Beowulf HOWTO

Jacek Radajewski and Douglas Eadline (traduction : Emmanuel PIERRE, epierre@e-nef.com) v1.1.1, 22 Novembre 1998

Ce document est une introduction à l'architecture Beowulf Supercomputeur. Il fournit les informations de base sur la programmation parallèle, et inclut des liens vers des documents plus spécifiques et des pages web.

Table des matières

1 Préambule

1.1 Mise en garde

Nous n'accepterons aucune responsabilité pour toute information incorrecte présente dans ce document, ni pour aucun des dommages qui pourraient en résulter.

1.2 Copyright

1.3 Au sujet de ce HOWTO

Jacek Radajewski a commencé à travailler sur ce document en novembre 1997 et a été ensuite rejoint par Douglas Eadline. En quelques mois, le HOWTO Beowulf est devenu un document consistant, et en août 1998, il a été découpé en trois: Beowulf HOWTO, Beowulf Architecture Design HOWTO, the Beowulf Installation and Administration HOWTO. La Version 1.0.0 de ce document a été soumise au Linux Documentation Project le 11 novembre 1998. Nous espérons que ce ne soit que le début de ce qui deviendra une documentation complète du Projet de Documentation Beowulf (Beowulf Documentation Project).

1.4 Au sujet des auteurs

– Jacek Radajewski est Administrateur Réseau, et prépare un degré honorifique en Informatique à l'Université du Southern Queensland, Australie. Le premier contact de Jacek avec Linux eut lieu en 1995 et il en tomba amoureux du premier coup. Jacek construisit son premier cluster Beowulf en Mai 1997 et a joué avec cette technologie depuis, toujours à la recherche de la meilleure manière de tout organiser. Vous pouvez joindre Jacek par courriel à *jacek@usq.edu.au*

- Douglas Eadline, Ph.D. est le President et le Principal Scientifique (Principal Scientist) à Paralogic, Inc., Bethlehem, PA, USA. Formé en tant que Chimiste Physique/Analytique, il s'est investi dans les ordinateurs depuis 1978, année où il a construit sa première machine pour l'utiliser avec l'instrumentation chimique. A ujourd'hui, le Dr. Eadline s'intéresse à Linux, aux clusters Beowulf, et aux algorithmes parallèles. Le Dr. Eadline peut être joint par courriel à *deadline@plogic.com*

1.5 Remerciements

L'écriture du HOWTO Beowulf a été longue, et il est finalement complet grâce à de nombreuses personnes. Nous voudrions remercier celles qui suivent pour leur aide et leurs contributions à ce HOWTO:

- Becky pour son amour, son soutien, et sa compréhension.
- Tom Sterling, Don Becker, et les autres personnes de la NASA qui furent à l'origine du projet Beowulf.
- Thanh Tran-Cong et la Faculty of Engineering and Surveying pour avoir donné la machine Beowulf *topcat* pour les tests de Beowulf.
- Mon supérieur Christopher Vance pour de nombreuses bonnes idées.
- Mon ami Russell Waldron pour de grandes idées de programmation, son intérêt général pour le projet, et son soutien.
- Mon ami David Smith pour la relecture de ce document.
- Et de nombreuses autres personnes sur la liste de diffusion Beowulf qui nous ont fournis beaucoup de retour et d'idées.
- Toutes les personnes qui sont responsables du système d'exploitation Linux et de tous les autres outils gratuits utilisés sur *topcat* et les diverses machines Beowulf.

2 Introduction

Au fur et à mesure que les niveaux de performance et de commodité des ordinateurs et des réseaux augmentent, il devient de plus en plus facile de construire des systèmes informatiques parallèles à partir de composants facilement disponibles, plutôt que de construire des processeurs sur de très coûteux Superordinateurs. En fait, le rapport prix/performances d'une machine de type Beowulf est de trois à dix fois meilleur que celui des superordinateurs traditionnels. L'architecture Beowulf s'échelonne bien, elle est facile à construire et vous ne payez que pour le matériel, puisque la pluspart des logiciels sont gratuits.

2.1 A qui s'adresse ce HOWTO?

Ce HOWTO s'adresse aux personnes qui ont déjà eu au moins des contacts avec le système d'exploitation Linux. La connaissance de la technologie Beowulf ou d'un système d'exploitation plus complexe et de concepts réseaux n'est pas essentielle, mais des aperçus de la programmation parallèle sont bienvenus (après tout, vous devez avoir de bonnes raisons de lire ce document). Ce HOWTO ne répondra pas à toutes les questions que vous pourriez vous poser au sujet de Beowulf, mais, espérons-le, vous donnera des idées et vous guidera dans la bonne direction. Le but de ce HOWTO est de fournir des informations de base, des liens et des références vers des documents plus approfondis.

2.2 Qu'est-ce que Beowulf?

Famed was this Beowulf: far flew the boast of him, son of Scyld, in the Scandian lands. So becomes it a youth to quit him well with his father's friends, by fee and gift, that to aid him, aged, in after days, come warriors willing, should war draw nigh, liegemen loyal: by lauded deeds shall an earl have honor in every clan. Beowulf est le poème épique le plus ancien en Anglais qui ait été conservé. C'est l'histoire d'un héros d'une grande force et d'un grand courage qui a défait un monstre appelé Grendel. Voir l'5.7 (Historique) pour en savoir plus sur le héros Beowulf.

Il y a peut-être de nombreuses définitions de Beowulf, autant que de personnes qui construisent ou utilisent des Superordinateurs Beowulf. Certains disent qu'ils peuvent appeler leur système Beowulf seulement s'il est construit de la même façon que la machine d'origine de la NASA. D'autres vont à l'extrême inverse et appellent ainsi n'importe quel système de stations qui exécutent du code parallèle. Ma définition d'un Beowulf se situe entre ces deux avis, et est fondée sur de nombreuses contributions dans la liste de diffusion Beowulf.

