Parallélisme, calcul formel et résolution de problèmes complexes.

Denis Naddef*, Jean-Louis Roch*, Denis Trystram*
[Denis.Naddef@imag.fr, Jean-Louis.Roch@imag.fr, Denis.Trystram@imag.fr]

Complexité algorithmique et Calcul Formel sont deux domaines très liés. D'une part, la definition de la complexité algorithmique peut être formulée sous un formalisme proche du calcul formel (par exemple pour les circuits arithmétiques). D'autre part, les techniques et outils du calcul formel peuvent être utilisés pour la construction d'algorithmes plus performants.

Ce cours est articulé en 3 parties. La première partie rappelle les principales classes de complexité, séquentielles mais aussi parallèles. La seconde partie est centrée sur la complexité parallèle (classification NC) et sa formulation sous forme de circuits arithmétiques. La technique dite de cascade algorithmique est présentée pour la construction d'algorithmes optimisant plusieurs critères simultanément; elle est illustrée sur quelques exemples, en particulier la résolution parallèle de systèmes linéaires denses.

Pour faire face à la complexité intrinsèque de certains problèmes, la construction d'algorithmes d'approximation est utilisée; nous étudions en particulier la résolution du problème de l'ordonnancement crucial en parallélisme.

La dernière partie montre, à travers la résolution du problème du voyageur de commerce (problème de référence en optimisation combinatoire), comment une formulation mathématique duale (en l'occurrence, sous forme de programme linéaire) peut être utilisée pour la résolution d'un problème qui n'admet pas d'approximation polynomiale.

Ce cours est basé principalement sur les documents suivants, dont il reprend certains paragraphes:

- *Complexité parallèle et Algorithmique PRAM*, Jean-Louis Roch, chapitre 5 de l'ouvrage Algorithmes parallèles - Analyse et Conception, G.Authié&al eds, Hermès, 1994
- *Parallel Computer Algebra*, Jean-Louis Roch et Gilles Villard, Tutorial ISSAC 97, Hawai, http://www-id.imag.fr/ jlroch/perso.html/ps/97-issac.ps.gz
- *Ordonnancement de programmes parallèles sur grappes : théorie versus pratique.*, Jean-Louis Roch, Actes du Congrès International *Algèbre linéaire et Arithmétique : Calcul Numérique, Symbolique et parallèle*, ALA 2001, Université Mohammed V, S. El Hajji éditeur, p. 131–144, 28–31 mai 2001.
- *Fondements théoriques pour la conception d'algorithmes efficaces de gestion de ressources*, Olivier Beaumont, Vincent Boudet, Pierre-François Dutot, Yves Robert et Denis Trystram, chapitre d'ouvrage à paraître 2004.
- *Polyhedral Theory and Branch-and-Cut Algorithms for the Symmetric TSP*, Denis Naddef, chapitre 2 de l'ouvrage *The Traveling Salesman Problem and its Variation*, G. Gutin&al. eds, Kluwer Academic Publishers, 2002

*Laboratoire ID-IMAG (UMR CNRS-INRIA-INPG-UJF 5132) – ENSIMAG – 51 Av. Jean Kuntzmann, F-38330 Montbonnot Saint-Martin

1 Complexité parallèle et calcul formel

1.1 Temps parallèle et nombre d'opérations

Soit A un algorithme résolvant un problème P; on suppose que l'instance P_n de P a $n^{O(1)}$ entrées. Sur le modèle séquentiel RAM, deux caractéristiques (l'une temporelle, l'autre matérielle) sont considérées pour évaluer A:

- le *temps*, noté $T_s(n)$ défini comme le nombre d'opérations effectuées sur des données bornées (ex : opérations flottantes, opérations sur des entiers machines, lecture ou écriture en mémoire d'un mot machine...). La classe $\mathcal{P}$ (resp. $\mathcal{NP}$) est définie comme l'ensemble des problèmes qui admettent un algorithme de temps polynomial $n^{O(1)}$ sur une machine RAM déterministe (resp. non déterministe).
- l'*espace mémoire*, noté $S(n)$ (S pour *space*) défini comme le nombre de places en mémoire nécessaires à l'exécution de l'algorithme.

Par analogie, dans le cadre du calcul parallèle, la qualité d'un algorithme parallèle A résolvant le problème P est basée sur deux caractéristiques, l'une temporelle, l'autre matérielle :

- le *temps parallèle* ou *profondeur* , noté $T_\infty(n)$, qui est le nombre de pas nécessaires à l'exécution de l'algorithme avec un nombre infini de processeurs;
- le nombre d'opérations $T_1(A)$ effectuées par l'algorithme qui est aussi une borne sur le nombre de processeurs permettant une exécution en temps effectif $T_\infty(n)$.

Il est possible d'exécuter l'algorithme parallèle A sur un nombre de processeurs inférieur à $T_1(A)$, en ordonnançant plusieurs instructions sur un même processeurs. Plus précisément, le "principe de Brent" stipule que, si l'on ne prend pas en compte le surcoût pour la réalisation de l'ordonnancement, l'algorithme parallèle A peut être exécuté sur p processeurs identiques en temps T_p :

$$T_p \leq \frac{T_1}{p} + \left(1 - \frac{1}{p}\right) T_\infty.$$

Ce résultat, dû à Graham, donne aussi une méthode pour la construction d'un tel ordonnancement.

1.2 La classification NC

Une question de base en calcul parallèle est de déterminer les problèmes intrinsèquement parallèles, c'est-à-dire qui peuvent être résolus beaucoup plus rapidement avec plusieurs processeurs plutôt qu'avec un seul [6] [15].

La classe NC, formalisée par Nicholas Pippenger [21] (et nommée par Cook NC pour "Nick's class" [6]), est la classe des problèmes qui peuvent être résolus en temps poly-logarithmique (c'est-à-dire résolues plus rapidement qu'il ne faut de temps pour lire séquentiellement leurs entrées) sur une machine parallèle ayant un nombre polynomial de processeurs, autrement dit de surface raisonnable. NC peut donc être caractérisée par:

$$\mathcal{NC} = \left\{ \text{problèmes } P / \exists A \text{ algorithme qui résout } P_n \text{ en coût } T_\infty(n) = \log^{O(1)} n \text{ et } T_1(n) = n^{O(1)} \right\}.$$

NC est trivialement un sous-ensemble de la classe P des fonctions qui peuvent être calculées séquentiellement en temps polynomial. Ainsi $P \supseteq NC$ mais l'inclusion stricte reste conjecturelle.

