Université de Lille

FST- Département Informatique

ACT

2018–2019
Master 1 d'Informatique

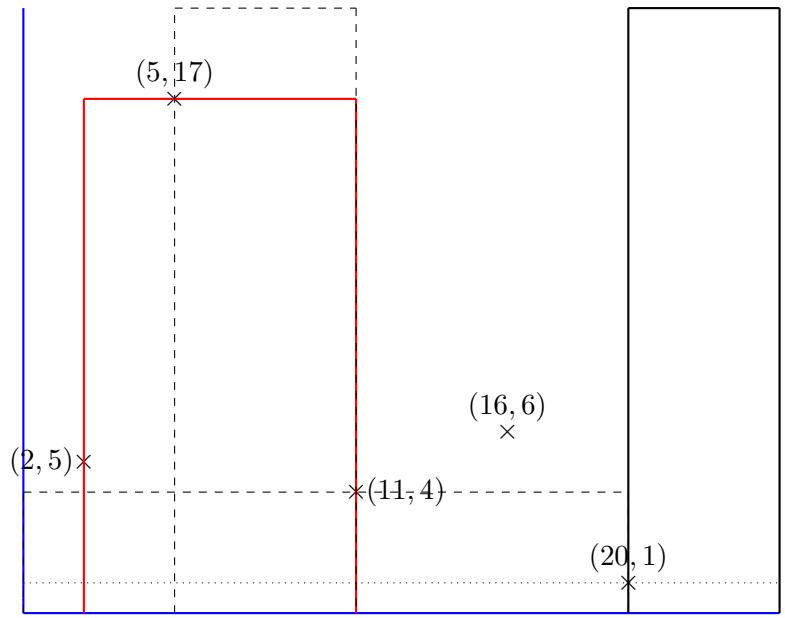
TP1 - Diviser pour régner
Le plus grand rectangle.

Objectif pédagogique : Ce premier TP aborde un problème d'algorithmique simple [1]. L'objectif pédagogique est de réfléchir à différentes solutions algorithmiques tout en se posant les questions de complexité et de correction. Une façon de résoudre efficacement le problème utilise le paradigme « Diviser et Régner ». Pour ceux qui veulent aller plus loin, nous vous proposons également d'expérimenter la parallélisation de votre code « Diviser et Régner ».

Le problème : Soit la région rectangulaire du plan déterminée par le rectangle $(0,0)(0,h)(l,h)(l,0)$, h et l éant deux entiers positifs et soient n points à coordonnées entières à l'intérieur de cette région.

On souhaite dessiner un rectangle dont la base est sur l'axe des x, dont l'intérieur ne contienne aucun des n points et qui soit de *surface maximale*.

Par exemple si h=20, l=25, et qu'il y a 5 points (2,5), (5,17), (11,4), (16,6) (20,1), voici quelques exemples de rectangle qu'on peut dessiner : (5,0) (5,20) (11,20) (11,0) de surface 120, (0,0),(0,1),(25,1),(25,0) de surface 25, (0,0),(0,4),(20,4),(20,0) de surface 80, (20,0),(20,20),(25,20)(25,0) de surface 100, ... La surface maximale qu'on peut obtenir est 153 avec le rectangle (2,0) (2,17), (11,17) (11,0).



Donc l'entrée du problème est :
- l, h, deux entiers strictement positifs ;
- n, un entier positif ;
- les coordonnées - entières - des n points (x_i, y_i) , $0 < x_i < l, 0 < y_i < h$.

La sortie attendue est la surface maximum d'un rectangle vérifiant les contraintes.

A faire :

L'objectif du TP est de concevoir et implémenter un/des algorithmes pour ce problème, permettant si possible de traiter rapidement des données de grande taille. Si vous voulez pouvoir parallaléliser un de vos codes, vous devrez programmer en C, C++ ou Java.

1. Ce sujet est fortement inspiré d'un problème du site CodeChef

Pour pouvoir tester, quelques jeux de données sont à votre disposition. Attention à la taille des entiers manipulés. Pour les expérimentations, vous pouvez utiliser la plateforme http://contest.fil.univ-lille1.fr/ ou/et bien sûr enrichir votre code pour mesurer le temps d'exécution.

Pour vous guider, nous vous proposons des pistes mais vous pouvez bien sûr en explorer d'autres...

A rendre sur Prof pour la semaine du 1er octobre : les sources de vos algorithmes, un rapport synthétique sur l'expérimentation et l'évaluation théorique de vos algorithmes.

Q 1. *Une première approche*

Supposons les points triés par abscisse croissante - on pourra aussi supposer (quitte à les rajouter) qu'on a un point $(0, 0)$ et un point $(l, 0)$.

Une première approche peut être basée sur la remarque suivante : un rectangle de surface maximale respectant les contraintes a nécessairement deux sommets de la forme $(x_i, 0), (x_j, 0)$ avec $0 \leq i < j \leq n - 1$ (Pourquoi ?).

Comment exprimer la surface du rectangle de surface maximale respectant les contraintes et dont deux sommets sont $(x_i, 0), (x_j, 0)$? Pouvez-vous en déuire un algorithme en $\Theta(n^3)$? en $\Theta(n^2)$?

Implémentez un algorithme basé sur cette approche et testez-le. Que constatez-vous ? Analysez la complexité de votre algorithme. Codechef demande que l'algorithme s'exécute sur leur plateforme en moins de 1s pour n=100.000. Pensez-vous que ce soit compatible avec la complexité de votre algorithme ?