Beowulf est une architecture multi-ordinateurs qui peut être utilisée pour la programmation parallèle. Ce système comporte habituellement un noeud serveur, et un ou plusieurs noeuds clients connectés entre eux à travers Ethernet ou tout autre réseau. C'est un système construit en utilisant des composants matériels existants, comme tout PC capable de faire tourner Linux, des adaptateurs Ethernet standards, et des switches. Il ne contient aucun composant matériel propre et est aisément reproductible. Beowulf utilise aussi des éléments comme le système d'exploitation Linux, Parallel VirtualMachine (PVM) et Message Passing Interface (MPI). Le noeud serveur contrôle l'ensemble du cluster et sert de serveur de fichiers pour les noeuds clients. Il est aussi la console du cluster et la passerelle (gateway) vers le monde extérieur. De grandes machines Beowulf peuvent avoir plus d'un noeud serveur, et éventuellement aussi d'autres noeuds dédiés à des tâches particulières, par exemple comme consoles ou stations de surveillance. Dans de nombreux cas, les noeuds clients d'un système Beowulf sont idiots (dumb): plus ils sont idiots, mieux ils sont. Les noeuds sont configurés et contrôlés par le noeud serveur, et ne font que ce qu'on leur demande de faire. Dans une configuration client sans disque (diskless), les noeuds clients ne connaissent même pas leur adresse IP ou leur nom jusqu'à ce que le serveur leur dise qui ils sont. Une des principales différences entre Beowulf et un Cluster de Stations de travail (COW) est le fait que Beowulf se comporte plus comme une simple machine plutôt que comme plusieurs stations de travail. Dans de nombreux cas, les noeuds clients n'ont pas de claviers ni de moniteurs, et on n'y accède que par une connection distante ou par un terminal série. Les noeux Beowulf peuvent être envisagés comme un CPU + des ensembles de mémoires qui peuvent être branchés dans le cluster, exactement comme un CPU ou un module mémoire peut être branché dans une carte mère.

Beowulf n'est pas un ensemble de matériels spécialisés, une nouvelle topologie réseau ou le dernier hack du kernel. Beowulf est une technologie de clustering d'ordinateurs Linux pour former un superordinateur parallèle, virtuel. Même s'il y a de nombreux paquetages comme des patches du noyau, PVM, les librairies MPI, et des outils de configuration qui rendent l'architecture Beowulf plus rapide, plus facile à configurer, et plus facilement utilisable, on peut construire une machine de classe Beowulf en utilisant une distribution Standard de Linux sans ajouter d'autres logiciels. Si vous avez deux Linux en réseau qui partagent au moins le même système de fichier `racine` via NFS, et qui se font confiance pour exécuter des sessions distantes (rsh), alors on peut dire que vous avez un simple Beowulf de deux noeuds.

2.3 Classification

Les systèmes Beowulf ont été construits à partir de nombreux constituants. Pour des considérations de performances, des composants moins communs (i.e. produits par un seul fabricant) ont été utilisés. Afin de recenser les différents types de systèmes et de rendre les discussions au sujet des machines un peu plus faciles, nous proposons la méthode simple de classification suivante:

CLASSE I BEOWULF:

Cette classe concerne des machines faites d'éléments globalement disponibles. Nous devrons utiliser les tests de certification "Computer Shopper" pour définir les composants d'assemblage. ("Computer Shopper" est un mensuel sur les PC et leurs composants.) [NdT: US seulement ; pour un équivalent, on peut évoquer par exemple "PC Direct".] Le test est le suivant:

Un Beowulf CLASSE I est une machine qui peut être assemblée à partir de pièces trouvées dans au moins quatre journaux de publicité de grande diffusion.

Les avantages des systèmes de CLASS I sont:

- le matériel est disponible de noubreuses sources (faible coût, maintenance facile)
- ne dépendant pas d'un seul vendeur de matériel
- support des drivers par les commodités Linux
- basé habituellement sur des standards (SCSI, Ethernet, etc.)

Les désavantages d'un système de CLASSE I sont:

- de meilleures performances peuvent nécessiter du matériel de CLASSE II

CLASSE II BEOWULF

Un Beowulf CLASSE II Beowulf est simplement une machine qui ne passe pas le test de certification "Computer Shopper". Ce n'est pas une mauvaise chose. D'autre part, il s'agit plutôt d'une classification de la machine.

Les avantages d'un système de CLASSE II sont:

- les performances peuvent être assez bonnes !

Les désavantages des systèmes de CLASSE II sont:

- le support des drivers peut varier
- reposent sur un seul vendeur de matériel
- peuvent être plus chers que les systèmes de CLASSE I.

Une CLASSE n'est pas nécessairement meilleure qu'une autre. Cela dépend surtout de vos besoins et de votre budget. Cette classification des systèmes sert seulement à rendre les discussions sur les systèmes Beowulf un peu plus succintes. La "Conception du Système" peut aider à déterminer quelle sorte de système est le plus approprié à vos besoins.

3 Aperçu de l'Architecture

3.1 A quoi cela ressemble-t-il?

Je pense que la meilleure façon de décrire l'architecture d'un superordinateur Beowulf est d'utiliser un exemple qui est très proche du vrai Beowulf, mais aussi familier à beaucoup d'administrateurs systèmes. L'exemple le plus proche d'une machine Beowulf est un Unix de laboratoire avec un serveur et un certain nombre de clients. Pour être plus spécifique, j'utiliserai le DEC Alpha au laboratoire d'informatique de la Faculté des Sciences de l'USQ comme exemple. Le serveur est appelé *beldin* et les machines clientes sont *scilab01*, *scilab02*, *scilab03*, jusqu'à *scilab20*. Tous les clients ont une copie locale du système d'exploitation Digital Unix 4.0 installé, mais ont l'espace disque utilisateur (`/home`) et `/usr/local` du serveur via NFS (Network File System). Chaque client a une entrée pour le serveur et tous les autres clients dans son fichier `/etc/hosts.equiv`: ainsi tous les clients peuvent exécuter une cession distante (rsh) vers tout autre. La machine serveur est un serveur NIS pour tout le laboratoire, ainsi les informations des comptes sont les mêmes sur toutes les machines. Une personne peut s'asseoir à la console de *scilab02*, se logue, et a le même environnement que s'il était logué sur le serveur, ou *scilab15*. La raison pour laquelle les clients ont la même présentation est que le système d'exploitation est installé et configuré de la même façon sur toutes les machines, les espaces `/home` et `/usr/local` sont physiquement sur le même serveur et les clients y accèdent via NFS. Pour plus d'informations sur NIS et NFS, reportez-vous à *NIS* et *NFS*.

