas de decomposer cette matrice\n");
  }
  // tout le monde fait ce qui suit si il n'y a pas eu d'erreur
  else {
    if (myrank==0) {
      // initialisation de la premiere ligne de U et de la premiere colonne de L :
      L[0][0]=A[0][0];
      for (j=1;j<n;j++) {
       U[0][j]=A[0][j]/A[0][0];
       L[j][0]=A[j][0];
      }
    }
    //////////////////////
    // Communication de ces valeurs de U et L calculées à tout le monde :
    //////////////////////
    // allocation de NewRowU, prêt à être diffusé ou à recevoir : la dernière ligne de U
calculée
    // allocation de NewColL, prêt à être diffusé ou à recevoir : la dernière colonne de L
calculée
    NewRowU=(double*)malloc((n-1)*sizeof(double));
    NewColL=(double*)malloc((n)*sizeof(double));
    //remplissage
    if (myrank==0) {
      for (i=0;i<n-1;i++) NewRowU[i]=U[0][i+1];
      for (i=0;i<n;i++) NewColL[i]=L[i][0];
    }
    //on diffuse maintenant NewRowU et NewColL
    MPI_Bcast(NewRowU,n-1,MPI_DOUBLE,0,MPI_COMM_WORLD);
    MPI_Bcast(NewColL,n,MPI_DOUBLE,0,MPI_COMM_WORLD);
    //maintenant chaque processus met à jour ses matrices locales L et U qui sont concentrées
dans LUcompact
    for (i=0;i<n-1;i++) LUcompact[0][i+1]=NewRowU[i];
    for (i=0;i<n;i++) LUcompact[i][0]=NewColL[i];
    //on desalloue
    free(NewRowU);
    free(NewColL);
```

```
    //////////////////////
    // boucle sur k
    //////////////////////
    for (k=1;k<n-1;k++){
#ifdef DEBUG
      if (k==K_TEST) {
       if (myrank==0) {
         printf("%d - les tailles : ",myrank);
         for (i=0;i<n_procs;i++) printf("%d ",tab_tailles[i]);
         printf("\n");
       }
       printf("%d - mon index = %d\n",myrank,index_i);
      }
#endif
      //////////////////////
      // kième colonne de L
      //////////////////////
      // allocation des buffers de réception : Aik_f :
      Aik_f=(double*) malloc (taille*sizeof(double));
      // le père prépare son buffer d'émission : Aik_p : les données de A à distribuer aux
processus selon leur besoin
      if (myrank==0) {
       Aik_p=(double*)malloc((n-k)*sizeof(double));
       for (i=0;i<n-k;i++) Aik_p[i]=A[k+i][k];
      }
      // on dispatche Aik_p à tous les processus, Aik_p étant un tableau de taille n-k :

MPI_Scatterv(Aik_p,tab_tailles,index,MPI_DOUBLE,Aik_f,taille,MPI_DOUBLE,0,MPI_COMM_WORLD);
#ifdef DEBUG_L
      // on attend que tous les processus aient reçu les données pour avoir un affichage plus
joli
      MPI_Barrier(MPI_COMM_WORLD);
      if (k==K_TEST) {
       printf("%d - Aik_f : ",myrank);
       for (i=0;i<taille;i++) printf(" %lf * ",Aik_f[i]);
       printf("\n");
      }
#endif
      if (myrank==0) free(Aik_p);
      //chaque processus a reçu dans Aik_f les données de A dont il aura besoin
#ifdef DEBUG_L
      if (k==K_TEST) {
       for (i=0;i<n;i++) {
         printf("%d - LUcompact =",myrank);
         for (j=0;j<n;j++) printf(" %lf *",LUcompact[i][j]);
         printf("\n");
       }
      }
#endif
      // calcul de Lik
      for (i=index_i;i<index_i+taille;i++) {
       S=LUcompact[i+k][0]*LUcompact[0][k];
#ifdef DEBUG_L
       if (k==K_TEST) {
         printf("%d - L - i=%d\n",myrank,i);
       }
#endif
       for (j=1;j<k;j++) S+=LUcompact[i+k][j]*LUcompact[j][k];
       Aik_f[i-index_i]=Aik_f[i-index_i]-S;
#ifdef DEBUG_L
       if (k==K_TEST) {
         printf("%d - L - i=%d - S=LUcompact[%d][0]*LUcompact[0][%d]\n",myrank,i,i+k,k);
         printf("%d - L - i=%d - S=%lf\n",myrank,i,S);
       }
#endif
      }
      // rassemblement des résultats dans L du pere :
      if (myrank==0) NewColL=(double*)malloc((n-k)*sizeof(double));

