

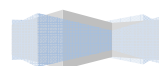
Université de Savoie

MPI-2 : Des concepts à la réalisation

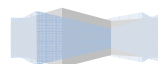
Barbier Keller, Bouleis Romain, Martel Alberic
Novembre 2007

Table des matières

Table des matières	2
Du parallélisme à MPI	4
1. Pourquoi programmer des applications parallèles ?.....	4
2. Quels sont les problèmes liés à l'exécution de programmes parallèles ?.....	4
3. Quels sont les architectures existantes ?	4
4. Comment MPI-2 répond à nos attentes ?	6
Installation de l'environnement et premier programme.....	8
1. Les composants nécessaires au développement	8
a. OpenMPI.....	8
b. Eclipse et PTP.....	8
2. Installation des composants.....	11
a. Eclipse.....	12
b. CDT	12
c. OpenMPI.....	12
d. PTP.....	13
e. PLDT.....	13
3. Configuration préliminaire	14
a. Lancement d'Eclipse	14
b. Configuration.....	14
4. Hello World.....	16
Les concepts par l'exemple	22
1. Squelette d'un programme MPI.....	22
2. L'envoi et la réception de message	23
a. Les fonctions Send et Recv	23
b. Les types de données OpenMPI.....	23
c. L'envoi de messages point à point en mode synchrone	24
d. Envoie de messages en mode asynchrone.....	27
3. Le rendez-vous.....	28
4. La diffusion	30
Cas d'utilisation : le supercalculateur	32
1. Présentation de l'application	32
2. Fonctionnement	32



3. Vu de l'intérieur.....	33
4. Conclusion sur le supercalculateur.....	37
Conclusion.....	38
Table des illustrations	39
Table des sources.....	40
Glossaire.....	41
Bibliographie	42



Du parallélisme à MPI

1. Pourquoi programmer des applications parallèles ?

De nos jours, les avancées technologiques orientent les concepteurs de systèmes vers une augmentation du nombre de ressources pour effectuer un traitement plutôt qu'augmenter la puissance d'une ressource pour exécuter ce même traitement. Ceci est la conséquence de phénomènes physiques empêchant une augmentation infinie de la puissance des ressources. Nous en voulons pour preuve l'annonce par Gordon Moore en septembre 2007 prédisant la désuétude de sa loi selon laquelle le nombre de transistors sur une puce de silicium double tous les deux ans.

De ce fait, afin d'accroître la puissance de calcul, la mise en parallèle de plusieurs ressources est l'alternative la plus crédible. Une ressource peut faire référence à :

- Un cœur de processeur : les processeurs récents possèdent plusieurs cœurs (unités de calcul) fonctionnant en parallèle,
- Un processeur (pouvant être multi-cœurs) : certains serveurs possèdent plusieurs processeurs fonctionnant en parallèle,
- Une machine (pouvant être multiprocesseurs) : il existe des « grappes » de machines fonctionnant en parallèle.

Mais pour bénéficier de l'exécution en parallèle de nos programmes, quelque soit la ressource, ceux-ci doivent être prévus pour.

2. Quels sont les problèmes liés à l'exécution de programmes parallèles ?

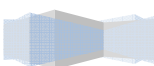
Lors de l'élaboration d'un programme devant être exécuté de manière parallèle, il est nécessaire de respecter un certain nombre de règles :

- Aucune hypothèse ne doit être faite sur le type des ressources : le programme doit avoir le même comportement indépendamment de son environnement d'exécution ; qu'il s'agisse d'un système multi-cœurs, multiprocesseurs, multi-machines voir même « uni-cœur » avec un système d'exploitation multitâche.
- Aucune hypothèse ne doit être faite sur la capacité de calcul à l'instant t des ressources : le programme doit avoir le même comportement quelque soit la vitesse d'exécution des différentes unités de calcul.

3. Quels sont les architectures existantes ?

Plusieurs architectures et leurs modèles de programmations respectifs ont donc été imaginés. Voici les trois principales :

- SISD (Single Instruction Single Data) : machines monoprocesseur, PC.
- SIMD (Single Instruction Multiple Data) : architectures vectorielles monoprocesseur.
- MIMD (Multiple Instruction Multiple Data) : architectures parallèles multiprocesseurs. Cette dernière architecture peut être décomposée en deux modèles de programmation :



- MPMD (Multiple Program Multiple Data) : les processus exécutent des programmes différents.
- SPMD (Single Program Multiple Data) : les processus exécutent le même programme. Avec ce modèle de programmation, on utilise deux outils de parallélisations :
 - OpenMP pour les systèmes à mémoire partagée.
 - MPI pour les systèmes à mémoire distribuée.

Dans un système à mémoire partagée, tous les processeurs ont accès à la même mémoire, des conflits sont donc possibles. La communication entre les processeurs se fait par l'intermédiaire des données stockées en mémoire, ce qui est peu gourmand en ressources. De plus, pour garder une application performante, le nombre de processeurs doit être relativement faible, ce type d'architecture peut donc être relativement onéreux dès lors que l'on a besoin d'augmenter ses capacités. Le schéma suivant illustre ceci.

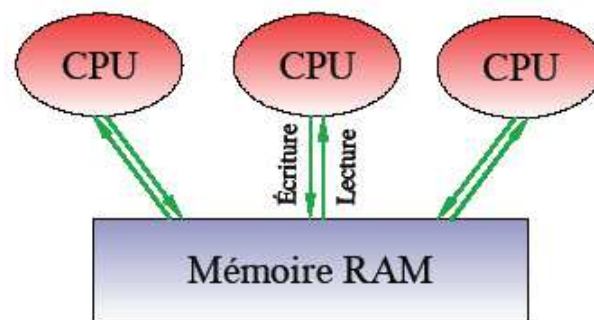


Figure 1 : Système à mémoire partagée

Dans un système à mémoire répartie, chaque processeur dispose de sa propre mémoire. Les processeurs ne peuvent donc pas avoir accès à la mémoire des autres processeurs. Ils doivent donc communiquer entre eux. Une telle architecture est plus abordable qu'un système à mémoire partagée, l'ajout d'une machine suffit pour augmenter les performances. Il est donc théoriquement possible de disposer d'un nombre illimité de ressources. L'inconvénient de cette architecture est que le protocole de communication mobilise plus de ressources que le partage de la mémoire entre les processeurs. Il faut donc mettre en place un réseau de communication performant. Le schéma suivant illustre ceci :

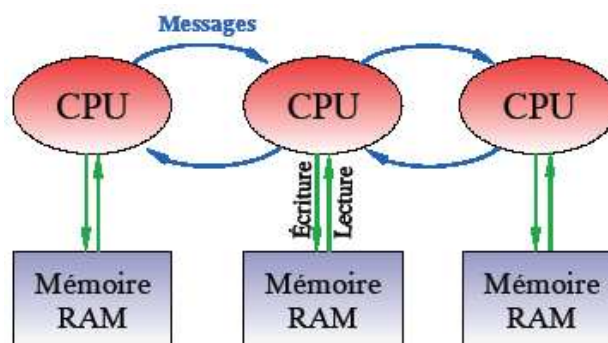
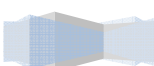


Figure 2 : Système à mémoire répartie



4. Comment MPI-2 répond à nos attentes ?

Le sigle MPI signifie « Message Passing Interface ». C'est une norme conçue en 1994 par une association d'une quarantaine d'organisations, qui définit une bibliothèque implémentée en C, C++ et Fortran 95 dédiée à la programmation parallèle par passage de message, c'est-à-dire que les processus communiquent entre eux en s'envoyant des messages simples. Grâce à cette bibliothèque, il est possible d'écrire des applications parallèles indifféremment du type de système sur lequel elles sont exploitées. Elles fonctionnent donc aussi bien sur des machines multiprocesseurs que sur des clusters de machines, des machines à mémoire partagée ou distribuée. Il existe d'autres bibliothèques de passage de message plus anciennes, mais l'avantage de MPI est qu'elle est grandement portable car implémentée sur de très nombreuses architectures de mémoire.

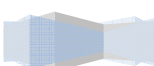
La norme MPI est disponible pour toutes les machines parallèles grâce aux implémentations LAM et MPICH relevant du domaine public. A l'époque, ces implémentations sont réalisées en Fortran 77 et en C. Cette version de MPI est notée « 1 ».

MPI-1 propose les fonctionnalités suivantes :

- Fourniture d'un environnement de développement parallèle.
- Communications point-à-point.
- Communications collectives.
- Types de données dérivés : MPI fournit pour chaque type de donnée existant (int, char, ...) un type correspondant dans son implémentation afin de les rendre portables sur tous types de machines (par exemple, un entier, « int » en C, aura pour type en MPI « MPI_INT »). Cela fonctionne pour les types de données classiques mais aussi pour les structures.
- Topologies : Il est possible, en faisant appel aux fonctions adaptées, d'agencer les processus en différentes topologies (en anneau, en grille).
- Groupes et communicateurs : Il est possible de regrouper les processus actifs, une utilité est d'envoyer des données à traiter dans ce groupe sans choisir précisément quel processus sera concerné par le traitement. Les communicateurs sont utilisés pour désigner les processus concernés par la communication, par défaut les processus communiquent dans le communicateur MPI_COMM_WORLD.

Après quelques modifications mineures, des améliorations majeures sont apportées à la norme MPI en 1997, et devient donc MPI-2. Ces améliorations sont :

- La gestion dynamique des processus : il devient possible de créer et de supprimer des processus durant l'exécution du programme.
- Les copies de mémoire à mémoire : avec MPI-1, il existait les communications point-à-point par échange de messages. Les deux processus devaient donc être disponibles en même temps pour que la communication puisse avoir lieu. Avec MPI-2, il est désormais possible d'accéder directement à la mémoire d'un processus distant. On appelle cela RMA pour « Remote Memory Access ». Cette nouvelle forme de communication permet de libérer le processus distant, ce qui apporte de meilleures performances.
- MPI I/O : Les applications effectuant d'importants calculs manipulent également de grandes quantités de données et génèrent donc beaucoup d'entrées-sorties. MPI-2 apporte une grosse dose d'optimisations dans ce domaine afin d'augmenter les performances globales. Ces optimisations se font par la combinaison de la



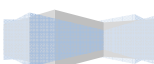
parallélisation des entrées-sorties, de techniques mises en œuvre explicitement dans le code (lecture/écriture asynchrone, opérations collectives), et d'opérations prises en charge par le système d'exploitation (regroupement des requêtes, gestion des tampons d'entrée/sortie).

- Un interfaçage avec Fortran 95

De nouvelles implémentations voient le jour :

- MPICH2
- LAM
- OpenMPI
- MPI d'IBM
- MPI de Fujitsu
- MPI de NEC
- MPI de SUN

Toutes ces implémentations gèrent les fonctions apportées par MPI-2 à l'exception de LAM qui n'est pas disponible en Fortran 95 et qui gère partiellement les copies de mémoire à mémoire et l'implémentation d'IBM qui ne gère pas les processus dynamiquement et qui n'est également pas disponible en Fortran 95.



