Laboratoire de l'Informatique du Parallélisme

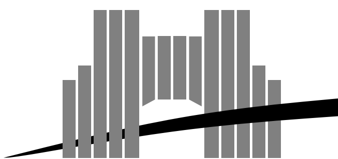
Ecole Normale Supérieure de Lyon
Institut IMAG
Unité de recherche associée au CNRS n°1398

Tests des Performances des Communications de la Machine VOLVOX IS-860

Frédéric Desprez, Cyrille Gavoille, Bruno Jargot et Makan Pourzandi

Mars 1993

Technical Report N° 93-02



Ecole Normale Supérieure de Lyon
46, Allée d'Italie, 69364 Lyon Cedex 07, France,
Téléphone : + 33 72 72 80 00; Télécopieur : + 33 72 72 80 80;
Adresses électroniques :
lip@frensl61.bitnet; lip@lip.ens−lyon.fr (uucp).

Tests des Performances des Communications de la Machine VOLVOX IS-860

Frédéric Desprez, Cyrille Gavoille, Bruno Jargot et Makan Pourzandi

Mars 1993

Abstract

In this document we present the distributed memory computer ARCHIPEL VolVox IS-860 and different programming environments TROLLIUS 2.1, VOLCOM 1.0 and VOLCOM 2.0. We measure the performances for different levels of these environments as a fonction of time and different hardware parameters.

Keywords: Archipel, VolVox IS-860, i860, T800, Trollius, VolCom

Résumé

Dans ce rapport, nous présentons une modélisation des communications de la machine à mémoire distribuée ARCHIPEL VolVox IS-860 sous les environnements de programmation TROLLIUS 2.1, VOLCOM 1.0 et VOLCOM 2.0. Les coûts de communication des différents niveaux sont modélisés comme une fonction du temps, des paramètres de la machine et de l'environnement.

Mots-clés: Archipel, VolVox IS-860, i860, T800, Trollius, VolCom

Chapitre 1

Introduction

L'étude de la complexité des algorithmes parallèles nécessite une bonne connaissance de la machine cible et de ses environnements de programmation. Une étude préalable de la machine doit donc être faite pour en extraire des modèles précis qui permettront par la suite d'étudier de la manière la plus réaliste le futur comportement des algorithmes lors de leur exécution.

Le coût le plus pénalisant sur une machine à mémoire distribuée est le coût des communications. Si celles-ci sont mal gérées, les performances d'un algorithme parallèle s'en trouvent amoindries. Cette pénalisation pouvant même aller jusqu'à l'obtention de performances moins intéressantes que celles obtenues sur une machine séquentielle.

Il convient donc de bien équilibrer les communications dans les algorithmes (et si possible de les masquer) et d'utiliser à bon escient les environnements de programmation disponibles, afin d'obtenir le plus rapidement possible les meilleures performances.

Ce rapport est découpé en 8 chapitres. Dans le premier chapitre, nous présentons rapidement les caractéristiques de la machine cible. Les deux chapitres suivants sont, quant à eux, consacrés à une présentation précise des deux environnements de programmation et des solutions choisies dans le domaine des communications. Puis, après une rapide présentation de la méthodologie de tests et du modèle de communication, nous présentons dans les deux chapitres suivants, les résultats obtenus avec les différents niveaux des environnements testés. Pour chaque niveau et fonctionnalité, nous donnons un modèle précis qui permettra au lecteur de pouvoir prévoir le comportement de son programme parallèle.

Chapitre 2

La machine VOLVOX IS-860

2.1 Architecture de la machine VolVox IS-860

La machine ARCHIPEL VolVox IS-860 est une machine à mémoire distribuée constituée de 8 à 48 nœuds de calcul (8 nœuds de calcul sur la machine testée) et de 4 nœuds d'entrée.

Chaque nœud d'entrée est un Transputer T800 utilisé pour la connexion avec l'extérieur (réseaux Internet et stations SUN).

Chaque nœud de calcul est lui-même constitué d'un processeur de calcul Intel i860 [13, 10] et d'un processeur INMOS T800 [16, 17]. A l'intérieur d'un nœud, ces processeurs sont reliés par une mémoire partagée double port (voir figure 2.1).

Tous ces nœuds sont reliés entre eux par un crossbar programmable INMOS C004 qui permet de réaliser n'importe quelle topologie de degré inférieur ou égal à quatre.

Pour de plus amples détails, se référer à la documentation Archipel [3].

2.2 Le Transputer T800

Le Transputer INMOS T800 a été, le plus rapide des processeurs 32 bits et également le premier processeur dédié à la construction de machines parallèles à mémoire distribuée.

En interne, le Transputer T800 possède une CPU et une unité flottante séparées, une unité hardware de gestion de processus, deux timers, une interface de gestion de la mémoire externe, une mémoire interne de 4 ko ainsi que 4 interfaces de communication possédant chacune 2 DMA. Ces DMA permettent au Transputer de faire des communications bidirectionnelles en parallèle sur ses quatre liens, en même temps que des calculs. Cette fonctionnalité extrêmement intéressante permet de masquer les communications par les calculs. La bande passante des liens est de 20 Mbits/s.

L'unité hardware de gestion de processus fait du Transputer un processeur multitâche très performant.

Ses performances scalaires sont de 20 Mips et de 1,5 Mflops en crête.

2.3 Le processeur Intel i860

Dans la course aux performances des processeurs, Intel a sorti l'i860 en 1990. Il a été baptisé "Cray on a chip" à son arrivée, grâce à des performances en crête dignes d'un supercalculateur (40 Mips, 80 Mflops en simple précision, 60 Mflops en double précision). Ces performances font de lui un processeur extrêmement intéressant pour des calculs intensifs.

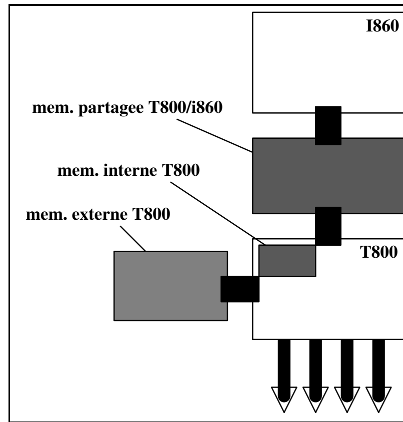


FIG. 2.1 - Architecture d'un nœud de la machine ARCHIPEL VolVox IS-860

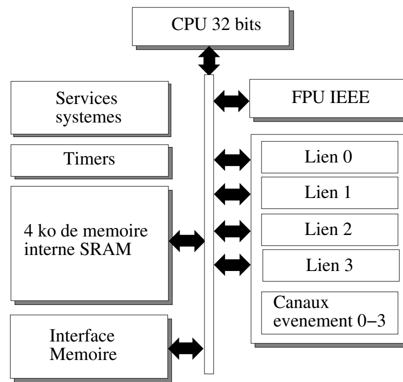


FIG. 2.2 - Architecture du Transputer T800 d'INMOS

L'architecture est de type pipeline superscalaire, ce qui lui permet de réaliser plusieurs opérations par cycle grâce à des unités parallèles [13].

Le processeur i860 constitue la brique de base de machines parallèles très performantes comme l'iPSC/860, la DELTA ou encore la Paragon. Ce processeur est également présent dans d'autres machines parallèles à mémoire distribuée, comme l'Archipel VolVox IS-860 où il est utilisé comme co-processeur de calcul et sur des cartes additionnelles (comme pour les SUNs).

Le pipeline de l'unité RISC est constitué de 4 étages [6].

L'i860 peut exécuter en crête 3 instructions par cycle. Le mode superscalaire est sélectionnable (mode double-instructions). Celui-ci reste malgré tout assez difficile à programmer, d'où des performances réelles très loin des performances en crête (5.4 Mflops en Fortran).

Ne possédant pas de liens avec l'extérieur, l'i860 doit être associé à un processeur de communication (Mesh Routing Chip (MRC) sur la Paragon), accessible par le biais d'une mémoire partagée (comme sur la machine Archipel).

L'i860 possède une unité graphique assez performante permettant d'effectuer des opérations de Z-buffer et d'élimination des parties cachées.

Contrairement au Transputer, le processeur i860 est monotâche. Cela est assez pénalisant dans le cas de machines parallèles en ce qui concerne la gestion des communications (comme pour

l'iPSC/860). C'est pour cette raison que sur la Paragon, on associe à l'i860 dédié aux calculs un autre i860 gérant les communications en liaison avec le MRC.

Il faut également noter que la difficulté de gestion des caches internes, des registres et des chemins de données ne permet pas aux compilateurs actuels de dépasser 15 Mflops, et les routines codées en assembleur atteignent avec peine la moitié de la puissance en crête [8].

Chapitre 3

L'environnement VOLCOM

3.1 Présentation

L'environnement développé sur la machine VolVox par la société Archipel fournit une plateforme conviviale et efficace pour la programmation parallèle. VolAps constitue l'ensemble des logiciels de bas niveau faisant tourner la machine VolVox. VolUse est l'environnement de programmation de la machine, il comprend VolCom, VolConf, Paragraph et VolXdr. VolCom permet les communications entre l'hôte, les nœuds de calcul et le système de gestion de fichiers. VolConf sert à configurer le réseau des processeurs, allouer les ressources et mapper automatiquement les applications sur la machine. Paragraph permet de visualiser les performances de la machine. L'utilisation de VolXdr permet d'éviter le problème de la représentation des données dans différents processeurs de l'environnement VolVox. Les routines de communications sur la machine VolVox ont été implémentées par couches. La couche supérieure hérite des propriétés de la couche inférieure. Donc, il est possible de faire appel à des fonctions *Low level* au niveau *High level* (l'inverse n'est évidemment pas possible).

Pour de plus amples renseignements sur l'environnement de programmation de la machine VolVox, se référer à la documentation Archipel [3].

3.2 Communications intra-processeur

Comme ceci a été dit dans la section 2.3 le processeur i860 est mono-tâche. Il n'y a donc pas de communications intra-processeur possible. Par contre, le T800 est un processeur multitâches. Il est possible de lancer plusieurs tâches sur un même nœud au moment du chargement des programmes, ou d'avoir une tâche mère qui crée d'autres tâches sur le même nœud. Dans ce dernier cas, les tâches créées héritent du même espace mémoire que la tâche mère. Pour créer des processus fils et les lancer en série ou en parallèle, la tâche mère appelle les procédures d'Inmos C Toolset (ProcAlloc, ProcRun, ProcPar ...). Elle a aussi la possibilité de définir des canaux entre ces processus. Les processus créés par un processus peuvent ainsi communiquer entre eux par l'intermédiaire de messages. Les communications à ce niveau sont bloquantes.

3.3 Communications inter-processeurs

Dans la machine VolVox IS-860, nous avons la possibilité de faire des communications dans un réseau de i860, de T800 ou un réseau hybride constitué de T800 et d'i860. Dans les sections

suivantes nous expliquons brièvement le fonctionnement de VolConf. Puis nous nous intéressons plus particulièrement à VolCom. La section 3.4 concernent la première version de VolUse.