MPI_Gatherv(Aik_f,taille,MPI_DOUBLE,NewColL,tab_tailles,index,MPI_DOUBLE,0,MPI_COMM_WORLD);
      // on n'a plus besoin des Aik_f :
      free(Aik_f);
      // le pere qui a reçu toutes les nouvelles valeurs de NewColL vide ce buffer dans L
      if (myrank==0) {
       for (i=0;i<n-k;i++) L[k+i][k]=NewColL[i];
       free(NewColL);
```

```
      }
#ifdef DEBUG_L
      if (k==K_TEST) {
       if (myrank==0) {
         for (i=0;i<n;i++) {
           printf("%d - L =",myrank);
           for (j=0;j<n;j++) printf(" %lf *",L[i][j]);
           printf("\n");
         }
       }
      }
#endif
      //////////////////////
      // Communication des dernières valeurs de L calculées
      //////////////////////
      // allocation de NewColL, prêt à être diffusé ou à recevoir : la dernière colonne de L
calculée
      NewColL=(double*)malloc((n-k)*sizeof(double));
      // remplissage
      if (myrank==0) for (i=0;i<n-k;i++) NewColL[i]=L[k+i][k];
      // on diffuse maintenant NewColL
      MPI_Bcast(NewColL,n-k,MPI_DOUBLE,0,MPI_COMM_WORLD);
      // maintenant chaque processus met à jour sa matrice locales L dans LUcompact
      for (i=0;i<n-k;i++) LUcompact[k+i][k]=NewColL[i];
      // on desalloue
      free(NewColL);

      //////////////////////
      // kième ligne de U
      //////////////////////
      // allocation des buffers de réception : Akj_f :
      Akj_f=(double*) malloc (taille*sizeof(double));
      // le père prépare son buffer d'émission : Akj_p : les données de A à distribuer aux
processus selon leur besoin
      if (myrank==0) {
       Akj_p=(double*)malloc((n-k)*sizeof(double));
       for (j=0;j<n-k;j++) Akj_p[j]=A[k][j+k];
      }
      // on dispatche Akj_p à tous les processus, Akj_p étant un tableau de taille n-k :

MPI_Scatterv(Akj_p,tab_tailles,index,MPI_DOUBLE,Akj_f,taille,MPI_DOUBLE,0,MPI_COMM_WORLD);
#ifdef DEBUG_U
      // on attend que tous les processus aient reçu les données pour avoir un affichage plus
joli
      MPI_Barrier(MPI_COMM_WORLD);
      if (k==K_TEST) {
       printf("%d - Akj_f : ",myrank);
       for (i=0;i<taille;i++) printf(" %lf * ",Akj_f[i]);
       printf("\n");
       for (i=0;i<n;i++) {
         printf("%d - LUcompact =",myrank);
         for (j=0;j<n;j++) printf(" %lf *",LUcompact[i][j]);
         printf("\n");
       }
      }
#endif
      if (myrank==0) free(Akj_p);
      //chaque processus a reçu dans Akj_f les données de A dont il aura besoin
      // calcul de Ukj :
      // pour U, on ne doit pas calculer l'élément diagonal :
      if (index_i==0) {
       for (j=index_i+1;j<index_i+taille;j++) {
         S=LUcompact[k][0]*LUcompact[0][j+k];
         for (i=1;i<k;i++) S+=LUcompact[k][i]*LUcompact[i][j+k];
         Akj_f[j-index_i]=Akj_f[j-index_i]-S;
#ifdef DEBUG_U
         if (k==K_TEST) {
           printf("%d - j=%d - S=LUcompact[%d][0]*LUcompact[0][%d]\n",myrank,j,k,j+k);
           printf("%d - j=%d - S=%lf\n",myrank,i,S);
         }
#endif
       }
      }
      // pour les tranches suivantes, tous les éléments sont à calculer
      else {
       for (j=index_i;j<index_i+taille;j++) {
         S=LUcompact[k][0]*LUcompact[0][j+k];
```

