

UNIVERSITÉ PIERRE ET MARIE CURIE

TRAVAUX DIRIGÉS

HPC

Calcul Haute Performance

Auteur:
Benjamin BIELLE

Responsable:
Pierre FORTIN

June 19, 2015

Sommaire

1	TD 1: Algorithme Parallèle	2
	Exercice 1: Tri pair-impair	2
	Exercice 2: Produit Matriciel	3
2	TD 2-3: Ensemble de Mandelbrot	5
	Exercice 1: Ensemble de Mandelbrot	5
3	TD 4: Convolution	8
	Exercice 1: Convolution	8
4	TD 5-6: MPI-OpenMP	10
	Exercice 1: Calcul de π	10
	Exercice 2: 1 thread \leftrightarrow 1 <i>case</i>	11
	Exercice 3: Produit Matriciel	11
	Exercice 4: Calcul de fractales	12
	Exercice 5: Convolution	12
	Exercice 6: MPI-OpenMP	12
5	TD 8: Produit matriciel et routines BLAS	14

Chapter 1

TD 1: Algorithme Parallèle

Exercice 1: Tri pair-impair

On suppose que nous disposons des procédures suivantes :

- *compare echage min*
- *compare echage max*

Question 1:

Nous pouvons simplement donner la boucle permettant de faire le calcul.

```
P : Nombre de processus
N : Donnees

for (i=0; i < P-1; i++)
{
    if (i%2 == id%2)
        compare_echange_min(id+1);
    else
        compare_echange_max(id-1);
}
```

Question 2:

Maintenant nous disposons de $P = \frac{N}{k}$ processeurs. Nous utilisons les procédures *compare echage mins* et *compare echage maxs* qui prennent en plus de l'identifiant du voisin, le nombre de données, ainsi nous pouvons écrire.

```
for (i=0; i < N-1; i++)
{
    if (i%2 == id%2)
        compare_echange_mins(id+1, k);
    else
        compare_echange_maxs(id-1, k);
}
```

Question 3:

On suppose que notre algorithme de tri optimal est en $\Theta(N * \ln(N))$ pour le tri initial local à chaque processeur.

Essayons de trouver la complexité théorique en parallèle.

$$\Theta(k * \log(k)) + P * \Theta(k) = \Theta\left(\frac{N}{P} * \log\left(\frac{N}{P}\right) + \Theta(N)\right)$$

Notre accélération peut s'exprimer ainsi :

$$SpeedUp(N, P) = \frac{\Theta(N * \log(N))}{\Theta\left(\frac{N}{P} * \log\left(\frac{N}{P}\right)\right)} + \Theta(N)$$

$$\lim_{N \rightarrow \infty} SpeedUp(N, P) = \frac{\Theta(N * \log(N))}{\Theta\left(\frac{N}{P} * \log\left(\frac{N}{P}\right)\right)} \quad (1.1)$$

$$= \Theta(P) * \frac{\Theta(N * \log(N))}{\Theta(N * \log\left(\frac{N}{P}\right))} \quad (1.2)$$

$$= \Theta(P) * \frac{\Theta(\log(N))}{\Theta(\log(N) - \log(P))} \quad (1.3)$$

$$= \Theta(P) * \frac{\Theta(1)}{\Theta\left(1 - \frac{\log(N)}{\log(P)}\right)} \quad (1.4)$$

$$= \Theta(P) \quad (1.5)$$

Exercice 2: Produit Matriciel

Nos matrices sont des matrices carrées de taille $N * N$ et nous disposons de P processeurs.

Question 1:

On attribue un bloc de $h = \frac{N}{P}$ lignes complètes de la matrice C à chaque processeur. Chaque processeur est identifié par numéro $0 \leq r \leq P - 1$.