Q 2. *Diviser pour régner*

En poursuivant l'approche précédente, pouvez-vous concevoir un algorithme selon le paradigme « diviser pour régner » ?

Est-il plus efficace que le précédent ? Testez-le sur les jeux de données proposés. Comment se comporte-t-il ? Pourquoi ? Quelle est ca somplexité dans le meilleur des cas ? Dans le pire des cas ?

Q 3. *Hauteur limitée*

Si on fixe h à une valeur relativement petite, par exemple $h = 500$, pouvez-vous améliorer la complexité de votre algorithme ? Expérimentez.

Q 4. *Linéaire ?*

Supposons les points triés par abscisse croissante.

Proposez un algorithme en $\Theta(n)$ pour résoudre le problème.

Remarque : vous pouvez supposer pour simplifier qu'il y a au plus un point ayant une abscisse donnée, quitte à transformer la donnée pour qu'elle vérifie cette propriété.

Q 5. *Diviser pour régner en parallèle*

A l'aide de l'annexe A, parallélisez votre implémentation de l'algorithme de type « diviser pour régner ». Quelle performance obtenez-vous sur un processeur multi-cœur ? (précisez le nombre de cœurs utilisés)
Comment se compare cette implémentation parallèle à l'algorithme linéaire en séquentiel ?

A rendre sur Prof pour la semaine du 1er octobre : les sources de vos algorithmes, un rapport synthétique sur l'expérimentation et l'évaluation théorique de vos algorithmes.

A Parallélisme de tâches pour les algorithmes « diviser pour régner »

A.1 Principe

Le paradigme « diviser pour régner »offre généralement un parallélisme naturel qui peut être intéressant (et facile) à exploiter sur les processeurs multi-cœurs actuels. A la différence de la programmation dynamique, on décompose en effet avec ce paradigme un problème en sous-problèmes ne se recouvrant pas. Ces problèmes peuvent alors généralement être résolus en parallèle.

Le plus simple pour exploiter ce type de parallélisme sur votre machine est de s'appuyer sur le parallélisme de tâches. Au lieu de demander à l'utilisateur de gérer lui-même les threads de calcul, celui-ci n'aura qu'à indiquer les « tâches »qui peuvent être exécutées en parallèle du thread courant. Chaque tâche (code et données qui lui sont propres) sera alors placée dans un ensemble de tâches : au fur et à mesure de l'exécution du programme, chaque thread (le thread courant ou les autres threads disponibles) viendra prendre, et exécuter entièrement, une tâche dans l'ensemble jusqu'à ce que celui-ci soit vide. Ceci facilite la programmation parallèle, et permet d'équilibrer dynamiquement (au cours de l'exécution) la charge de calcul entre les threads (si les tâches n'impliquent pas toutes la même charge de calcul).

A.2 Exemple

A titre d'exemple, voilà comment paralléliser une implémentation en C du tri-rapide avec OpenMP [2] (un standard de programmation parallèle en mémoire partagée) :

```
void QuickSort(int tableau[], int debut, int fin){
  /* Cas terminal: */
  if(debut >= fin)
    return;

  /* ... choix du pivot ... */

  /* ... partitionnement autour du pivot ... */

  /* Traitement des sous-problemes : */
#pragma omp task
  QuickSort(tableau, debut, indice_pivot-1);
#pragma omp task
  QuickSort(tableau, indice_pivot+1, fin);
}
```

La directive de compilation #**pragma** omp task permet au compilateur (par exemple `gcc` avec l'option `-fopenmp`) de créer une tâche pour chaque appel récursif.

L'appel initial est effectué par un seul thread (directive single), dans une région parallèle OpenMP (directive parallel) :

```
#pragma omp parallel
#pragma omp single
  QuickSort(tableau, 0, taille-1);
```

Le nombre total de threads peut être indiqué par la variable d'environnement `OMP_NUM_THREADS`, et peut êre positionné par défaut au nombre de cœurs de la machine.

En évitant de créer des tâches trop « petites »(dont les coûts de création et de gestion ne sont pas amortis par le coût de calcul), on peut ainsi obtenir des accélérations de 3,5 sur 4 cœurs CPU pour ce tri-rapide (pour des tableaux de $N = 2^{27}$ éléments).

Au besoin, la directive taskwait permet d'indiquer une barrière de synchronisation : la tâche courante attend alors la terminaison de ses tâches « filles ». Les variables à partager entre les tâches mère et filles pourront être indiquées comme « partagées »avec la clause shared de la directive task.

Pour plus d'informations sur :

— la programmation OpenMP : voir la norme OpenMP disponible sur `http://www.openmp.org/specifications` (et les nombreux tutoriels disponibles sur le web) ;

2. http ://www.openmp.org/

— la programmation par tâches en OpenMP : voir *Tasking in OpenMP*, A. Duran, 5th International Workshop on OpenMP, 2009.

A.3 Implémentations

Voici quelques autres implémentations du parallélisme de tâches (les principes de création, gestion et synchronisation entre tâches étant similaires à OpenMP) :

— en C/C++ : OpenMP, Intel TBB [3], Intel Cilk Plus [4] ... ;

— en Java : on pourra essayer *Future* [5].

3. `https://www.threadingbuildingblocks.org/`

4. `https://www.cilkplus.org/`

5. `http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/Future.html` Voir par exemple : `https://blog.codecentric.de/en/2011/10/executing-tasks-in-parallel-using-java-future/`