3.4 VolConf et le système de routage

On ne peut pas parler des communications sans connaître un peu le système de routage. C'est pourquoi nous expliquons ce système dans cette section. Celui-ci est distribué à travers la machine. Il est constitué de différents processus appelés `Nrouter`. Ces processus collaborent pour le routage des messages. Le processus `Nserver` gérant les entrées/sorties vers le monde extérieur, se trouve sur le point d'entrée de la machine.

Chaque processeur possède un processus nommé `Nrouter`. Ce processus contrôle les entrées/sorties de chaque processeur. Les entrées/sorties des processus de chaque nœud passent obligatoirement par le `Nrouter` associé à celui-ci. Chaque processus `Nrouter` possède un tableau de routage et un tableau d'identification. Pour un système à p processeurs, le tableau de routage est constitué de p lignes et $p+1$ colonnes (une ligne et une colonne par processeur et une colonne consacrée au processus `Nserver`). Chaque case (i,j) indique le lien à utiliser pour envoyer un message de `Nrouter` i à `Nrouter` j. Ce tableau est établi au moment de l'initialisation de l'application. Le tableau d'identification, associe à chaque tâche un `Nrouter`. Chaque tâche est associée à au moins un `Nrouter`.

Prenons l'exemple où une tâche t_1 envoie un message à la tâche t_2. La tâche t_1 est associée au processus `Nrouter` i. Elle envoie le message à ce dernier. Celui-ci se réfère à sa table d'identification pour trouver j, le numéro de `Nrouter` associé à la tâche t_2. Puis il utilise la table de routage afin de trouver le lien à utiliser pour envoyer le message. Si le `Nrouter` j est un des voisins de `Nrouter` i, il recevra à ce stade le message. Sinon c'est un `Nrouter` intermédiaire k qui va recevoir le message. Celui-ci va utiliser sa table d'identification pour router le message vers le `Nrouter` j. Ainsi le message est routé au travers de la machine pour arriver au `Nrouter` j. Grâce à VolConf, le programmeur n'a plus la tâche d'écrire le fichier de configuration. Ceci élimine une source importante d'erreurs au moment de la programmation.

3.5 VolCom

VolCom regroupe l'ensemble des logiciels gérant des communications sur la machine VolVox. Ceci comprend des procédures d'envoi et de réception, des procédures d'établissement des chemins virtuels .... VolCom fournit différents niveaux de programmation : *High level*, *Intermediate level*, *Low level*.

Au niveau *High level*, VolCom facilite la programmation grâce à ses multiples procédures de communications. Mais comme toute surcouche évoluée, il implique un surcoût relativement important. La programmation au niveau *Low level* correspond à l'utilisation de l'Inmos Ansi C Toolset. Au niveau *Intermediate level*, il y a l'intégration de quelques outils logiciels simplifiant la programmation.

Les procédures de communication sont identiques pour les deux niveaux *Intermediate* et *Low level*. Donc, notre étude des performances de communications pour le niveau *Low level* est aussi valable pour le niveau *Intermediate* .

3.5.1 Niveaux Low level & Intermediate level

Ces niveaux correspondent pratiquement à la programmation à l'aide des outils fournis par l'Inmos Ansi C Toolset [11].

La surcouche logicielle à ce niveau est très faible. Ce qui nous laisse envisager des performances proches des performances matérielles. La programmation à ce niveau n'est par contre pas très conviviale. Pour communiquer entre les processus, on doit établir les canaux explicitement à l'aide des procédures prédéfinies. Ces canaux doivent être aussi définis de façon cohérente avec les possibilités matérielles de la machine. Il n'y pas de procédure ou de mécanisme de multiplexage. Cependant, il est possible d'implémenter des processus spécifiques pour une telle réalisation. Au niveau *Intermediate* l'intégration de plusieurs outils permet le mapping entre les processus logiques, les processeurs et les connexions entre les processeurs [3].

3.5.2 Niveau High level

L'environnement *High level* comprend une série d'outils logiciels. Ces outils ont pour but de simplifier la programmation et permettre l'exécution des programmes sur la machine parallèle VolVox. Le niveau *High level* fournit un environnement de programmation plus convivial que celui du *Low level*. A ce niveau, VolCom place automatiquement les tâches sur les processeurs. Le programmeur peut donc se contenter seulement de préciser les tâches sans se soucier des problèmes de communications entre elles. Ceci a l'avantage de rendre la programmation facile. VolCom n'implémente cependant pas forcément la topologie la plus adaptée. Il n'était pas possible de faire des communications non-bloquantes au niveau *High level* dans notre version de VolCom (version 1.0). Ceci rend la programmation asynchrone impossible. Ce problème a été résolu dans la version 2.0 de VolCom.

3.6 Communications à partir de l'i860

Les communications entre i860 et T800 sur un même nœud de calcul se font à travers la mémoire partagée. Dans la première version de VolCom, il était impossible d'utiliser les routines de communications pour envoyer des messages entre le T800 et l'i860. Ce problème a été résolu dans la deuxième version de VolCom. Les communications entre des i860 sur des nœuds différents passent par les T800 associés à chaque nœud. Les T800 transmettent les messages entre les nœuds et les i860 les recoivent à travers la mémoire partagée.

Chapitre 4

L'environnement TROLLIUS

Cette partie, basée sur une étude du source de Trollius, ne vise pas à présenter la syntaxe des différents niveaux de Trollius mais à en comprendre le fonctionnement interne, tout en gardant à l'esprit que le but final est d'en mesurer les performances. Pour une présentation plus générale de Trollius, on pourra se reporter à [12].

4.1 Présentation

En raison du manque d'outils de développement sur les machines multiprocesseurs à mémoire distribuée, des chercheurs américains ont décidé de développer un système d'exploitation assez souple pour pouvoir s'adapter à un large nombre de machines présentes et futures. Ce projet a débuté en 1985 au Cornell Theory Center et continue maintenant à l'Ohio State University. Le développement a été effectué sur une machine à base de transputers. Cela a fortement marqué la structure du système et, à l'heure actuelle, Trollius n'existe que sur des machines à base de transputers.

Trollius vise à constituer une fondation sur laquelle se développeraient des fonctionnalités plus étendues. De ce fait, Trollius se veut très simple. Il est bâti sur le modèle client/serveur. Utilisant les capacités multitâches hardware du T800, Trollius crée un ensemble de processus systèmes serveurs qui, sur requête du processus utilisateur, renvoient un service tel qu'une information de routage, une allocation de mémoire, ... Au cœur du système se trouve le kernel qui, contenu dans une taille extrêmement réduite, a pour rôle de mettre en rendez-vous les messages.

Trollius a été porté sur l'i860 pour les machines IS-860 d'Archipel.

Pour plus d'information sur les buts de Trollius et pour avoir un aperçu de ses services, se référer aux documents [4, 5] dont cette partie s'est fortement inspirée. Pour des renseignements plus complets, se référer à [2, 12]

4.2 In/out

Toutes les communications interprocessus de Trollius s'effectuent physiquement avec les instructions `in` et `out` du transputer, que ce soient les communications internes au transputer ou bien les communications entre deux transputers. Le processus envoyeur utilise `out`, le processus receveur utilise `in`.

La syntaxe est :

```
in(chan,buf,nbytes)
int *chan,nbytes;
char *buf;

out(chan,buf,nbytes)
int *chan,nbytes;
char *buf;
```

où:

`chan` identifie le canal de communication. Pour les communications entre deux processus sur un même transputer, il doit être initialisé comme pointeur sur une adresse en RAM. Les deux processus doivent alors utiliser la même adresse. Pour des communications entre deux processus sur deux transputers voisins, chan doit pointer sur une adresse identifiant le lien et le sens utilisés. L'autre processus doit alors pointer sur l'autre extrémité du lien dans le sens inverse. Ces adresses sont fixées par le T800.

`buf` est un pointeur sur le buffer du receveur ou envoyeur.

`nbytes` indique le nombre d'octets à transférer. Il doit être le même pour les deux processus sous peine de blocage.

Les envois et réceptions avec `in` et `out` sont bloquants.

4.3 Communications intra-transputer

Comme nous l'avons vu dans la section 4.2, deux processus désirant communiquer doivent avoir l'adresse commune d'un canal de communication. Dans ce but, ils vont effectuer tous deux une requête auprès d'un processus système appelé `kernel`. Ce processus, dont l'adresse du canal avec lequel il communique est fixée dans le source de Trollius, se contente de mettre en contact les deux processus et d'échanger leurs adresses.

Trollius propose deux types de fonctions de communications:

Fonctions bloquantes Le processus envoyant ou recevant reste bloqué tant qu'il n'a pas passé le message à un autre processus[1] (pour l'envoyeur) ou bien tant qu'il n'a pas recu de message (pour le receveur).

Fonctions non bloquantes La tentative est abandonnée si la communication ne peut pas avoir lieu immédiatement.

La différenciation se situe au niveau de la requête auprès du `kernel`. Si le processus n'y trouve pas l'autre partie et si la fonction est bloquante alors le processus est retenu par le `kernel`. Par contre, si la fonction est non bloquante alors le processus est immédiatement relaché avec un code d'erreur.

[1]...qui n'est pas obligatoirement le processus recevant

L'identification des deux processus communicants se fait par deux entiers appelés événement et type. Les deux processus doivent avoir le même événement. De plus, à moins que l'un des deux types soit nul, un ET logique entre les deux types doit avoir un résultat non nul.

En raison du coût d'une requête, Trollius offre la possibilité pour les deux parties de conserver l'adresse obtenue lors du rendez-vous. Cela permettra dorénavant une communication directe sans rendez-vous préalable chez le `kernel`. C'est ce que Trollius appelle "créer un circuit virtuel ". Le carnet d'adresse de chaque processus est limité à seize entrées. Le correspondant est identifié par les quatres bits de poids faible de l'événement utilisé lors de l'établissement du circuit virtuel.

Notons que chaque processus système peut être assimilé à l'événement sur lequel il se synchronise. Afin d'éviter toute collision avec les processus systèmes, l'utilisateur doit se restreindre à l'utilisation des événements positifs ou nuls.

4.4 Communications inter-transputer

Comme nous l'avons vu dans la section 2.2, le transputer peut faire des communications bidirectionnelles en parallèle sur ses quatres liens. L'accès en exclusion mutuelle des liens est assuré par huit processus systèmes propriétaires (quatre `dli_t4`[2] et quatre `dlo_t4`[3])

Un processus désirant utiliser un lien[4] devra d'abord faire une requête auprès du routeur de son nœud qui lui renverra l'événement correspondant au `dlo_t4` propriétaire du lien à emprunter.

Notons tout de suite que le modèle de communication est du type *store and forward*.

4.4.1 Routage

- **Le langage NaIL**[5] : La configuration de la VolVox étant statique sous Trollius, la table de routage est déterminée avant le boot lors de la compilation avec l'outil map du fichier de configuration appelé `bnail` [1]. Le minimum à spécifier est l'ensemble des nœuds utilisés ainsi que leurs connexions physiques.

  L'esprit d'initiative et la stratégie de l'outil `map` peuvent être paramétrés à loisir par l'utilisateur. Il est capable de générer un chemin de l'hôte vers chaque nœud et vice versa, et l'ensemble des chemins possibles qui n'ont pas déjà été spécifiés par l'utilisateur.