3.2 Comment utiliser les autres noeuds?

Maintenant que nous avons une vision correcte de l'architecture du système, regardons comment nous pouvons utiliser les cycles CPU des machines dans le laboratoire. Toute personne peut se loguer sur n'importe laquelle des machines, et lancer un programme dans son répertoire de base, mais peut aussi éclater la même tâche sur différentes machines simplement en exécutant un shell distant. Par exemple, si nous voulons calculer la somme des racines carrées de tous les entiers inclus strictement entre 1 et 10, nous écrivons un simple programme appelé `sigmasqrt` (voir 6.2 (code source)) qui fait cela exactement. Pour calculer la somme des racines carrées des nombres de 1 à 10, nous exécutons :

```
[jacek@beldin sigmasqrt]$ time ./sigmasqrt 1 10
22.468278

real    0m0.029s
user    0m0.001s
sys     0m0.024s
```

La commande `time` nous permet de vérifier le temps mis en exécutant cette tâche. Comme nous pouvons le voir, cet exemple a pris seulement une petite fraction de seconde (0.029 sec) pour s'exécuter, mais que se passe-t-il si je veux ajouter la racine carrée des entiers de 1 à 1 000 000 000? Essayons ceci, et calculons le temps écoulé:

```
[jacek@beldin sigmasqrt]$ time ./sigmasqrt 1 1000000000
21081851083600.559000

real    16m45.937s
user    16m43.527s
sys     0m0.108s
```

Cette fois, le temps d'exécution de ce programme est considérablement supérieur. La question évidente qui se pose est: est-il possible de diminuer le temps d'exécution de cette tâche et comment? La réponse évidente est de découper la tâche en un ensemble de sous-tâches et d'exécuter ces sous-tâches en parallèle sur tous les ordinateurs. Nous pouvons séparer la grande tâche d'addition en 20 parties en calculant un intervalle de racines carrées et en les additionnant sur un seul noeud. Quand tous les noeuds ont fini les calculs et retournent leurs résultats, les 20 nombres peuvent être additionnés ensemble et fournir la solution finale. Avant de lancer ce processus, nous allons créer un "named pipe" qui sera utilisé par tous les processus pour écrire leurs résultats:

```
[jacek@beldin sigmasqrt]$ mkfifo output
[jacek@beldin sigmasqrt]$ ./prun.sh & time cat output | ./sum
[1] 5085
21081851083600.941000
[1]+  Done                    ./prun.sh

real    0m58.539s
user    0m0.061s
sys     0m0.206s
```

Cette fois, cela prend 58.5 secondes. C'est le temps qui a été nécessaire entre le démarrage du processus et le moment où les noeuds ont fini leurs calculs et écrit leurs résultats dans la pipe. Ce temps n'inclut pas l'addition finale des 20 nombres, mais il représente une petite fraction de seconde et peut être ignoré. Nous pouvons voir qu'il y a un avantage significatif à exécuter une tâche en parallèle. En fait la tâche en parallèle s'est exécutée 17 fois plus vite, ce qui est très raisonnable pour un facteur 20 d'augmentation du nombre de CPU. Le but de l'exemple précédent est d'illustrer la méthode la plus simple de paralléliser du code concurrent. En pratique, des exemples aussi simples sont rares et différentes techniques (les API de PVM et PMI) sont utilisées pour obtenir le parallélisme.

3.3 En quoi un Beowulf diffère-t-il d'un COW ?

Le laboraroire d'informatique décrit plus haut est un exemple parfait d'un cluster de stations (COW). Qu'est-ce qui rend donc Beowulf si spécial et en quoi diffère-t-il d'un COW ? En réalité il n'y a pas beaucoup de différence, mais un Beowulf a quelques caractéristiques uniques. La première est que dans la plupart des cas, les noeuds clients dans un cluster Beowulf n'ont pas de clavier, de souris, de carte graphique ni de moniteur. Tous les accès aux noeuds clients sont faits par une connection distante du noeud serveur, un noeud dédié à une console, ou une console série. Cela parce qu'il n'y a aucun besoin pour un noeud client d'accéder à des machines en dehors du cluster, ni pour des machines en dehors du cluster d'accéder à des noeuds clients directement; c'est une pratique habituelle que les noeuds clients utilisent des adresses IP privées comme les plages d'adresses 10.0.0.0/8 ou 192.168.0.0/16 (RFC 1918 *http://www.alternic.net/rfcs/1900/rfc1918.txt.html*). D'habitude la seule machine qui est aussi connectée au monde externe en utilisant une seconde carte réseau est le noeud serveur. La façon la plus habituelle d'accéder au système est soit d'utiliser la console du serveur directement, soit de faire un telnet ou un login distant (rlogin) sur le noeud serveur d'une station personnelle. Une fois sur celui-ci, les utilisateurs peuvent éditer et compiler leur code, et aussi distribuer les tâches sur tous les noeuds du cluster. Dans la plupart des cas, les COW sont utilisées pour des calculs parallèles la nuit et les week-ends, quand les stations ne sont pas utilisées pendant les journées de travail, utilisant ainsi les périodes de cycles libres des CPU. D'autre part, le Beowulf est une machine dédiée au calcul parallèle, et optimisée pour cette tâche. Il donne aussi un meilleur rapport prix/performance puisqu'il est constitué de composants grand public et qu'il tourne principalement à partir de logiciels libres. Beowulf donne aussi davantage l'image d'une seule machine, ce qui permet aux utilisateurs de voir le cluster Beowulf comme une seule station de calcul.

4 Conception du Système

Avant d'acheter du matériel, il serait de bon aloi de considérer le design de votre système. Il y a deux approches matérielles qui sont impliquées dans le design d'un système Beowulf: le type de noeuds ou d'ordinateurs que vous allez utiliser, et la méthode que vous allez utiliser pour vous connecter aux noeuds d'ordinateurs. Il n'y a qu'une seule approche logicielle qui puisse affecter votre choix matériel: la librairie de communication ou API. Une discussion plus détaillée sur le matériel et les logiciels de communication est fournie plus loin dans ce document.

Alors que le nombre de choix n'est pas grand, il y a des considérations de conception qui doivent être prises pour la construction d'un cluster Beowulf. La science (ou art) de la "programmation parallèle" étant l'objet de nombreuses interprétations, une introduction est fournie plus bas. Si vous ne voulez pas lire les connaissances de base, vous pouvez survoler cette section, mais nous vous conseillons de lire la section 4.4 (Convenance) avant tout choix défnnitif de matériel.

4.1 Brefs rappels sur la programmation parallèle

Cette section fournit des informations générales sur les concepts de la programmation parallèle. Ceci n'est PAS exhaustif, ce n'est pas une description complète de la programmation parallèle ou de sa technologie. C'est une brève description des enjeux qui peuvent influer fortement sur le concepteur d'un Beowulf, ou sur son utilisateur.

Lorsque vous déciderez de construire votre Beowulf, de nombreux points décrits plus bas deviendront importants dans votre processus de choix. A cause de la nature de ses "composants", un Superordinateur Beowulf nécessite de prendre de nombreux facteurs en compte, car maintenant ils dépendent de nous. En général, il n'est pas du tout difficile de comprendre les objectifs impliqués dans la programmation parallèle. D'ailleurs, une fois que ces objectifs sont compris, vos attentes seront plus réalistes, et le succès plus probable. Contrairement au "monde séquentiel", où la vitesse du processeur est considérée comme le seul facteur important, la vitesse des processeurs dans le "monde parallèle" n'est que l'un des paramètres qui détermineront les performances et l'efficacité du système dans son ensemble.