Installation de l'environnement et premier programme

1. Les composants nécessaires au développement

a. OpenMPI

MPI étant un standard, il en existe plusieurs implémentations comme MPICH, LAM/MPI, FT-MPI, OpenMPI, etc., avec chacune leurs avantages et leurs inconvénients.

Afin de réaliser les exemples contenus dans l'ouvrage, nous avons préféré OpenMPI pour les raisons suivantes :

- Compatibilité complète avec MPI-2
- « Thread safe »
- Tolérant aux pannes réseaux
- Support de réseaux hétérogènes
- Support de plusieurs systèmes d'exploitation
- Très documenté et Open Source

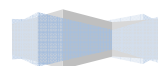
OpenMPI est donc un ensemble de fonctions respectant la norme MPI-2, permettant d'écrire des programmes parallèles en s'abstrayant complètement de la couche réseau. Cette librairie est écrite en trois langages : C, C++ et Fortran. Il appartient donc au développeur de choisir un des ces langages pour écrire son application. Pour faire ce choix, nous avons d'abord exclu Fortran ; en effet, ce langage n'est aujourd'hui plus très utilisé et peu de monde y est formé. Restant donc C et C++, nous avons préféré C++ uniquement pour l'aspect objet qu'il apporte par rapport au C. Tous les exemples contenu dans cet ouvrage seront donc écrit en C++, néanmoins la traduction dans un des deux autres langages est assez facile, les fonctions ayant généralement les mêmes types de paramètres.

b. Eclipse et PTP

Initialement prévu pour le développement en Java, Eclipse s'impose peu à peu comme étant l'EDI de référence pour de multiples langages tel que Ruby, PHP, Perl, C et C++. De plus, Eclipse possède d'innombrables plugins facilitant le développement de toutes sortes d'applications, y compris un plugin pour écrire et débogger des programmes parallèles : PTP, pour « Parallel Tools Platform ».

Eclipse et PTP apportent donc un environnement complet et adapté aux besoins liés à la programmation parallèle, facilitant la programmation grâce à notamment :

- Coloration syntaxique
- Auto complétion
- Organisation des sources et des exécutables
- Visualisation des ressources et des processus mis en jeux
- Débogage avancé



Les captures d'écran suivantes montrent différents cas d'utilisation d'Eclipse avec PTP :

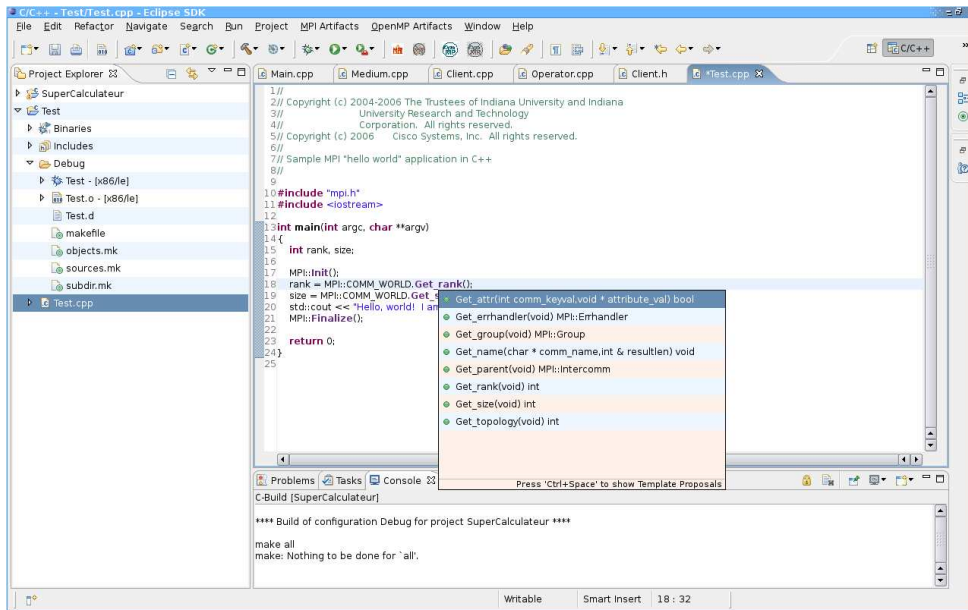


Figure 3 : Autocomplétion avec Eclipse

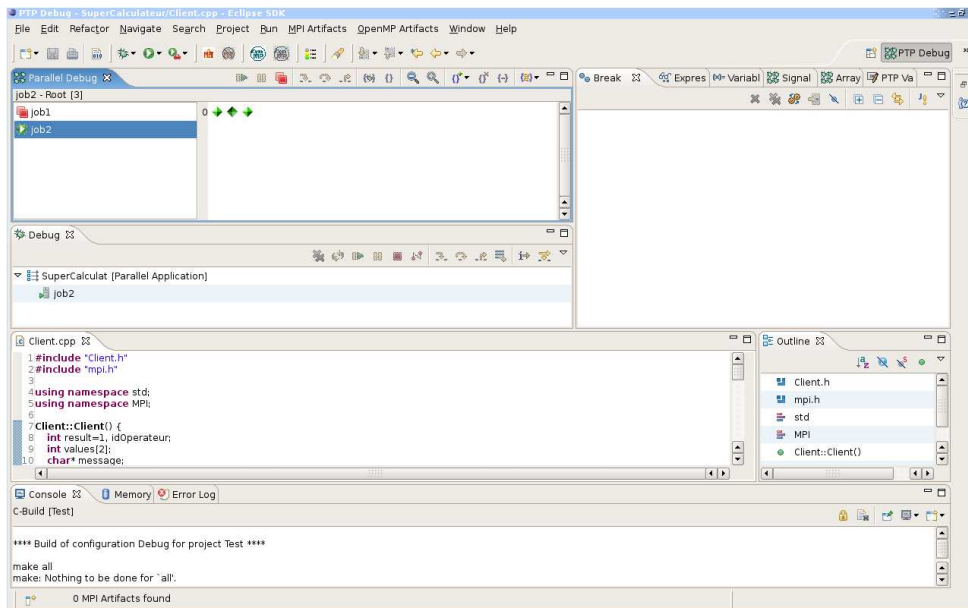
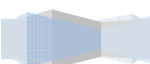


Figure 4 : Perspective d'exécution



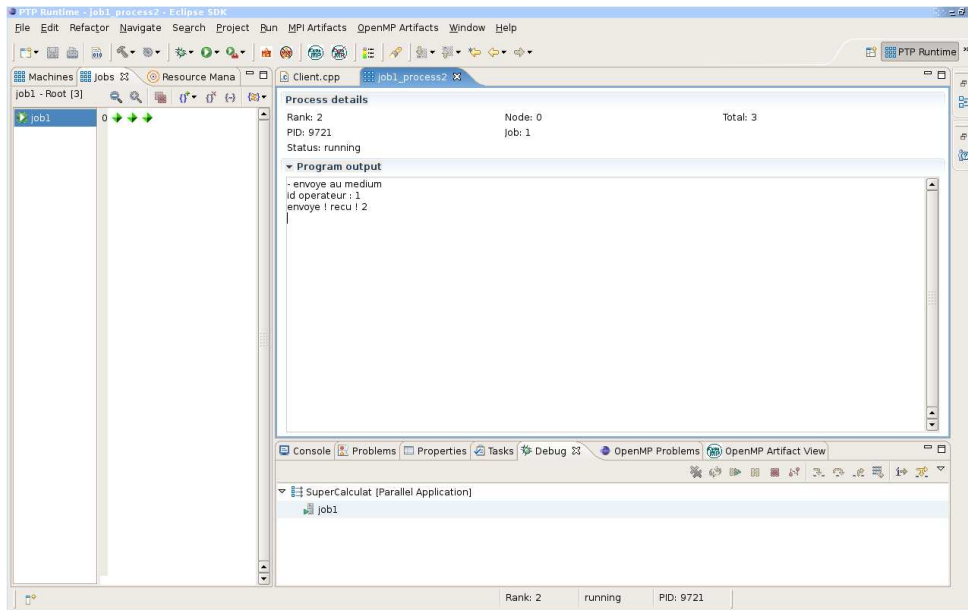


Figure 5 : Perspective de débogage

Les architectures sur lesquelles les programmes parallèles peuvent être exécutés étant très diverses, PTP a été pensé de manière générique. De ce fait, PTP est séparé en deux modules principaux communiquant entre eux lors de l'exécution ou du débogage d'une application :

- L'un intégré à Eclipse, permettant la visualisation, le contrôle et le débogage.
- L'autre externe à Eclipse, capturant l'état des processus mis en jeu.
- La figure suivante montre l'interaction entre ces deux modules.

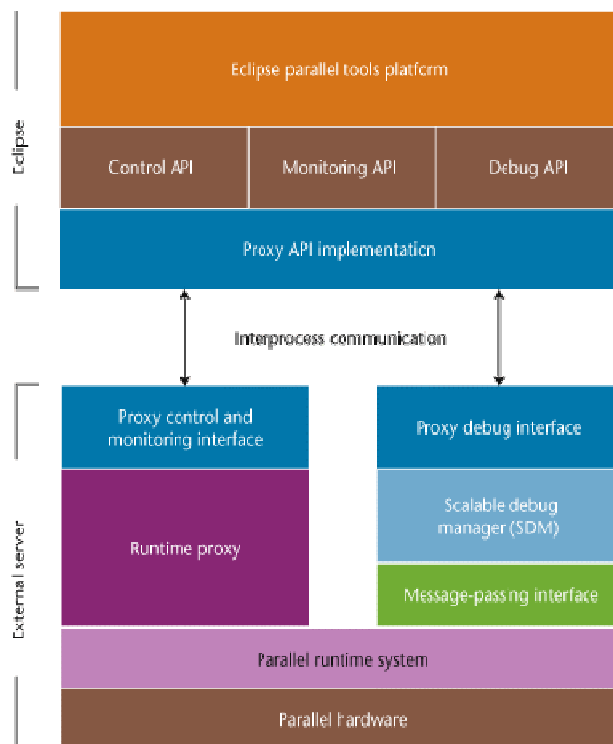
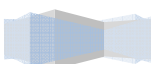


Figure 6 : Architecture de PTP (Source : <http://www.computer.org>)



Le « Runtime proxy » est un élément important du dispositif, c'est lui qui va permettre à Eclipse de communiquer avec les processus. Ce proxy est un programme dépendant de l'implémentation choisie ; il devra donc être compilé lors de l'installation de PTP. Il en va de même pour le « Scalable debug manager » (sdm) qui va permettre la communication entre le debugger d'Eclipse et les processus.