Al : tableau de taille $h * N$
Bl : tableau de taille $N * N$

On charge dans Al, les lignes de $(r * h)$ à $(r + 1) * h - 1$ de la matrice A.
On charge dans Bl toute la matrice B.

```
for (i=0; i < h-1; i++)
  for (j=0; j < N-1; j++)
  {
    s=0
    for (k=0; k < N-1; k++)
      s += Al(i, k) * Bl(k, j)
    C(i, j) = s
  }
```

Question 2:

Cet algorithme pose un certain inconvenient. En effet, nous chargeons toute la matrice B sur chaque processeur ce qui prend trop de place.

Question 3:

Nous considérons maintenant que les processeurs forment un anneau.

```
Al : tableau de taille h * N
Cl : tableau de taille h * N
Bl : tableau de taille N * h

for (e=0; e < P-1; e++)
{
    for (i=0; i < h-1; i++)
        for (j=0; j < h-1; j++)
        {
            s=0
            for (k=0; k < N-1; k++)
                s += Al(i, k) * Bl(k, j)
            Cl(i, ((r+e)%P)*h+j) = s
        }
    Envoie de Bl sur le processeur (r-1) % P
    Reception de Bl depuis le processeur (r+1) % P
}
```

Chapter 2

TD 2-3: Ensemble de Mandelbrot

Exercice 1: Ensemble de Mandelbrot

Algorithme Séquentiel:

```
b = ymax
for (i=0; i < h-1; i++)
{
    a = xmin
    for (j=0; j < w-1; j++)
    {
        image[j+1*w] = ab2color(a, b, color)
        a += (xmax - xmin) / w-1
    }
    b -= (ymax - ymin) / h-1
}

ab2color(a, b, color)
{
    x = a
    y = b
    for (j=0; j < w-1; j++)
    {
        x1 = x^2 - y^2 + a
        y1 = 2*x*y + b
        x = x1
        y = y1
        if (x^2 + y^2 >= 4)
            return i % 255
    }
    return 255
}
```

Question 1:

Le calcul de chaque pixel est indépendant donc on peut paralléliser le calcul des pixels.
On découpe l'image en zone de pixel.

Question 2:

Nous allons maintenant paralléliser l'algorithme séquentiel de façon équilibrée les données à calculer sur chaque processeur.
On utilisera les transmissions basiques de MPI (MPI_Send et MPI_Recv).

```

k = MPI_Rank()
b = ymax - k * (h/N) * (ymax - ymin)/h-1
for (i=0; i < (h/N)-1; i++)
{
    a = xmin
    for (j=0; j < w-1; j++)
    {
        image[j+1*w] = ab2color(a, b, color)
        a += (xmax - xmin) / w-1
    }
    b -= (ymax - ymin) / h-1
}

if (h == 0)
{
    for (i=1; i < N-1; i++)
    {
        MPI_Recv(&image[i - w * (h/N)], w*(h/N), MPLCHAR, i, MPLCOMM_WORLD, NULL)
    }
}
else
    MPI_Send(image, w*(h/N), MPLCHAR, 0, MPLCOMM_WORLD)

ab2color(a, b, color)
{
    x = a
    y = b
    for (j=0; j < w-1; j++)
    {
        x1 = x^2 - y^2 + a
        y1 = 2*x*y + b
        x = x1
        y = y1
        if (x^2 + y^2 >= 4)
            return i % 255
    }
    return 255
}

```

Question 3:

Maintenant nous allons étudier les performances de nos algorithmes (séquentiel et parallèle).

$$\text{Accélération} = \frac{T_{seq}}{T_{para}} \text{ et Efficacité} = \frac{\text{Accélération}}{nbproc}$$

	séquentiel	2 processeurs	4 processeurs	8 processeurs
temps (s)	3.75484	1.89138	1.89521	1.57206
accélération		1.9852382916	1.9812263549	2.3884839001
efficacité		≈99.29%	≈49.53%	≈29.85%

On peut en conclure que notre algorithme parallèle n'est pas très efficace, en effet on aurait dû observer une augmentation de l'efficacité en fonction du nombre de processeurs or nous observons complètement l'inverse.