- **Routeur** : Lors du boot de Trollius, un processus système `router` est créé sur chaque transputer et initialisé avec la partie de la table de routage le concernant. Cette table est gérée en hash-table. A chaque nœud destination est associé un événement correspondant au processus propriétaire du lien à emprunter et un entier indiquant le type du nœud destination (ITB[6], OTB[7] ...)

- **La mémoire cache** : A chaque communication vers un nœud distant, il faut faire appel au processus `router` pour obtenir l'événement correspondant. Afin de minimiser l'overhead, chaque processus gère un cache. Ce cache, d'une taille de 29 entrées, est lui aussi géré en hash-table. Notons que la présence de ce cache[8] interdit tout changement de routage en cours

[2] Data Link Input
[3] Data Link Output
[4] Par lien, nous entendons un seul sens du lien physique
[5] Node and Interconnect Language
[6] In The Box
[7] Out The Box
[8] géré en FIFO

d'exécution.

4.4.2 Dli/dlo

Les processus systèmes `dli_t4` et `dlo_t4` sont les seuls processus autorisés à écrire ou lire dans leur lien respectif. Mais, afin d'accroitre les performances du système, ils peuvent céder leur droit à un autre processus afin que ce dernier puisse écrire ou lire directement sur le lien physique. Le processus système restera alors bloqué (*i.e.* il ne répondra pas aux requêtes éventuelles d'autres processus) tant que le nouveau propriétaire ne lui aura pas rendu la propriété du lien.

Un processus désirant communiquer avec un processus situé sur un autre transputer se synchronisera donc avec le `dlo_t4` indiqué par le routeur. Afin d'éviter un intermédiaire inutile et coûteux, le `dlo_t4` fournit au processus demandeur l'adresse du lien lors du rendez-vous chez le `kernel`.

Sur le transputer receveur, un `dli_t4` reçoit les informations. Mais comme il ne peut pas connaitre à l'avance la taille du message, Trollius doit accompagner chaque message d'un en-tête de taille fixe contenant la longueur du message contenu dans une deuxième partie. Cette deuxième partie est toutefois limitée à 4096 octets. Trollius alloue en effet de façon statique le buffer de `dli_t4`, car une allocation dynamique aurait entrainé un overhead important par la nécessité d'une requête auprès du processus gestionnaire de mémoire. Cette limitation, qui rend nécessaire une paquetisation des longs messages, fait apparaitre un surcoût que nous quantifierons.

A ce niveau, deux possibilités se présentent:

- **Le message n'est pas pour un processus local**: `dli_t4` va faire du *store and forward*, c'est à dire qu'il reçoit le message puis le renvoit vers le nœud suivant, après avoir fait une requête auprès du routeur local.
- **Le message est pour un processus local**: `dli_t4` va optimiser la transmission de la deuxième partie du message en cédant la propriété du lien après avoir transmis l'en-tête. Ainsi, le processus receveur lira directement sur le lien la partie la plus volumineuse du message.

Lors d'un transfert de message inter-transputer, les deux processus communiquants reçoivent la propriété du lien. Elle est normalement rendue au dli/dlo dès que le message a été transmis. Mais, de même que lors des communications intra-transputer, il est possible de créer un circuit virtuel qui permettra aux prochaines communications d'être directes (*i.e.* sans passage par les dli/dlo). L'inconvénient majeur des circuits virtuels est qu'ils interdisent à tous les autres processus l'utilisation du lien, leur avantage étant bien entendu de réduire le coût des communications.

4.5 Bufferisation

Le but de la bufferisation est d'éviter le blocage du processus émetteur. Pour cela, un processus système `bufferd` se charge de recevoir, stocker et faire parvenir à destination les messages ne pouvant pas être reçu immédiatement par d'autres processus.

4.5.1 Intra-T800

Un message sera envoyé vers `bufferd` si l'utilisation d'une fonction non bloquante d'envoi a échoué. `bufferd` recevra et stockera alors le message puis il créera un autre processus système

appelé `bfworker` à qui il renverra le message. Ce dernier est chargé d'effectuer un envoi bloquant du message bufferisé avec l'événement et le type d'origine. Ainsi, à chaque message envoyé à `bufferd` est associé un processus `bfworker`. Quand un `bfworker` a envoyé son message, il est mis en attente : le surcoût important dû à sa création est donc limité à la première utilisation.

Notons que si `bufferd` ne dispose pas de `bfworker` (ils sont limités en nombre), tous les messages reçus sont placés dans une file d'attente jusqu'à la libération d'un `bfworker`. Les messages ainsi mis de côté sont inertes et ne tentent donc pas de se synchroniser chez le `kernel`.

4.5.2 Inter-T800

Une communication inter-T800 fait intervenir deux rendez-vous (envoyeur↔`dlo_t4` et `dli_t4`↔receveur). Il y a donc une bufferisation possible à deux endroits.

envoyeur↔`dlo_t4` Ce cas est simple. Si le `dlo_t4` n'est pas prêt, alors le message est pris en charge par un `bfworker` qui tente de se synchroniser avec le `dlo_t4` comme expliqué ci-dessus.

`dli_t4`↔receveur Ce cas est plus complexe. On a vu qu'après la réception de l'en-tête du message, le `dli_t4` tente de se synchroniser avec le receveur et lui cède alors la propriété du lien pour une transmission plus rapide du corps du message. Si le receveur n'est pas prêt, alors le `dli_t4` lira lui-même le corps du message sur le lien (d'où un premier surcout). Le `dli_t4` retentera ensuite un rendez-vous avec le receveur et on se ramène alors au cas d'une communication intra-T800.

4.6 Les couches de Trollius

Les différents niveaux de Trollius sont construits sur le principe des couches ISO. Chaque niveau ajoute des fonctionnalités au niveau immédiatement inférieur tout en conservant les acquis. L'un des concepteurs de Trollius parle à ce sujet de la technique de l'ognon que l'on peut peler vers les plus bas niveaux de fonctionnalités avec le moins d'overhead [4]. Cette technique conduit l'utilisateur, on le verra, à faire un choix entre la rapidité et le confort.

in/out C'est le niveau de base correspondant aux instructions `in` et `out` du transputer. On se reportera à la section 4.2 pour de plus amples informations.

Kernel Le rôle de ce niveau est de fournir au niveau inférieur l'adresse d'un canal de communication commun aux deux parties, à l'aide d'une requête auprès du `kernel`. C'est à ce niveau que s'effectue la gestion des circuits virtuels. Notons que ce niveau ne peut fonctionner que si les deux processus se trouvent sur le même nœud.

Datalink En accompagnant chaque message d'un en-tête, ce niveau permet des communications inter-transputer. De plus, l'utilisation des fonctions non bloquantes du niveau inférieur lui permet de détourner, si nécessaire, les messages vers le processus `bufferd`. Notons, d'une part que l'en-tête du message ouvre un circuit virtuel qui est normalement refermé par la deuxième partie, et que d'autre part le routeur doit être appelé par l'utilisateur et le message ne doit pas dépasser 4096 octets.

Network Ce niveau comble les deux lacunes précitées en appelant lui-même le routeur et en effectuant une paquetisation des messages.

Transport Ce niveau permet de communiquer en suivant une méthode *request-to-send*: tout expéditeur est bloqué tant qu'un receveur potentiel n'a pas émis un message d'acquittement. Notons que ce niveau interdit l'utilisation de circuit virtuel, et impose une éventuelle bufferisation.

Physical Ce niveau ne suit pas le principe des couches. Il a pour but d'utiliser aussi efficacement que possible un circuit virtuel entre deux transputers voisins. De ce fait, après avoir retrouvé l'adresse du canal de communication, il fait appel aux instructions `in/out` du transputer. Notons que ce niveau ne fonctionne que si un circuit virtuel a déjà été établi à un autre niveau entre deux processus situés sur deux transputers voisins.

4.7 Communications à partir de l'i860

Les communications entre l'i860 et le T800 se font par la mémoire partagée et par interruptions. L'idée de base est que toutes les communications de l'i860 sont effectuées par un agent appelé `i860d` et situé sur le T800 (n'oublions pas que l'i860 est monotâche).

On a vu dans la section 4.6 que toutes les communications inter-processus s'effectuent avec les instructions `in/out` du transputer. Il suffit, donc, qu'à chaque fois que le processus de l'i860 rencontre les fonctions `in/out` de transmettre l'instruction à l'agent qui l'exécute à sa place. La requête est placée à un endroit fixé dans le source, et l'agent en est averti par interruption. Afin de rester en conformité avec le `in/out` du transputer, l'i860 reste bloqué tant que le `in/out` n'est pas complètement exécuté, un flag mis à jour par l'agent servant d'indicateur.

Le problème provient du cache de l'i860 qui fonctionne en lecture et en écriture. Cela impose à l'i860 de vider l'intégralité de son cache avant d'appeler l'agent, afin d'être assuré de la présence de la requête et du message dans la mémoire partagée. De même avant *chaque* lecture du flag, l'i860 doit vider son cache afin de ne pas lire sans cesse la copie de sa précédente lecture se trouvant dans son cache.

L'intégration de l'i860 dans Trollius présente l'avantage que la plupart des programmes tournant sur le T800 peuvent aussi tourner sur l'i860 sans subir de modifications. Par contre, l'inconvénient majeur de cette intégration est l'absence de possibilités de réaliser des communications non bloquantes. En effet, comme nous l'avons vu ci-dessus, l'i860 reste bloqué tant que l'agent `i860d` n'a pas fini d'effectuer la communication. Nous ne pouvons que déplorer ce temps perdu et regretter l'absence d'une fonction non bloquante associée à une fonction `probe` qui aurait indiqué la valeur du flag mis à jour par l'agent.

Chapitre 5

Le modèle de tests

5.1 Modèle de communication

Le coût d'une communication d'un message de taille L entre deux nœuds voisins peut être modélisé de la façon suivante [15] :

$$t = \beta_c + L\tau_c$$

où

β_c : temps de latence - correspond au temps d'initialisation de la communication
τ_c : temps de transfert élémentaire - correspond au débit

Afin de tenir compte de la paquetisation des messages effectuée au niveau Network de Trollius, nous avons introduit un coût τ_p par paquet supplémentaire. D'où finalement la modélisation suivante :

$$t = \beta_c + L\tau_c + N_p\tau_p$$

où

$$\begin{aligned} N_p &= \lceil \frac{L}{L_p} \rceil - 1 \text{ si } L > L_p \\ &= 0 \text{ sinon} \end{aligned}$$

avec

- β_c : temps de latence (en μs)
- L : taille du message (en octets)
- τ_c : temps de transfert élémentaire (μs/octet)
- L_p : taille d'un paquet (en octets, 4096 par défaut)
- τ_p : coût par paquet supplémentaire (en μs/paquet)

5.2 Présentation des tests effectués

Nous avons utilisé le test Ping-Pong (voir figure 5.1).