2. Installation des composants

Si l'utilisation d'un EDI facilite grandement le développement, sa mise en place est elle relativement périlleuse et son installation doit suivre un protocole précis. Ce chapitre est donc consacré à l'installation des différents composants. Pour l'écriture de cet ouvrage, nous avons fait le choix d'utiliser Linux Kubuntu 7.04, néanmoins, cette procédure doit pouvoir s'appliquer à la majorité des systèmes Unix. Tous les fichiers nécessaires sont disponibles sur le CD-ROM joint. Bien qu'il reste possible d'utiliser les dernières versions disponibles sur les sites respectifs des différentes applications, il sera nécessaire de faire attention à la compatibilité entre les composants, notamment entre PTP et OpenMPI.

Si une étape de l'installation ne fonctionne pas comme il sera décrit par la suite, il sera alors nécessaire de consulter la documentation en ligne du composant.

Pour pouvoir installer les composants, il faudra s'assurer de disposer de Linux, du compilateur gratuit g++ et de java 1.5 minimum (Attention, sur certaines distributions, la version de java n'est pas celle développé par Sun et est incompatible avec PTP, il faut donc supprimer les paquets concernant « gcj » et à installer les paquets « sun-java5 »).

Pour comprendre les dépendances entre les différents programmes, voici un diagramme récapitulatif :

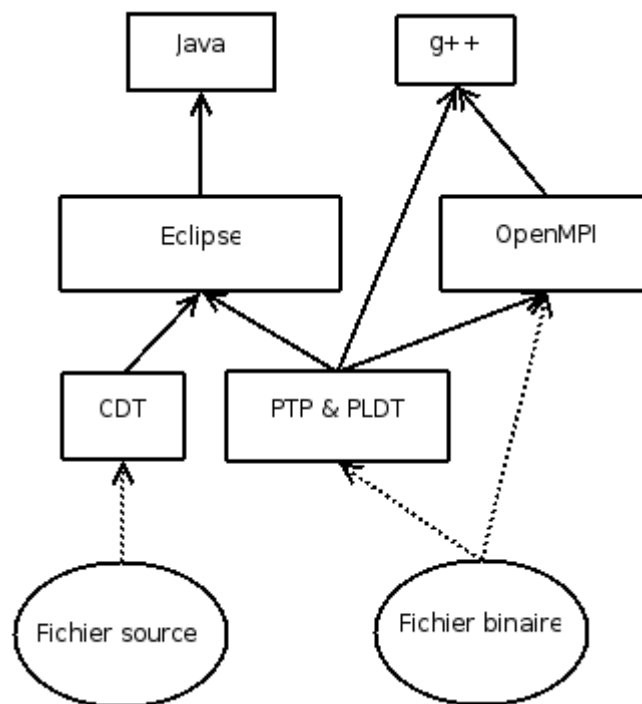
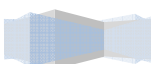


Figure 7 : Graphe de dépendance des composants



a. Eclipse

Récupérer le fichier *eclipse-SDK-3.3.1-linux-gtk.tar.gz* présent sur le cd-rom et l'extraire dans le répertoire désiré pour l'installation grâce à la commande :

```
tar -xvzf eclipse-SDK-3.3.1-linux-gtk.tar.gz <eclipse_dir>
```

Eclipse se lancera alors grâce à commande `./eclipse &` à la racine du dossier d'installation.

b. CDT

L'installation par défaut d'eclipse ne contient pas le plugin permettant de développer en C++; il faut donc l'installer manuellement. Ce plugin se trouve sur le cd-rom et à pour nom de fichier *cdt-master-4.0.1.zip*. Le décompresser dans le répertoire d'Eclipse et redémarrer Eclipse.

c. OpenMPI

OpenMPI ne consiste pas seulement en une collection de fonctions permettant de programmer des applications parallèles. En effet, les programmes incluant les bibliothèques doivent être compilés et exécutés différemment. Pour ce faire, OpenMPI propose d'utiliser les exécutables suivants :

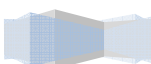
- `mpic++`
Compile et lie un programme écrit en c++.
- `mpicc`
Compile et lie un programme écrit en c.
- `mpiCC`
Equivalent à `mpic++`. Gardé pour assurer une compatibilité avec d'autres implémentations.
- `mpicxx`
Equivalent à `mpic++`. Gardé pour assurer une compatibilité avec d'autres implémentations.
- `mpiexec`
Exécute un programme utilisant `mpi`.
- `mpif77`
Compile et lie un programme écrit en Fortran77.
- `mpif90`
Compile et lie un programme écrit en Fortran90.
- `mpirun`
Equivalent à `mpiexec`. Gardé pour assurer une compatibilité avec d'autres implémentations.

Une fois OpenMPI installé, une description précise de ses commandes est accessible via la commande « `man` » (ex : `man mpiCC`).

Etant multiplateforme, OpenMPI est distribué sous forme de sources. Les fichiers binaires précédemment cités doivent donc être compilés.

L'installation d'OpenMPI a donc plusieurs aspects :

- La copie des librairies MPI



- La compilation des exécutables
- L'ajout des documentations des exécutables et des fonctions MPI au système. Une fois OpenMPI installé, il est possible de faire un « man » sur n'importe quelle fonction pour obtenir de l'aide à propos de celle ci (ex : man mpi_isend).

L'installation s'effectue comme suit :

- Décompression de l'archive *openmpi-1.2.4.tar.gz* dans un dossier temporaire.
- Dans ce dossier, effectuer la commande *./configure --with-devel-headers*, puis la commande *sudo make install*.

Afin de tester la validité de l'installation, essayer la commande *mpirun -np 2 hostname*. Cette commande aura pour effet d'exécuter en parallèle la commande *hostname*. L'argument *-np 2* indique que deux processus seront créés. Séquentiellement, ceci revient à exécuter deux fois « hostname ». Ceci retournera donc deux lignes identiques contenant le nom de la machine sur laquelle la commande est effectuée.

Il en sera de même pour exécuter tout programme mpi en parallèle en ligne de commande : *mpirun -np x un_prog*, ou *x* est le nombre de processus désiré et *un_prog* un programme mpi.

d. PTP

PTP est un plugin particulier puisqu'il est nécessaire de compiler deux exécutables : le proxy permettant le lien entre Eclipse et l'exécution des processus et *sdm* qui, lui, permet l'interfaçage entre le débogueur d'Eclipse et les processus.

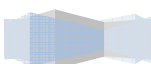
Pour pouvoir être compilé et exécuté, ces exécutables doivent connaître l'emplacement des bibliothèques mpi. Si la procédure a été suivie, elles se trouvent dans */usr/local/lib*. Pour indiquer cet emplacement, il faut utiliser la variable d'environnement *LD_LIBRARY_PATH* grâce à la commande : *export LD_LIBRARY_PATH=/usr/local/lib*.

Comme tout plugin, il faut d'abord le décompresser dans le répertoire où réside Eclipse. L'archive est disponible sur le CD-ROM sous le nom *org.eclipse.ptp-1.1.1.tar.gz*.

Une fois décompressé, aller dans *<eclipse>/plugin/org.eclipse.ptp.linux.<arch>1.1.0*, puis exécuter la commande *sh BUILD* qui installera alors le proxy et *sdm*.

e. PLDT

PLDT est un plugin pour Eclipse permettant l'auto complétion des fonctions MPI. C'est une composante du plugin PTP mais contrairement à celui ci, il est compatible Windows, c'est pourquoi il n'y est pas directement intégré. Pour l'installer, décompresser le fichier *org.eclipse.ptp.pldt-1.1.1.tar.gz* présent sur le cdrom dans le répertoire d'éclipse.



3. Configuration préliminaire

a. Lancement d'Eclipse

Désormais, tout est normalement installé ; il ne reste plus que quelques petits détails de configuration pour avoir un environnement fonctionnant parfaitement.

Le proxy de PTP sera lancé directement par Eclipse, et comme mentionné plus haut, il est nécessaire de connaître l'emplacement des bibliothèques MPI grâce à la variable d'environnement `LD_LIBRARY_PATH`. Cette variable devra donc toujours être valide avant l'exécution d'Eclipse. Pour ce faire, un simple script dont le contenu est le suivant suffit :

```
#!/bin/bash
export LD_LIBRARY_PATH=/usr/local/lib
<repertoire_eclipse>/eclipse
```

Source 1 : Script lancement d'éclipse

Pour que PTP fonctionne, ce script doit être le seul point d'entrée pour exécuter Eclipse.

b. Configuration

Avant de créer un nouveau projet, il faut configurer PTP. Les captures d'écrans suivantes montrent les points clés. Pour accéder à la fenêtre de configuration, utiliser le menu **Window, Preferences**.

- Chemin de SDM

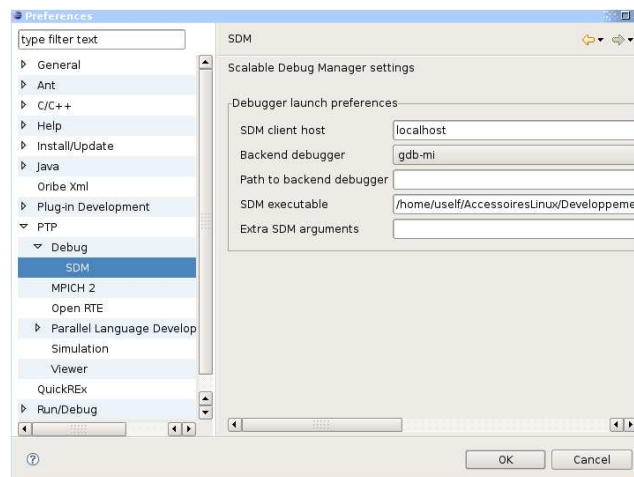
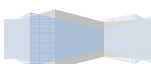


Figure 8 : Configuration, chemin de SDM

La valeur du champ **SDM executable** doit être le chemin vers l'exécutable compilé lors de l'installation de PTP ; ce fichier se trouve dans le répertoire suivant : `<eclipse>/plugins/org.eclipse.ptp.linux.<arch>_1.1.0/bin/`.



- Chemin du proxy

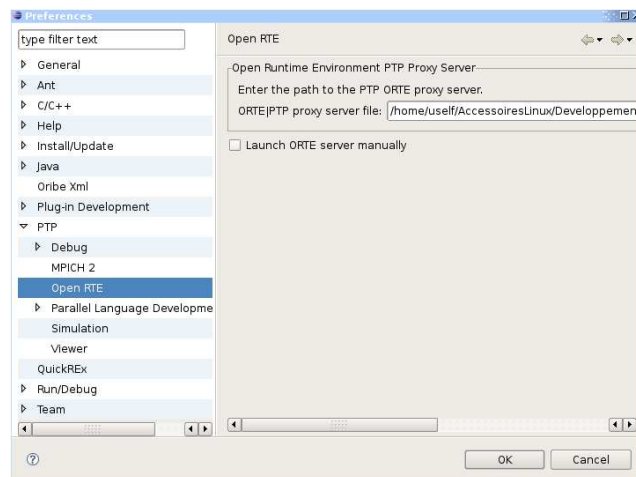


Figure 9 : Configuration, chemin du proxy

La valeur du champ **ORTE|PTP proxy server file** doit être le chemin vers le binaire du proxy. Ce fichier se trouve dans `<eclipse>/plugins/org.eclipse.ptp.linux.<arch>_1.1.0/bin/`.