4.2 Les méthodes de programmation parallèle

La programmation parallèle peut prendre plusieurs formes. Du point de vue de l'utilisateur, il est important de tenir compte des avantages et inconvénients de chaque méthodologie. La section suivante tente de fournir quelques aperçus sur les méthodes de programmation parallèle et indique où la machine Beowulf fait défaut dans ce continuum.

4.2.1 Pourquoi plus d'un CPU?

Répondre à cette question est important. Utiliser 8 CPU pour lancer un traitement de texte sonne comme "trop inutile" – et ce l'est. Et qu'en est-il pour un serveur web, une base de données, un programme de ray-tracing, ou un planificateur de projets? Peut-être plus de CPU peuvent-ils améliorer les performances. Mais qu'en est-il de simulations plus complexes, de la dynamique des fluides, ou d'une application de Fouille de Données (Data Mining)? Des CPU supplémentaires sont absolument nécessaires dans ces situations. D'ailleurs, de multiples CPU sont utilisés pour résoudre de plus en plus de problèmes.

La question suivante est habituellement: "Pourquoi ai-je besoin de deux ou quatre CPU? Je n'ai qu'à attendre le méga super rapide processeur 986." Il y a de nombreuses raisons:

1. Avec l'utilisation de systèmes d'exploitations multi-tâches, il est possible de faire plusieurs choses en même temps. Cela est un "parallélisme" naturel qui est exploité par plus d'un CPU de bas prix.
2. La vitesse des processeurs double tous les 18 mois mais qu'en est-il de la vitesse de la mémoire? Malheureusement, celle-ci n'augmente pas aussi vite que celle des processeurs. Gardez à l'esprit que beaucoup d'applications ont besoin de mémoire autre que celle du cache processeur et de l'accès disque. Faire les choses en parallèle est une façon de contourner ces limitations.
3. Les prédictions indiquent que la vitesse des processeurs ne continuera pas à doubler tous les 18 mois après l'an 2005. Il y a divers obstacles à surmonter pour maintenir ce rythme.
4. Suivant l'application, la programmation parallèle peut accélérer les choses de 2 à 500 fois (et même plus dans certains cas). De telles performances ne sont pas disponibles sur un seul processeur. Même les Superordinateurs qui utilisaient à un moment un seul processeur spécialisé très rapide sont maintenant constitués de nombreux CPU plus banals.

Si vous avez besoin de vitesse – à cause d'un problème lié au calcul et/ou aux entrées/sorties –, il vaut la peine de considérer l'approche parallèle. Comme le calcul parallèle est implémenté selon de nombreuses voies, résoudre votre problème en parallèle nécessitera de prendre quelques décisions importantes. Ces décisions peuvent affecter dramatiquement la protabilité, la performance, et le coût de votre application.

Avant d'être par trop technique, regardons un vrai "problème de calcul parallèle" en utilisant un exemple qui nous est familier: faire la queue à une caisse.

4.2.2 La "caisse" en programmation parallèle

Considérons un grand magasin avec 8 caisses regroupées devant le magasin. Imaginons que chaque caisse est un CPU et chaque client un programme informatique. La taille du programme (quantité de calcul) est la taille de la commande de chaque client. Les analogies suivantes peuvent être utilisées pour illustrer les concepts de la programmation parallèle:

Systèmes d'exploitation Mono-Tâche: Une caisse ouverte (et en service) qui ne peut traiter qu'un client à la fois.

Exemple en Informatique : MS DOS

Systèmes d'exploitation Multi-Tâches: Une caisse ouverte, mais maintenant nous pouvons traiter une partie de chaque commande à un instant donné, aller à la personne suivante et traiter une partie de sa commande. Tout le monde "semble" avancer dans la queue en même temps, mais s'il n'y a personne dans la queue, vous serez servi plus vite.

Exemple en Informatique : UNIX, NT avec un seul CPU

Systèmes d'exploitation Multi-Tâches avec plusieurs CPU: Maintenant on ouvre plusieurs caisses dans le magasin. Chaque commande peut être traitée par une caisse différente et la queue peut avancer plus vite. Ceci est appelé SMP - Gestion Multiple Symétrique (Symmetric Multi-processing). Même s'il y a plus de caisses ouvertes, vous n'avancerez pas plus vite dans la queue que s'il n'y avait qu'une seule caisse.

Exemple en Informatique : UNIX, NT avec plusieurs CPU

Sous-tâches (Threads) sur les autres CPU d'un Système d'exploitation Multi-Tâches: Si vous "séparez" les objets de votre commande, vous pouvez être capable d'avancer plus vite en utilisant plusieurs caisses en même temps. D'abord, nous postulons que vous achetez une grande quantité d'objets, parce que le temps que vous investirez pour "séparer" votre commande doit être regagné en utilisant plusieurs caisses. En théorie, vous devriez être capables de vous déplacer dans la queue "n" fois plus vite qu'avant, où "n" est le nombre de caisses. Quand les caissiers ont besoin de faire des sous-totaux, ils peuvent échanger rapidement les informations visuellement et en discutant avec toutes les autres caisses "locales". Ils peuvent aussi aller chercher directement dans les registres des autres caisses pour trouver les informations dont ils ont besoin pour travailler plus vite. La limite étant le nombre de caisses qu'un magasin peut effectivement installer.

La loi de Amdals montre que l'accélération de l'application est liée à la portion séquentielle la plus lente exécutée par le programme (NdT: i.e. majorée par la tâche la plus lente).

Exemple en Informatique : UNIX ou NT avec plusieurs CPU sur la même carte-mère avec des programmes multi-threads.

Envoyer des messages sur des Systèmes d'exploitation Multi-Tâches avec plusieurs CPU: De façon à améliorer la performance, la Direction ajoute 8 caisses à l'arrière du magasin. Puisque les nouvelles caisses sont loin du devant du magasin, les caissiers doivent téléphoner pour envoyer leurs sous-totaux vers celui-ci. La distance ajoute un délai supplémentaire (en temps) dans la communication entre caissiers, mais si la communication est minimisée, cela ne pose pas de problème. Si vous avez vraiment une grosse commande, une qui nécessite toutes les caisses, alors comme avant votre vitesse peut être améliorée en utilisant toutes les caisses en même temps, le temps sopplémentaire devant être pris en compte. Dans certains cas, le magasin peut n'avoir que des caisses (ou des îlots de caisses) localisés dans tout le magasin : chaque caisse (ou îlot) doit communiquer par téléphone. Puisque tous les caissiers peuvent discutter par téléphone, leur emplacement importe peu.

Exemple en Informatique : Une ou plusieurs copies d'UNIX ou NT avec plusieurs CPU sur la même, ou différentes cartes-mères communiquant par messages.