Une propriété fondamentale de NC est d'être **résistante** : elle reste la même quel que soit le modèle parallèle, par exemple PRAM, circuits, etc.

Remarque. Classes probabilistes. Comme en séquentiel, le préfixe R (resp. Z) désigne une classe de complexité probabiliste de type Monte Carlo – R pour Random – (resp. de type Las Vegas – Z pour Zero-error –). Ainsi la classe $\mathcal{RNC}$ est l'ensemble des problèmes pouvant être résolus par un algorithme Monte Carlo de temps parallèle poly-logarithmique et effectuant un nombre polynômial d'opérations.

1.3 Modèle booléen et NC-Réduction

De façon à pouvoir classer les problèmes par ordre de difficulté à l'intérieur de NC, et préciser où peut se trouver la différence entre P et NC, une relation d'ordre entre les problèmes est définie dans le cadre du modèle booléen [2] : la NC-réductibilité.

1.3.1 Le modèle booléen

Dans ce modèle, une machine parallèle est une famille *uniforme* $(B_n)_{n\in I\!N}$ de circuits booléens (i.e. graphes orientés et sans cycle – DAG –) telle que B_n a $n^{O(1)}$ entrées. Un nœud du circuit est ici une porte logique effectuant une opération booléenne (et, ou, négation). On distingue essentiellement deux sous-modèles, selon que le nombre d'entrées d'une porte (fan-in) est borné (i.e. vaut 2 ici) ou non borné. Le nombre de sorties d'une porte (fan-out) est quant à lui non borné [1].

Les nÏuds d'entrée (respectivement de sortie) ont un fan-in (respectivement fan-out) de 0. L'uniformité permet de limiter la complexité architecturale du circuit [24]. On utilise le plus souvent la "log-uniformité" qui signifie que la description du circuit B_n (pour $n \in I\!N$) peut être calculée sur une machine de Turing avec un espace logarithmique.

La surface $T_1(n)$ est ici définie comme le nombre de nœuds du circuit B_n, et le temps parallèle $T_\infty(n)$ comme sa profondeur.

Dans ce modèle on distingue les sous-classes NC^k (respectivement AC^k) pour les circuits dont les portes ont un fan-in borné (resp. non borné) et qui sont de profondeur $O(\log^k n)$ et de taille $n^{O(1)}$. On a alors les relations suivantes [15] :

$$NC^k =\subseteq AC^k \subseteq NC^{k+1}$$

La classe NC est alors définie comme l'union de toutes les classes NC^k :

$$NC = \bigcup_{k=0}^{\infty} NC^k$$

Remarque : circuits arithmétiques. Des extensions du modèle booléen [8] permettent de considérer que les portes du circuit peuvent faire en temps unité des opérations sur un espace donné E (par exemple les rationnels, ou les polynômes à coefficients rationnels). Les classes de complexité correspondantes sont alors notées NC^k_E pour préciser que les opérations de base considérées sont des opérations sur E. Par exemple, le produit de n entiers de n bits appartient à $NC^1_{I\!N}$ (produit itéré) mais le même algorithme ne permet que de prouver l'appartenance à NC^2 (la multiplication de deux entiers de n bits appartenant à NC^1).

[1] Un circuit de fan-in borné et de fan-out non borné peut en effet être transformé en un circuit de même surface et de même temps -en ordre- qui soit de fan-in et de fan-out borné [13].

1.3.2 NC-réductibilité et problèmes P-complets.

Une fois le modèle booléen défini, il est maintenant possible de classifier les problèmes selon leur complexité, grâce à une relation d'ordre : la NC^1-réductibilité [6]. On dit qu'une fonction (ou un problème) f est NC^1-réductible à une fonction g (ce qui est note $f \leq_{NC^1} g$) s'il existe une famille uniforme de circuits qui calcule f en temps logarithmique ($O(\log n)$), et dont les nœuds sont soit des portes booléennes, soit des oracles permettant de calculer g. Un oracle pour g est ici un nœud ayant r entrées $(e_1, \ldots, e_r)$ et t sorties $(s_1, \ldots, s_t)$ et qui calcule le résultat $(s_1, \ldots, s_t) = g(e_1, \ldots, e_r)$. La profondeur d'un tel nœud est assimilée à $\log(rt)$.
Il est clair que la relation de NC^1-réductibilité est réflexive et transitive et que NC^k est close par NC^1-réductibilité.

Soit E une classe de problèmes. On dira qu'une fonction (un problème) f est $NC^1 - dur$ pour l'ensemble E (ou E-dur) si et seulement si :

$$\forall g \in E : g \leq_{NC^1} f$$

f est dit complet pour E (E-complet) si f est E-dur et si $f \in E$.

Nous avons vu que $NC \subseteq P$. L'inclusion stricte restant conjecturelle, on peut se demander quels sont les problèmes complets pour P, qui sont ceux contenant -ou susceptibles de contenir- le moins de parallélisme intrinsèque.

Le problème P-complet de référence (l'analogue de la satisfaisabilité pour la complexité séquentielle et la classe NP) est le MCVP (monotone circuit value problem) [10] : " Etant donné une séquence de n équations booléennes du type $e_1 = 0$, $e_2 = 1$ et $e_k = e_i \wedge e_j$ ou $e_k = e_i \vee e_j$ pour $1 \leq i \leq j < k \leq n$, calculer la valeur de e_n [2]."

1.4 Evaluation de circuits arithmétiques

Nous avons vu que tout problème dans P pouvait être réduit au problème MCVP, qui est P-complet. Toutes les techniques permettant d'évaluer rapidement en parallèle des instances du MCVP pourront donc être appliquées à n'importe quel problème polynômial (pour autant que l'on puisse construire "facilement" les instances du MCVP correspondant au problème que l'on cherche à paralléliser)[22].

De manière générale, une instance du MCVP se présente comme un programme arithmétique sans boucle, de longueur n, (i.e. un graphe de précédence comportant n nœuds) ne comportant que des affectations ou des opérations booléennes $\wedge$ ou $\vee$.

Différentes techniques ont été proposées pour évaluer rapidement des programmes sans boucles dans des structures algébriques.

En particulier, l'évaluation d'expression arithmétique dans un anneau ou un corps appartient à NC [4] [9]. Ainsi, si le DAG correspondant au programme sans boucle peut être transformé en un arbre ayant $m = n^{O(1)}$ nœuds, alors une évaluation parallèle de cet arbre permet de l'évaluer en temps $T_\infty = O(\log m)$ avec $T_1 = \Theta(m)$ opérations.