- Sources MPI

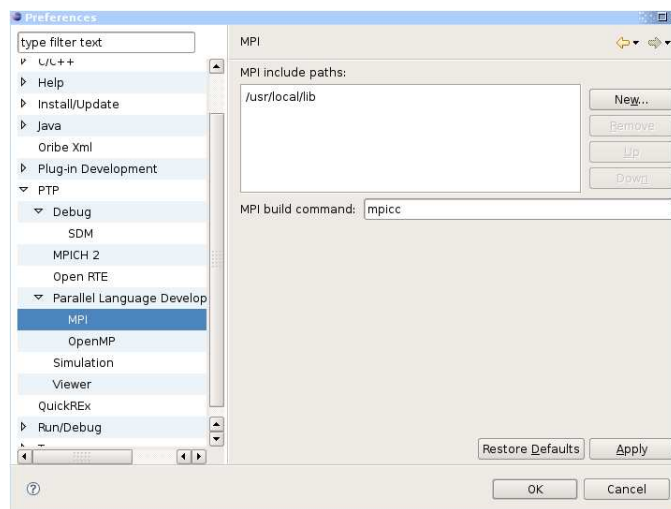
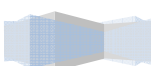


Figure 10 : Configuration, sources MPI

- Le champ **MPI include paths** doit contenir l'emplacement des librairies MPI ; normalement `/usr/local/lib`.
- **MPI build command** indique la commande permettant de compiler les programmes mpi : `mpicc`.



4. Hello World

Comme tout bon premier programme qui se respecte, celui proposé ici permettra à chaque processus exécuté d'afficher « Hello, je suis le processus x de X » (ou x est le numéro du processus MPI et X le nombre total de processus mis en jeu par le programme).

Toutes les étapes pour obtenir ce programme seront soigneusement détaillées. Néanmoins, pour la suite de l'ouvrage, toutes les manipulations ne seront pas décrites, seul le code source sera expliqué, c'est pourquoi, il est nécessaire de suivre les instructions qui suivent de manière rigoureuses.

- Création d'un nouveau projet

Pour créer un nouveau projet dans Eclipse : **File, new, Project, C++, C++ Project.**

L'écran suivant apparaîtra alors :

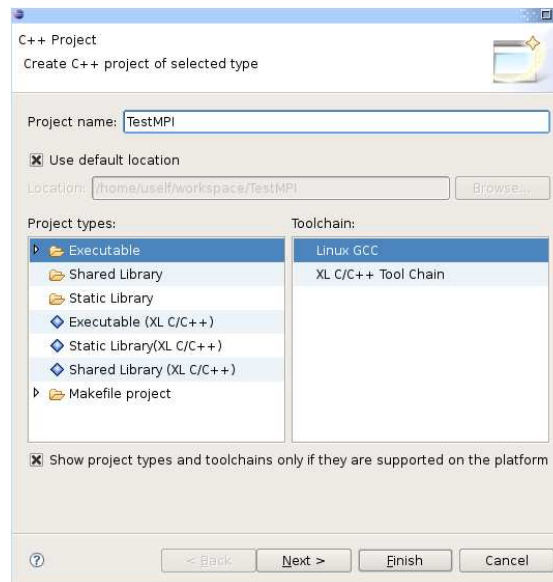
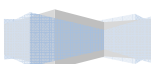


Figure 11 : Nouveau projet c++

Choisir un nom de projet puis **Executable**, **Linux GCC** et cliquer sur **Next** jusqu'à la page **MPI Project Settings**.



- Préférences MPI

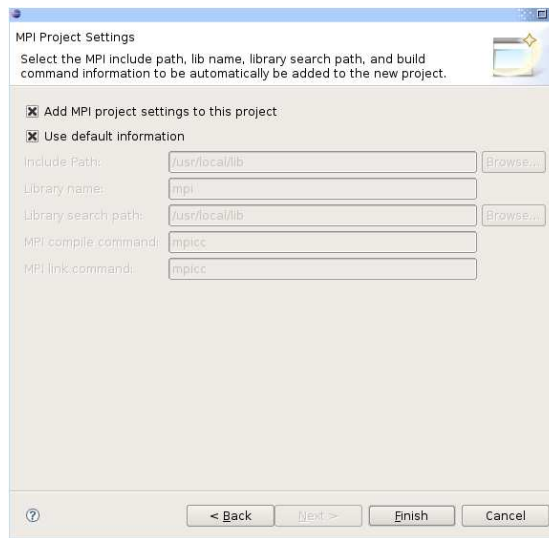


Figure 12 : Préférences MPI

Simplement cocher **Add MPI project settings to this project**. Si l'installation a été effectuée selon la procédure, il n'y a rien d'autre à changer. Cliquer sur **Finish**.

Un nouveau projet sera alors créé. Il est visible depuis la vue « Project explorer ».

- Paramètres du projet

Eclipse se chargeant de la compilation, il est nécessaire de lui préciser les commandes pour compiler (*mpiCC*). Ces préférences se modifient dans les propriétés du projet. Dans la vue **Project explorer**, cliquer droit sur le projet nouvellement créé, puis **properties**. La fenêtre de dialogue suivante apparaîtra :

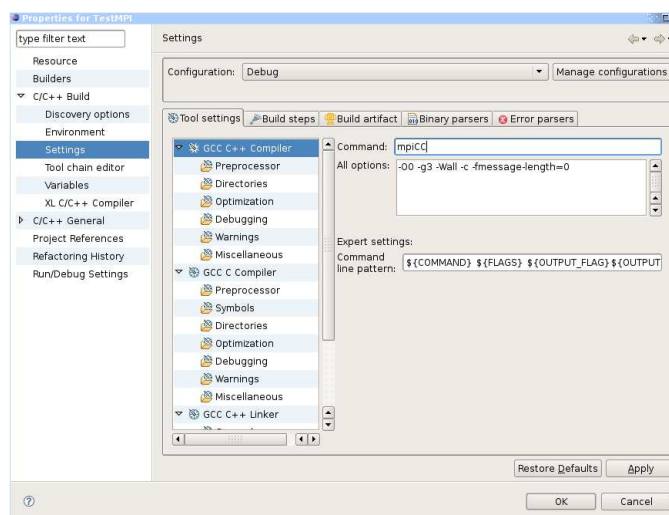
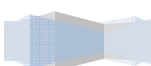


Figure 13 : Préférences du projet

Remplacer *gcc* par *mpiCC* dans le champ **Command**. Faire de même après avoir sélectionné **GCC C++ Linker** dans la liste de choix centrale. Valider.



- Création d'un fichier source

Dans la vue « Project explorer », cliquer droit sur le projet, puis **new, Source File**. L'écran suivant apparaîtra alors :

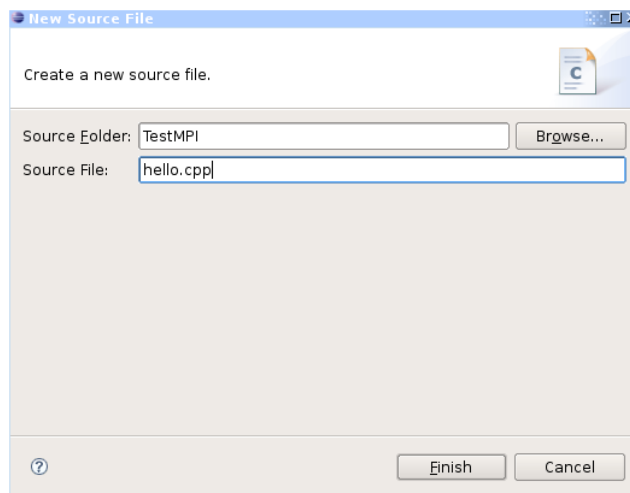


Figure 14 : Création du fichier source

Donner un nom au nouveau fichier. Celui ci doit avoir pour extension « cpp » (extension pour les sources c++). Cliquer sur *Finish*.

L'éditeur de code s'ouvre alors.

OpenMPI est fourni avec quelques exemples d'utilisation. Copier/Coller le contenu du fichier *hello_cxx.cc* se trouvant dans *examples* dans le dossier où OpenMPI a été décompressé dans l'éditeur de code d'Eclipse. Le contenu de ce fichier est le suivant :

```
//
// Copyright (c) 2004-2006 The Trustees of Indiana University and Indiana
//                           University Research and Technology
//                           Corporation. All rights reserved.
// Copyright (c) 2006      Cisco Systems, Inc. All rights reserved.
//
// Sample MPI "hello world" application in C++
//

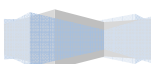
#include "mpi.h"
#include <iostream>

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

    MPI::Init();
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    std::cout << "Hello, world! I am " << rank << " of " << size << std::endl;
    MPI::Finalize();

    return 0;
}
```

Source 2 : Hello World MPI



Lors de l'exécution, x processus de ce programme seront créés. A ce stade, il faut comprendre que la seule chose qui différencie les processus est leur identifiant ; leur « rank ». Cet identifiant est stocké dans la variable *rank* et est obtenu grâce à la fonction *Get_rank*. La variable *size* quant à elle contient le nombre total de processus et est obtenue grâce à la fonction *Get_size*. La variable globale « COMM_WORLD » est un communicateur MPI. Grossièrement, il s'agit du groupe par défaut dans lequel des processus peuvent communiquer. Noter ici qu'il n'y a aucune communications entre les différents processus. Il s'agit juste d'afficher son identifiant puis de quitter. Les fonctions *Init* et *Finalize* initialisent et finalisent respectivement un programme MPI.

