Composition Mineure "Calcul Parallèle"
Corrigé

Eric Goubault, Régis Cridlig

Mercredi 2 avril 1997

Problème

Le but du sujet est d'estimer la valeur de plusieurs algorithmes classiques de calcul de transformée de Fourier bi-dimensionnelle discrète sur un réseau de stations selon les performances du réseau, des stations et la taille du calcul.

Le réseau que nous considérerons dans toute la suite du problème est composé de N ($N \geq 2$) stations nommées $P_1, P_2, \cdots, P_N$. Il s'agit typiquement d'un réseau de stations de travail connectées par Ethernet et travaillant ensemble grâce aux fonctionalités de la librairie PVM.

La donnée de départ est une matrice complexe carrée $M = (M_{i,j})_{1 \leq i,j \leq K}$ de taille $K \times K$ (N divise K), entièrement stockée dans la mémoire locale de P_1. On veut construire des algorithmes qui répartissent au mieux le travail sur les $P_1, P_2, \cdots, P_N$ de telle façon qu'à la fin de leur exécution, la transformée de Fourier $\hat{M}$ se trouve stockée dans la mémoire locale de P_1. Les algorithmes impliqueront des opérations exécutées localement sur une station, telles l'envoi non bloquant (*pvm_send*) ou la réception bloquante (*pvm_recv*) de tableaux de complexes, ou telles le calcul de la transformée de Fourier uni-dimensionnelle par l'algorithme FFT (vu en cours et que l'on considèrera être une fonction C donnée fft) d'un tableau de complexes. On supposera avoir à disposition un type C prédéfini *complex* représentant un nombre complexe. Toutes les opérations élémentaires (addition complexe $+$, multiplication complexe $\times$, exponentielle complexe exp, compactage ou décompactage d'un complexe dans un buffer de communication *pkcmplx*, *upkcmplx* etc.) sont supposées être calculées en un cycle d'horloge sur chaque station, pour simplifier. A l'exception de certaines questions, on n'utilisera pour le passage de messages que les primitives *pvm_send* et *pvm_recv*.

On considérera que les performances de nos algorithmes ne dépendront que des données suivantes :

- le temps d'un envoi d'un message, sans compter l'attente éventuelle d'un partenaire pour la communication, est le débit γ multiplié par la taille en nombre de complexes du tableau envoyé plus le temps de latence δ.
- le temps d'un cycle sur P_i ($1 \leq i \leq N$) est de α_i (une constante). Sauf mention contraire, on supposera pour simplifier dans toute la suite $\alpha_i = 1$ ($1 \leq i \leq N$). On pourra également supposer que le temps de réception d'un message sur la station P_i est de α_i multiplié par la taille du tableau transmis.
- la création de processus (par *spawn* que l'on suppose permettre de choisir exactement la station P_i, $1 \leq i \leq N$ sur laquelle une fonction va être exécutée) met un temps τ pour établir la connection au démon local *pvmd*. Ce temps est inchangé si on demande à *spawn* de lancer plusieurs copies d'un même programme.

1 Question :

On examine d'abord l'algorithme A suivant, inspiré de la solution naturelle en mode SIMD.

P_1 (le maître) crée des processus "*esclave*" sur chacune des autres stations (P_i, $2 \leq i \leq N$) puis envoie à chacune d'entre elles le ième paquet de $\frac{K}{N}$ lignes de la matrice M. P_1 exécute ensuite fft sur les $\frac{K}{N}$ premières lignes de M puis se met en attente des transformées de Fourier exécutées sur chaque autre paquet de $\frac{K}{N}$ lignes de M par les esclaves. Une fois reçues et stockées dans la matrice N, P_1 envoie à chaque esclave P_j le jème paquet de $\frac{K}{N}$ colonnes sur lesquelles P_j effectue encore une fois fft. Enfin P_1, qui a calculé pendant ce temps la transformée du premier paquet de colonnes, réceptionne les transformées effectuées par les autres stations et les stocke dans $\hat{M}$.

(1.1) Quel est l'ordre de grandeur du temps d'exécution de fft en fonction de la taille du tableau en argument, si fft est un algorithme de transformée de Fourier rapide (on pourra se permettre d'être imprécis sur les constantes multiplicatives et additives pour le calcul du nombre d'opérations élémentaires effectuées par l'algorithme) ? Justifier en quelques lignes.