Mais si le nombre de nœuds de l'arbre correspondant à l'expansion du circuit est exponentiel, l'évaluation ne pourra se faire en parallèle qu'en temps linéaire . Comme exemple, on peut considérer le programme suivant : $x_i := x_{i-1} + x_{i-1}$ pour $i = 1, \ldots, n$ avec comme entrée x_0 un entier dans le monoïde $(I\!N, +)$. Pour pallier à ce problème, il est cependant possible d'utiliser la puissance de la structure algébrique de l'ensemble dans lequel est défini le problème. Pour l'exemple précédent, il est clair que le programme calcul $2^n.x$; l'existence de la multiplication, distributive par rapport à

[2] Tout problème s'exécutant en temps polynômial sur une machine de Turing déterministe peut être NC^1-réduit à ce problème [15], ce qui prouve, étant clairement dans P, qu'il est P-complet.

l'addition, peut permettre de ramener le problème à un produit itéré pour calculer 2^n. Plus précisément, il est possible d'évaluer plus rapidement le circuit en tirant parti à la fois de l'associativité et de la distributivité d'une loi par rapport à l'autre; ce résultat a été donné par Kaltofen, Miller et Ramachandran dans [14] pour un anneau (i.e. les opérations + et × du programme sont dans un anneau) et étendu dans [22] au cas des treillis, en particulier des booléens cadre du problème P-complet de référence (MCVP).

On définit le degré arithmétique d d'un circuit (DAG correspondant à un programme sans boucle) comme suit. le degré d'une entrée est 1; le degré d'un nœud + le maximum des degrés de ses opérandes; le degré d'un nœud × la somme des degrés de ses opérandes. Le degré du circuit est le maximum du degré de ses nœuds.

Dans la proposition ci-dessous, $M(n)$ est le nombre d'opérations nécessaires pour effectuer un produit de matrices en temps $\log n - M(n) < n^3 -$.

Proposition 1 *[14] Tout programme sans-boucle de n nÏuds et de degré d dans un semi-anneau peut être évalué en temps* $T_\infty(n) = O(\log n \log(n.d))$ *avec* $T_1(n) = O(M(n) \log n \log(n.d))$ *opérations.*

Exemple: résolution de système triangulaire. Considérons en exemple la résolution du système linéaire triangulaire inversible $Ax = b$ par la technique classique d'élimination itérative dite de descente triangulaire. Soit n la dimension de A. Le circuit associé à ce programme sans boucle comporte $\Theta(n^2)$ nÏuds. Son degré est celui du nÏud associé à x_n; or le degré de x_k est le degré de $x_{k-1} + O(1)$, pour tout k. Le degré du programme est donc $\Theta(n)$. On en déduit que la résolution d'un système linéaire peut être réalisée en temps $T_\infty(n) = O(\log^2 n)$ avec $T_1(n) = M(n^2) = O(n^6)$ processeurs. Ce problème appartient donc à NC^2; mais le nombre important d'opérations interdit son utilisation pratique.

Remarque : cas des treillis. Des extensions de ce résultat à la structure algébrique de treillis ont été données [22]. Cette structure est intéressante, car elle permet de mieux prendre en compte la structure des booléens, cadre du MCVP.

1.5 Algorithmes en cascade

Par rapport à un algorithme séquentiel, introduire du parallélisme nécessite l'introduction d'opérations supplémentaires afin d'obtenir un algorithme de temps parallèle T_∞ minimal, quitte à augmenter le nombre d'opérations T_1 requis par rapport à T_s..

Cependant, lors de l'exécution, on ne peut utiliser qu'un nombre limité de ressources. Le problème est alors de replier efficacement le parallélisme. Le principe de Brent motive la construction d'algorithmes parallèles qui minimisent donc T_∞ tout en gardant T_1 proche du nombre d'opérations du meilleur algorithme séquentiel possible.

Nous présentons deux solutions pour effectuer un tel repliage de l'algorithme parallèle. La première est basée sur la construction d'ordonnancements efficaces; c'est l'objet de la section suivante. La deuxième est algorithmique; elle est basée sur un couplage de plusieurs algorithmes, généralement au moins un algorithme séquentiel et un algorithme parallèle. Cette technique est appelée *cascade algorithmique*. Nous l'illustrons sur trois exemples : maximum de n éléments, inversion de matrices et découpe récursive à grain adaptatif.

1.5.1 Circuit à fan-in non borné pour le calcul du maximum

Avec un circuit à fan-in non borné, le maximum de n éléments peut être calculé par -entre autres- les 3 algorithmes suivants:

- calcul séquentiel itératif classique : $T_1 = n$; $T_\infty = n$
- calcul parallèle récursif en arbre d'arité K (par exemple $K = 2$ ou $K = \sqrt{n}$): $T_1 = n$; $T_\infty = \log n$
- calcul parallèle avec comparaison parallèle de chaque élément à tous les autres; seul l'élément trouvé supérieur à tous les autres est gardé en temps constant avec une porte "ET-logique" n-aire. $T_1 = n^2$; $T_\infty = O(1)$

L'algorithme qui minimise T_∞ (le troisième) demande un accroissement important du nombre d'opérations T_1.

Le couplage de ces 3 algorithmes différents permet la construction d'un algorithme qu vérifie $T_\infty = \log\log n$ tout en gardant $T_1 = \Theta(n)$.

Transparents; transparents 12–20, pages 3,4 du document:
`http://www-id.imag.fr/~jlroch/perso.html/COURS/2003-dea-algo-par/2001-10-22-cours2.pdf`

1.5.2 Résolution de systèmes triangulaires

La technique d'évaluation du circuit (§1.4) a montré qu'il était possible de résoudre en temps $T_\infty = \log^2 n$ un système triangulaire; le problème est ici de trouver un algorithme parallèle qui effectue le même nombre d'opérations qu'un algorithme séquentiel, i.e. $T_1 = n^2$.

Analyse du parallélisme du meilleur algorithme séquentiel L'algorithme de résolution par descente triangulaire a un coût $T_s(n) = \Theta(n^2)$. L'analyse des dépendances de cet algorithme (fig. 1) conduit au coût parallèle: $T_\infty = n$ et $T_1 = n^2$.