- **Compilation et exécution**

Sauvegarder votre fichier grâce à *ctrl + s*. Pour compiler, *ctrl + b*. Il faut maintenant créer un profil d'exécution qui servira à déterminer le nombre de processus devant être créés. Pour ce faire, Cliquer droit sur le projet, **Run As, Open run dialog....** . La boîte de dialogue suivante s'ouvrira :

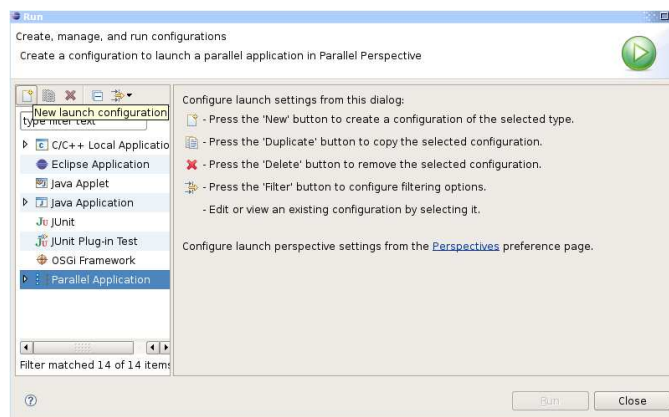


Figure 15 : Profil d'exécution

Sélectionner **Parallel Application** puis cliquer sur l'icone sur le coin supérieur gauche pour créer un profil.

Le paramétrage du profil se fait comme suit :

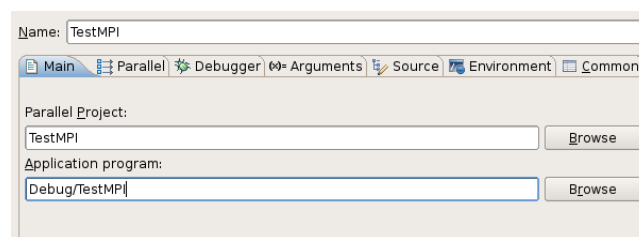
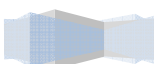


Figure 16 : Options d'exécution, choix de l'exécutable

Dans l'onglet **Main** : le champ **Application program** doit contenir l'exécutable du programme : *Debug/TestMPI*.



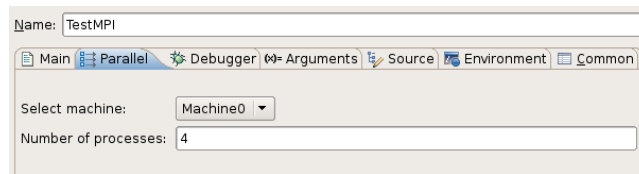


Figure 17 : Options d'exécution, choix du nombre de processus

Dans l'onglet **Parallel** : le champ **Number of processes** doit contenir le nombre de processus devant être créés (argument « -np x » de la commande mpirun).

Cliquer sur **Run** pour lancer les processus. Il faut alors afficher la perspective **PTP Runtime** pour afficher les traces d'exécution. Pour ce faire **Window, Open Perspective, Other..., PTP Runtime, OK**.

La procédure est la même pour afficher la perspective de débogage. L'utilisation de celle-ci ne sera pas détaillée car elle se limite à des cas spécifiques.

L'écran suivant présente la perspective d'exécution du programme.

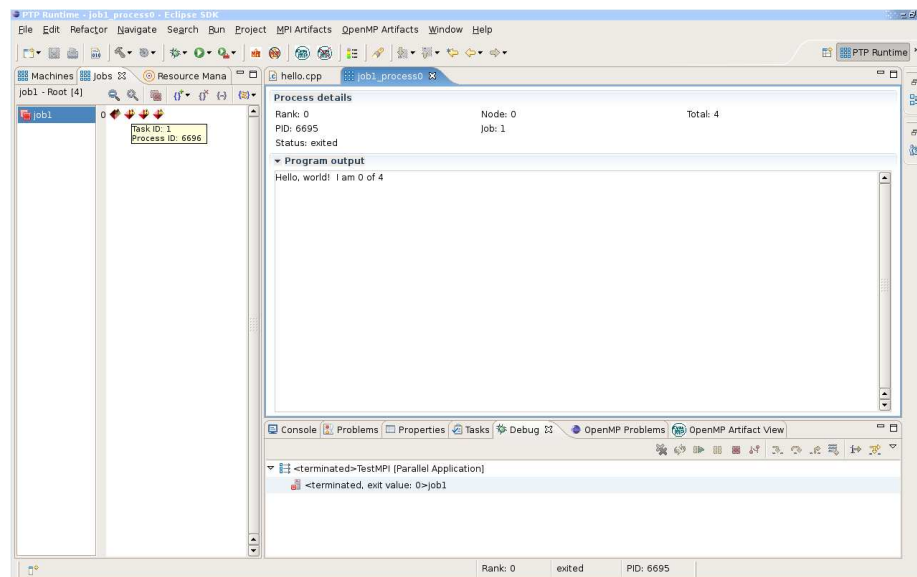
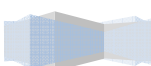


Figure 18 : Runtime Perspective

Cette perspective est constituée de trois vues :

- Les ressources à droite.
- Le code et la trace au centre.
- Des informations complémentaires en bas.

Les processus sont représentés par des logos rouges. Un double clic sur l'un d'eux permet d'afficher sa trace d'exécution. La légende suivante donne une explication des différents icônes pouvant représenter les processus :



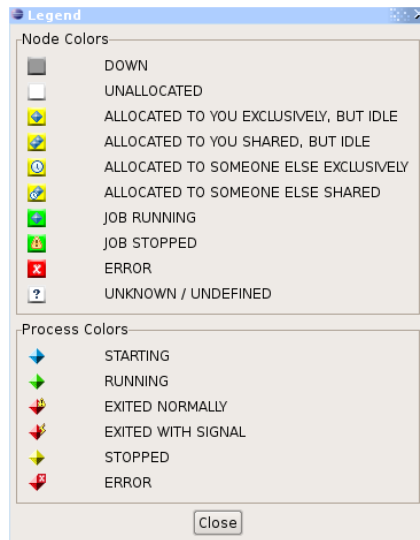
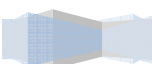


Figure 19 : Légende des icones des processus



Les concepts par l'exemple

La meilleure façon de s'imprégner de la logique de MPI est de comprendre l'implémentation des schémas simples de la programmation parallèle. Les exemples suivants seront donc développés et expliqués :

- L'envoi et la réception de messages
- Le schéma du « rendez-vous »
- La diffusion (broadcast)

Mais avant tout, il convient d'expliquer le squelette d'un programme MPI.

Remarque 1 : tous les exemples ci-dessous ont été testés sur une machine monoprocesseur. De ce fait, le parallélisme simulé est tributaire de l'ordonnanceur des tâches du système d'exploitation. Celui-ci étant multitâche, ceci n'aura aucune incidence sur les résultats obtenus.

Remarque 2 : toutes les sources des exemples sont disponibles sur le CD-ROM.

1. Squelette d'un programme MPI

Quelque soit la complexité du programme voulu, il sera toujours nécessaire :

- D'inclure la librairie *mpi.h*
- D'appeler les fonctions mpi uniquement à l'intérieur des deux fonctions que sont *MPI::Init* et *MPI::Finalize*.

Il sera aussi très souvent intéressant de connaître l'identifiant du processus et le nombre de processus total grâce à *MPI::Get_size* et *MPI::Get_rank*.

Voici un exemple de squelette. Attention, l'exécution de ce programme n'aura aucun effet.

```
#include "mpi.h"

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

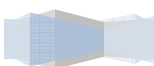
    MPI::Init();
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();

    //programme

    MPI::Finalize();

    return 0;
}
```

Source 3 : Squelette d'un programme MPI



2. L'envoi et la réception de message

Lors des communications interprocessus, il existe différentes méthodes de transfert de messages. OpenMPI implémente donc un mode synchrone et un mode asynchrone qui permettent d'effectuer des émissions et des réceptions bloquantes ou non.

Un message peut être considéré comme une enveloppe contenant une donnée (une lettre) et une adresse de destination.

a. Les fonctions Send et Recv

Quelque soit les fonctions d'envoi et de réception utilisées, les arguments sont toujours les mêmes. Les voici pour les deux fonctions de bases que sont Send et Recv :

```
MPI::COMM_WORLD.Send(&message, taille, type, dest, tag);
MPI::COMM_WORLD.Recv(&message, taille, type, source, tag);
```

Source 4 : Arguments des fonctions Send et Recv

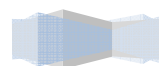
- *message* : pointeur sur la variable devant être envoyée dans le cas de Send, devant être reçue dans le cas de Recv.
- *taille* : la taille du message.
- *type* : type mpi de la valeur envoyée, par exemple : MPI::INT si le message est un *int*.
- *dest / source* : numéro du processus destination / source.
- *tag* : un entier devant être le même pour l'émetteur et le récepteur. Utilisé pour différencier les catégories de messages (le message est une donnée ou le message est un contrôle par exemple).

Il existe deux « jokers » :

- MPI_ANY_SOURCE qui est une variable globale à OpenMPI permettant de remplacer l'argument dest / source pour que l'échange se fasse sans connaître à l'avance l'identifiant de son interlocuteur.
- MPI_ANY_TAG qui est aussi une variable globale permettant d'émettre et de recevoir quelque soit le tag de l'émetteur et du récepteur.

b. Les types de données OpenMPI

- MPI::CHAR : char.
- MPI::SHORT, MPI::INT, MPI::LONG : int short, int et long.
- MPI::UNSIGNED_CHAR, MPI::UNSIGNED_SHORT : unsigned char, unsigned int short.
- MPI::UNSIGNED_LONG : unsigned long.
- MPI::FLOAT, MPI::DOUBLE, MPI::LONG_DOUBLE : float, double.
- MPI::BYTE.
- MPI::PACKED : une structure.



c. L'envoi de messages point à point en mode synchrone

Un médium garantit que le transfert d'information n'est possible que lors d'une synchronisation globale des processus émetteur et récepteur, c'est-à-dire que la communication ne peut commencer que lorsque le processus récepteur est prêt à recevoir. Il y a toutefois quelques variantes de comportement dépendantes de l'utilisation de fonctions bloquantes ou non bloquantes.

i. Envoi synchrone bloquant et réception bloquante

Le processus émetteur ne commencera l'envoi de son message que lorsque le processus récepteur sera prêt à recevoir. Les deux processus devront donc être parfaitement synchronisés.

La fonction d'envoi synchrone bloquant est `Ssend` et la fonction de réception bloquante est `Recv`. Ici l'émetteur est bloqué jusqu'à l'envoi du message et ne peut donc pas continuer l'exécution de son programme.

Le code suivant est un exemple d'utilisation de ces fonctions :

```
/* Le processus émetteur (0) envoie l'entier « message » au processus récepteur (1) de façon synchrone bloquante, c'est-à-dire qu'il attend que le processus récepteur exécute Recv avant le début de l'envoi. */

int rank = MPI::Get_size() ;

if(rank == 0) {
    MPI::COMM_WORLD.Ssend(&message, 1, MPI::INT, 1, tag);
    cout << "Le processus 0 a envoyé " << message << " au processus 1" << endl;
}
else {
    cout << "Le processus 1 est en attente d'un message." << endl;
    MPI::COMM_WORLD.Recv(&messageReceived, 1, MPI::INT, 0, tag);
    cout << "Le processus 1 a reçu " << message << endl;
}
```

Source 5 : Envoi synchrone bloquant et réception bloquante

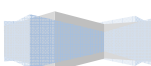
Le processus récepteur choisit d'attendre un message du processus 0 de façon bloquante.