Ecrire le processus "*maitre*" et "*esclave*" en PVM correspondant à la spécification informelle de l'algorithme A. On ne cherchera ni à coder fft ni la structure *complex*, ni les fonctionnalités *pkcmplx*, *upkcmplx*.

(1.2) Quel est le temps de communication ? Y a t-il a priori attente d'établissement des communications ?

(1.3) Quel est le temps de calcul pour cet algorithme ?

(1.4) Quel est le problème avec cet algorithme si les α_i étaient différents ? Comment pourrait-on le modifier si ces coefficients étaient connus de P_1 (on ne cherchera pas à réécrire les programmes correspondants) ?

Réponse :

(1.1) Aux constantes près, comme il s'agit d'un algorithme "divide and conquer" dichotomique, l'ordre de grandeur du temps d'exécution est de $nlogn$.

Le code pour le maître est,

```
struct complex M[K][K];
int tids[N-1];

fft_une_dimension()
{ int i;
  for(i=1; i<=N-1; i++)
  { pkcmplx(&M[i*K/N][0],K*K/N,1);
    pvm_send(tids[i-1],0);
  }
  for(i=0; i<K/N; i++) fft(M[i],K);
  for(i=1; i<=N-1; i++)
  { pvm_recv(tids[i-1],0);
    upkcmplx(&M[i*K/N][0],K*K/N,1);
  }
}

transpose()
{ int i,j;
  for(i=0; i<K; i++)
    for(j=0; j<i; j++)
```

```
        M[j][i]=M[i][j];
}

maitre()
{ pvm_spawn("esclave",NULL,0,0,N-1,tids);
  fft_une_dimension();
  transpose();
  fft_une_dimension();
  transpose();
}
```

Le code pour les esclaves est,

```
fft_une_passe()
{ int i,tid=pvm_parent();
  complex T[K/N][K];
  pvm_recv(tid,0);
  upkcmplx(T,K*K/N,1);
  for(i=0; i<K/N; i++) fft(T[i],K);
  pkcmplx(T,K*K/N,1);
  pvm_send(tid,0);
}

esclave()
{ fft_une_passe();
  fft_une_passe();
}
```

(1.2) Le temps de communication du maître est $2(N-1)(\frac{K^2}{N}\gamma+\delta)$ pour les envois et $2(N-1)\frac{K^2}{N}\alpha_1$ pour les réceptions. Le temps de communication d'un esclave est de même $2(\frac{K^2}{N}\gamma + \delta)$ et $2\frac{K^2}{N}\alpha_i$ respectivement.

L'esclave numéro P_i attend $i-1$ packings et envois pour la réception des données lors de sa première passe. Il attend en outre la transposition et $N-i+1$ unpackings et réceptions lors de sa deuxième passe.

Le maître n'attend pas pour la réception des données des esclaves dès lors que les esclaves sont aussi rapides que le maître et que le nombre d'esclaves est supérieur ou égal à deux.

Par contre si les α_i étaient différents, le maître devrait attendre les esclaves les plus lents.

(1.3) C'est le temps de calcul du maître plus son temps de communication.

- `spawn` : τ
- `transpose` : $(K^2-2K)\alpha_1$
- `fft` : $2\frac{K}{N}\theta(KlogK)\alpha_1$
- envois : $2(N-1)(\frac{K^2}{N}\gamma+\delta)$
- réceptions : $2(N-1)\frac{K^2}{N}\alpha_1$
- packing et unpacking : $4(N-1)\frac{K^2}{N}\alpha_1$

(1.4) Si les α_i étaient différents, ce serait la plus lente des stations qui imposerait la cadence. On pourrait modifier l'algorithme en connaissant les coefficients, de telle façon à répartir plus de lignes (respectivement colonnes) sur les stations les plus rapides.

Plus précisément P_i doit traiter $\frac{1/\alpha_i}{\sum 1/\alpha_i}K$ lignes.

2 Question :

(2.1) Indiquer les modifications à apporter aux programmes *maitre* et *esclave* de la question (1.1) afin que chaque esclave gère lui-même l'envoi de ses parties de colonnes directement (sans passer par P_1) aux stations correspondantes.