Nous avons vu que l'évaluation par circuit arithmétique conduisait à un temps parallèle $T_\infty = \log^2 n$. On considère ici l'algorithme récursif [3]. Soient A, b et x partitionnés comme suit :

$$A = \begin{bmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{bmatrix} \qquad b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \qquad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}. \tag{1.1}$$

Ici A_{11} est de taille $h \times h$, x_1 et x_2 sont de taille h. On a :

$$A_{11}x_1 = b_1 \quad \text{et} \quad A_{22}x_2 = b_2 - A_{21}x_1. \tag{1.2}$$

où x_1 et x_2 sont calculés récursivement en utilisant le même algorithme; $A_{21}x_1$ est calculé par un produit scalaire.

Supposons que l'on stoppe la découpe récursive quand les matrices sont de taille $k \times k$, et que l'on utilise alors un algorithme séquentiel pour les inversions de système triangulaire et produits matrice-vecteur dès que le système ou la matrice est de dimension inférieure à une valeur k. Le coût de l'algorithme parallèle résultant est alors :

$$T_\infty = \Theta(n.k) \qquad T_1 = \Theta(n^2) \tag{1.3}$$

```
Update( x : out E,
        a : in  E,
        y : in E )
begin
  x.Cumul<+>( -a.Read()*y.Read() );
end


FinalDivision( x : in and out E,
               a : in         E )
begin
  x.Write( x.Read() / a.Read() );
end




TriangularSolve ( n : in  integer,
                  a : in  array[1..n, 1..n] of E,
                  b : in  array[1..n] of E,
                  x : out array[1..n] of E )
begin
  i,j : local integer;

  for i = 1..n loop
    x[i].Cumul<+>( b[i].Read() );
    fork FinalDivision(x[i], a[i,i]);
    for j = (i+1)..n loop
      fork Update(x[j], a[j,i], x[i]);
    end loop
  end loop
end
```

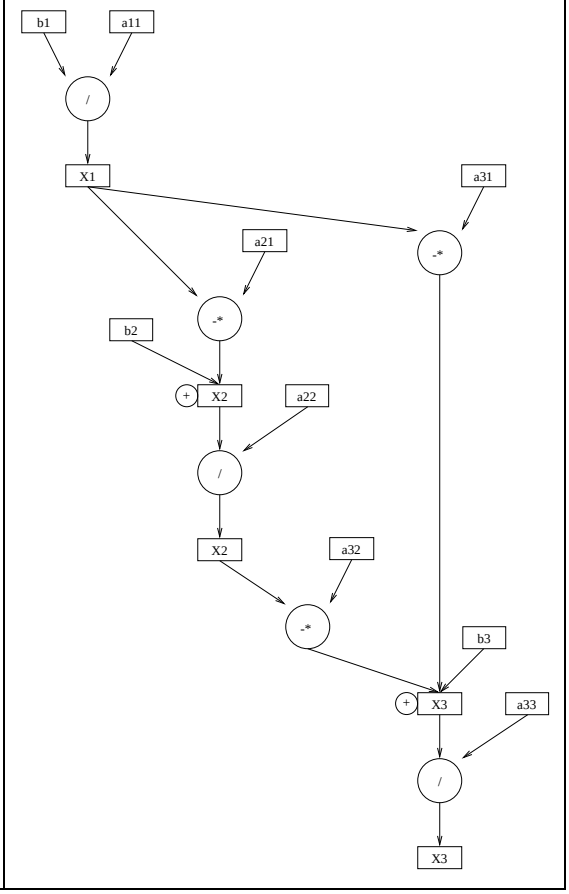

Figure 1: Graphe de dépendance pour la résolution d'un système triangulaire inversible de dimension 3.

Augmenter le parallélisme La dépendance entre x_1 et x_2 dans 1.2 peut être éliminée en calculant directement en parallèle les inverses des matrices inversibles A_{11} et A_{22}.

Considérons la matrice A partitionnée en quatre blocs de dimension $n/2$ (1.1 avec $h = n/2$). On a alors :

$$A^{-1} = \begin{bmatrix} A_{11}^{-1} & 0 \\ -A_{22}^{-1} A_{21} A_{11}^{-1} & A_{22}^{-1} \end{bmatrix} \tag{1.4}$$

Dans la suite, nous considérons que le produit de deux matrices de dimension n peut être calculé en $T_\infty = \log n$ avec $T_1 = O(n^3)$ opérations.

Selon 1.4, l'inverse de A peut être calculée comme suit ; on calcule d'abord en parallèle A_{11}^{-1} et A_{22}^{-1}; puis on calcule le dernier bloc A_{21}^{-1} de A^{-1} en effectuant séquentiellement deux produits paralèles de matrices. Le coût d'inversion de A est alors $T_\infty = \log^2 n$ avec $T_1 = O(n^3)$ opérations. Une fois A^{-1} calculée, on peut calculer $x = A^{-1}b$ avec un coût négligeable devant celui de l'inversion de A.

Cependant, même si de coût polylogarithmique, cet algorithme effectue n fois plus d'opérations que l'algorithme séquentiel. Dans le paragraphe suivant, l'utilisation de la technique de cascade permet de limiter ce surcoût en nombre d'opérations.

Remarque. On peut remarquer qu'en utilisant un produit rapide de matrices, le nombre d'opérations devient $T_1 = n^omega$ avec $\omega < 2.38$ [20].

Résolution de système linéaire en cascade L'algorithme précédent n'est pas efficace mais peut être couplé à l'algorithme séquentiel récursif par vloc pour l'accélérer (formule 1.2). En effet, il peut être utilisé sur des matrices de petite dimension (disons inférieure à h) lorsque le surcoût $O(h^3)$ dû à l'inversion rapide de ce bloc devient négligeable par rapport au coût de la mise à jour suite à l'inversion du bloc (environ nh).