Nous avons testé les communications pour différentes tailles de messages. En ce qui concerne Trollius, deux séries de mesures ont été effectuées :

- L'une de 0 à 4 Ko par pas de 64 octets: Nous évaluons ainsi précisément les communications pour des messages de petites tailles (inférieures à la taille du paquet par défaut de Trollius).
- L'autre de 0 à 16 Ko par pas de 512 octets: Nous évaluons ainsi la paquetisation effectuée par le niveau Network.

Procedure Ping
 t_1 := time()
 for i:=0 to $nb_{iterations}$ do
   send(Pong,Buffer,Longueur)
   receive(Buffer,Longueur)
 endfor
t_2 := time()
afficher$(t_2 - t_1)/(2 * nb_{iterations})$
endProcedure

Procedure Pong
 for i:=0 to $nb_{iterations}$ do
   receive(Buffer,Longueur)
   send(Ping,Buffer,Longueur)
 endfor
endProcedure

FIG. 5.1 - Algorithme PingPong

Nous effectuons ainsi 10000 aller-retour pour chaque test puis nous interprétons les résultats par la méthode de régression linéaire. Nous effectuons nos tests sur 1, 2 et 3 liens en uni et bi-directionnel. Pour les tests multi-unidirectionnels (nous entendons par ceci les tests en 1, 2 ou 3 liens en unidirectionnels), nous effectuons un Ping-Pong sur chacun des liens. Les Ping-Pongs sont synchronisés au départ et nous veillons à ne pas fausser les mesures par le rapatriement des résultats.

Les tests multi-bidirectionnels sont un peu particuliers en raison de la difficulté de synchroniser deux processus situés sur des nœuds différents. Nous voulions être certains que chaque lien fonctionne réellement en bidirectionnel. Dans ce but, nous avons utilisé deux processus supplémentaires qui étaient chargés d'occuper de façon continue un sens du lien tandis que l'on effectuait les mesures avec l'autre sens sur une suite d'envois réceptionnés sur le nœud destination. Afin de vérifier la validité des mesures, nous avons mesuré une suite d'envois sans occuper le lien dans l'autre sens. Les résultats, comparés au Ping-Pong équivalent, ont confirmé la validité de la méthodologie.

Chapitre 6

Les expérimentations sous VOLCOM

6.1 Introduction

Les tests intra-processeur ont été effectués entre deux processus fils lancés par un processus père sur un même T800. Les tests inter-nœuds, ont été effectués entre des nœuds voisins. Ces tests concernent les communications entre les T800, les i860, les T800 et les i860 situés sur deux nœuds voisins.

Par la suite nous donnons les temps de latence et de transfert élémentaire en micro secondes (μs) et le débit en Mégaoctets par secondes (Mo/s).

6.2 Communications intra-T800

Le processus père lance les processus fils à l'aide des fonctions Inmos C Toolset(ProcAlloc). Au moment de les créer, il alloue les canaux de communication entre ces processus. Puis il passe ces canaux ainsi créés comme paramètre aux processus fils, pour leur permettre de communiquer entre eux [11].

Le temps de transfert élémentaire est de l'ordre de 10^{-2} microsecondes. Dans ce cas, le transfert des données s'effectue à travers la mémoire du T800, ce qui explique une bande passante aussi élevée. Par contre le temps de latence est relativement élevé. Ceci est dû à l'allocation de la mémoire par les processus et les appels de procédures pour l'initialisation des canaux de communications. Nous donnons les mesures exactes dans le tableau 6.1.

6.3 Communications inter-processeurs

Dans la section suivante nous testons d'abord des communications entre deux T800 voisins dans les différents niveaux de VolCom. Puis en section 6.5 nous testons les communications entre les i860 voisins et finalement entre un T800 et un i860 situés sur deux nœuds voisins.

6.4 Communications inter-T800

Nous mesurons les performances des routines de communication entre deux T800 placés sur deux nœuds voisins aux différents niveaux de VolCom. Dans les sections suivantes, nous détaillons nos tests par niveau.

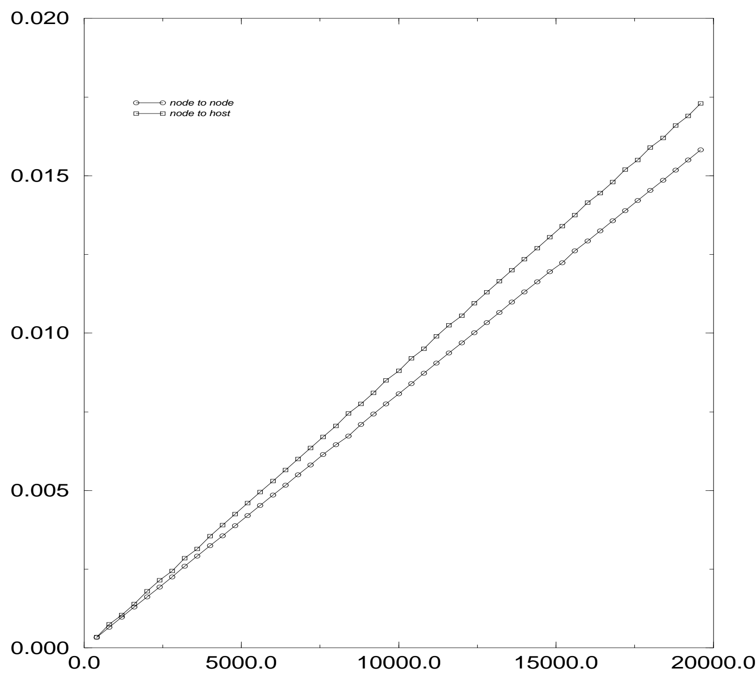


FIG. 6.1 - Temps de communication entre deux nœuds au niveau Low level en fonction de la taille du message

test	β_s	τ_c	débit
Noeud-Noeud	11.5	0.81	1.24
Noeud-Hôte	12.2	0.88	1.13
Intra-Noeud	18.3	0.08	13.4

TAB. 6.1 - La bande passante au Niveau *Low level* pour un lien

6.4.1 Niveau Low level

A ce niveau il y a seulement possibilité de communiquer entre les nœuds reliés par un lien physique. Il n'y a pas de possibilité d'utiliser les procédures de VolConf. Donc au besoin, l'utilisateur doit mettre au point ses propres procédures de routage.

Nous effectuons des tests concernant les liaisons entre un nœud et ses voisins sans faire de tests concernant le routage dans le réseau de processeurs.

Dans le chapitre 2.1 nous avons vu que les nœuds sont reliés les uns aux autres par un switch COO4.

L'hôte est relié aux switches C004 à travers un transputer T222.

Les mesures montrent que les temps de latence sont sensiblement les mêmes dans les deux cas. Il existe cependant un écart de l'ordre de 10% entre les temps de transfert élémentaires. Ceci est dû principalement au temps d'acheminement entre la machine et le frontal. Il faut aussi souligner le temps de passage des messages par le T222 (les T222 font l'interface entre les nœuds de travail et l'hôte).

Pour les communications entre 2 nœuds nous obtenons une bande passante très proche de la bande passante maximale donnée par le constructeur.

La première version de VolCom ne permet pas des envois de messages par des liens entre T800

test	β_s	τ_c	débit par lien logique
1 lien Uni	11.5	0.807	1.23
1 lien Bi	10.1	0.96	1.03
2 liens Uni	11.66	0.806	1.24
2 liens Bi	10.2	0.806	1.24
3 liens Uni	11.3	0.807	1.23
3 liens Bi	10.4	0.805	1.24

TAB. 6.2 - La bande passante au Niveau *Low level* pour plusieurs liens

et i860 d'un même nœudde calcul. Cependant il est possible de faire des communications entre les deux à travers la mémoire commune. Ce problème a été résolu dans la deuxième version de VolCom.

Nous effectuons en parallèle des séries de communications sur 1, 2 et 3 liens. Dans les trois cas nous trouvons les mêmes valeurs pour le débit par lien. Ceci montre qu'il n'y a pas d'interaction entre le fonctionnement des différentes unités d'interface (Link interface). Pour les communications en bidirectionel sur 2 et 3 liens, nous ne constatons pas la baisse du débit comme pour un seul lien bidirectionel. Dans le cas d'un lien, cette baisse est dûe aux messages d'acquittement envoyés par le récepteur. Pour les cas de 2 et 3 liens, nous ne savons pas comment expliquer cette absence de baisse. Malheureusement nous n'avons pas pu effectuer nos tests pour 4 liens. Il semble que le nombre important de processus créés dans ce dernier cas cause un effondrement du système de contrôle du transputer dû à l'insuffisance de la mémoire. Nous pensons cependant que dans une machine avec assez de mémoire il sera possible d'obtenir les mêmes résultats que pour les cas 1, 2 et 3.
Nous avons voulu vérifier l'influence des calculs sur les performances des procédures de communication. Nous avons fait exécuter une série de calculs (en l'occurrence un *daxpy* [7]) en même temps que nous effectuions des communications entre un nœud et ses deux voisins.

Nous avons trouvé $10.2\mu s$ pour le temps de latence et $0.806\mu s$ pour le temps de transfert élémentaire. Ceci montre que les calculs ont très peu d'influence sur les performances des procédures de communication.

Les deux séries de mesures précédentes montrent l'indépendance entre les calculs et les communications. Ceci est logique et est dû à la répartition de ces tâches entre différentes unités. Dans un cas il s'agit de la FPU[1] et dans l'autre cas des unités d'interface (Link interface gérées par DMA[2]).

6.4.2 Niveau High level

A ce niveau il y a des processus de routage permettant d'envoyer des messages aux processeurs non-voisins.

Nous testons d'abord les communications entre deux nœuds voisins. Puis nous effectuons des mesures sur plusieurs nœuds pour déterminer les performances des procédures de communication et de routage.

Dans la première version de VolUse il n'était pas possible de lancer plusieurs tâches sur un même processeur au niveau *High level*. Ceci est possible au niveau *Low level* mais la difficulté de programmation à ce niveau rend l'utilisation de plusieurs tâches sur un même processeur, assez

[1] Floating Point Unit
[2] Direct Memory Access

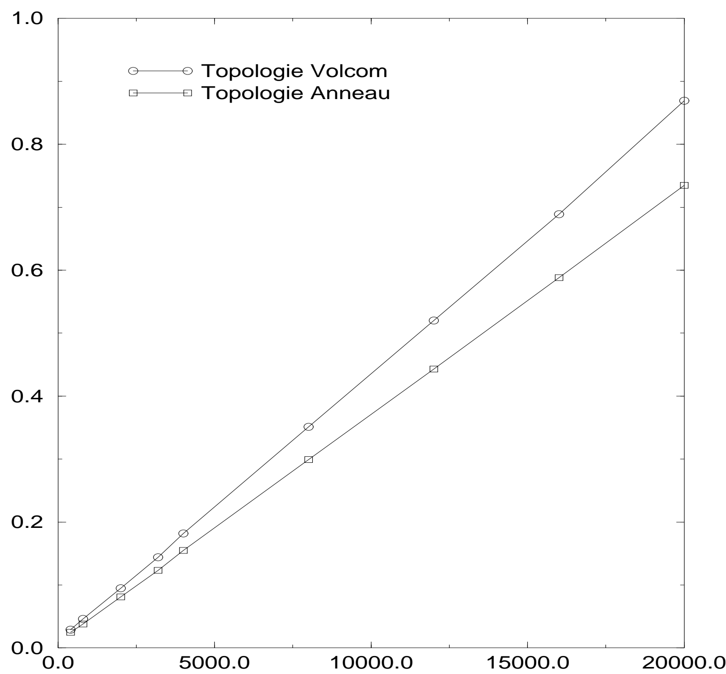


FIG. 6.2 - Temps de communication selon les différentes topologies utilisées au niveau High level

$test$	β_s	τ_c	débit
$VolCom1$	906.6	4.1	0.25
$VolCom2$	766.5	1.1	0.94

TAB. 6.3 - La bande passante au Niveau *High level* pour 1 lien entre 2 nœuds dans les versions 1 et 2 de VolCom

difficile. Ce problème a été résolu dans la deuxième version.