Question 4:

Pour résoudre notre problème de perte d'efficacité, nous allons mettre en place un équilibrage de charge.

Nous allons nous inspirer du modèle maître-esclave.

Master

```
hlignes
indice_bloc = N
i

while indice_bloc < (h/hlignes)
{
    MPI_Recv(&i, 1, MPI_INT, MPLANY_SOURCE, REQ, MPLCOMM_WORLD, &status)
    MPI_Send(&indice_bloc, 1, MPI_INT, status.source, REQ, MPLCOMM_WORLD)
    indice_bloc++
}

for (i=1; i < N; i++)
{
    MPI_Recv(&i, 1, MPI_INT, MPLANY_SOURCE, REQ, MPLCOMM_WORLD, &status)
    MPI_Recv(image[i*(h/hligne)], 1, MPI_INT, status.source, DATA, MPLCOMM_WORLD, &status)
    MPI_Send(&i, 1, MPI_INT, status.source, END, MPLCOMM_WORLD)
}
```

Slave

```
image[w * hlignes]
k = MPI_Rank() - 1

while tag != END
{
    b = ymax - k * (ymax - ymin / h-1)
    for (i=0; i < hlignes; i++)
    {
        for (j=0; j < hlignes-1; j++)
        {
            a = xmin
            for (p=0; p < w-1; p++)
            {
                image[p+1*w] = ab2color(a, b, color)
                a += (xmax - xmin) / w-1
            }
            b -= ((ymax - ymin) / hlignes-1) * N
        }
    }

    MPI_Send(&k, 1, MPI_INT, 0, REQ, MPLCOMM_WORLD)
    MPI_Send(image, 1, MPI_INT, 0, DATA, MPLCOMM_WORLD)
    MPI_Recv(&k, 1, MPI_INT, i, MPLCOMM_WORLD, &status)
    tag = status.tag
}
```

Nous pouvons faire quelques optimisations, par exemple en ajoutant 2 indices de bloc pour l'esclave, ainsi on peut envoyer les données de manière asynchrone et donc les esclaves continuent leur calcul pendant qu'ils envoient des données (pas de perte de temps pendant les communications) : Recouvrement des communications par du calcul.

Maintenant on peut observer les performances de notre équilibrage de charge.

	séquentiel	2 processeurs	4 processeurs	8 processeurs	16 processeurs
temps (s)	3.75484	3.74962	1.31427	0.662957	0.285439
accélération		1.00138412919	2.85707111481	5.66377608201	13.154614471
efficacité		≈50%	≈71%	≈70%	≈82%

Question 5:

cf TME

Chapter 3

TD 4: Convolution

Exercice 1: Convolution

Question 1:

Dans la fonction *Convolution*, on doit préparer un tampon intermédiaire au lieu de faire le calcul directement sur l'image car nous avons besoin de ces valeurs pour calculer toute l'image ce qui pourrait nuire au reste du calcul.

Question 2:

On peut paralléliser le travail sur chaque pixel.

Question 3:

Pour le calcul de un pixel, nous avons 9 multiplications et 9 additions.

$$I * k = \sum_{i=-1}^1 \sum_{j=-1}^1 I(x + i, y + j) * k(i, j)$$

La complexité s'exprime donc en $\Theta(1)$.
Ici nous pouvons utiliser un équilibrage de charge statique car le nombre de calcul est constant.

Question 4:

Ici nous pouvons utiliser le découpage par bande car celui ci est plus simple.