Les scénarios précédents, même s'ils ne sont pas exacts, sont une bonne représentation des contraintes qui agissent sur les systèmes parallèles. Contrairement aux machines avec un seul CPU (ou caisse), la communication est importante.

4.3 Architectures pour le calcul parallèle

Les méthodes et architectures habituelles de la programmation parallèle sont représentées ci-dessous. Même si cette description n'est en aucun cas exhaustive, elle est suffisante pour comprendre les impératifs de base dans la conception d'un Beowulf.

4.3.1 Architectures Matérielles

Il y a typiquement deux façons d'assembler un ordinateur parallèle:

1. La mémoire locale des machines qui communiquent par messages (Clusters Beowulf)
2. Les machines à mémoire partagée qui communiquent à travers la mémoire (machines SMP)

Un Beowulf typique est une collection de machines mono-processeurs connectées utilisant un réseau Ethernet rapide, et qui est ainsi une machine à mémoire locale. Une machine à 4 voies SMP est une machine à mémoire partagée et peut être utilisée pour du calcul parallèle – les applications parallèles communiquant via la mémoire partagée. Comme pour l'analogie du grand magasin, les machines à mémoire locale (donc à caisse individuelle) peuvent être scalairisées jusqu'à un grand nombre de CPU ; en revanche, le nombre de CPU que les machines à mémoire partagée peuvent avoir (le nombre de caisses que vous pouvez placer en un seul endroit) peut se trouver limité à cause de l'utilisation (et/ou de la vitesse) de la mémoire.

Il est toutefois possible de connecter des machines à mémoire partagée pour créer une machine à mémoire partagée "hybride". Ces machines hybrides "ressemblent" à une grosse machine SMP pour l'utilisateur et sont souvent appelées des machines NUMA (accès mémoire non uniforme) parce que la mémoire globale vue par le programmeur et partagée par tous les CPU peut avoir différents temps d'accès. A un certain niveau d'ailleurs, une machine NUMA doit "passer des messages" entre les groupes de mémoires partagées.

Il est aussi possible de connecter des machines SMP en tant que noeuds de mémoire locale. Typiquement, les cartes-mères de CLASSE I ont soit 2 ou 4 CPU et sont souvent utilisées comme moyens pour réduire le coût global du système. L'arrangeur (scheduler) interne de Linux détermine combien de ces CPU sont partagés. L'utilisateur ne peut (à ce jour) affecter une tâche à un processeur SMP spécifique. Cet utilisateur peut quand même démarrer deux processus indépendants ou un programme multi-threads et s'attendre à voir une amélioration de performance par rapport à un système à simple CPU.

4.3.2 Architectures Logicielles et API

Il y a basiquement deux façons d'"exprimer" la concurrence dans un programme:

1. En envoyant des Messages entre les processeurs
2. En utilisant les threads du système d'exploitation (natives)

D'autres méthodes existent, mais celles-là sont le plus généralement employées. Il est important de se souvenir que l'expression de concurrence n'est pas nécessairement contrôlée par la couche matérielle. Les Messages et les Threads peuvent être implémentés sur des SMPn NUMA-SMP, et clusters – même si, comme expliqué ci-dessous, l'efficacité et la portabilité sont des facteurs importants.

Messages Historiquement, la technologie de passage de messages reflétait les débuts des ordinateurs parallèles à mémoire locale. Les messages nécessitent la copie des données tandis que les Threads utilisent des données à la place. Le temps de latence et la vitesse à laquelle les messages peuvent être copiés sont les facteurs limitants des modèles de passage de messages. Un message est assez simple: des données et un processeur de destination. Des API de passage de messages répandues sont entre autres *PVM* ou *MPI*. Le passage de Messages peut être implémenté avec efficacité en utilisant ensemble des Threads et des Messages entre SMP et machines en cluster. L'avantage d'utiliser les messages sur une machine SMP, par rapport aux Threads, est que si vous décidez d'utiliser des clusters dans le futur, il est facile d'ajouter des machines ou de scalairiser vos applications.

Threads Les Threads ont été développés sur les systèmes d'exploitation parce que la mémoire partagée des SMP (moutiprocessorage symmétrique) permettait une communication très rapide et une synchronisation de la mémoire partagée entre les parties d'un programme. Les Threads marchent bien sur les systèmes SMP parce que la communication a lieu à travers la mémoire partagée. Pour cette raison, l'utilisateur doit isoler les données locales des données globales, sinon les programmes ne fonctionneront pas correctement. Cela est en contraste avec les messages: une grande quantité de copie peut être éliminée avec les threads car les données sont partagées entre les processus (threads). Linux implémente les Threads POSIX. Le problème avec les Threads vient du fait qu'il est difficile de les étendre au-delà d'une machine SMP, et, comme les données sont partagées entre les CPU, la gestion de la cohérence du cache peut contribuer à le charger. Etendre les Threads au-delà des limites des performances des SMP nécessite la technologie NUMA qui est chère et n'est pas nativement supportée par Linux. Implémenter des Threads par dessus les messages a été fait (*(http://syntron.com/ptools/ptools_pg.htm)*), mais les Threads sont souvent inefficients une fois implémentés en utilisant des messages.

On peut résumer ainsi les performances:

```
         performance       performance
         machine SMP     cluster de machines  scalabilité
         -----------     -------------------  -----------
messages    bonne             meilleure        meilleure

threads   meilleure           mauvaise*        mauvaise*

* nécessite une technologie NUMA coûteuse.
```

4.3.3 Architecture des Applications

Pour exécuter une application en parallèle sur des CPU multiples, celle-ci doit être explicitement découpée en parties concurrentes. Une application standard mono-CPU ne s'exécutera pas plus rapidement même si elle est exécutée sur une machine multi-processeurs. Certains outils et compilateurs peuvent découper les programmesn mais la parallélisation n'est pas une opération "plug and play". Suivant l'application, la parallélisation peut être facile, extrêmement difficile, voire impossible suivant les contraintes de l'algorithme.

Avant de parler des besoins applicatifs, il nous faut introduire le concept de Convenance (Suitability).

4.4 Convenance

Beaucoup de questions au sujet du calcul parallèle ont la même réponse:

"Cela dépend entièrement de l'application."

Avant de passer directement aux opportunités, il y a une distinction très importante qui doit être faite: la différence entre CONCURRENT et PARALLELE. Pour clarifier cette discussion, nous allons définir ces deux termes ainsi:

les parties CONCURRENTES d'un programme sont celles qui peuvent être calculées indépendamment.

Les parties PARALLELES d'un programme sont celles qui sont exécutées sur des éléments de calculs au même moment.