Par défaut, VolUse place automatiquement les tâches sur les processeurs. L'utilisateur peut également préciser l'emplacement des tâches. Plus l'utilisateur précise l'emplacement des tâches et les liens entre elles et plus il tend vers une programmation niveau *Low level*. C'est un handicap quand on veut avoir des topologies spécifiques pour effectuer des pipelines ou mettre au point les stratégies de communication efficaces telles que celles utilisant des arbres de recouvrement [14, 9] ....

Le fait que VolCom au niveau *High level* ne supporte pas des communications asynchrones rend impossible la programmation asynchrone (à moins que l'utilisateur implémente des processus tampons intermédiaires). Ce problème a été résolu dans la version 2.0 de VolCom.

Les communications entre 2 processeurs voisins

Dans le tableau 6.3, nous présentons le temps de latence et de transfert élémentaire entre deux nœuds voisins dans la version 1.0 et 2.0 à l'aide des procédures *vsend* et *vreceive*.

Le temps de latence et le temps de transfert sont élevés. Il semble que ceci soit dû au temps passé à reconfigurer le C004 lors de l'établissement de chaque nouveau lien et au temps du routage du message dans le réseau.

Le ring test sur plusieurs processeurs

Nous effectuons des envois du premier processeur au dernier processeur, en effectuant du *store & forward* sur les nœuds intermédiaires. Nous faisons deux séries de tests :

1. En formant un anneau explicitement, en placant les tâches sur chacun des nœuds et en envoyant le message explicitement au processeur suivant.
2. En laissant à VolUse le soin de placer les tâches sans lui préciser la topologie à choisir et en envoyant le message d'une tâche à la suivante, jusqu'à la dernière tâche

Nous présentons les résultats dans la figure 6.2. La comparaison entre ces 2 séries de valeurs montre que le temps mis par VolCom pour acheminer les messages est plus long que celui des *stores & forwards* successifs. VolCom détermine le chemin du routage à chaque étape à l'aide des tableaux d'identification. Il y a donc un surcoût pour trouver le processeur qui détient la tâche suivante dans le tableau d'identification. Cependant le plus important est de regarder l'écart entre les 2 temps. On peut en déduire que les procédures de routages de VolCom ne sont pas très efficaces. Nous avons effectué une dernière série d'expérience. Nous avons laissé VolUse placer les tâches et nous avons effectué des communications entre le premier et le dernier nœud. Le temps de communication correspond au temps de communications entre 2 nœuds. Ceci montre qu'en dépit de la surcharge dûe à la table d'identification, on peut gagner énormément de temps par le choix du chemin optimal entre deux nœuds.

6.5 Communications inter-i860

Comme cela a été dit précédemment, les communications entre les i860 passent par les T800 associés à chaque nœud. Dans chaque nœud, les messages entre T800 et i860 passent par la mémoire partagée. Les T800 sont reliés entre eux par le biais d'un *crossbar*. Les communications entre les T800 sont longuement discutées plus haut. Nous obtenons pour le temps de latence au niveau *High level* $1305\mu s$ et pour le temps de transfert élémentaire $1.23\mu s$ (ces résultats sont obtenus sous VolCom 2.0).

6.6 Communications T800-i860

Dans la première version de VolCom il est impossible d'envoyer des messages entre un T800 et un i860 situés sur le même nœud. Nous étudions des performances des communications entre un T800 et un i860 sur deux nœuds voisins. Nous obtenons pour le temps de latence $1215\mu s$ et $1.23\mu s$ pour le temps de transfert élémentaire (ces résultats sont obtenus sous VolCom 2.0).

Nous obtenons sensiblement les mêmes temps de transfert élémentaire pour les communications entre T800-i860 et i860-i860. Ceci est logique, car les communications entre deux nœuds voisins, quels que soient les processeurs utilisés, passent par les T800. La transmission du message à travers de la mémoire commune est la seule différence dans les deux cas. Le temps de latence est plus long pour le cas i860-i860. Ceci montre le temps important nécessaire pour initialiser les canaux de communications entre l'i860 et le T800 à travers la mémoire commune.

test	β_s	τ_c	débit
T800-T800 *Low level*	11.5	0.807	1.23
T800-T800 *High level*	766.5	1.1	0.94
T800-i860 *High level*	1215	1.23	0.81
i860-i860 *High level*	1305	1.23	0.81

TAB. 6.4 - La bande passante pour les communications inter processeurs. Pour le cas T800-i860, les processeurs sont situés sur deux nœuds voisins

6.7 Conclusions concernant VolCom

Concernant les communications inter T800, la différence entre les temps de transfert élémentaires et de latence entre les deux niveaux *High level* and *Low level* est importante. Il est donc raisonnable de penser à diminuer ce temps d'une manière significative. L'utilisation des User tasks a l'avantage de rendre la programmation facile. Effectivement le programmeur peut se passer de placer les tâches lui-même sur le réseau de processeurs. Cette approche a cependant l'inconvénient d'imposer au système de créer lui-même les fichiers de configuration. On sait que la génération automatique des fichiers de configuration est un problème assez compliqué, sans oublier des problèmes posés au moment de l'utilisation des techniques de plus en plus courantes que sont le pipelinage, le recouvrement calcul/communications .... L'ensemble de ces problèmes pourrait aboutir à considérer le niveau *High level* de VolCom beaucoup plus comme un logiciel d'initiation au parallélisme qu'un environnement de développement de logiciels parallèles.

Il y a une nette amélioration des performances dans la version 2.0 de VolCom par rapport à la version 1.0. Nous croyons cependant qu'il est encore possible de les améliorer. VolUse est un environnement de programmation de haut niveau bien adapté pour effectuer des approches parallèles des problèmes. Cependant, il ne nous semble pas adapté à l'implémentation d'applications ayant besoin de performances élevées au niveau des communications. Car ses procédures de communications sont peu performantes.

VolCom offre un environnement de programmation convivial permettant une mise au point rapide des programmes parallèles (pour peu qu'on ait quelques connaissances dans le domaine). En tenant compte des différents niveaux de programmation de cette machine, nous pensons qu'il est possible d'obtenir des performances satisfaisantes. D'abord, on fait une première approche et une étude des différentes solutions parallèles du problème à l'aide du niveau *High level*. Ceci permet de voir la faisabilité et les problèmes algorithmiques en un minimum de temps. Puis on essaye d'augmenter les performances de l'implémentation en passant au niveau *Low level*. Cette méthode a l'inconvénient de ne pas tenir compte des contraintes physiques dûes à la machine au moment de la conception de l'algorithme. Elle a cependant l'avantage d'éviter des erreurs algorithmiques et facilite la conception des algorithmes parallèles.

Chapitre 7

Les expérimentations sous TROLLIUS

7.1 Avertissement

En répétant les tests, nous avons remarqué, de façon non systématique, des variations à la baisse des temps de latence allant jusqu'à 10%. Comme nous ignorons les raisons de ces variations, nous ne savons pas quels sont les bons résultats. D'autre part, nous avons remarqué des variations des performances selon le nœud utilisé, et cela même pour des communications intra-transputer.

7.2 Introduction

La table 7.1 décrit tous les tests effectués pour les différents niveaux de Trollius ainsi qu'avec diverses fonctionnalités.

Test	**Niveau**
in/out	instructions `in/out`
K	Kernel
Kcv	Kernel avec circuit virtuel
Dnb	Datalink sans bufferisation
Dcv	Datalink avec circuit virtuel
Db	Datalink avec bufferisation forcée
Nnb	Network sans bufferisation
Ncv	Network avec circuit virtuel
P	Physical

TAB. 7.1 - Description des tests effectués

Tous les tests ont été effectués sur le même nœud.

En plus des paramètres présentés dans le chapitre 5, nous indiquons pour les niveaux Datalink et Network le temps de transmission de l'en-tête du message qui peut contenir des informations utiles. Cette transmission est calculée par l'envoi d'un message de taille nulle, d'où l'appellation t_0.

Il est à noter que l'étude du source de TROLLIUS indique que les temps de réception et d'envoi d'un message doivent être sensiblement égaux.

Les tests montrent que le temps de transfert élémentaire est indépendant du niveau utilisé. La taille du message n'intervient que lors du transfert effectif du message avec `in/out`. Seule la bufferisation modifie le temps de transfert élémentaire (en le triplant).

Nous avons noté dans la section 4.4.1 la présence d'un cache pour les appels au routeur. Lors de chacun de nos tests, nous avons initialisé le cache par un échange préalable de messages.

7.3 Communications intra-T800

D'après la section 2.1 Chaque nœud a trois types de mémoire: MAIN (mem. externe T800), ONCHIP (mem. interne T800) et I860MAIN (mem. partagée T800/i860). Les tests ont montré que selon la mémoire où se trouve le buffer, le temps de transfert élémentaire varie. Nous avons testé les transferts entre les mémoires MAIN et I860MAIN. Le tableau 7.2 indique la valeur de τ_c selon l'emplacement du buffer ainsi que la valeur du débit associé. Il convient également de noter que la mémoire interne ONCHIP peut être utilisée sur le transputer pour stocker du code. Celui-ci pouvant alors s'effectuer avec un gain de vitesse d'environ 40%.