Question 5:

A faire

Question 6:

```
convolution (choix, imageloc, hloc, w)
{
    unsigned char image2[hloc][w]
    int i, j

    for (i=1; i < hloc - 1; i++)
        for (j=1; j < w - 1; j++)
            image2[i][j] = filtre(choix, imageloc[i-1][j-1], imageloc[i+1][j+1])
}
```

arguments : fichier image, filtre , nombre iteration (nbIter)

Le processeur 0 lit le fichier image puis broadcast la hauteur **et** la largeur.
Il fait un "Scatter" de l'image globale

```
for (i=0; i < nbIter-1; i++)
{
    if (proc != 0)
        Envoyer ligne 1 au processus de rang proc - 1
    if (proc != N-1)
        Recevoir de (proc + 1) la ligne (hloc - 1)
        Envoyer la ligne (hloc - 2) au processus (proc + 1)
    if (proc != 0)
        Recevoir de (proc + 1) la ligne (hloc - 1)

    convolution(choix, imageloc, hloc, w)
}
```

Le processus 0 fait un "Gather" puis sauvegarde l'image

Question 7:

A faire

Chapter 4

TD 5-6: MPI-OpenMP

Exercice 1: Calcul de π

$$\pi \approx \sum_{i=0}^N \frac{4N}{N^2+i^2} = \frac{1}{N} \sum_{i=0}^N \frac{4}{1+\frac{i}{N}^2}$$

Version Séquentielle

```
res = 0
for (i=0; i <= N; i++)
{
    res += 4 / (1+pow(i/N, 2))
}
res *= 1/N
return res
```

Versions Parallélisées

Version 1

```
res = 0
#pragma omp parallel
{
    #pragma omp for reduction(+:res) private(i)
    for (i=0; i <= N; i++)
    {
        res += 4 / (1+pow(i/N, 2))
    }
}
res *= 1/N
return res
```

Version 2

```
res = 0
#pragma
#pragma omp for reduction(+:res) private(i)
    for (i=0; i <= N; i++)
    {
        #pragma omp atomic
        res += 4 / (1+pow(i/N, 2))
    }
}
```

```
res *= 1/N
return res
```

La version 3 ne diffère pas de la version 2, il faut juste remplacer le `atomic` par `critical`. Sur un dual-core, il faut faire un `export omp_num_thread=2` pour ne lancer que 2 threads. Ici nous allons faire nos tests pour un N fixé à 1 000 000 000.

	séquentiel	atomic	critical	reduction
temps (s)	8.7	≈ 60	≈ 30	4.79

Exercice 2: 1 thread ↔ 1 case

Version 1

```
tab[N]
#pragma omp parallel num_thread
{
    tab[omp_get_thread_num()] = omp_get_thread_num()
}
```

Version 2

```
tab[N]
int cpt=0
#pragma private (rang)
{
    rang = omp_get_thread_num()
    #pragma omp critical
    {
        tab[cpt++] = rang
    }
}
```

Petit bonus, une troisième version.

Version 2

```
tab[N]
int cpt=0
#pragma private (rang)
{
    rang = omp_get_thread_num()
    #pragma omp atomic capture
    {
        tab[cpt++] = rang
    }
}
```

Exercice 3: Produit Matriciel

Nous allons utiliser un équilibrage de charge statique (car tous les calculs sont identiques) et nous allons appliquer la parallélisation sur la boucle i (pour avoir un grain de calcul suffisant).

```

#pragma omp parallel
{
    #pragma omp for private (j, k, s)
    {
        for (i=0; i < N; i++)
            for (j=0; j < N; j++)
            {
                s=0
                for (k=0; k < N; k++)
                    s += A[i][k] * B[k][j]
                C[i][j]=s
            }
    }
}

```

Exercice 4: Calcul de fractales

Nous allons utiliser un équilibrage de charge dynamique car la répartition des calculs n'est pas égale, nous appliquerons notre parallélisation sur la boucle i (pourquoi ? cf exercice précédent).

```

#pragma omp parallel for schedule (dynamic) private (j, x, y)
{
    for (i=0; i < h; i++)
    {
        x = xmin
        y = ymin + i * incy
        for (j=0; j < w; j++)
        {
            image[i*w+j] = ab2color(x, y, color);
            x += incx
        }
    }
}

```

Exercice 5: Convolution

cf TME

Exercice 6: MPI-OpenMP

- Le master ne travaille pas (pas de calculs utiles).
- "Surcharge" d'esclaves (hiérarchie des masters).
- "Extensibilité" mémoire (cf cours).
- Granularité de calcul suffisante.