La distinction est très importante, parce que la CONCURRENCE est une propriété d'un programme et l'efficacité en PARALLELISME est une propriété de la machine. Idéalement, l'exécution en parallèle doit produire des performances plus grandes. Le facteur limitant les performances en parallèle est la vitesse de communication et le temps de latence entre les noeuds de calcul. (Le temps de latence existe aussi dans les applications TMP threadées à cause de la cohérence du cache). De nombreux tests de performances communs sont hautement parallèles, et ainsi la communication et le temps de latence ne sont pas les points importants. Ce type de problème peut être appelé "évidemment parallèle". D'autres applications ne sont pas si simples et exécuter des parties CONCURRENTES du programme en PARALLELE peut faire en sorte que le programme fonctionne plus lentement, et ainsi décaler toute performance de gain dans d'autres parties CONCURRENTES du programme. En termes plus simples, le coût en temps de communication doit en pâtir au profit de celui gagné en temps de calcul, sinon l'exécution PARALLELE des parties CONCURRENTES est inefficace.

La tâche du programmeur est de déterminer quelles parties CONCURRENTES le programmeur DOIT exécuter en PARALLELE et pour quelles parties il NE DOIT PAS le faire. Sa réponse déterminera l'EFFICACITE de l'application. Le graphe suivant résume la situation pour le programmeur:

```
        | *
        | *
        | *
% des   | *
appli-  |  *
cations |  *
        |  *
        |  *
        |    *
        |     *
        |      *
        |        ****
        |            ****
        |                ********************
        +------------------------------------
         temps de communication/temps de calcul
```

Dans un ordinateur parallèle parfait, le rapport communication/calcul devrait être égal et tout ce qui est CONCURRENT pourrait être implémenté en PARALLELE. Malheureusement, les vrais ordinateurs parallèles, incluant les machines à mémoire partagée, sont sujets aux effets décrits dans ce graphe. En concevant un Beowulf, l'utilisateur devrait garder celui-ci en tête parce que la performance dépend du rapport entre le temps de communication et le temps de calcul pour un ORDINATEUR PARALLELE SPECIFIQUE. Les applications peuvent être portables entre les ordinateurs parallèles, mais il n'y a aucune garantie qu'elles seront efficaces sur une plateforme différente.

EN GENERAL, IL N'EXISTE PAS DE PROGRAMME PORTABLE EFFICACE EN PARALLELE

Il y a encore une autre conséquence au graphe précédent. Puisque l'efficacité dépend du rapport communication/calcul, changer juste un composant du rapport ne signifie pas nécessairement qu'une application s'exécutera plus rapidement. Un changement de vitesse processeur, en gardant la même vitesse de communication, peut avoir des effets inattendus sur votre programme. Par exemple, doubler ou tripler la vitesse du processeur, en gardant la même vitesse de communication, peut maintenant rendre des parties de votre programme qui sont efficaces en PARALLELE, plus efficaces si elles étaient exécutées SEQUENTIELLEMENT. Cela dit, il se peut qu'il soit plus rapide maintenant d'exécuter les parties qui étaient avant PARALLELES en tant que SEQUENTIELLES. D'autant plus qu'exécuter des parties inefficaces en PARALLELE empêchera votre application d'atteindre sa vitesse maximale. Ainsi, en ajoutant un processeur plus rapide, vous avez peut-être ralenti votre application (vous enpêchez votre nouveau CPU de fonctionner à sa vitesse maximale pour cette application).

UPGRADER VERS UN CPU PLUS RAPIDE PEUT REELLEMENT RALENTIR VOTRE APPLICATION

Donc, en conclusion, pour savoir si oui ou non vous pouvez utiliser un environnement matériel parallèle, vous devez avoir un bon aperçu des capacités d'une machine particulière pour votre application. Vous devez tenir compte de beaucoup de facteurs: vitesse de la CPU, compilateur, API de passage de messages, réseau... Notez que se contenter d'optimiser une application ne donne pas toutes les informations. Vous pouvez isoler une lourde partie de calcul de votre programme, mais ne pas connaître son coût au niveau de la communication. Il se peut que pour un certain système, le coût de communication ne rende pas efficace de paralléliser ce code.

Une note finale sur une erreur commune: on dit souvent qu'"un programme est PARALLELISE", mais en réalité seules les parties CONCURRENTES ont été identifiées. Pour toutes les raisons précédentes, le programme n'est pas PARALLELISE. Une PARALLELISATION efficace est une propriété de la machine.

4.5 Ecrire et porter des logiciels parallèles

A partir du mmoment où vous avez décidé de concevoir et de construire un Beowulf, considérer un instant votre application en accord avec les observations précédentes est une bonne idée.

En général, vous pouvez faire deux choses:

1. Y aller et construire un Beowulf CLASSE I et après y ajuster votre application. Ou exécuter des applications parallèles que vous savez fonctionner sur votre Beowulf (mais attention à la portabilité et à l'efficacité en accord avec les informations citées ci-dessus).
2. Examiner les applications dont vous avez besoin sur votre Beowulf, et faire une estimation quant au type de matériel et de logiciels qu'il vous faut.

Dans chaque cas, vous devrez considérer les besoins en efficacité. En général, il y a trois choses à faire:

1. Déterminer les parties concurrentes de votre programme
2. Estimer le parallélisme efficacement
3. Décrire les parties concurrentes de votre programme

Examinons-les successivement:

4.5.1 Déterminer les parties concurrentes de votre programme

Cette étape est couvent considérée comme "paralléliser votre programme". Les décisions de parallélisation seront faites à l'étape 2. Dans cette étape, vous avez besoin de déterminer les liens et les besoins dans les données.

D'un point de vue pratique, les applications peuvent présenter deux types de concurrence: calcul (travaux numériques) et E/S (Bases de Données). Même si dans de nombreux cas, la concurrence entre calculs et E/S est orthogonale, des applications ont besoin des deux. Des outils existants peuvent faire l'analyse de la concurrence sur des applications existantes. La plupart de ces outils sont conçus pour le FORTRAN. Il y a deux raisons pour lesquelles le FORTRAN est utilisé: historiquement, la majorité des applications gourmandes en calculs numériques étaient écrites en FORTRAN et c'était donc plus facile à analyser. Si aucun de ces outils n'est disponible, alors cette étape peut être quelque peu difficile pour des applications existantes.

4.5.2 Estimer le parallélisme efficacement

Sans l'aide d'outils, cette étape peut nécessiter un cycle de tests et erreurs, ou seulement de bons vieux réflexes bien éduqués. Si vous avez une application spécifique en tête, essayez de déterminer la limite du CPU (liée au calcul) ou les limites des disques (liées aux E/S). Les spécifités de votre Beowulf peuvent beaucoup dépendre de vos besoins. Par exemple, un problème lié au calcul peut ne nécessiter qu'un petit nombre de CPU très rapides et un réseau très rapide à faible temps de latence, tandis qu'un problème lié aux E/S peut mieux travailler avec des CPU plus lents et un Ethernet rapide.