```
         for (i=1;i<k;i++) S+=LUcompact[k][i]*LUcompact[i][j+k];
         Akj_f[j-index_i]=(Akj_f[j-index_i]-S)/LUcompact[k][k];
#ifdef DEBUG_U
         if (k==K_TEST) {
           printf("%d - j=%d - S=LUcompact[%d][0]*LUcompact[0][%d]\n",myrank,j,k,j+k);
           printf("%d - %d - S=%lf\n",myrank,i,S);
         }
#endif
       }
      }
      // rassemblement des résultats dans U du pere :
      if (myrank==0) NewRowU=(double*)malloc((n-k)*sizeof(double));

MPI_Gatherv(Akj_f,taille,MPI_DOUBLE,NewRowU,tab_tailles,index,MPI_DOUBLE,0,MPI_COMM_WORLD);
      // on n'a plus besoin des Akj_f :
      free(Akj_f);
      // le père qui a reçu toutes les nouvelles valeurs de NewColL vide ce buffer dans L
(sauf le premier élément qui correspond à l'élément diagonal, qui n'est pas calculé pour U)
      if (myrank==0) {
       for (j=1;j<n-k-1;j++) U[k][k+j+1]=NewRowU[j];
       free(NewRowU);
      }

      //////////////////////
      // Communication des dernières valeurs de U calculées
      //////////////////////
      // allocation de NewRowU, prêt à être diffusé ou à recevoir : la dernière colonne de U
calculée
      NewRowU=(double*)malloc((n-k-1)*sizeof(double));
      // remplissage
      if (myrank==0) for (i=0;i<n-k-1;i++) NewRowU[i]=U[k][k+i+1];
      // on diffuse maintenant NewRowU
      MPI_Bcast(NewRowU,n-k-1,MPI_DOUBLE,0,MPI_COMM_WORLD);
      // maintenant chaque processus met à jour sa matrice locales U dans LUcompact
      for (i=0;i<n-k+1;i++) LUcompact[k][k+i+1]=NewRowU[i];
      // on desalloue
      free(NewRowU);

      //////////////////////
      // on met à jour tab_tailles et l'index :
      //////////////////////
      if (myrank==0) {
       //  tab_tranche : on décrémente de 1 l'élément correspondant
       //  au processus numéro "reste de la division entière de n par n_procs -1" dans
tab_tailles
       //  en vérifiant que ce reste soit positif, pour rester dans les bornes du tableau
(d'où le "+n_procs)%n_procs")
       tab_tailles[((n-k)%n_procs-1+n_procs)%n_procs]--;
#ifdef DEBUG
       printf("0 - k=%d - formule = %d\n",k,((n-k)%n_procs-1+n_procs)%n_procs);
#endif
       // index :
       // index[0]=0; ça y est déjà
       for (i=1;i<n_procs;i++) index[i]=index[i-1]+tab_tailles[i];
      }
      // le processus met à jour la valeur locale de sa propre taille et de son indice de
début
      MPI_Scatter(tab_tailles,1,MPI_INT,&taille,1,MPI_INT,0,MPI_COMM_WORLD);
      MPI_Scatter(index,1,MPI_INT,&index_i,1,MPI_INT,0,MPI_COMM_WORLD);
#ifdef DEBUG
      if (k==K_TEST) {
       printf("%d - fini\n",myrank);
      }
#endif
    } // fin de la boucle sur k

    // calcul de la derniere valeur de L (qui est L[n][n]):
    if (myrank==0) {
      S=0;
      for (j=0;j<n-1;j++) S+=A[n-1][j]*A[j][n-1];
      L[n-1][n-1]-=S;
    }

    MPI_Barrier(MPI_COMM_WORLD);
#ifdef RESULTAT
    //////////////////
    // Résolution linéaire (non parallèle)
```