(2.2) Y gagne-t-on en temps par rapport au cas (1.1) (on négligera le temps de passage des *tid*s des tâches créées par P_1 à toutes les autres stations) ?

Réponse :

(2.1) Une fois la transformée en ligne calculée sur une station P_i (paquet de ligne p_i), on peut exécuter les *pvm_send* des paquets de colonnes c_j vers les $P_1, \cdots, P_{i-1}$, puis dans un ordre quelconque les *pvm_recv* des paquets de colonnes c_i appartenant aux stations $P_1, P_2, \cdots, P_{i-1}, P_{i+1}, \cdots P_N$ et les *pvm_send* restants (colonnes $c_{i+1}, \cdots, c_N$). Ceci assure qu'il n'y aura pas d'interblocage. On peut trouver des algorithmes de transposition plus rapides sur certains réseaux, mais ce n'est pas nécessaire ici.

(2.2) On y gagne en temps de communication, $(N-1)(\frac{K^2}{N}\gamma + \delta)$ pour les envois et $(N-1)\frac{K^2}{N}\alpha$ pour les réceptions, plus en temps de packing/unpacking : $2(N-1)\frac{K^2}{N}\alpha$.

3 Question :

On propose un algorithme B qui répartit le travail différement. L'idée est de suivre une variante de la méthode du cours (création récursive).

(3.1) Montrer que la transformée de Fourier discrète bi-dimensionelle de $M = (M_{i,j})_{1 \leq i,j \leq K}$ pour K pair, s'exprime en fonction des transformées de Fourier des quatre matrices M^{pp}, M^{ip}, M^{pi} et M^{ii} définies comme suit : $M^{pp}_{i,j} = M_{2i,2j}$, $M^{ip}_{i,j} = M_{2i-1,2j}$, $M^{pi}_{i,j} = M_{2i,2j-1}$ et $M^{ii}_{i,j} = M_{2i-1,2j-1}$ pour $1 \leq i,j \leq K/2$.

(3.2) En déduire un algorithme parallèle maître/esclave résolvant notre problème pour N de la forme $\frac{4^h-1}{3}$, où le maître P_1, puis ses esclaves, exécutent h *spawn* récursifs de quatre nouveaux esclaves à chaque fois. On n'aura pas besoin pour cette question d'écrire le programme en PVM. On se contentera d'indiquer l'activité des stations au cours du temps, ainsi que les communications entre elles.

(3.3) Peut-on modifier l'algorithme trouvé en (3.2) de façon à minimiser le temps inactif de chaque station ? Indication : le maître, puis ses esclaves, exécutent toujours h *spawn* récursifs de nouveaux esclaves et on pourra prendre $N = 4^{h-1}$.

(3.4) Comparer les temps de communication et le temps global de l'algorithme (3.2) et (3.3) avec ceux des questions (1.2) et (1.3).

(3.5) (Question facultative) Modifier l'algorithme de la question (3.2) de telle façon que P_1 ne fasse qu'un *spawn* unique des esclaves correspondant aux feuilles de l'arbre de création des processus du (3.2) (on choisira N au mieux). Obtiendrait-on un algorithme plus efficace ? Répondre en quelques lignes.

Réponse :

(3.1) On suppose $i, j \leq \frac{K}{2}$:

$$\hat{M}_{ij} = \frac{1}{K^2} \sum_{k=0}^{K} \sum_{l=0}^{K} M_{kl} e^{-\frac{2i\pi}{K}ik} e^{-\frac{2i\pi}{K}jl}$$