Theorem 2 *La solution d'un système traiangulaire inversible peut être calculée en $T_\infty = n^{1/2}$ et $T_1 = n^2$) en utilisant un algorithme de multiplication de matrices effectuant. n^3 opérations.*
Si un algorithme en n^ω est utilisé pour la multiplication, le coût devient : $T_\infty = n^{(\omega-2)/(\omega-1)} \log^2 n$ et $T_1 = n^2$

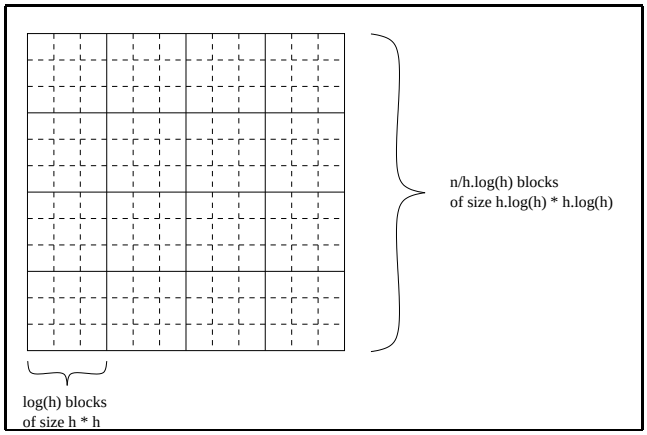

Figure 2: Partitionnement avec $h = 8$, $h \log h = 24$, $n = 96$

Selon 1.3, partitionnons A en n^2/h^2 blocs de taille $h \times h$. Un calcul direct (cf théorème 1.3) conduit à un temps paralèle $T_\infty = O(n^{1/2} \log^2 n)$. Pour éviter le facteur de surcoût $\log n$ overhead factor in the parallel time, on regroupe les calculs sur des blocs de dimension $\log^2 h$.
Soit $k = h \log h$; la matrice A peut être découpée en $(n/k)^2$ blocs, chaque bloc contenant $\log^2 h$ sous-blocs de dimension h (cf fig. 2).
On utilise l'algorithme itératif sur les matrices de taille $(n/k) \times (n/k)$

A l'étape i, nous devons inverser le système triangulaire correspondant au bloc diagonal (i, i). Pour ce calcul, on inverse d'abord les $\log h$ diagonales de ce bloc. Ensuite, on met à jour les autres sous blocs liés à x_i. A la fin de cette étape i, les blocs x_j, pour $j > i$, sont mises à jour. L'algorithme est le suivat:

Initialisation.

On partitionne A en n/k blocs $M_{i,j}$ de dimension k ($k = h \log h$). Pour $1 \leq j \leq i \leq n/k$, on partitionne $M_{i,j}$ en $\log h \times \log h$ blocs $m_{i,j}^{k,l}$ de dimension h.
Soit x intialisé à b et partitionné selon A.

for $i = 1..n/k$ do

1. for $j = 1..\log h$ do
   fork $\left(m_{i,i}^{j,j}\right)^{-1} = \text{invert}(m_{i,i}^{j,j})$.
   En utilisant l'inevrsion et le principe de Brent, le coût est $T_\infty = \log^2 h$ et $T_1 =, h^3 \log h$.
2. for $j = 1..\log h$ do
   update x_i^j in parallel
   $x_i^j = \left(m_{i,i}^{j,j}\right)^{-1} \left(x_i^j - \sum_{l=1}^{j-1} m_{i,i}^{j,l} x_i^l\right)$
   Les produits scalaires sont effecués en parallèle: ainsi, x_i est calculé avec un coût $T_\infty = \log^2 h$ et $T_1 = h^2 \log h$).
3. for $j = i + 1..n/k$ fork update x_j in parallel
   $x_j = x_j - M_{j,i} x_i$
   En effectuant les produits scalaires en parallèle, le coût est $T_\infty = \log h$ et $T_1 = nh \log h$).

Finalement, le coût de cet algorithme en cascade est $T_\infty = n \log^2 h/k$ et le nombre d'opérations est $T_1 = n/k \max(h^3 \log h, nh \log h))$. Avec $k = h \log h$ et choisissant $h = n^{1/2}$, on ontient

$$T_\infty = n^{O,5} \quad T_1 = n^2$$

La même technique est appliquée pour obtenir la borne supérieure lorsqu'un algorithme de multiplication rapide est utilisé. □

Dans les exemples précédents, la cascade était basée initialement sur un algorithme tres parallele qui effectuait trop d'opérations; l'objectif de la cascade était alors de réduire le nombre d'opérations en cascadant cet algorithme parallèle sur un algorithme effectuant moins d'opérations – typiquement un algorithme séquentiel –.

La section suivante montre une manière duale de réaliser la cascade: en cascadant un algorithme séquentiel sur un algorithme parallèle pour réduire cette fois le temps parallèle.

1.5.3 Granularité adaptive

Cette technique est décrite dans [16]: elle consiste à cascader à utiliser un algorithme parallèle pour accélérer un algorithme séquentiel.

Nous illustrons brièvement cette technique, duale de la précédente, sur la résolution du système linéaire précédent. Par défaut, en supposant le nombre de ressources faible, le programme est démarré selon un algorithme séquentiel par bloc. Si un processeur est inactif, il cherche à extraire des calculs pouvant être effectués en parallèle chez un processeur actif. Cette extraction peut être réalisée en accédant aux prochains blocs qui doivent être traités par l'algorithme séquentiel.

Ainsi, l'algorithme séquentiel dégénère en exécution parallèle. Cette dégénération est récursive, et ne limite donc pas le parallélisme: l'extraction peut être effectuée jusqu'à des blocs de dimension minimale.

Ce couplage, qui réalise une cascade dynamiquement en fonction de l'inactivité des processeurs, est plus général que la cascade précédente qui était statique. On obtient ainsi un algorithme dont le temps d'exécution sur un processeur est garanti le même que le temps de l'exécution du meilleur algorithme séquentiels; et sur un grand nombre de processeurs, on obtient un temps d'exécution parallèle minimal.

2 Ordonnancement et algorithmes d'approximation

Après une description du problème d'ordonnancement et de sa complexité, nous montrons qu'il existe des algorithmes d'approximation pour résoudre certaines instances.

Nous montrons ensuite comment un ordonnancement peut être construit pour optimiser plusieurs critères, comme temps parallèle, nombre d'opérations et espace mémoire avec le vol de travail.

Dans cette section, nous allons illustrer la démarche générale sur un des problèmes les plus simples en ordonnancement, la distribution de tâches sur des machines homogènes, sans prendre en compte le coût des communications.

2.1 Problème d'ordonnancement et complexité

La plupart des problèmes d'allocation de ressources s'expriment comme des problèmes d'optimisation combinatoire opérant sur des graphes (minimisation ou maximisation de certains critères sous contraintes). Ainsi, il est classique de chercher à déterminer une exécution de durée minimale d'un programme à paralléliser représenté par un graphe orienté acyclique (DAG, directed acyclic graph) $G = (V, E)$ [5] appelé graphe de tâches ou graphe de précédence.