```
    //////////////////
        descente(LUcompact /* qui symbolise ici L */ ,Y,B,n,n_procs);
        MPI_Barrier(MPI_COMM_WORLD);
        remontee(LUcompact /* qui symbolise ici U */ ,X,Y,n,n_procs);
        //puis afficher le resultat
        for (i=0;i<n;i++){
          printf("X = ");
          for (j=0;j<n;j++) printf(" %lf *",A[i][j]);
          printf("\n");
        }
#endif

  } // fin du "if (fabs(A[0][0])<EPS)"

  //désallocation mémoire
  if (myrank==0) {
#ifdef RESULTAT
    for (i=0;i<n;i++) {
      printf("%d - L =",myrank);
      for (j=0;j<n;j++) printf(" %lf *",L[i][j]);
      printf("\n");
    }
    printf("\n");
    for (i=0;i<n;i++) {
      printf("%d - U =",myrank);
      for (j=0;j<n;j++) printf(" %lf *",U[i][j]);
      printf("\n");
    }
#endif
    for (i=0;i<n;i++) free(A[i]);
    free(A);
    for (i=0;i<n;i++) free(B[i]);
    free(B);
    for (i=0;i<n;i++) free(Y[i]);
    free(Y);
    for (i=0;i<n;i++) free(X[i]);
    free(X);
    for (i=0;i<n;i++) free(U[i]);
    free(U);
    for (i=0;i<n;i++) free(L[i]);
    free(L);
    free(index);
    free(tab_tailles);
  }

  // timing
  if (myrank==0) {
    time2=MPI_Wtime();
    timetotal=time2-time1;
    printf("temps ecoule : %lf sec.\n\n",timetotal);

  }
#ifdef DEBUG
  printf("%d - program complete\n",myrank);
#endif
  MPI_Finalize();

  exit(EXIT_SUCCESS);
}
```

3.3. Listing 2 (CROUT avec le simulateur) :

```
subroutine croutpara(n,nProc,ok,bar,A,B,X,L,U,xtmp,tmp3)

  !parametres
  integer,intent(in)::n,nProc
  real,intent(inout),dimension(n)::tmp3
  real,intent(inout),dimension(n,n)::A,L,U
  real,intent(inout),dimension(n)::B,X,xtmp
  integer,intent(inout),dimension(4)::bar
  logical,intent(inout)::ok

  !variables
  INTEGER i,j,k,r,id,process_fork
  REAL,PARAMETER::eps=1E-9
  real::tmp1,tmp2,tmp4

  L=0
  U=0

  !saisie de la matrice A, du vecteur B, du nombre de processus
  DO i=1,N
     WRITE(*,'(A,I2,A)') 'Elements de la ligne', i, ' de A ?'
     READ(*,*) A(I,:)
     WRITE(*,'(A,I2,A)') 'Element ', i, ' de B ?'
     READ(*,*) B(i)
  END DO

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  write(*,*) "affichage de la matrice A"
  do i=1,n
   write(*,2) A(i,:)
2  format(20f10.6)
  end do
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  l=0
  u=0

  call barrier_init(bar,nProc)

  ! décomposition
  ok=.true.

  id=process_fork(nProc)

  do i=1+id,n,nProc  !on met des 1 dans la diagonale de U
     U(i,i)=1
  end do

  do r=1,n

     call barrier(bar,nProc) !tous les processus font les boucles en r
     !ils se partagent le travail interne
     !les barrieres permettent de les garder tous dans
     !                            la meme boucle en r

     if (.NOT. ok) exit    !on le place après cette barrière et au début car
     !nous sommes sur que tous les processus
     !s'arreterons sans effectuer une boucle de plus
     !si une division par zero intervient

     do i=r+id,n,nProc       !chaque processus calcule une composante de L
        !           dans la colonne r sous la ligne r
        tmp1=0
        do k=1,r-1
           tmp1=tmp1+L(i,k)*U(k,r)
        end do
        L(i,r)=A(i,r)-tmp1;
     end do

     call barrier(bar,nProc) !on regroupe les processus

     do j=r+id+1,n,nProc    !chaque processus calcule une composante de U
        !            dans la ligne r et à droite de
```