Comment y remédier ?

Sans modifier le code

Nous pouvons utiliser des noeuds mono-cœur, nous pouvons faire des modifications dans le fichier hostfile (on double la première ligne ce qui nous donne 10 machines dans le hostfile mais 11 processus).

En modifiant le code mais on garde le même algorithme

En version mono-thread, on peut utiliser une boucle avec calcul et communications mais comment déterminer la quantité de calcul à effectuer à chaque itération ?
En version multi-thread, on utilise deux threads, un pour les communications et un autre pour les calculs (cf `MPI_THREAD_MULTIPLE`).

En modifiant le code et l'algorithme

Pour cela nous utilisons un équilibrage de charge dynamique de type "auto-régulé" ou "vol de tâche".

A faire

Chapter 5

TD 8: Produit matriciel et routines BLAS

Question 1:

Cette question sera vue en tme.

Question 2:

N est une puissance de 2 dans notre cas, le stride se trouve être le pas (il permet d'arrêter la récursion). Nous allons utiliser un découpage de la matrice en bloc de N puissance de 2.

Exemple :

$$\mathbf{A} = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \quad \mathbf{C} = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

- $M1 = (A_{1,1} + A_{2,2}) * (B_{1,1} + B_{2,2})$
- $M2 = (A_{2,1} + A_{2,2}) * B_{1,1}$
- $M3 = A_{1,1} * (B_{1,2} - B_{2,2})$
- $M4 = A_{2,2} * (B_{2,1} - B_{1,1})$
- $M5 = (A_{1,1} + A_{1,2}) * B_{2,2}$
- $M6 = (A_{2,1} - A_{1,1}) * (B_{1,1} + B_{1,2})$
- $M7 = (A_{1,2} - A_{2,2}) * (B_{2,1} + B_{2,2})$

Ce qui nous donne la matrice

$$\mathbf{C} = \begin{pmatrix} M1 + M4 - M5 + M7 & M3 + M5 \\ M2 + M4 & M1 - M2 + M3 + M6 \end{pmatrix}$$

Maintenant un peu de "programmation".

```
mm(...)
{
    if (n > seuil)
    {
        p = n/2
        (1) mm(crow, ccol, crow, arow, acol, brow, bcol, p, stride, A, B, C)
        (2) mm(crow, ccol, crow, arow, acol+p, brow+p, bcol, p, stride, A, B, C)
        (3) mm(crow, ccol+p, crow, arow, acol, brow, bcol+p, p, stride, A, B, C)
        (4) mm(crow, ccol+p, crow, arow, acol+p, brow+p, bcol+p, p, stride, A, B, C)
        (5) mm(crow+p, ccol, crow, arow+p, acol, brow, bcol, p, stride, A, B, C)
        (6) mm(crow+p, ccol, crow, arow+p, acol+p, brow+p, bcol, p, stride, A, B, C)
        (7) mm(crow+p, ccol+p, crow+p, arow+p, acol, brow, bcol+p, p, stride, A, B, C)
        (8) mm(crow+p, ccol+p, crow+p, arow+p, acol+p, brow+p, bcol+p, p, stride, A, B, C)
    }
    else
    {
        for (i=0; i < n; i++)
            for (j=0; j < n; j++)
                for (k=0; k < n; k++)
                    C[(crow+i)*stride+(ccol+j)] += A[(arow+i)*stride+(acol+k)] *
                                                    B[(brow+i)*stride+(bcol+j)]
    }
}

appel : mm(0, 0, 0, 0, 0, 0, N, N, A, B, C)
```

Question 3:

Ici nous allons découper notre matrice en "méga-bloc", notre parallélisation s'effectuera sur ces "méga-bloc".