Dans le graphe G, les sommets représentent les différentes tâches à traiter, généralement pondérées par leur temps d'exécution. Une arête (orientée) entre les tâches i et j signifie que la tâche i doit être traitée avant le début de la tâche j (ce qui justifie le caractère acyclique du graphe). Quand on considère des coûts de communication, l'arête (i, j) est généralement pondérée par le temps de communication nécessaire si les tâches i et j ne sont affectées au même processeur. Sous forme de problème de décision, le problème que nous allons considérer (problème central de l'ordonnancement multiprocesseur - PCOM) peut s'écrire de la manière suivante :

PCOM:

Instance: Soit $G = (V, E)$ un graphe d'ordre n (dont E est codé sous forme de matrice d'adjacence par exemple), on note p_j la pondération du sommet j (c'est-à-dire la durée d'exécution de la tâche j) et un ensemble de m processeurs identiques complètement connectés (logiquement) par un réseau homogène (et infiniment rapide). On se donne également une borne k (un entier) sur la durée d'exécution.

Question. Existe-t-il un ordonnancement valide des n tâches en moins de k unités de temps ?
On cherche donc une application σ qui à chaque tâche de V associe la date où elle débutera son exécution en respectant les contraintes de précédence de E et une application π qui à chaque tâche de V associe le processeur où elle s'exécutera.

Mathématiquement, ce problème s'exprime de la façon suivante : $\forall j \in V$, déterminer σ de V dans $I\!N$ et π de V dans $\{1; m\}$ telle que si $(i, j) \in E$ alors $\sigma(j) \geq \sigma(i) + p_i$ et pour tout couple (i, j) si $\pi(i) = \pi(j)$ alors on a soit $\sigma(j) \geq \sigma(i) + p_i$, soit $\sigma(i) \geq \sigma(j) + p_j$.

Theorem 3 *Le problème PCOM est NP-complet.*

La démonstration de ce résultat s'obtient assez facilement par une réduction à partir du problème 3-Partition. Il est NP-complet au sens fort pour m arbitraire et le reste pour m fixé.

Bien que ce problème soit NP-dur, il existe des algorithmes d'approximation efficaces. Nous étudions dans la section suivante l'ordonnancement glouton qui donne une preuve constructive du principe de Brent (§1.1).

2.2 Exemple d'algorithmes d'approximation

Sur machine parallèle, les ordonnancements classiquement utilisés sont de type liste : ils consistent à affecter à un processeur inactif une tâche prête à être exécutée si il en existe. En effet, en supposant $T_\infty \ll T_1$ et si on néglige le coût de l'interprétation, un tel algorithme fournit un temps d'exécution asymptotiquement optimal comme le montre le résultat suivant. Ce théorème est important; preuve constructive du principe de Brent (§1.1), il justifie le fait de s'intéresser à des algorithmes parallèles de temps critique très faible.

Theorem 4 *[12] Si l'on ne prend pas en compte le coût de calcul et de réalisation de l'ordonnancement, tout ordonnancement de type liste appliqué à un programme parallèle de coût T_1 et T_∞ conduit à un temps d'exécution T_p sur p processeurs majoré par :*

$$T_p \leq \frac{T_1}{p} + T_\infty$$

Preuve Nous rappelons la preuve car son schéma est utilisé pour l'analyse de nombreuses variantes.

Soit t_1 une tâche qui s'est terminée à la date T_p, et soit d_1 la date du début de cette tâche. Avant d_1, deux situations peuvent être distinguées : i) soit aucun processeur n'a été inactif avant d_1; ii) soit au contraire il existe une date $d < d_1$ à laquelle au moins un processeur était inactif.
Dans ce cas ii), soit d' la plus grande de ces dates. L'ordonnancement étant de liste, si à la date d' la tâche t_1 avait été prête alors elle aurait débuté son exécution. Il existe donc une tâche t_2 en cours d'exécution à la date d' telle que $t_2 \prec_G t_1$. Soit d_2 la date de début d'exécution de cette tâche.

L'application récursive de ce schéma permet de construire une séquence $t_k \prec_G \cdots \prec_G t_2 \prec_G t_1$ de tâches telles qu'à tout instant de l'exécution, soit tous les processeurs sont actifs, soit un des processeurs est en train d'exécuter une des tâches de cette séquence. La durée de la première situation est majorée par $\frac{T_1}{p}$ et celle de la seconde par la durée d'un chemin critique du graphe, c'est-à-dire par T_∞. □

Par une preuve similaire [17], on montre que, sur m machines, un ordonnancement de liste est une approximation à un facteur $\left(2 - \frac{1}{m}\right)$ de l'ordonnancement optimal sur m machines. Dans la section suivante, nous montrons que ce ratio de performance est optimal.

2.3 Bornes inférieures sur le ratio de performance

Une question naturelle est alors de déterminer s'il est possible d'obtenir un ratio de performance inférieur à $\left(2 - \frac{1}{m}\right)$, soit sur le même problème soit en utilisant des opérations plus puissantes pour réaliser l'ordonnancement, comme le tirage aléatoire ou la migration. Ce problème est étudié dans [25] où la proposition suivante est prouvée.

Theorem 5 *[25] Si les temps de communication ne sont pas pris en compte, $\left(2 - \frac{1}{m}\right)$ est une borne inférieure pour le ratio d'un algorithme d'ordonnancement déterministe avec ou sans migration.*
Un algorithme probabiliste sans préemption ne peut avoir un ratio de performance inférieur à $\left(2 - \frac{1}{\sqrt{m}}\right)$, ratio qui est obtenu par un algorithme de liste dans lequel les tâches affectées aux processeurs inactifs sont tirées au hasard parmi l'ensemble des tâches prêtes.

Seule la preuve de la première partie est ici présentée car elle utilise la construction d'un adversaire, technique fréquemment utilisée pour obtenir des bornes inférieures sur un algorithme à la volée. Il s'agit de construire une instance du problème réalisant le pire cas et de montrer que ce pire cas ne peut être évité par aucun algorithme d'ordonnancement.

Le pire cas est obtenu pour l'instance suivante due à Graham [11]. G contient $1 + p(p-1)$ tâches indépendantes; une tâche α est de longueur p tandis que les β_k, $1 \leq k \leq p(p-1)$ autres sont de longueur 1. L'ordonnancement optimal est de longueur p; il est obtenu en exécutant α_1 sur un processeur et les $p(p-1)$ tâches β_k sur les $p-1$ autres processeurs.