Voici, pour chaque processus, la trace obtenue lors de l'exécution du code avec un message ayant pour valeur 2 :

```
Processus 0 :   Le processus 0 a envoyé 2 au processus 1
Processus 1 :   Le processus 1 est en attente d'un message.
                Le processus 1 a reçu 2.
```

Source 6 : Trace envoi synchrone bloquant et réception bloquante

Le schéma suivant illustre l'échange du message :



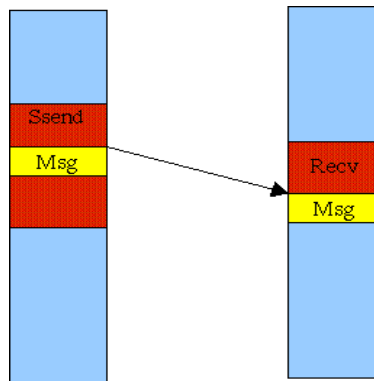


Figure 20 : Envoi synchrone bloquant et réception bloquante

ii. Envoi synchrone bloquant et réception non bloquante

La fonction de réception non bloquante est `Irecv`. Ici, deux cas sont possibles :

- Le processus récepteur exécute la fonction `Irecv` avant que le processus émetteur n'exécute `Ssend` :
Le processus émetteur peut envoyer son message dès l'exécution de la fonction `Send` car le processus récepteur s'est d'ores et déjà déclaré prêt à recevoir, et inversement la réception du message ne se fait pas nécessairement lors de l'exécution de la fonction `Irecv`.
- Le processus récepteur exécute la fonction `Irecv` après que le processus émetteur exécute `Ssend` :
Dans ce cas, le processus émetteur est en attente d'une réception de la part du récepteur et l'exécution de son programme stagne. Lorsque le processus récepteur exécute la fonction `Irecv`, le processus émetteur est débloqué et envoie le message qui est alors réceptionné par le récepteur.

Le code suivant est un exemple d'utilisation de ces fonctions :

```

/* Le processus émetteur (0) envoie l'entier « message » au processus récepteur (1) de façon
synchrone bloquante, c'est-à-dire qu'il attend que le processus récepteur exécute Irecv avant
le début de l'envoi. */

int rank = MPI::Get_size() ;

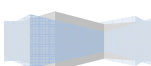
if(rank == 0) {
    MPI::COMM_WORLD.Ssend(&message, 1, MPI::INT, 1, tag);
    cout << "Le processus 0 a envoyé " << message << " au processus 1" << endl;
}
else {
    cout << "Le processus 1 est en attente d'un message." << endl;
    MPI_Request request = MPI::COMM_WORLD.Irecv(&messageReceived, 1, MPI::INT, 0, tag);

    //autres instructions

    MPI_Wait(&request, MPI_STATUS_IGNORE);
    cout << "Le processus 1 a reçu " << message << endl;
}

```

Source 7 : Envoi synchrone bloquant et réception non bloquante



Le processus récepteur choisit d'attendre un message du processus 0 de façon non bloquante. Ce type de réception doit nécessairement être suivi d'un appel à la fonction MPI_Wait() qui permet d'attendre que l'on ait bien reçu un message.

Voici, pour chaque processus, la trace obtenue lors de l'exécution du code avec un message ayant pour valeur 2 :

```
Processus 1 : Le processus 1 est en attente d'un message.
              Le processus 1 a reçu 2
Processus 0 : Le processus 0 a envoyé 2 au processus 1
```

Source 8 : Trace envoi synchrone bloquant et réception non bloquante

Le schéma suivant illustre l'échange du message :

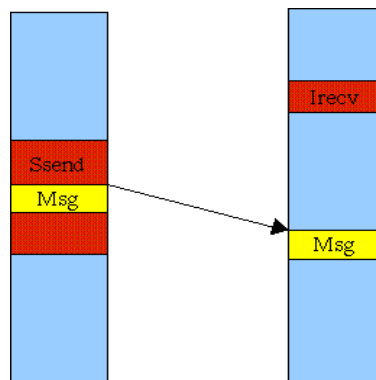


Figure 21 : Envoi synchrone bloquant et réception non bloquante

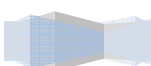
iii. Envoi synchrone non bloquant et réception bloquante

L'émission ne peut se terminer que si le récepteur a commencé l'exécution d'une réception. La fonction d'envoi synchrone non bloquant est lssend.

On a également ici deux scénarios possibles :

- L'émetteur exécute sa fonction d'envoi avant que le récepteur n'exécute sa fonction de réception :
Le message ne sera transmis que lorsque le récepteur sera disponible. Il se passe donc un délai entre l'exécution de la fonction d'émission et l'envoi du message, mais ce processus ne se bloque pas et continue l'exécution de son programme. Par contre, le récepteur n'est pas en attente, et le message est reçu dès l'exécution de la fonction de réception.
- L'émetteur l'exécute après le récepteur :
Le récepteur est bloqué en attente de la réception d'un message, il ne peut pas continuer l'exécution de son programme et ne sera débloqué que lorsque l'émetteur enverra le message. Ce dernier envoie le message dès l'exécution de sa commande d'envoi.

Le code suivant est un exemple d'utilisation de ces fonctions :



```

/* Le processus émetteur (0) envoie l'entier « message » au processus récepteur (1) de façon
synchrone non bloquante, c'est-à-dire qu'il n'attend pas que le processus récepteur exécute
Recv avant le début de l'envoi. */

int rank = MPI::Get_size() ;

if(rank == 0) {
    MPI::COMM_WORLD.Issend(&message, 1, MPI::INT, 1, tag);
    cout << "Le processus 0 a envoyé " << message << " au processus 1" << endl;
}
else {
    cout << "Le processus 1 est en attente d'un message." << endl;
    MPI::COMM_WORLD.Recv(&messageReceived, 0, MPI::INT, 0, tag);
    cout << "Le processus 1 a reçu " << message << endl;
}

```

Source 9 : Envoi synchrone non bloquant et réception bloquante

Voici, pour chaque processus, la trace obtenue lors de l'exécution du code avec un message ayant pour valeur 2 :

```

Processus 0 : Le processus 0 a envoyé 2 au processus 1
Processus 1 : Le processus 1 est en attente d'un message.
                Le processus 1 a reçu 2

```

Source 10 : Trace envoi synchrone non bloquant et réception bloquante

Le schéma suivant illustre l'échange du message :

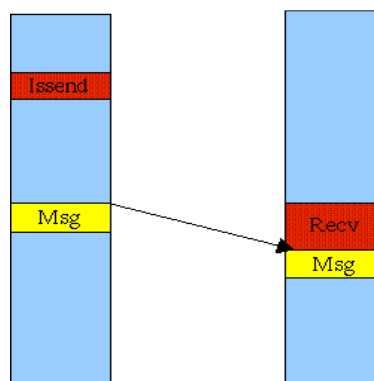


Figure 22 : Envoi synchrone non bloquant et réception bloquante

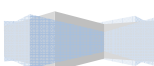
d. Envoi de messages en mode asynchrone

Ce mode de transmission fait appel, en plus des traditionnels processus émetteur et récepteur, à une mémoire tampon utilisée dans le canal de communication dans laquelle un message sera stocké lors d'un éventuel envoi. Dès qu'un processus désire recevoir un message, il va le chercher dans cette mémoire tampon, le message est ensuite consommé par le processus récepteur, la mémoire tampon est donc désormais disponible pour accueillir un nouveau message.

Les fonctions fournies par la bibliothèque MPI pour l'émission asynchrone bloquante et non bloquante sont respectivement Send et Isend. Ce sont les mêmes méthodes de réception que pour la récupération d'un message envoyé de façon synchrone qui sont utilisées.

Le programme suivant illustre un envoi asynchrone non bloquant et une réception bloquante :

Les concepts par l'exemple



```

/* Le processus émetteur (0) envoie l'entier « message » au processus récepteur (1) de façon
asynchrone, c'est-à-dire qu'il attend que le processus récepteur exécute Recv avant le début
de l'envoi. */

int rank = MPI::Get_size() ;

if(rank == 0) {
    MPI::COMM_WORLD.Isend(&message, 1, MPI::INT, 1, tag);
    cout << "Le processus 0 a envoyé " << message << " au processus 1" << endl;
}
else {
    cout << "Le processus 1 est en attente d'un message." << endl;
    MPI::COMM_WORLD.Recv(&messageReceived, 1, MPI::INT, 0, tag);
    cout << "Le processus 1 a reçu " << message << endl;
}
}

```

Source 11 : Envoi asynchrone non bloquant et réception bloquante

Voici, pour chaque processus, la trace obtenue lors de l'exécution du code avec un message ayant pour valeur 2 :

```

Processus 0 : Le processus 0 a envoyé 2 au processus 1
Processus 1 : Le processus 1 est en attente d'un message.
               Le processus 1 a reçu 2

```

Source 12 : Trace envoi asynchrone non bloquant et réception bloquante

Le schéma suivant illustre l'échange du message :

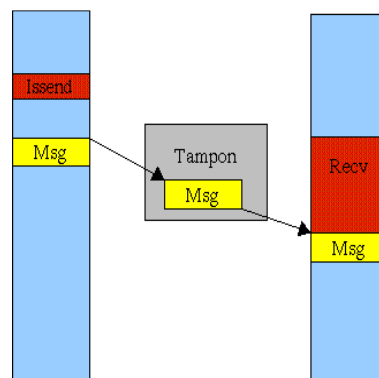


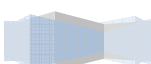
Figure 23 : Envoi asynchrone non bloquant et réception bloquante

3. Le rendez-vous

MPI fournit une fonction permettant à des processus de « s'attendre » lors de l'exécution d'un programme. Les processus seront donc bloqués sur cette fonction tant qu'ils ne l'auront pas tous exécutés.

Dans le code présenté ci-dessous, le processus 0 attend pendant 5 secondes avant d'afficher son identifiant ; les autres l'affichent sans attente. Tous les processus sont ensuite bloqués sur la directive *Barrier* avant de pouvoir se terminer.

En exécutant ce code avec trois processus par exemple, les traces d'exécution indiquent que les processus 1 et 2 peuvent afficher leur identifiant sans attente, néanmoins, ils ne se terminent pas.



Ils sont donc bloqués sur la fonction *Barrier*. Le processus 0 affiche lui son identifiant 5 secondes plus tard. Tous les processus sont alors débloqués et se terminent.