Nous reprenons notre code de tout à l'heure en prenant soin d'arranger les tâches.

```
mm(...)
{
    if (n > seuil)
    {
        p = n/2

        #pragma omp task
        {
            (1) mm(crow, ccol, crow, arow, acol, brow, bcol, p, stride, A, B, C)
            (3) mm(crow, ccol+p, crow, arow, acol, brow, bcol+p, p, stride, A, B, C)
            (5) mm(crow+p, ccol, crow, arow+p, acol, brow, bcol, p, stride, A, B, C)
            (7) mm(crow+p, ccol+p, crow+p, arow+p, acol, brow, bcol+p, p, stride, A, B, C)
        }

        taskwait

        #pragma omp task
        {
            (2) mm(crow, ccol, crow, arow, acol+p, brow+p, bcol, p, stride, A, B, C)
            (4) mm(crow, ccol+p, crow, arow, acol+p, brow+p, bcol+p, p, stride, A, B, C)
            (6) mm(crow+p, ccol, crow, arow+p, acol+p, brow+p, bcol, p, stride, A, B, C)
            (8) mm(crow+p, ccol+p, crow+p, arow+p, acol+p, brow+p, bcol+p, p, stride, A, B, C)
        }
    }
    else
    {
        for (i=0; i < n; i++)
            for (j=0; j < n; j++)
                for (k=0; k < n; k++)
                    C[(crow+i)*stride+(ccol+j)] += A[(arow+i)*stride+(acol+k)] *
                                                    B[(brow+i)*stride+(bcol+j)]
    }
}
```



```
}
}
```

```
appel : mm(0, 0, 0, 0, 0, 0, N, N, A, B, C)
```

Et maintenant le cadeau Bonux !!
 Voici un petit graphique montrant les performances en fonction de la taille des matrices (se sont des matrices carrées) en GigaFlop/s.

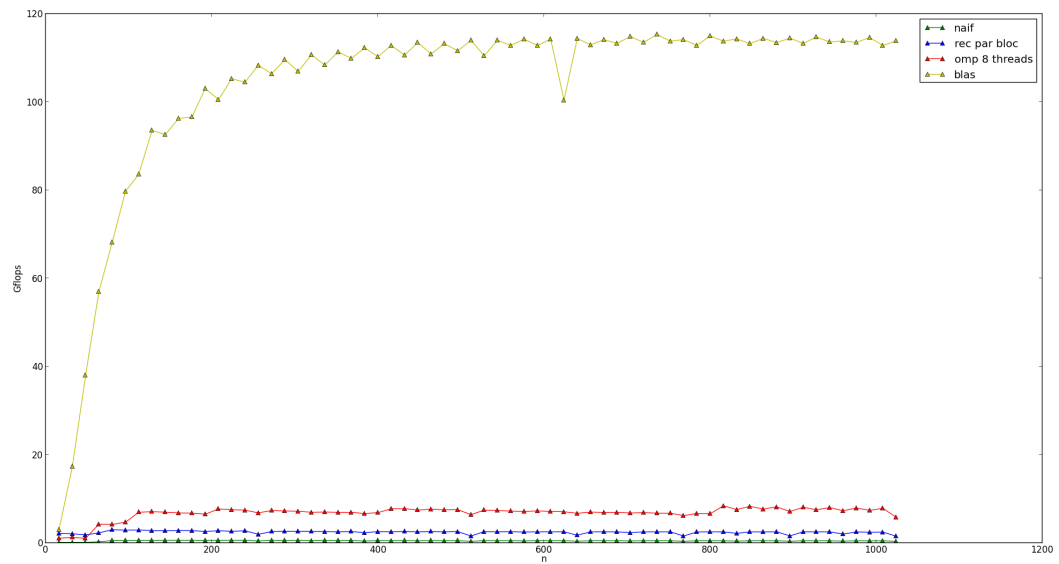


Figure 5.1: Performance