```
      !            la colonne r
      tmp2=0
      do k=1,r-1
         tmp2=tmp2+L(r,k)*U(k,j)
      end do
      if (abs(L(r,r))<EPS) then
         ok=.FALSE.               !arret si diision par zero
      else
         U(r,j)=(A(r,j)-tmp2)/L(r,r)
      end if
   end do

end do

!fin de la decomposition

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
write(*,*) "affichage de la matrice L"
do i=1,n
 write(*,2) L(i,:)

end do
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
write(*,*) "affichage de la matrice U"
do i=1,n
 write(*,2) U(i,:)

end do
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!


if (ok) then

!résolution du premier système triangulaire

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
write(*,*) "Vecteur B"
write(*,2) B(:)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!


   if (id.EQ.0) then           !le pere calcul la premiere composante
      if(abs(L(1,1))<EPS) then
         ok=.FALSE.
      else
         xtmp(1)=B(1)/L(1,1)
      end if
   end if

   do i=2,n    !tous les processus bouclent de 1 à n et se partagent
      !                               le travaille interne
      if (id.EQ.0) tmp3=0
      if (.NOT. ok) exit
      call barrier(bar,nProc)  !les fils attendent que le pere ait fini
                             !le calcul precedent et celui de xtmp(1)
      do j=1+id,i-1,nProc      !calcul de la somme en parallele
         tmp3(j)=tmp3(j)+L(i,j)*xtmp(j)
      end do

      call barrier(bar,nProc)  !regroupement pour que le pere puisse ensuite
      !                              acceder a tmp3
      if (id.EQ.0) then
         if(abs(L(i,i))<EPS) then
            ok=.FALSE.
         else
             write(*,*) "affichage de tmp3"
             write(*,*) tmp3(:)


             tmp4=0
             do j=1,n
                    tmp4=tmp4+tmp3(j)
             end do
```

```
               write(*,*) "affichage de tmp4"
               write(*,2) tmp4


               xtmp(i)=(B(i)-tmp4)/L(i,i)
            end if
         end if

     end do


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  write(*,*) "Vecteur xtmp (tq L*xtmp=B)"
  write(*,2) xtmp(:)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!



     !résolution du deuxième système triangulaire

     if (id.EQ.0) X(n)=xtmp(n) ! le pere calcule la derniere composante

     do i=n-1,1,-1         !tous les processus bouclent  et se partagent
        !                        le travaille interne
        if (id.EQ.0) tmp3=0
        if (.NOT. ok) exit     !si division par zero dans la derniere boucle
        !du système triangulaire inferieur
        call barrier(bar,nProc)  !les fils attendent que le pere ait fini
        !   le calcul precedent et celui de x(1)
        do j=i+1+id,n,nProc
           tmp3(j)=tmp3(j)+U(i,j)*X(j)
        end do

        call barrier(bar,nProc)  !regroupement pour que le pere puisse ensuite
        !                              acceder a tmp3

        if (id.EQ.0) then


               write(*,*) "affichage de tmp3"
               write(*,*) tmp3(:)


               tmp4=0
               do j=1,n
                      tmp4=tmp4+tmp3(j)
               end do



               write(*,*) "affichage de tmp4"
               write(*,2) tmp4



               X(i)=xtmp(i)-tmp4
        end if
     end do


  end if

  call process_join(nProc,id)     !mort des processus fils

  if (ok) then
     !Affichage du résultat
     WRITE(*,1) x(:)
1    FORMAT(F15.7)
  else
     write(*,*) 'arret car division par zero'
  endif