Voici la source de ce programme :

```
//declaration des variables
int rank, size;

//initialisation de MPI
MPI::Init();

//récupération de l'identifiant du processus
rank = MPI::COMM_WORLD.Get_rank();

//récupération du nombre de processus
size = MPI::COMM_WORLD.Get_size();

if (rank == 0) {
    //attente de 5 secondes pour le processus 0
    sleep(5);
    cout << "Je suis le processus " << rank << endl;
} else {
    cout << "Je suis le processus " << rank << endl;
}

//attente que tous les processus soient arrivés ici
MPI::COMM_WORLD.Barrier();
cout << "Le processus " << rank << " est arrivé." << endl;

MPI::Finalize();
```

Source 13 : Le rendez-vous

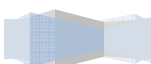
Voici la trace d'exécution de ce programme pour cinq processus (ici, afin d'observer le résultat, il est préférable de lancer le programme via l'interpréteur de commandes grâce à *mpirun -np 5 prog*) :

```
Je suis le processus 1
Je suis le processus 2
Je suis le processus 4
Je suis le processus 3
Je suis le processus 0
Le processus 0 est arrivé.
Le processus 1 est arrivé.
Le processus 4 est arrivé.
Le processus 3 est arrivé.
Le processus 2 est arrivé.
```

Source 14 : Trace du rendez-vous

Afin d'illustrer la fonction de *Barrier*, voici la trace obtenue en supprimant cette instruction :

```
Je suis le processus 1
Le processus 1 est arrivé.
Je suis le processus 2
Le processus 2 est arrivé.
Je suis le processus 4
Le processus 4 est arrivé.
Je suis le processus 3
Le processus 3 est arrivé.
Je suis le processus 0
```



```
Le processus 0 est arrivé.
```

Source 15 : Trace du rendez-vous sans *Barrier*

4. La diffusion

L'application ci-dessous illustre le principe de la diffusion.

Le processus 0, ou processus source, émet l'entier 1 à tous les processus, lui y compris, grâce à la fonction *Bcast*. Chaque processus affiche ensuite la valeur reçue. Ceci illustre donc une émission « un vers plusieurs ».

Puis les processus s'attendent grâce à un rendez-vous. Enfin, ils renvoient tous la valeur qu'ils ont précédemment reçue au processus source grâce à la fonction *Gather*. Ceci illustre une émission « plusieurs vers un ».

Une fois les valeurs reçues, le processus source les affiche.

```
//declaration des variables
int rank, size;
int source = 0;
int msg = 0;

MPI::Init();

rank = MPI::COMM_WORLD.Get_rank();
size = MPI::COMM_WORLD.Get_size();

// tableau dans lequel seront stockées les valeurs reçues
// après diffusion et récupération.
int res[size];

if (rank == source) msg = 1;

// Envoi en diffusion du contenu de msg.
MPI::COMM_WORLD.Bcast(&msg, 1, MPI::INT, source);

// Affichage de la valeur reçue par chaque processus.
cout << "Le processus " << rank << " a reçu " << msg << endl;

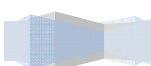
// Cette instruction est utilisée pour s'assurer que les processus
// aient tous affiché la valeur reçue avant de poursuivre.
// C'est uniquement pour regrouper les affichages dans la trace
// d'exécution.
MPI::COMM_WORLD.Barrier();

// Récupération des valeurs de chaque processus et stockage de chacune
// d'elles dans le tableau res du processus source.
MPI::COMM_WORLD.Gather(&msg, 1, MPI::INT, res, 1, MPI::INT, source);

// Affichage des valeurs stockées dans le tableau res pour s'assurer
// que le programme s'est déroulé correctement.
if (rank == source) {
    out << "Le processus 0 a reçu ";
    for (int i = 0; i < size; i++)
        cout << res[i] << " ";
    cout << endl;
}

MPI::Finalize();
```

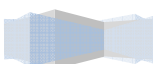
Source 16 : La diffusion



Voici la trace d'exécution pour quatre processus :

```
Processus 1 : Le processus 1 a reçu 1
Processus 2 : Le processus 2 a reçu 1
Processus 3 : Le processus 3 a reçu 1
Processus 0 : Le processus 0 a reçu 1
               Le processus 0 a reçu 1 1 1 1
```

Source 17 : Trace de la diffusion



Cas d'utilisation : le supercalculateur

Dans cette dernière partie, un programme complet et pouvant avoir une réelle utilité sera expliqué. Lui aussi est développé en C++, mais contrairement aux exemples précédents, l'aspect objet sera utilisé. Le code source complet de ce programme se trouve sur le CD-ROM.

1. Présentation de l'application

Ce programme a pour but d'effectuer des calculs de façon répartie. Ici, les calculs seront restreints aux opérations de base que sont l'addition, la multiplication, la soustraction et la division. Il sera néanmoins très facile de le modifier pour effectuer des calculs bien plus compliqués.

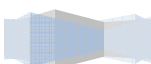
L'intérêt réside dans le fait qu'une ressource de calcul ne sait effectuer qu'une seule opération. Le raisonnement est le suivant : une ressource pouvant être un processeur spécifique ne sachant effectuer de façon optimisée qu'un seul type d'opération, en parallélisant plusieurs processeurs dont les opérations spécifiques sont différentes, il est possible d'obtenir une palette d'opérateurs formant un système de calcul à hautes performances.

Cette application est donc constituée d'opérateurs, mais aussi d'un ou plusieurs clients et d'un médium. Les clients sont des programmes ayant des calculs à réaliser. Ces calculs seront répartis sur les opérateurs. Enfin, le médium permet aux clients de connaître l'adresse d'un opérateur. En effet, ce système pouvant être utilisé de manière dynamique, les clients ne peuvent connaître les adresses des opérateurs en avance.

2. Fonctionnement

Les trois éléments que sont les opérateurs, les clients et le médium ont un cycle de vie différent ; le voici pour chacun d'eux :

- Les clients :
 1. Acquérir un calcul à effectuer,
 2. Demander au médium l'adresse (l'identifiant MPI) de l'opérateur permettant de réaliser le calcul,
 3. Envoyer les données de calcul à l'opérateur concerné,
 4. Acquérir le résultat et en disposer.
- Les opérateurs :
 1. S'enregistrer auprès du médium,
 2. Attendre des valeurs d'un client pour effectuer l'opération,
 3. Effectuer l'opération,
 4. Envoyer le résultat au client,
 5. Recommencer à la seconde étape.
- Le médium :
 1. Attendre une requête d'un opérateur souhaitant s'enregistrer ou d'un client souhaitant connaître l'identifiant d'un opérateur,
 2. Enregistrer l'identifiant de l'opérateur souhaitant s'enregistrer ou répondre au client,
 3. Recommencer à la première étape.



Voici un schéma illustrant ce fonctionnement :

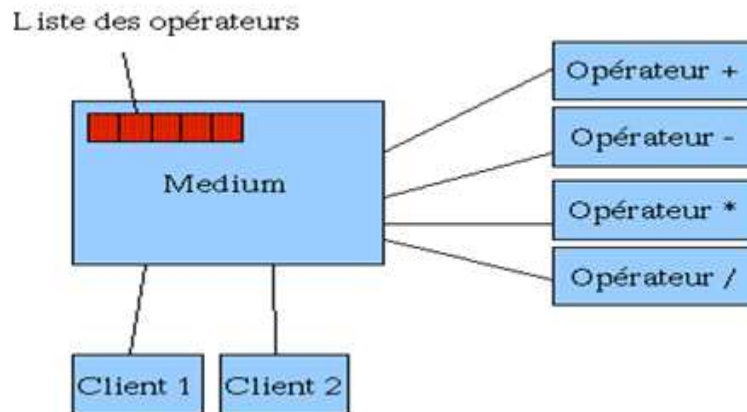


Figure 24 : Fonctionnement du supercalculateur

3. Vu de l'intérieur

Pour une meilleure compréhension du code, les trois éléments ont été répartis en trois classes :

- Client, contenue dans les fichiers Client.cpp et Client.h, est la classe qui contient le code d'un client. Voici la source commentée du fichier Client.cpp :

```
#include "Client.h"
#include "mpi.h"
#include <time.h>
#include <vector>

using namespace std;
using namespace MPI;

Client::Client() {

    //resultat de l'operation et identifiant de l'operateur
    int result=1, idOperateur;

    //tableau des valeurs à calculer
    int values[2];

    //operation voulue
    char* message;

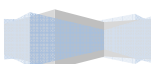
    srand(time(NULL));

    //choix de l'opérateur
    strcpy(message, "*");
    //strcpy(message, "+");
    //strcpy(message, "/");
    //strcpy(message, "-");

    // choix aléatoire de valeur à adjoindre à l'opérateur
    values[0] = (int)((double)rand() / ((double)RAND_MAX + 1) * 20) + 1;
    values[1] = (int)((double)rand() / ((double)RAND_MAX + 1) * 20) + 1;

    // Demande au médium l'id de l'opérateur dont le nom est contenu dans message.
    COMM_WORLD.Ssend(message, strlen(message)+1, MPI::CHAR, MEDIUM_ID, CLIENT_TAG_ASK_ID);

    // Affichage des entiers générés par le client.
```



```

cout << "Envoi des nombres " << values[0] << " et " << values[1] << endl;
// Affichage de l'opération choisie.
cout << "Opération choisie : " << message << endl;

// Réception de l'id de l'opérateur.
COMM_WORLD.Recv(&idOperateur, 1, MPI::INT, MEDIUM_ID, MEDIUM_TAG);

// Affichage de l'id de l'opérateur reçu.
cout << "Id operateur : " << idOperateur << endl;

// Envoi à l'opérateur les valeurs à traiter.
MPI::COMM_WORLD.Issend(values, 2, MPI::INT, idOperateur, CLIENT_TAG_ASK_ID);

// Réception du résultat.
MPI::COMM_WORLD.Recv(&result, 1, MPI::INT, idOperateur, OPERATOR_TAG);

// Affichage du résultat reçu.
cout << "Recu : " << result << endl;
}

/* Destructeur */
Client::~Client() {
    exit(0);
}

```

Source 18 : Client.cpp

- Operator, contenue dans les fichiers Operator.cpp et Operator.h, est une classe abstraite ; elle ne peut donc être instanciée. Lors de l'écriture d'un nouvel opérateur, il suffira donc d'écrire une classe dérivant d'Operator et de rajouter la méthode de calcul qui y est associé pour avoir un nouvel opérateur. Ceci simplifie grandement l'écriture d'un nouvel opérateur, puisque seule une fonction de calcul a besoin d'être écrite. Pour l'exemple ci-dessus, quatre classes ont été créées : OperatorAdd (+), OperatorSub (-), OperatorMul (*), OperatorDiv (/). Voici tout d'abord la source de l'opérateur *OperatorSub* :

```

#include "OperatorSub.h"

/* Constructeur */
OperatorSub::OperatorSub() {}

/* Destructeur */
OperatorSub::~OperatorSub() {}

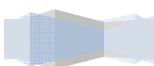
/* Retourne l'opérateur de l'opération effectuée */
/* Methode hérité de la classe abstraite Operator */
char OperatorSub::getOperation() {
    return '-';
}

/* Soustraction de deux entiers */
/* Methode hérité de la classe abstraite Operator */
int OperatorSub::compute(int nb1, int nb2) {
    return nb1 - nb2;
}