Cette recommandation arrive souvent comme une surprise pour beaucoup, la croyance habituelle étant que plus le processeur est rapide, mieux c'est. Mais cela n'est vrai que si vous avez un budget illimité: les vrais systèmes peuvent avoir des contraintes de coûts qui doivent être optimisées. Pour les problèmes liés aux E/S, il existe une loi peu connue (appelée la loi de Eadline-Dedkov) qui est assez utile:

Soient deux machines parallèles avec le même index de performance CPU cumulée, celle qui a les processeurs les plus lents (et probablement un réseau de communication interprocesseur plus lent) aura les meilleures performances pour des applications dominées par les E/S.

Même si les preuves de cette règle vont au-delà de ce document, vous pouvez trouver intéressant de lire l'article *Performance Considerations for I/O-Dominant Applications on Parallel Computers* (format Postscript 109K) *(ftp://www.plogic.com/pub/papers/exs-pap6.ps)*

Une fois que vous aurez déterminé quel type de concurrence vous avez dans votre application, vous devrez estimer à quel point elle sera efficace en parallèle. Voir la Section 5.4 (Logiciels) pour une description des outils Logiciels.

En l'absence d'outils, il vous faudra peut-être improviser votre chemin lors de cette étape. Si une boucle liée aux calculs est mesurée en minutes et que les données peuvent être transférées en secondes, alors c'est un bon candidat pour la parallélisation. Mais souvenez-vous que si vous prenez une boucle de 16 minutes et la coupez en 32 morceaux, et que vos transferts de données ont besoin de quelques secondes par partie, alors cela devient plus réduit en termes de performances. Vous atteindrez un point de retours en diminution.

4.5.3 Décrire les parties concurrentes de votre programme

Il y a plusieurs façons de décrire les parties concurrentes de votre programme:

1. L'exécution parallèle explicite
2. L'exécution parallèle implicite

La différence principale entre les deux est que le parallélisme explicite est déterminé parl'utilisateur, alors que le parallélisme implicite est déterminé par le compilateur.

Les méthodes explicites Il y a principalement des méthodes où l'utilisateur peut modifier le code source spécifique pour une machine parallèle. L'utilisateur doit soit ajouter des messages en utilisant *PVM* ou *MPI*, soit ajouter des threads POSIX. (Souvenez vous que les threads ne peuvent se déplacer entre les cartes-mères SMP).

Les méthodes explicites tendent à être les plus difficiles à implémenter et à déboguer. Les utilisateurs ajoutent typiquement des appels de fonctions dans le code source FORTRAN 77 standard ou C/C++. La librairie MPI a ajouté des fonctions pour rendre certaines méthodes parallèles plus faciles à implémenter (i.e. les fonctions scatter/gather). De plus, il est aussi possible d'ajouter des librairies standard qui ont été écrites pour des ordinateurs parallèles. Souvenez-vous quand même du compromis efficacité/portabilité.

Pour des raisons historiques, beaucoup d'applications gourmandes en calculs sont écrites en FORTRAN. Pour cette raison, FORTRAN dispose du plus grand nombres de supports pour le calcul parallèle (outils, librairies ...). De nombreux programmeurs utilisent maintenant C ou réécrivent leurs applications FORTRAN existantes en C, avec l'idée que C permettra une exécution plus rapide. Même si cela est vrai puisque C est la chose la plus proche du code machine universel, il a quelques inconvénients majeurs. L'utilisation de pointeurs en C rend la détermination des dépendances entre données et l'analyse automatique des pointeurs extrêmement difficiles. Si vous avez des applications existantes en FORTRAN et que vous voudrez les paralléliser dans le futur - NE LES CONVERTISSEZ PAS EN C !

Méthodes Implicites Les méthodes implicites sont celles dans lesquelles l'utilisateur abandonne quelques décisions de parallélisation (ou toutes) au compilateur. Par exemple le FORTRAN 90, High Performance FORTRAN (HPF), Bulk Synchronous Parallel (BSP), et toute une série de méthodes qui sont en cours de développement.

Les méthodes implicites nécessitent de la part de l'utilisateur des informations concernant la nature concurrente de leur application, mais le compilateur prendra quand même beaucoup de décicions sur la manière d'exécuter cette concurrence en parallèle. Ces méthodes procurent un niveau de portabilité et d'efficacité, mais il n'y a pas de "meilleure façon" de décrire un problème concurrent pour un ordinateur parallèle.

5 Ressources Beowulf

5.1 Points de départ

- Liste de diffusion US Beowulf. Pour s'inscrire, envoyer un courriel à *beowulf-request@cesdis.gsfc.nasa.gov* avec le mot *subscribe* dans le corps du message.
- Homepage Beowulf *http://www.beowulf.org*
- Extreme Linux *http://www.extremelinux.org*
- Extreme Linux Software pour Red Hat *http://www.redhat.com/extreme*

5.2 Documentation

- La dernière version du Beowulf HOWTO en Anglais*http://www.sci.usq.edu.au/staff/jacek/beowulf.*
- La dernière version du Beowulf HOWTO en Français*http://www.e-nef.com/linux/beowulf.*
- Construire un système Beowulf *http://www.cacr.caltech.edu/beowulf/tutorial/building.html*
- Les liens de Jacek sur Beowulf *http://www.sci.usq.edu.au/staff/jacek/beowulf.*
- Beowulf Installation and Administration HOWTO (DRAFT) *http://www.sci.usq.edu.au/staff/jacek/beowulf.*

– Linux Parallel Processing HOWTO *http://yara.ecn.purdue.edu/˜pplinux/PPHOWTO/pphowto.html*

5.3 Publications

– Chance Reschke, Thomas Sterling, Daniel Ridge, Daniel Savarese, Donald Becker, and Phillip Merkey *A Design Study of Alternative Network Topologies for the Beowulf Parallel Workstation*. Proceedings Fifth IEEE International Symposium on High Performance Distributed Computing, 1996. *http://www.beowulf.org/papers/HPDC96/hpdc96.html*

– Daniel Ridge, Donald Becker, Phillip Merkey, Thomas Sterling Becker, and Phillip Merkey. *Harnessing the Power of Parallelism in a Pile-of-PCs*. Proceedings, IEEE Aerospace, 1997. *http://www.beowulf.org/papers/AA97/aa97.ps*