Aucun algorithme d'ordonnancement à la volée ne peut faire d'hypothèse sur la longueur d'une tâche qui est inconnue tant que la tâche n'est pas terminée. La technique de l'adversaire consiste alors à faire en sorte que la la tâche α soit démarrée le plus tard possible. Chaque fois que l'ordonnanceur lance l'exécution d'une tâche sur un processeur, nous supposerons donc que cette tâche est une tâche β_k, de longueur 1. Ainsi l'ordonnanceur ne peut demander l'exécution de la tâche α qu'après que toutes les tâches β_k aient été exécutées, donc au plus tôt au top $(p-1)$. La longueur de l'ordonnancement délivré est alors de longueur au moins $(p-1)+p$, ce qui montre la borne inférieure. Il est clair que la préemption ou la migration ne peuvent améliorer ce ratio. □.

En conséquence, ni la migration ni l'introduction d'aléatoire ne peuvent améliorer le ratio de performance en comparaison d'un algorithme de liste.

Pour un programme à grain fin (T_∞ petit), un algorithme de type liste apparaît donc pertinent puisque asymptotiquement optimal. Cependant, la preuve ne prend pas en compte le coût de gestion de calcul de l'ordonnancement, à savoir la gestion de la liste des tâches prêtes. Ce problème est étudié dans la section suivante avec les algorithmes de vol de travail.

2.4 Ordonnancement par vol de travail (work-stealing)

D'un point de vue performance effective, le coût de la réalisation d'un ordonnancement doit aussi être pris en compte [17]. Pour un algorithme de liste, il est a priori borné par le nombre n de tâches; comme $n > \frac{T_1}{T_\infty}$, le surcoût peut être considérable pour un programme de grain fin.

Toutefois, la preuve précédente montre que le nombre de tops d'inactivité est majoré par T_∞ sur chaque processeur; ainsi, si les processeurs sont capables de trouver facilement des tâches prêtes à exécuter lorsqu'il en existe, le surcoût d'ordonnancement, dans ce cas majoré par $O(pT_\infty)$, sera alors négligeable pour des programmes possédant un grand degré de parallélisme.

Ce constat est à la base de la stratégie "travail d'abord" ("work-first principle" [7]), traditionnellement utilisée pour la compilation des langages fonctionnels [18]. Cette stratégie elle consiste à mettre la plus grande partie du surcoût de réalisation de l'ordonnancement sur le chemin critique, i.e. lorsqu'un processeur devenu inactif doit voler une tâche à un autre processeur qui possède des tâches prêtes. Par rapport à la preuve du théorème 4, le surcoût est mis sur le terme en T_∞ plutôt que sur celui en T_1/p. Ainsi, pour des programmes très parallèles où T_∞ est négligeable devant T_1, le surcoût dû à l'ordonnancement est moindre.

La solution est donc de n'effectuer la gestion locale de tâches que lorsque cela est nécessaire, à savoir lors d'un vol. Pour cela, la création d'une tâche est compilée en appel de fonction locale. Cela nécessite une hypothèse fondamentale : toute tâche créée doit pouvoir être directement exécutée localement, donc être prête. Cette hypothèse est facilement vérifiée pour les langages offrant un parallélisme de type série-parallèle, comme les langages fonctionnels qui ont été les premiers à l'exploiter. Elle n'est cependant pas restrictive à ce type de parallélisme et est implantée dans des langages plus généraux comme Athapascan [23].

Le programme est alors *dégénéré en exécution séquentielle profondeur d'abord*. Le point critique est que cette dégénérescence ne doit pas entraîner de perte de parallélisme au niveau de l'algorithme.

La dégénération séquentielle est basée sur un ordre total sur les tâches, compatible avec l'ordre partiel défini par la relation de précédence. Cet ordre total est celui suivi par l'exécution séquentielle, par exemple profondeur d'abord. La section suivante montre que le choix de cet ordre peut être fait

de manière à optimiser l'espace mémoire requis.

2.5 Optimisation sous contraintes: temps parallèle et espace mémoire

Sur un nombre fini de ressources, optimiser l'espace mémoire conduit à un ordonnancement séquentiel alors que minimiser le temps d'exécution conduit à exploiter au mieux le parallélisme; les deux objectifs, mémoire et temps, sont donc antagonistes.

Pourtant, il est possible de construire des ordonnancements qui sont asymptotiquement optimaux en temps (i.e. si T_∞ est asymptotiquement négligeable devant T_1) et qui garantisse simultanément un espace mémoire borné par rapport à une exécution séquentielle.

Plus précisément, l'ordonnancement séquentiel du programme peut lui aussi influencer conséquemment sur la consommation mémoire. Considérons par exemple un programme comportant n tâches a_i allouant 1 mot chacune et n tâches b_i effectuant les libérations correspondantes; les seules contraintes de précédence soient du type $a_i \prec l_i$. Considérons les deux ordonnancements séquentiels suivants :

$$a_1 < b_1 < \ldots < a_i < b_i < \ldots < a_n < b_n \quad (1)$$
$$a_1 < \ldots < a_i < \ldots < a_n < b_1 < \ldots < b_i < \ldots < b_n \quad (2)$$

Le volume mémoire requis pour l'exécution correspondant à l'ordre (1) est de 1 mot, tandis que celui correspondant à l'ordre (2) est de n mots. Le volume mémoire requis dépendant fortement de l'ordre, le principe pour contrôler cette consommation lors d'une exécution parallèle est de suivre l'ordre séquentiel dit de *référence* qui a été utilisé pour définir l'espace mémoire séquentiel S_1 qui sert de référence [1, 19]. Il est alors possible de comparer S_p à S_1.

Theorem 6 *[1] Un ordonnancement de liste qui, parmi les tâches prêtes, affecte à un processeur inactif la plus prioritaire selon l'ordre séquentiel de référence requiert un espace mémoire s majoré par :*

$$S_p \leq S_1 + pKT_\infty.$$

où S_1 est l'espace mémoire requis par l'exécution séquentielle de référence et K l'espace mémoire maximum alloué par une tâche.

Preuve Les tâches restant à exécuter sont stockées dans une liste triée selon l'ordre d'une exécution séquentielle[3]. Lorsqu'un processeur termine une tâche, il prend la tâche en tête de cette liste pour l'exécuter. Les tâches les plus vieilles selon l'ordre séquentiel sont donc prioritaires. La durée d'une tâche étant majorée par T_∞, à chaque instant au plus T_∞ tâches peuvent être exécutées en avance par rapport à l'ordre séquentiel sur chaque processeur. Chaque tâche allouant au plus un espace mémoire de K, l'espace mémoire total requis est majoré par $S_1 + pKT_\infty$. □

Aussi, la gestion de l'ordre des tâches dans la liste est crucial pour les performances. Dans le contexte de machines à mémoire partagée (SMP), Cilk utilise un tel ordonnancement. Dans un contexte distribué, Athapascan implémente une gestion distribuée d'une pile cactus qui garantit l'espace mémoire utilisé.