```

Source 19 : OperatorSub.cpp

C'est la méthode *compute*, commune à tous les opérateurs, qui effectue le traitement. Ecrire un nouvel opérateur revient donc à changer le corps de cette fonction et à donner un autre caractère de retour à la fonction *getOperation*. Voici maintenant le code de la classe mère des opérateurs ; Operator :



```

#include "Operator.h"
#include "Global.h"

using namespace std;

/* Constructeur */
Operator::Operator() {}

/* Destructeur */
Operator::~Operator() {}

/* Fonction permettant la réception des valeurs des clients,
 * leur traitement, et le renvoi du résultat. */
void Operator::ListenAllClients() {
    // Tableau de réception accueillant deux entiers
    int values[2];

    //resultat et id du client
    int result, idClient;

    //status pour connaitre la source
    MPI::Status status;

    while(1) {
        // Attend la demande d'une opération.
        MPI::COMM_WORLD.Recv(values, 2, MPI::INT, MPI_ANY_SOURCE, CLIENT_TAG_ASK_ID, status);
        cout << "Valeurs recues de " << status.Get_source();

        // Récupère l'id du client.
        idClient = status.Get_source();

        // Effectue l'opération.
        result = compute(values[0],values[1]);

        // Retourne le résultat à la source (non bloquant).
        MPI::COMM_WORLD.Issend(&result, 1, MPI::INT, idClient, OPERATOR_TAG);
    }
}

void Operator::start() {
    char op[255];
    op[0] = this -> getOperation();
    op[1] = '\0';

    //inscription auprès du medium
    MPI::COMM_WORLD.Ssend(&op, 255, MPI::CHAR, 0, OPERATOR_TAG);

    //attente des clients
    this->ListenAllClients();
}

```

Source 20 : Operator.cpp

Après avoir été instancié, un opérateur doit être lancé grâce à sa méthode *start*.

- Medium, contenue dans les fichiers Medium.cpp et Medium.h, est la classe qui contient le code du médium. Voici la source commentée du fichier Medium.cpp :

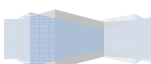
```

#include <string>
#include "Medium.h"
#include "mpi.h"

using namespace std;

/* Constructeur */
Medium::Medium() {
    this->start();
}

```



```

}

/* Methode qui lance l'ecoute */
void Medium::start() {
    this->listenEveryProcess();
}

/* Ajout d'un opérateur dans la map des opérateurs connus du médium */
void Medium::addOperator(string name, int number) {
    this -> operateurs[name] = number;
}

/* Récupération de l'id du processus dont le nom est en paramètre */
int Medium::getOperatorId(string name) {
    return this -> operateurs[name];
}

/* Réception des messages des processus autres que le médium.
 * Origine du message :
 * - Opérateur : ajout de l'opérateur à la liste des opérateurs connus.
 * - Client : envoi de l'id de l'opérateur recherché au client. */
void Medium::listenEveryProcess() {
    char message[255];
    int id = -1;
    MPI::Status status;

    while(true) {
        // Recoit un message d'un opérateur ou d'un client
        MPI::COMM_WORLD.Recv(message, 255, MPI::CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG, status);

        switch (status.Get_tag()) {
            // Si c'est un opérateur
            case OPERATOR_TAG:
                // Associe l'id de l'opérateur a son "opération"
                this->addOperator(message, status.Get_source());
                break;
            // Si c'est un client qui demande l'id d'un opérateur
            case CLIENT_TAG_ASK_ID:
                // Récupère l'id de l'opérateur exécutant l'opération recherchée
                id = this->getOperatorId(message);
                // Renvoie l'id au client (non bloquant)
                MPI::COMM_WORLD.Issend(&id, 1, MPI::INT, status.Get_source(),
MEDIUM_TAG);
                break;
            default :
                break;
        }
    }
}
}

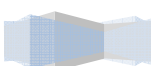
```

Source 21 : Medium.cpp

Un nouveau type *MPI::Status* est utilisé ; il permet, lors d'une réception, de connaître le tag et l'identifiant de l'émetteur. En effet, la fonction *Recv* est utilisée avec les variables globales *MPI_ANY_SOURCE* et *MPI_ANY_TAG*, il fallait donc un moyen d'identifier la source et le but du message : est-ce un opérateur souhaitant s'inscrire ou est-ce un client souhaitant connaître l'adresse d'un opérateur ?

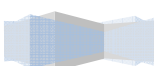
Deux autres fichiers sont nécessaires au bon fonctionnement :

- *Global.h* : ce fichier contient des variables globales utilisées dans les fichiers précédents telles que *CLIENT_TAG_ASK_ID* ou *OPERATOR_TAG*.
- *Main.cpp* : celui-ci contient la méthode principale, appelée lors du lancement des différents processus.



4. Conclusion sur le supercalculateur

Ce programme concluant cet ouvrage concernant MPI montre à quel point la programmation parallèle est facilitée par cette librairie. Si le supercalculateur n'a pas de réel intérêt en l'état, sa structure permet de le faire évoluer facilement vers une application très concrète et très performante.



Conclusion

Cet ouvrage permet à tout programmeur d'appréhender la programmation parallèle d'une manière relativement simple et efficace.

En effet, aujourd'hui MPI-2 est très abouti et permet d'effectuer tout ce que les modèles de parallélisme proposent. Sa structure basée sur l'envoi de messages permet d'élaborer des applications fonctionnant sur tout type d'architecture, s'abstrayant totalement de l'aspect réseau mais aussi des systèmes d'exploitation et des types de processeurs.

Enfin, grâce à Eclipse et PTP, le développement avec MPI-2 devient réellement facile à utiliser et à maintenir.

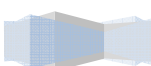


Table des illustrations

Figure 1 : Système à mémoire partagée	5
Figure 2 : Système à mémoire répartie.....	5
Figure 3 : Autocomplétion avec Eclipse	9
Figure 4 : Perspective d'exécution	9
Figure 5 : Perspective de débogage	10
Figure 6 : Architecture de PTP (Source : http://www.computer.org).....	10
Figure 7 : Graphe de dépendance des composants.....	11
Figure 8 : Configuration, chemin de SDM	14
Figure 9 : Configuration, chemin du proxy.....	15
Figure 10 : Configuration, sources MPI.....	15
Figure 11 : Nouveau projet c++	16
Figure 12 : Préférences MPI.....	17
Figure 13 : Préférences du projet.....	17
Figure 14 : Création du fichier source	18
Figure 15 : Profil d'exécution	19
Figure 16 : Options d'exécution, choix de l'exécutable	19
Figure 17 : Options d'exécution, choix du nombre de processus	20
Figure 18 : Runtime Perspective	20
Figure 19 : Légende des icônes des processus.....	21
Figure 20 : Envoi synchrone bloquant et réception bloquante	25
Figure 21 : Envoi synchrone bloquant et réception non bloquante	26
Figure 22 : Envoi synchrone non bloquant et réception bloquante	27
Figure 23 : Envoi asynchrone non bloquant et réception bloquante	28
Figure 24 : Fonctionnement du supercalculateur	33

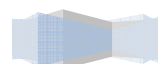
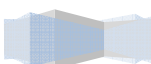


Table des sources

Source 1 : Script lancement d'éclipse	14
Source 2 : Hello World MPI	18
Source 3 : Squelette d'un programme MPI.....	22
Source 4 : Arguments des fonctions Send et Recv.....	23
Source 5 : Envoi synchrone bloquant et réception bloquante	24
Source 6 : Trace envoi synchrone bloquant et réception bloquante.....	24
Source 7 : Envoi synchrone bloquant et réception non bloquante	25
Source 8 : Trace envoi synchrone bloquant et réception non bloquante	26
Source 9 : Envoi synchrone non bloquant et réception bloquante	27
Source 10 : Trace envoi synchrone non bloquant et réception bloquante	27
Source 11 : Envoi asynchrone non bloquant et réception bloquante	28
Source 12 : Trace envoi asynchrone non bloquant et réception bloquante	28
Source 13 : Le rendez-vous	29
Source 14 : Trace du rendez-vous.....	29
Source 15 : Trace du rendez-vous sans <i>Barrier</i>	30
Source 16 : La diffusion	30
Source 17 : Trace de la diffusion	31
Source 18 : Client.cpp	34
Source 19 : OperatorSub.cpp	34
Source 20 : Operator.cpp	35
Source 21 : Medium.cpp	36



Glossaire

Eclipse : EDI pour le développement en JAVA, dont les nombreux plugins permettent de l'utiliser pour beaucoup d'autres langages.

EDI : Environnement de Développement Intégré : Outil permettant d'écrire, de compiler, d'exécuter et parfois de déboguer un programme.

G++ : Compilateur GNU pour C++.

GJ : Machine virtuelle java du projet GNU.

Medium : En parallélisme, programme garantissant le transfert des informations.

MPI : « Message Passing Interface », norme de programmation parallèle par envoi de messages.

Ordonnanceur : Système gérant la répartition du temps à attribuer aux processus actifs.

OpenMPI : OpenMPI est un ensemble de fonctions respectant la norme MPI-2, permettant d'écrire des programmes parallèles en s'abstrayant complètement de la couche réseau.

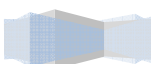
PLDT : Plugin d'Eclipse offrant une aide syntaxique pour la rédaction de programmes parallèles.

Processus : Entité informatique représentant l'exécution d'une succession d'instructions.

Proxy : Programme permettant à Eclipse de communiquer avec les processus en cours d'exécution.

PTP : Plugin d'Eclipse permettant d'écrire, d'exécuter et de déboguer des programmes parallèles.

SDM : « Scalable debug manager », partie du plugin PTP permettant l'interfaçage entre le débogueur d'Eclipse et les processus.



Bibliographie

- www.open-mpi.org : Site officiel d'OpenMPI.
- www.eclipse.org : Site officiel d'Eclipse.
- www.eclipse.org/ptp : Page du site officiel d'Eclipse présentant le plugin PTP.
- www.eclipse.org/cdt : Page du site officiel d'Eclipse présentant le plugin CDT.

Sites présentant les fonctions OpenMPI :

- www.msi.umn.edu/tutorial/scicomp/general/MPI/communicator.html
- www.idris.fr/data/cours/parallel/mpi/mpi1_aide_memoire_C.html

Sites d'aide à la programmation en C++ :

- www.developpez.com/c
- www.siteduzero.com