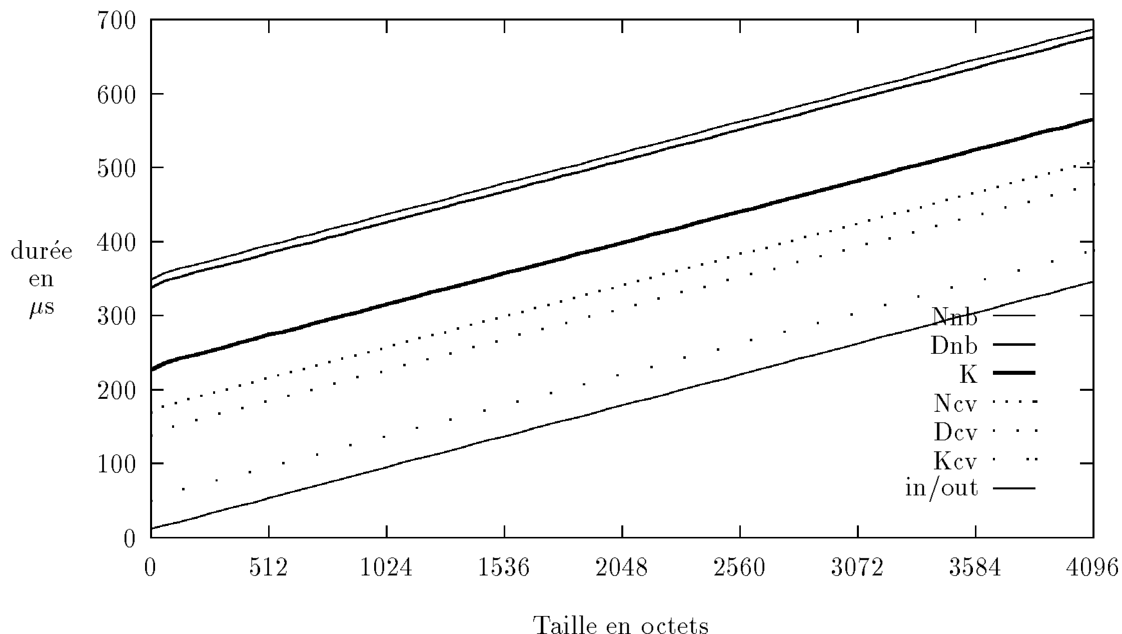


FIG. 7.1 - Communications intra-T800 - messages courts

La mémoire MAIN est plus de deux fois plus rapide que la mémoire partagée I860MAIN.

Le tableau 7.3 indique les temps de latence des différents niveaux de Trollius. Nous constatons que :

- le coût de la requête auprès du `kernel` peut être évalué à 180 μs;

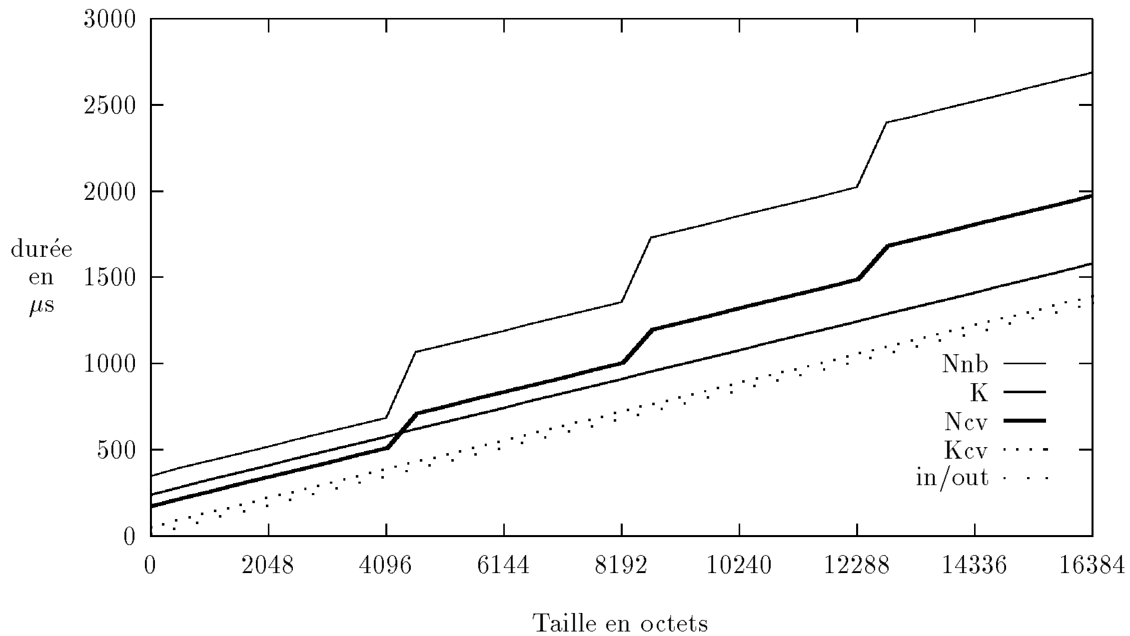


FIG. 7.2 - Communications intra-T800 - messages longs

	MAIN ⇔ MAIN	MAIN ⇔ I860MAIN	I860MAIN ⇔ I860MAIN
$\tau_c(\mu s/o)$	$0,08160$	$0,10286$	$0,18462$
débit (Mo/s)	$12,25$	$9,72$	$5,42$

TAB. 7.2 - Communications intra-T800 - temps de transfert élémentaire et débit

- les temps de latence élevés des niveaux Datalink et Network sont dûs à la transmission de l'en-tête du message;
- le surcoût d'un paquet au niveau Network est du même ordre que le temps de latence. La différence correspond à l'appel d'une fonction qui détermine s'il y a lieu ou non de router le message vers un autre nœud;
- l'en-tête du message peut contenir huit entiers. Il est à noter qu'un message de longueur nulle n'apporte pas de gain sensible. Cela est dû au fait que l'en-tête ouvre un circuit virtuel que le corps du message doit refermer. Or, la gestion des circuits virtuels s'effectuant au niveau Kernel, le corps doit obligatoirement passer par toutes les couches, même s'il n'y a finalement pas de transmission.

	t_0	$\beta_c(\mu s)$	$\tau_p(\mu s/p)$
in/out	•	12	0
Kcv	•	53	0
K	•	231	0
Dcv	138	143	•
Dnb	338	343	•
Ncv	169	174	154
Nnb	349	353	349

Tab. 7.3 - Communications intra-T800 - temps de latence et coût par paquet

7.4 Bufferisation

En modifiant légèrement le source de Trollius, nous sommes parvenus à forcer la bufferisation d'un message avec le niveau Datalink. Le test a été effectué entre deux processus sur un même nœud, les deux buffers étant situés en mémoire MAIN.

	t_0	$\beta_c(\mu s)$	$\tau_c(\mu s/o)$	débit(Mo/s)
Db	2304	2319	$0,2447$	$4,09$

Tab. 7.4 - Bufferisation - niveau Datalink

Le tableau 7.4 indique les résultats obtenus :

- Le τ_c correspond aux trois transmissions successives (cf. section 4.5) qui sont :
  `envoyeur` $\overset{1}{\Rightarrow}$ `bufferd` $\overset{2}{\Rightarrow}$ `bfworker` $\overset{3}{\Rightarrow}$ `receveur`
  On a bien $\tau_c = 0,08160 * 3$. D'autre part, nous constatons un temps de latence très élevé.

7.5 Communications inter-T800

Nous avons testé l'ensemble des niveaux avec les buffers en mémoire MAIN et I860MAIN. La différence, bien qu'existante, n'est pas assez significative pour pouvoir être chiffrée (de l'ordre de $5\mu s$ pour un message de 16 Ko).

Le tableau 7.5 indique les résultats obtenus :

- La différence entre l'utilisation ou non d'un circuit virtuel est de l'ordre de $270\mu s$. Or, comme nous l'avons vu dans la section 4.4.2, il y a deux rendez-vous (`envoyeur`↔`dlo_t4` et `dli_t4` ↔`receveur`). En fait, il y a un recouvrement partiel entre le temps de latence de l'envoi du corps du message par l'émetteur et le rendez-vous entre `dli_t4` et le processus destination.
- Nous n'arrivons pas à trouver de raisons valables à la valeur élevée de τ_p par rapport au β_c au niveau Nnb.

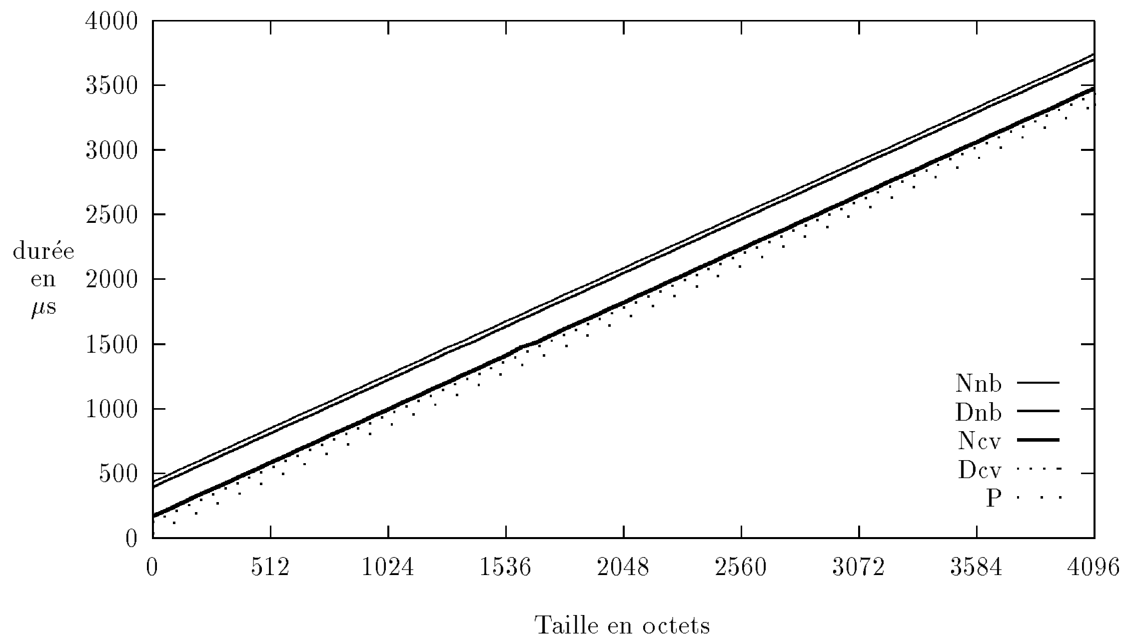


FIG. 7.3 - Communications inter-T800 - messages courts

	$T800 \Leftrightarrow T800$
$\tau_c(\mu s/o)$	$0,807$
débit (Mo/s)	$1,24$

TAB. 7.5 - Communication inter-T800 voisins - temps de transfert élémentaire

7.6 Communications T800 - i860

Les résultats obtenus sont assez étranges. Nous n'avons pas trouvé d'explications totalement satisfaisantes pour les justifier.