– Thomas Sterling, Donald J. Becker, Daniel Savarese, Michael R. Berry, and Chance Res. *Achieving a Balanced Low-Cost Architecture for Mass Storage Management through Multiple Fast Ethernet Channels on the Beowulf Parallel Workstation*. Proceedings, International Parallel Processing Symposium, 1996. *http://www.beowulf.org/papers/IPPS96/ipps96.html*

– Donald J. Becker, Thomas Sterling, Daniel Savarese, Bruce Fryxell, Kevin Olson. *Communication Overhead for Space Science Applications on the Beowulf Parallel Workstation*. Proceedings,High Performance and Distributed Computing, 1995. *http://www.beowulf.org/papers/HPDC95/hpdc95.html*

– Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, Charles V. Packer. *BEOWULF: A PARALLEL WORKSTATION FOR SCIENTIFIC COMPUTATION*. Proceedings, International Conference on Parallel Processing, 95. *http://www.beowulf.org/papers/ICPP95/icpp95.html*

– Publications sur le site de Beowulf *http://www.beowulf.org/papers/papers.html*

5.4 Logiciels

– PVM - Parallel Virtual Machine/Machine Parallèle Virtuelle *http://www.epm.ornl.gov/pvm/pvm_home.html*

– LAM/MPI - Local Area Multicomputer / Message Passing Interface Multi-Ordinateurs locaux / Interface de Transmission de Messages *http://www.mpi.nd.edu/lam*

– BERT77 - outil de conversion FORTRAN *http://www.plogic.com/bert.html*

– logiciels Beowulf de la page du Projet Beowulf *http://beowulf.gsfc.nasa.gov/software/software.html*

– Jacek's Beowulf-outils *ftp://ftp.sci.usq.edu.au/pub/jacek/beowulf-utils*

– bWatch - logiciel de surveillance de cluster*http://www.sci.usq.edu.au/staff/jacek/bWatch*

5.5 Machines Beowulf

– Avalon consiste en 140 processeurs Alpha, 36 Go de RAM, et est probablement la machine Beowulf la plus rapide, allant à 47.7 Gflops et classée 114ème sur la liste du Top 500. *http://swift.lanl.gov/avalon/*

– Megalon-A Massively PArallel CompuTer Resource (MPACTR) consiste en 14 quadri CPU Pentium Pro 200 noeuds, et 14 Go de RAM. *http://megalon.ca.sandia.gov/description.html*

– theHIVE - Highly-parallel Integrated Virtual Environment est un autre Superordinateur Beowulf rapide. theHIVE est de 64 noeuds, une machine de 128 CPU avec un total de 4 Go de RAM. *http://newton.gsfc.nasa.gov/thehive/*

– Topcat est une machine beaucoup plus petite, constituée de 16 CPU et 1.2 Go de RAM. *http://www.sci.usq.edu.au/staff/jacek/topcat*

– MAGI cluster – c'est un très bon site avec de nombreux liens de qualité. *http://noel.feld.cvut.cz/magi/*

5.6 D'autres Sites Intéressants

– Linux SMP*http://www.linux.org.uk/SMP/title.html*

– Paralogic - Achetez un Beowulf *http://www.plogic.com*

5.7 Histoire

- Légendes - Beowulf *http://legends.dm.net/beowulf/index.html*
- Les Aventures de Beowulf *http://www.lnstar.com/literature/beowulf/beowulf.html*

6 Code Source

6.1 sum.c

```
/* Jacek Radajewski jacek@usq.edu.au */
/* 21/08/1998 */

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

int main (void) {

  double result = 0.0;
  double number = 0.0;
  char string[80];


  while (scanf("%s", string) != EOF) {

    number = atof(string);
    result = result + number;
  }

  printf("%lf\n", result);

  return 0;

}
```

6.2 sigmasqrt.c

```
/* Jacek Radajewski jacek@usq.edu.au */
/* 21/08/1998 */

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

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

  long number1, number2, counter;
  double result;

  if (argc < 3) {
    printf ("usage : %s number1 number2\n",argv[0]);
```

```
    exit(1);
  } else {
    number1 = atol (argv[1]);
    number2 = atol (argv[2]);
    result = 0.0;
  }

  for (counter = number1; counter <= number2; counter++) {
    result = result + sqrt((double)counter);
  }

  printf("%lf\n", result);

  return 0;

}
```

6.3 prun.sh

```
#!/bin/bash
# Jacek Radajewski jacek@usq.edu.au
# 21/08/1998

export SIGMASQRT=/home/staff/jacek/beowulf/HOWTO/example1/sigmasqrt

# $OUTPUT doit être un canal nommé (named pipe)
# mkfifo output

export OUTPUT=/home/staff/jacek/beowulf/HOWTO/example1/output

rsh scilab01 $SIGMASQRT         1  50000000 > $OUTPUT < /dev/null&
rsh scilab02 $SIGMASQRT  50000001 100000000 > $OUTPUT < /dev/null&
rsh scilab03 $SIGMASQRT 100000001 150000000 > $OUTPUT < /dev/null&
rsh scilab04 $SIGMASQRT 150000001 200000000 > $OUTPUT < /dev/null&
rsh scilab05 $SIGMASQRT 200000001 250000000 > $OUTPUT < /dev/null&
rsh scilab06 $SIGMASQRT 250000001 300000000 > $OUTPUT < /dev/null&
rsh scilab07 $SIGMASQRT 300000001 350000000 > $OUTPUT < /dev/null&
rsh scilab08 $SIGMASQRT 350000001 400000000 > $OUTPUT < /dev/null&
rsh scilab09 $SIGMASQRT 400000001 450000000 > $OUTPUT < /dev/null&
rsh scilab10 $SIGMASQRT 450000001 500000000 > $OUTPUT < /dev/null&
rsh scilab11 $SIGMASQRT 500000001 550000000 > $OUTPUT < /dev/null&
rsh scilab12 $SIGMASQRT 550000001 600000000 > $OUTPUT < /dev/null&
rsh scilab13 $SIGMASQRT 600000001 650000000 > $OUTPUT < /dev/null&
rsh scilab14 $SIGMASQRT 650000001 700000000 > $OUTPUT < /dev/null&
rsh scilab15 $SIGMASQRT 700000001 750000000 > $OUTPUT < /dev/null&
rsh scilab16 $SIGMASQRT 750000001 800000000 > $OUTPUT < /dev/null&
rsh scilab17 $SIGMASQRT 800000001 850000000 > $OUTPUT < /dev/null&
rsh scilab18 $SIGMASQRT 850000001 900000000 > $OUTPUT < /dev/null&
rsh scilab19 $SIGMASQRT 900000001 950000000 > $OUTPUT < /dev/null&
rsh scilab20 $SIGMASQRT 950000001 1000000000 > $OUTPUT < /dev/null&
```