$$
\begin{aligned}
&= \frac{1}{4}\left(\frac{1}{K/4}\sum_{k\ \mathrm{pair}, l\ \mathrm{pair}} M_{kl}e^{-\frac{2i\pi}{K}ikjl} + \frac{1}{K/4}\sum_{k\ \mathrm{impair}, l\ \mathrm{pair}} M_{kl}e^{-\frac{2i\pi}{K}ikjl} + \right.\\
&\quad \left. \frac{1}{K/4}\sum_{k\ \mathrm{pair}, l\ \mathrm{impair}} M_{kl}e^{-\frac{2i\pi}{K}ikjl} + \frac{1}{K/4}\sum_{k\ \mathrm{impair}, l\ \mathrm{impair}} M_{kl}e^{-\frac{2i\pi}{K}ikjl}\right)\\
&= \frac{1}{4}\left(\frac{1}{K/4}\sum_{k=0}^{\frac{K}{2}}\sum_{l=0}^{\frac{K}{2}} M_{2k,2l}e^{-\frac{2i\pi}{K}i2kj2l} + \frac{1}{K/4}\sum_{k=0}^{\frac{K}{2}}\sum_{l=0}^{\frac{K}{2}} M_{2k+1,l}e^{-\frac{2i\pi}{K}i(2k+1)j2l} + \right.\\
&\quad \left. \frac{1}{K/4}\sum_{k=0}^{\frac{K}{2}}\sum_{l=0}^{\frac{K}{2}} M_{2k,2l+1}e^{-\frac{2i\pi}{K}i2kj(2l+1)} + \frac{1}{K/4}\sum_{k=0}^{\frac{K}{2}}\sum_{l=0}^{\frac{K}{2}} M_{2k+1,2l+1}e^{-\frac{2i\pi}{K}i(2k+1)j(2l+1)}\right)\\
&= \frac{1}{4}\left(\hat{M}_{ij}^{pp} + e^{-\frac{2i\pi}{K}i}\hat{M}_{ij}^{ip} + e^{-\frac{2i\pi}{K}j}\hat{M}_{ij}^{pi} + e^{-\frac{2i\pi}{K}(i+j)}\hat{M}_{ij}^{ii}\right)
\end{aligned}
$$

(3.2) Tous les esclaves sont créés en parallèle jusqu'à l'étape h où les feuilles calculent la `fft` séquentiellement. Dans une deuxième phase les esclaves remontent la FFT avec le calcul (3.1).

(3.3) On peut minimiser le temps inactif de chaque station en ne "spawnant" jamais la quatrième branche mais en conservant cette exécution locale.

(3.4) Le temps de communication de (3.2) est $2(h-1)[K^2(\gamma+\alpha)+\delta]$. Celui de (3.3) est $2h[3\frac{K^2}{4}(\gamma+\alpha)+\delta]$. Il est supérieur à celui de l'algorithme A dès que $h \geq 3$.

Le temps de calcul de la procédure séquentielle `fft` pour P_i aux feuilles (dont la taille est $\frac{K}{2^{h-1}}$) est

$$2\frac{K}{2^{h-1}}\theta(\frac{K}{2^{(h-1)}}log(\frac{K}{2^{(h-1)}}))\alpha_i$$

soit un ratio favorable de $\frac{1}{2}logN$ pour (3.3) vis-à-vis de l'algorithme A et de $\frac{3}{8}logN$ pour (3.2). Il faut ajouter le calcul (3.1) de la FFT, $\theta(\frac{K^2}{4^n})\alpha_i$ pour un esclave P_i à la profondeur $0 \leq n < h$.

(3.5) Il suffit de faire effectuer le tri dans le tableau M à répartir avant le *spawn*, séquentiellement sur P_1. Il y a donc un overhead séquentiel de calcul et autant d'esclaves après (4^{h-1}). Donc on perd plutôt.

4 Question :

Dans cette partie, on suppose que les α_i peuvent être très différents les uns des autres et ne sont connus par aucun des P_i. On peut même s'autoriser pour un nombre de stations $t < N-1$ parmi les P_2 à P_N un α_i égal à plus l'infini, modélisant ainsi la panne juste après la connection à *pvmd* (panne immédiate). Dans ce cas, on supposera qu'il n'y a aucun moyen pour aucune des stations de détecter la panne d'une autre des stations.

(4.1) Quel est le problème avec les algorithmes précédents ?

(4.2) Pour pallier le problème des pannes, indiquer les modifications à faire par rapport au programme écrit en (1.1), sans réécrire les programmes correspondants, si on s'autorise l'utilisation de *pvm_trec* qui rajoute un argument de "timeout" t à la réception bloquante d'un message (si aucun message ne vient avant le temps t, *pvm_trec* renvoie un code indiquant la non-réception). Y a t-il une façon de choisir à priori t ? Comment faire alors ? Peut-on optimiser la répartition des calculs au-delà de la simple gestion des pannes immédiates ?