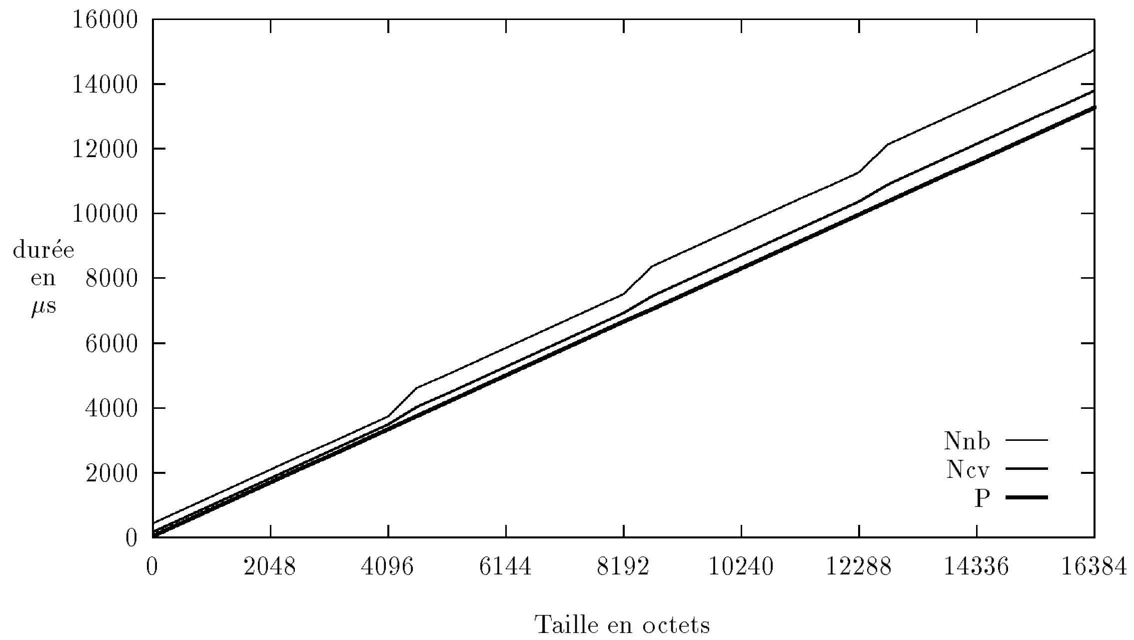


FIG. 7.4 - Communications inter-T800 - messages longs

	t_0	$\beta(\mu s)$	$\tau_p(\mu s/p)$
P	36	39	0
Dcv	124	126	•
Dnb	390	393	•
Ncv	167	169	134
Nnb	433	435	461

TAB. 7.6 - Communication inter-T800 voisins - temps de latence et coût par paquet

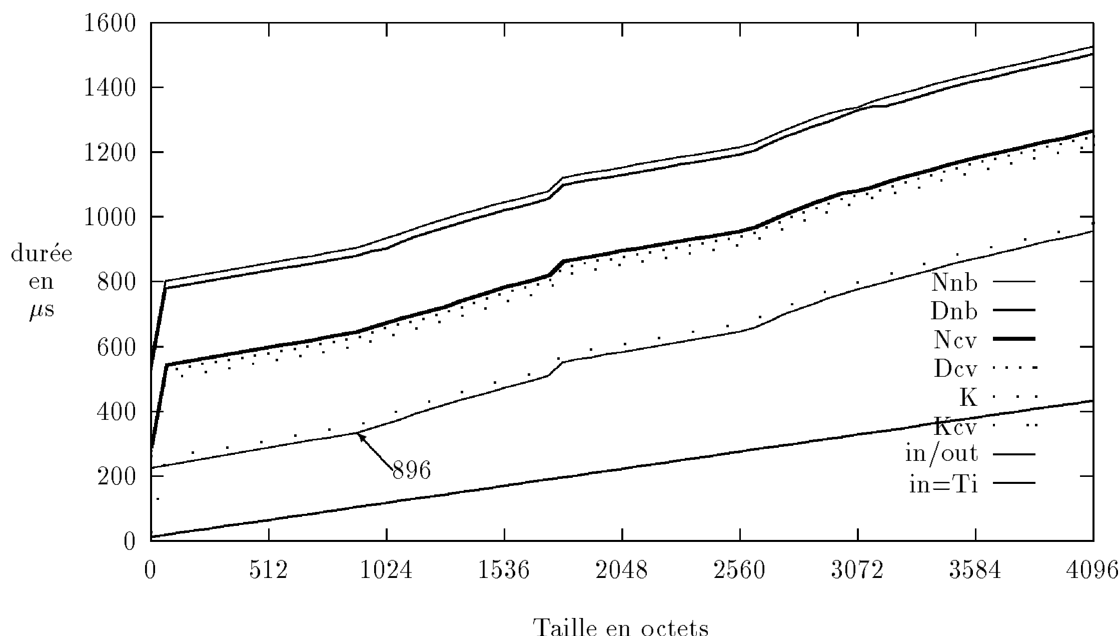


FIG. 7.5 - Communications T800-i860 - messages courts

La figure 7.5 représente les courbes obtenues avec les différents niveaux. Nous avons rajouté, à titre de comparaison, la courbe obtenue lors de communications intra-T800 du type MAIN ⇔ I860MAIN avec les instructions `in/out` du transputer (courbe in=Ti).

D'après la section 4.7, on devrait obtenir lors des communications i860-T800 une courbe de même pente que in=Ti. Cela se vérifie pour les messages de petite taille. Après, le temps de transfert élémentaire augmente de façon irrégulière et non fixe suivant le nombre d'itérations du test. La figure 7.6, certes beaucoup moins précise, permet toutefois de relativiser l'irrégularité. Le cache de l'i860 peut avoir un effet sur les résultats, dans la mesure où la lecture du flag indiquant la fin de la communication ne peut se faire qu'après une nouvelle mise à jour du cache, opération qui prend un temps non négligeable. Mais la courbe devrait alors avoir des paliers plus marqués.

Bien qu'un peu étonnant, le temps de transfert élémentaire permet de calculer avec une bonne précision les temps de communications de longs messages.

Le tableau 7.8 indique les temps de latence des différents niveaux de Trollius. Nous constatons :

- Un coût très élevé du niveau `in/out` qui se répercute sur les autres niveaux.
- Le coût de la requête auprès du `kernel` (qui se trouve sur le T800) peut être évalué à $260\mu s$. Ce chiffre peut être attribué à une optimisation du code de la requête qui tient compte du coût élevé des communications entre l'i860 et le T800.
- L'emploi de circuit virtuel apporte logiquement un gain très important par la requête qu'il permet d'éviter.
- En raison de l'appel coûteux à l'agent `i860d` situé sur le T800, l'envoi de messages de longueur nulle avec les niveaux Datalink et Network entraine un gain très significatif.

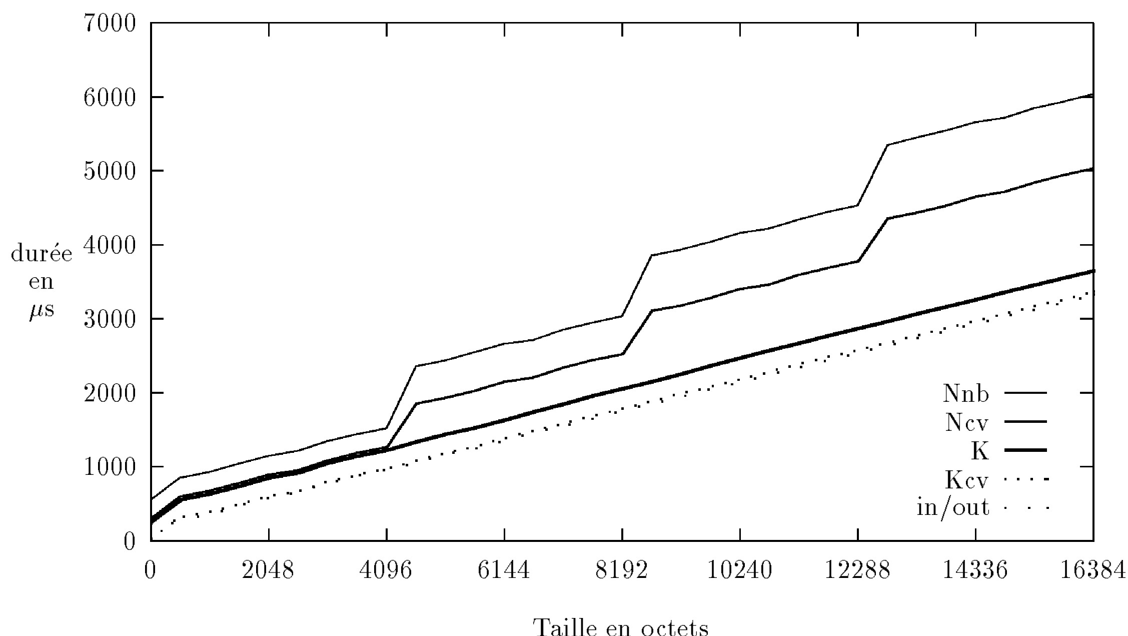


FIG. 7.6 - Communications T800-i860 - messages longs

	$i860 \Leftrightarrow T800$
$\tau_c(\mu s/o)$	$0,19$
débit (Mo/s)	$5,26$

TAB. 7.7 - Communication i860-T800 - temps de transfert élémentaire

- Ces coûts élevés des communications i860-T800 nous font regretter d'autant plus l'absence de communications non bloquantes.

7.7 Communications multi-unidirectionnelles

Le temps de transfert élémentaire est indépendant du nombre de liens utilisés unidirectionnellement. Cela confirme le fonctionnement en parallèle des quatre liens du T800 (cf. section 2.2).

Le tableau 7.10 indique les temps de latence obtenus suivant le niveau et le nombre de liens utilisés simultanément de façon unidirectionnelle :

- L'augmentation rapide du temps de latence avec le niveau Datalink sans circuit virtuel est due à l'unicité du `kernel` sur un nœud. Cela provoque des conflits entre les différentes requêtes et des attentes. La forme très irrégulière de la courbe sur la figure 7.7 illustre bien cela. Nous n'avons pas testé le niveau Nnb pour cette raison.

	t_0	$\beta(\mu s)$	$\tau_p(\mu s/p)$
in/out	•	177	•
Kcv	•	197	•
K	•	433	•
Dcv	277	480	•
Dnb	530	735	•
Ncv	290	497	470
Nnb	551	760	720

TAB. 7.8 - Communication i860-T800 - temps de latence et coût par paquet

	t_0	β	τ_p	τ_c	débit
T800-T800 P	36	39	0	0.807	1.24
T800-T800 Dcv	124	126	•	0.807	1.24
T800-T800 Dnb	390	393	•	0.807	1.24
T800-T800 Ncv	167	169	134	0.807	1.24
T800-T800 Nnb	433	435	461	0.807	1.24
i860-T800 in/out	•	177	•	0.19	5.26
i860-T800 Kcv	•	197	•	0.19	5.26
i860-T800 K	•	433	•	0.19	5.26
i860-T800 Dcv	277	480	•	0.19	5.26
i860-T800 Dnb	530	735	•	0.19	5.26
i860-T800 Ncv	290	497	470	0.19	5.26
i860-T800 Nnb	551	760	720	0.19	5.26

TAB. 7.9 - Le tableau récapitulatif des bandes passante pour les communications inter processeurs

- Le temps de latence des autres niveaux augmente très modérément avec le nombre de liens utilisés. Cela s'explique par un partage du processeur pour l'exécution de la partie *Trollius* des communications.
- La valeur du niveau Network avec circuit virtuel en utilisant quatre liens n'est probablement pas significative.

7.8 Communications multi-bidirectionnelles

On constate (le tableau 7.11) une baisse du temps de transfert élémentaire lors de l'utilisation d'un lien en bidirectionnel, en raison d'acquittements propres au transputer. Chaque octet envoyé doit être acquitté par deux bits par le T800 receveur. On constate le fonctionnement en parallèle des quatre liens dans les deux sens.

Le tableau 7.12 présente les temps de latence pour l'utilisation en bidirectionnel de 1 à 3 liens simultanément avec le niveau Physical.

Par rapport à l'utilisation en unidirectionnel, il y a une augmentation sensible, de l'ordre de 12%.