end subroutine croutpara
```

VII. Conclusion

Dans un premier temps, nous nous sommes consacrés à la configuration de LAM, pour qu'il marche correctement et que nous puissions développer des programmes tests afin de découvrir l'environnement étudié. Cet environnement était alors installé sur un seul serveur : sciora. Nous pouvions alors vérifier que nos programmes étaient corrects. Mais, la présence d'une seule machine ne nous permettait pas de tester le parallélisme. Par la suite, nous avons obtenu une machine supplémentaire, et enfin une troisième, ce qui nous a permis d'effectuer des tests sur des machines de puissances équivalentes (amipc6 et amipc7).

Parallèlement, nous avons eu quelques problèmes passagers, dus à des ajustements nécessaires au niveau de la sécurité du réseau de l'université.

Perspectives pour l'an prochain :

William GUYOT a mis à notre disposition trois ordinateurs afin de nous permettre de tester l'environnement LAM et sa mise en place. L'installation de machines supplémentaires est prévue, et est en cours. Cependant, pour les TP de l'année prochaine, il serait judicieux de disposer d'un nombre important de machines (même si celles-ci restent distantes), et d'une plus grande puissance de calcul.

Nous voulons souligner que l'utilisation de cette bibliothèque répandue nous a paru beaucoup plus intéressante et plus proche de cas réels que le travail que nous avons pu effectuer sur le simulateur.

Remerciements :

Nous tenons à remercier Mme GOURGAND pour son soutien et son aide tout au long de notre démarche, ainsi que M. GUYOT qui s'est mobilisé pour l'installation et la configuration de l'environnement LAM.

Sources :

Le site officiel :
www.lam-mpi.org

Divers sites qui nous ont aidé :
http://cui.unige.ch/spc/Teaching/ato-ii/Mpi/descr.html
http://www.crihan.fr/CRIHAN/calcul/par/doc/Doc/Descript_log/Parall_echanges/MPI.html
http://www.pdc.kth.se/training/Talks/MPI/Collective.I/more.html
http://www.idris.fr/data/cours/parallel/mpi/Aide_Memoire_C.html

Etude de cas MPI 2000-2001 : Julien BULICHELLI, Jacky CHABANON, Nicolas MARCON et Yannick MONTET.

Notre site en ligne, où serons disponibles toutes les informations nécessaires (y compris ce rapport) :

www.lam.fr.st

VIII. Annexe

Aide mémoire : MPI et le C

1 - Environnement
Initialiser MPI et quitter MPI :
int MPI_Init(int *argc,char** argv)
int MPI_Finalize(void)
Quitter brutalement MPI :
int MPI_Abort(MPI_Comm comm)
Savoir si un processus à fait un MPI_Init :
int MPI_Initialized(int *flag)
Récupérer la chaine de caractères associée au code d'erreur err:
int MPI_Error_string(int errcode,char *chaine,int *taille_chaine)
Fonctions de timing MPI :
doubleMPI_Wtime(void)
doubleMPI_Wtick(void)

---

2 - Communications point à point bloquantes
Envoyer un message à un processus :
int MPI_[R,S,B]send(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm)
Recevoir un message d'un processus :
int MPI_Recv(void *buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI_Status *status)
Envoyer et recevoir un message :
int MPI_Sendrecv(void *sendbuf,int sendcount,MPI_Datatype sendtype,int dest,int sendtag,void *recvbuf,int recvcount,MPI_Datatype recvtype,int source,int recvtag,MPI_Comm comm,MPI_Status *status)
Compter le nombre d'éléments reçus :
int MPI_Get_count(MPI_status *status,MPI_Datatype datatype,int *count)
Tester l'arrivée d'un message :
int MPI_Probe(int source,int tag,MPI_Comm comm,MPI_Status *status)
Autres fonctions :
MPI_Sendrecv_replace(), MPI_Get_elements()

---

3 - Communications point à point non bloquantes
Commencer à envoyer un message :