Réponse :

(4.1) Quand on ne connaît pas, et quand on ne peut pas déterminer les processeurs morts, ou leurs vitesses relatives, l'allocation statique des tâches comme au 2, 3 ou au 4 n'est pas bonne. Si un des α_i égale $+\infty$ alors le maître se bloque, et si les α_i sont très différents cela occasionne beaucoup d'attente.

(4.2) On décide qu'après le temps t les stations qui n'ont pas répondu sont mortes et on les ignore. Le problème est de choisir t. Si t fait décider que toutes sont mortes, on double t et on recommence ainsi de suite. Comme on a supposé que toutes les stations ne pouvaient pas mourir, il existe une borne au-dessus de laquelle l'une d'entre elles répondra.

5 Question subsidiaire :

On fait les mêmes suppositions sur les α_i qu'à la question 4.

Maintenant on s'autorise l'utilisation de réceptions non-bloquantes (asynchrones). L'idée est de simuler une mémoire partagée (contenant essentiellement M) par un "serveur" (P_1). Les accès par les autres stations ("clients") se font à travers des communications asynchrones : des messages avec le TAG "write" correspondent à une demande faite par un client d'écrire son paquet (de lignes ou de colonnes) dans M, stockée localement dans la mémoire de P_1. Des messages avec le TAG "read" correspondent à une demande de lecture d'un paquet de lignes faite par un client au serveur P_1. On ne s'intéressera pas ici à la façon réelle d'implémenter ce client/serveur en PVM (par signaux ou "active polling") et on simplifiera l'écriture des algorithmes en pseudo-code, en rajoutant un paramètre "*handler*" aux réceptions asynchrones, qui représente une fonction (définie par le programmeur) exécutée au moment de la réception d'un message avec le bon TAG.

La modification de l'algorithme A est alors comme suit :

- En plus de M, P_1 gère une collection de drapeaux (variables booléennes) : un drapeau $d(l)$ (respectivement $d(c)$) est associé à chaque ligne l (respectivement à chaque colonne c) de M. Ces drapeaux sont utilisés pour savoir (et faire savoir) si une ligne (respectivement une colonne) est utilisée par un client. Les drapeaux sont initialement à faux.
- Chaque esclave demande la ligne 1, puis la ligne 2 etc. Le premier qui l'obtient fait son calcul de fft puis demande à réécrire dans M la ligne correspondante. Quand tous les drapeaux des lignes sont à vrai, les esclaves font la compétition pour calculer les transformées en colonne cette fois-ci.

(5.1) On suppose d'abord que chacun des clients demande d'abord au serveur la valeur du drapeau de la ligne 1, de la ligne 2, etc. jusqu'à en trouver un à la valeur faux. A ce moment là, le client demande la lecture de la ligne correspondante (ce qui met automatiquement le drapeau correspondant à vrai). Exhiber une exécution avec un certain choix des α_i et de N pour laquelle il y a redondance des calculs.

(5.2) Maintenant, on suppose que les clients demandent au serveur la première ligne (ou colonne) dont le drapeau est à faux. Le serveur étant purement séquentiel, quelque soit la rapidité des différents clients, il répond aux requêtes dans un certain ordre. Montrer qu'alors on peut assurer une non-redondance des calculs (on se contentera de montrer que le type d'exemple du (5.1) ne peut pas se produire).

(5.3) Montrer que la méthode trouvée au (5.2) résoud le problème des pannes immédiates, mais pas d'une panne qui pourrait se produire à n'importe quel moment du calcul. Cette méthode sera-t-elle vraisemblablement plus efficace que celle du (4.2) ?

Réponse :

(5.1) Le problème est d'avoir dissocié le test du drapeau, de sa modification. Donc si on a un processeur qui se glisse entre le test et la demande de lecture, pour faire lui-même le test, on peut avoir deux processeurs exécutant le même calcul.

(5.2) Le serveur ne peut plus donner la même ligne (ou colonne) à deux clients. Les calculs sont donc répartis sans redondance.

(5.3) Un client qui tombe en panne après avoir lu une ligne (ou colonne) aura réservé le drapeau correspondant et finalement va bloquer le calcul global. Par contre en cas de pannes immédiates le calcul ira jusqu'à son terme si au moins un client reste actif, car les clients en panne immédiate ne peuvent pas lire de ligne (ou colonne).

La compétition entre les processeurs avantage en moyenne les plus rapides : cette méthode sera plus efficace (quand les α_i sont différents).