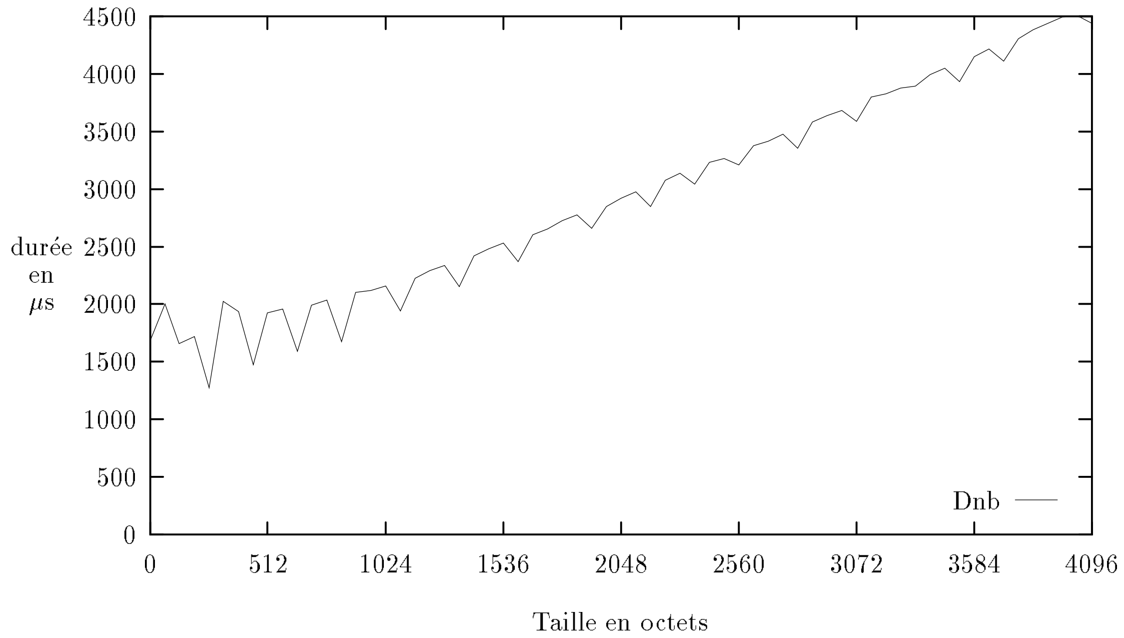


FIG. 7.7 - Communications Multi-unidirectionnelles avec 4 liens

$\beta(\mu s)$	1 lien	2 liens	3 liens	4 liens
P	$38,9$	$39,4$	$39,7$	$40,5$
Dcv	126	129	136	149
Dnb	390	620	874	1384
Ncv	167	181	231	217

TAB. 7.10 - Communications Multi-Unidirectionnelles - temps de latence

	$\tau(\mu s/o)$	débit
unidirectionnel	$0,808$	$1,24$
bidirectionnel	$0,968$	$1,03$

TAB. 7.11 - Communications bidirectionnelles - temps de transfert élémentaire

$\beta(\mu s)$	unidirectionnel	bidirectionnel
1 lien	40	$45,2$
2 liens	$40,4$	$45,2$
3 liens	$40,4$	$46,3$

TAB. 7.12 - Communications bidirectionnelles - temps de latence du niveau Physical

Chapitre 8

Conclusion

Nous n'avons pas voulu comparer les résultats entre les différents niveaux des deux environnements. Ces niveaux ne sont effectivement pas forcément comparables dans leur approche du système. On peut malgré cela considérer que les performances des bas niveaux sont assez proches dans les deux environnements. Elles s'approchent des performances des communications au niveau des liens physiques.

Nous avons testé les performances des communications sur les environnements VolCom et Trollius. Ceci nous a permis d'avoir une idée générale des performances des communications sur la machine VolVox IS-860. Nos tests montrent clairement que les communications de haut niveau sont trop chères dans les deux environnements de programmation. Il est donc possible de gagner énormément au niveau des performances en passant au niveau *Low level*. Ceci implique une perte importante de la convivialité au niveau de la programmation. Comme dans n'importe quel environnement de programmation évolué, le défi majeur est de trouver un compromis optimal entre les performances et la convivialité. L'environnement VolCom de ce point de vue nous paraît quelque peu "bancal". L'écart entre les performances au niveau *High level* et au niveau *Low level* et *Intermediate level* est trop important.

Néanmoins, étant données les bonnes performances de la machine au niveau *Low level*, nous pensons que cette machine devrait également être capable de fournir de très bonnes performances au niveau *High level*.

Remerciements

Nous tenons à remercier Thibaud Duboux et Jean-Yves Peterschmitt pour la relecture des versions préliminaires de ce rapport et pour leurs pertinentes remarques.

Bibliographie

[1] *TROLLIUS Manual Pages*, Mar. 1990.

[2] *TROLLIUS Reference Manual for C Programmers*, Mar. 1990.

[3] ARCHIPEL, *VolVox Machine Reference Manual*, 1992.

[4] G. BURNS, *Trollius: Early American Transputer Software*, Parallelogram, 13 (1989).

[5] G. BURNS, V. RADIYA, R. DAOUD, AND R. MACHIRAJU, *All About Trollius*, Occam User's Group Newsletter, (1990).

[6] H. CHARLES, *Le Processeur i860*, Tech. Rep. 90-02, LIP-IMAG, 1990.

[7] J. DONGARRA AND D. SORENSEN, *Linear algebra on high performance computers*, Parallel Computing, 85 (1986), pp. 221–236.

[8] T. DUNIGAN, *Performance of the intel iPSC/860*, Tech. Rep. TM-11491, Oak Ridge National Laboratory, June 90.

[9] P. FRAIGNIAUD, *Communications intensives dans les architectures à mémoires distribuées et Algorithme parallèle pour la recherche de racines de pôlynomes*, PhD thesis, Ecole Normale Superieure de Lyon, Dec. 1990.

[10] S. FRIED, *Personal supercomputing with the Intel i860*, BYTE January, (1991).

[11] INMOS, *C Ansi Toolset*, 1990.

[12] M. LOI AND B. TOURANCHEAU, *TROLLIUS, un système d'exploitation pour ordinateur multiprocesseur à mémoire distribuée*, Tech. Rep. 9102, LIP-IMAG, 1990.

[13] N. MARGULIS, *The Intel 80860*, BYTE, (1989), pp. 333–340.

[14] M. POURZANDI AND B. TOURANCHEAU, *Recouvrement calcul/communication dans l'élimination de Gauss sur iPSC/860*, Tech. Rep. 92-29, Laboratoire de l'Informatique du Parallelisme, Ecole Normale Superieure de Lyon, 69364 Lyon Cedex 07, France, 1992.

[15] Y. SAAD AND M. SCHULTZ, *Topological Properties of Hypercubes*, Tech. Rep. YALEU/DCS/RR-389, Computer Science Department - Yale University, 1987.

[16] D. TRYSTRAM AND F.VINCENT, *Programmation avancée du Transputer: architecture et mécanismes*, La lettre du Transputer et des calculateurs distribués - Avril, (1989).

[17] C. WHITBY-STEVENS, *Supernode: Transputer and Software*, in Proceedings of Supercomputing'88, 1988.

Table des matières

1 Introduction **1**

2 La machine VOLVOX IS-860 **2**
2.1 Architecture de la machine VolVox IS-860 2
2.2 Le Transputer T800 2
2.3 Le processeur Intel i860 2

3 L'environnement VOLCOM **5**
3.1 Présentation 5
3.2 Communications intra-processeur 5
3.3 Communications inter-processeurs 5
3.4 VolConf et le système de routage 6
3.5 VolCom 6
3.5.1 Niveaux Low level & Intermediate level 7
3.5.2 Niveau High level 7
3.6 Communications à partir de l'i860 7

4 L'environnement TROLLIUS **8**
4.1 Présentation 8
4.2 In/out 8
4.3 Communications intra-transputer 9
4.4 Communications inter-transputer 10
4.4.1 Routage 10
4.4.2 Dli/dlo 11
4.5 Bufferisation 11
4.5.1 Intra-T800 11
4.5.2 Inter-T800 12
4.6 Les couches de Trollius 12
4.7 Communications à partir de l'i860 13

5 Le modèle de tests **14**
5.1 Modèle de communication 14
5.2 Présentation des tests effectués 15

6 Les expérimentations sous VOLCOM **16**
6.1 Introduction 16
6.2 Communications intra-T800 16

6.3 Communications inter-processeurs 16
6.4 Communications inter-T800 16
6.4.1 Niveau Low level 17
6.4.2 Niveau High level 18
6.5 Communications inter-i860 20
6.6 Communications T800-i860 20
6.7 Conclusions concernant VolCom 21

7 Les expérimentations sous TROLLIUS **22**
7.1 Avertissement 22
7.2 Introduction 22
7.3 Communications intra-T800 23
7.4 Bufferisation 25
7.5 Communications inter-T800 25
7.6 Communications T800 - i860 26
7.7 Communications multi-unidirectionnelles 29
7.8 Communications multi-bidirectionnelles 30

8 Conclusion **32**

Table des figures

2.1 Architecture d'un nœud de la machine ARCHIPEL VolVox IS-860 3
2.2 Architecture du Transputer T800 d'INMOS 3

5.1 Algorithme PingPong 15

6.1 Temps de communication entre deux nœuds au niveau Low level en fonction de la taille du message 17
6.2 Temps de communication selon les différentes topologies utilisées au niveau High level 19

7.1 Communications intra-T800 - messages courts 23
7.2 Communications intra-T800 - messages longs 24
7.3 Communications inter-T800 - messages courts 26
7.4 Communications inter-T800 - messages longs 27
7.5 Communications T800-i860 - messages courts 28
7.6 Communications T800-i860 - messages longs 29
7.7 Communications Multi-unidirectionnelles avec 4 liens 31

Liste des tableaux

6.1 La bande passante au Niveau *Low level* pour un lien 17
6.2 La bande passante au Niveau *Low level* pour plusieurs liens 18
6.3 La bande passante au Niveau *High level* pour 1 lien entre 2 nœuds dans les versions 1 et 2 de VolCom 19
6.4 La bande passante pour les communications inter processeurs. Pour le cas T800-i860, les processeurs sont situés sur deux nœuds voisins 21

7.1 Description des tests effectués 22
7.2 Communications intra-T800 - temps de transfert élémentaire et débit 24
7.3 Communications intra-T800 - temps de latence et coût par paquet 25
7.4 Bufferisation - niveau Datalink 25
7.5 Communication inter-T800 voisins - temps de transfert élémentaire 26
7.6 Communication inter-T800 voisins - temps de latence et coût par paquet 27
7.7 Communication i860-T800 - temps de transfert élémentaire 29
7.8 Communication i860-T800 - temps de latence et coût par paquet 30
7.9 Le tableau récapitulatif des bandes passante pour les communications inter processeurs 30
7.10 Communications Multi-Unidirectionnelles - temps de latence 31
7.11 Communications bidirectionnelles - temps de transfert élémentaire 31
7.12 Communications bidirectionnelles - temps de latence du niveau Physical 31