int MPI_I[r,s,b]send(void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request * request)
Commencer à recevoir un message :
int MPI_Irecv(void *buf,int count,MPI_Datatype datatype,int source,int tag,MPI_Comm comm,MPI_Request *request)
Compléter une opération non bloquante :
int MPI_Wait(MPI_Request *request,MPI_Status *status)
Tester une opération non bloquante :
int MPI_Test(MPI_Request *request,int *flag,MPI_Status *status)

Libérer une requête avant de la réutiliser :
int MPI_Request_free(MPI_Request *request)
autres fonctions :
MPI_Waitany(), MPI_Waitall(), MPI_Cancel(), MPI_Testany(), MPI_Iprobe(), MPI_Test_cancelled()

---

4 - Communications persistantes
Décrire un schéma persistant :
int MPI_[R,S,B]send_init(void *sendbuf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request)
int MPI_Recv_init(void *recbuf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm,MPI_Request *request)
Démarrer une communication persistante :
int MPI_Start(MPI_Request *request)
Autres fonctions : MPI_Startall(), MPI_Request_free()

---

5 - Communications collectives
Diffusion générale d'un message :
int MPI_Bcast(void *buf,int count,MPI_Datatype datatype,int root,MPI_Comm comm)
Collecte de données :
int MPI_Gather(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount,MPI_Datatype recvtype ,int root,MPI_Comm comm)
Diffusion sélective d'un message :
int MPI_Scatter(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount,MPI_Datatype recvtype ,int root,MPI_Comm comm)
Collecte de données et rediffusion :
int MPI_Alltoall(void *sendbuf,int sendcount,MPI_Datatype sendtype,void *recvbuf,int recvcount,MPI_Datatype recvtype,MPI_Comm comm)
Calcul d'une réduction :
int MPI_Reduce(void *sendbuf,void *recvbuf,int count,MPI_Datatype datatype,MPI_Op operation,int root,MPI_Comm comm)
Calcul d'une réduction et rediffusion du résultat :
int MPI_Allreduce(void *sendbuf,void *recvbuf,int count,MPI_Datatype datatype,MPI_Op operation,MPI_Comm comm)
Synchronisation de processus :
int MPI_Barrier(MPI_Comm comm)
Autres fonctions :
MPI_Gatherv(), MPI_Scatterv(),MPI_Allgather[v](), MPI_Alltoallv(), MPI_Op_create(), MPI_Op_free(), MPI_Scan(), MPI_Reduce_scatter()

---

6 - Communicateurs
Création d'un intracommunicateur :
int MPI_Comm_create(MPI_Comm comm,MPI_Group group,MPI_Comm *newcomm)
Création d'un intercommunicateur :
int MPI_Intercomm_create(MPI_Comm comm_local,int leader_local,MPI_Comm comm_lien,int leader_distant,int tag,MPI_Comm *nouvel_intercom)
Nombre de processus dans un intracommunicateur :
int MPI_Comm_size(MPI_Comm comm,int *nbre)
Nombre de processus dans un intercommunicateur :
int MPI_Comm_remote_size(MPI_Comm intercomm,int *nbre)

Rang d'un processus dans un intracommunicateur :
int MPI_Comm_rank(MPI_Comm comm,int *rang)
Duplication d'un communicateur :
int MPI_Comm_dup(MPI_Comm comm,MPI_Comm *newcomm)
Comparaison de deux communicateurs :
int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2,int *result)
Partage d'un communicateur :
int MPI_Comm_split(MPI_Comm comm,int couleur,int clé,MPI_Comm *newcomm)
Libérer un communicateur :
int MPI_Comm_free(MPI_Comm comm)

---

7 - Groupes de processus
Récupérer le groupe associé à un communicateur :
int MPI_Comm_Group(MPI_Comm comm,MPI_Group *groupe)
Rang d'un processus dans un groupe :
int MPI_Group_rank(MPI_Group groupe,int *rang)
Comparer deux groupes de processus :
int MPI_Group_compare(MPI_Group groupe1,MPI_Group groupe2,int *result)
Fusion de deux groupes :
int MPI_Group_union(MPI_Group groupe1,MPI_Group groupe2,MPI_Group *nouveau_groupe)
Créer un groupe à partir d'un existant :
int MPI_Group_incl(MPI_Group groupe,int n,int *rangs,MPI_Group *nouveau_groupe)
int MPI_Group_excl(MPI_Group groupe,int n,int *rangs,MPI_Group *nouveau_groupe)
Libérer un groupe de processus :
int MPI_Group_free(MPI_Group groupe)
Autres fonctions :
MPI_Group_translate_ranks(), MPI_Group_intersection()

---

8 - Topologies de processus
Création d'une topologie cartesienne :
int MPI_Cart_create(MPI_Comm old_comm,int ndims,int *dims,int * periods,int reorder,MPI_Comm *comm_cart)
Extraire la topologie associée à un communicateur :
int MPI_Cart_get(MPI_Comm comm,int ndims,int *dims,int *periods,int *coords)
Faire générer par MPI une décomposition (x,y,z) :
int MPI_Dims_create(int npes_total,int ndims,int *dims)
Rang d'un processus dans une topologie cartesienne :
int MPI_Cart_rank(MPI_Comm comm,int *coords,int *rang)
Coordonnées (x,y,z) d'un processus dans la topologie :
int MPI_Cart_coords(MPI_Comm comm,int rang,int ndims,int *coords)
Décaler une topologie cartésienne (trouver les voisins) :
int MPI_Cart_shift(MPI_Comm comm,int direction,int disp,int *rang_source,int *rang_dest)
Autres fonctions :
MPI_Cart_sub(), MPI_Cart_map(), MPI_Graph_create(), MPI_Graph_neighbors(), MPI_Graph_neighbors_count(), MPI_Graph_get(), MPI_Topo_test()

---

9 - Types dérivés

Construire un type de données contigües :
int MPI_Type_contiguous(int nbre,MPI_Datatype ancien_type,MPI_Datatype *nouveau_type)
Type de données distantes d'un pas constant :
int MPI_Type_[h]vector(int nbre,int taille_bloc,MPI_Aint pas,MPI_Datatype ancien_type,MPI_Datatype *nouveau_type)
Type de données distantes d'un pas variable :
int MPI_Type_[h]indexed(int nbre,int *taille_bloc,MPI_Aint *pas,MPI_Datatype ancien_type,MPI_Datatype * nouveau_type)
Contruire un type structuré :
int MPI_Type_struct(int nbre,int *taille_bloc,MPI_Aint *pas,MPI_Datatype *anciens_types,MPI_Datatype *nouveau_type)
Valider un type :
int MPI_Type_commit(MPI_Datatype *dataype)
Routine portable pour retourner l'adresse d'une variable :
int MPI_Address(void *variable, MPI_Aint *adresse)
Autres fonctions :
MPI_Type_free, MPI_Type_extent, MPI_Type_size(), MPI_Type_[u,l]b()
10 - Constantes

Jokers :
MPI_ANY_TAG, MPI_ANY_SOURCE
Datatypes élémentaires :
MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_UNSIGNED, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT ,MPI_UNSIGNED_LONG, MPI_LOGICAL, MPI_BYTE, MPI_PACKED
Constantes reservées :
MPI_PROC_NULL, MPI_UNDEFINED
Communicateurs reservés :
MPI_COMM_WORLD, MPI_COMM_SELF
Opérateurs de MPI_Reduce et MPI_Allreduce :
MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_BAND, MPI_BOR, MPI_BXOR, MPI_LAND, MPI_LOR, MPI_LXOR
Résultat de MPI_Comm_Compare et MPI_Group_Compare :
MPI_CONGRUENT, MPI_IDENT, MPI_SIMILAR, MPI_UNEQUAL

Chaînes de caractères retournées par MPI_Error_string :

Code	Symptome
MPI_SUCCESS	Pas d’erreurs
MPI_ERR_BUFFER	Pointeur sur zone buffer invalide
MPI_ERR_COUNT	Argument count invalide
MPI_ERR_TYPE	Datatype invalide
MPI_ERR_TAG	Tag de message invalide
MPI_ERR_COMM	Communicateur invalide
MPI_ERR_RANK	Rang de processus invalide
MPI_ERR_REQUEST	Request de communication invalide
MPI_ERR_ROOT	Processus root invalide
MPI_ERR_GROUP	Groupe de processus invalide
MPI_ERR_OP	Opération de réduction impossible
MPI_ERR_TOPOLOGY	Topologie invalide
MPI_ERR_DIMS	Arguments de dimensions invalide
MPI_ERR_ARG	Argument invalide
MPI_ERR_UNKNOWN	Erreur inconnue
MPI_ERR_TRUNCATE	Message tronqué en réception
MPI_ERR_OTHER	Erreur connue non présente dans cette liste
MPI_ERR_INTERN	Erreur interne MPI
MPI_ERR_IN_STATUS	Erreur dans un champ status
MPI_ERR_PENDING	Communication en cours
MPI_ERR_LASTCODE	Dernier code d'erreur