[3] L'insertion et la suppression dans la liste peuvent être effectuées en un nombre constant d'opérations par chaînage : les tâches filles sont insérées à l'ancienne position de la mère.

2.6 Construction d'ordonnancements multi-critères

Usually, approximation algorithms are designed with respect to one criterion. However, several criteria could be used to describe the quality of a schedule. In the context of parallel processing, the choice of which criterion to choose depends on the priorities of the users.

However, one could wish to take advantage of several criteria in a single schedule. With the makespan and the sum of weighted completion times, it is easy to find examples where there is no schedule reaching the optimal value for both criteria. Therefore you can not have the cake and eat it, but you can still try to find for a schedule how far the solution is from the optimal one for each criterion. In this section, we will look at a generic way design algorithms with guarantees on two criteria and at a more specific algorithm family for the moldable case. More details can be found in the chapter of Dutot, Mounie and Trystram in the Handbook of Scheduling edited by Joseph Leung (April 2004).

Two Phases, Two Algorithms ($\mathcal{A}_{\sum C_i}$, $\mathcal{A}_{C_{max}}$) Let us use two known algorithms $\mathcal{A}_{\sum C_i}$ and $\mathcal{A}_{C_{max}}$ with performance ratios respectively $\rho_{\sum C_i}$ and $\rho_{C_{max}}$ with respect to the sum of completion time and the makespan.

Proposition. It is possible to combine $\mathcal{A}_{\sum C_i}$ and $\mathcal{A}_{C_{max}}$ in a new algorithm with a performance ratio of $2\rho_{\sum C_i}$ and $2\rho_{C_{max}}$ at the same time.

Let us remark that delaying by τ the starting time of the tasks of the schedule given by $\mathcal{A}_{C_{max}}$ increases the completion time of the tasks with the same delay τ.

The starting point of the new algorithm is the schedule built by $\mathcal{A}_{\sum C_i}$. The tasks ending in this schedule before $\rho_{C_{max}} C^*_{max}$ are left unchanged. All tasks ending after $\rho_{C_{max}} C^*_{max}$ are removed and rescheduled with $\mathcal{A}_{C_{max}}$, starting at $\rho_{C_{max}} C^*_{max}$. As $\mathcal{A}_{C_{max}}$ is able to schedule all tasks in $\rho_{C_{max}} C^*_{max}$ and it is always possible to remove tasks from a schedule without increasing its completion time, all these tasks will complete before $2\rho_{C_{max}} C^*_{max}$.

Now let us look at the new values of the two criteria. Any task scheduled by $\mathcal{A}_{\sum C_i}$ ending after $\rho_{C_{max}} C^*_{max}$ does not increase its completion time by a factor more than 2, thus the new performance ratio is no more than twice $\rho_{\sum C_i}$. On the other hand, the makespan is lower than $2\rho_{C_{max}} C^*_{max}$. Thus the performance ratios on the two criteria are the double of the performance ratio of each single algorithm.

We can also remark that the schedule presented has a lot of idle times and the makespan can be greatly improved by just starting every task as soon as possible with the same allocation and order. However, even if this trick can give very good results for practical problems, it does not improve the theoretical bounds proven on the schedules, as it cannot always be precisely defined.

Tuning performance ratios It is possible to decrease one performance ratio at the expense of the other. The point is to choose a border proportionally to $\rho_{C_{max}} C^*_{max}$, namely $\lambda * \rho_{C_{max}} C^*_{max}$. The performance ratios are a Pareto curve of λ.

Proposition. It is possible to combine $\mathcal{A}_{\sum C_i}$ and $\mathcal{A}_{C_{max}}$ in a new algorithm with a performance ratio of $\frac{1+\lambda}{\lambda}\rho_{\sum C_i}$ and $(1+\lambda)\rho_{C_{max}}$ at the same time.

Combining two existing algorithms it is possible to schedule independent moldable tasks with a performance ratio of 3 for the makespan and 16 for the sum of the completion time.

The former approach required the use of two algorithms, one per criterion and mixed them in order to design a bi-criterion scheduling algorithm. It is also possible to design an efficient bi-criterion algorithm just by adapting an algorithm $\mathcal{A}_{C_{max}}$ designed for the makespan criterion.

The main idea is to create a schedule which has a performance ratio on the sum of completion times based on the result of algorithm $\mathcal{A}_{C_{max}}$ without losing too much on the makespan. To have this performance ratio $\rho_{\sum C_i}$ on the sum of the completion times, we actually try to have the same performance ratio $\rho_{\sum C_i}$ on all the completion times.

3 Le problème du voyageur de commerce symétrique et la programmation linéaire

3.1 Le problème

Etant donnés un certain nombre de villes et les distances inter-villes, le célèbre problème du Voyageur de Commerce consiste pour un voyageur à visiter toutes les villes exactement une fois et revenir à son point de départ en parcourant la plus petite distance possible. Les distances sont symétriques, elles sont les mêmes dans chaque sens de parcours entre deux villes données.

On modélise le problème par un graphe complet avec des longueurs associées aux arêtes. On supposera le graphe complet, c'est à dire que toutes les liaisons existent. Si ce n'est pas le cas on peut rajouter les liaisons manquantes et mettre sur toute arête une longueur égale à la plus courte distance entre ses extrémités dans le problème original. Remarquez que dans ce cas une solution du nouveau problème ne correspond pas nécessairement à un parcours dans lequel chaque ville n'est visitée qu'une seule fois.

Un cycle est dit *hamiltonien* s'il passe par chaque sommet exactement une fois. Le problème est donc de trouver un cycle hamiltonien de longueur minimum.

Nous utiliserons les notations suivantes:

$K_n = (V, E)$, est un graphe complet avec $n = |V|$ sommets. Soit $S \subset V$, alors $\delta(S)$ (resp. $\gamma(S)$) représente l'ensemble des arêtes avec exactement une extrémité dans S (resp. les deux extrémité dans S), i.e. $\delta(S) = \{(u, v) \in E : u \in S, v \notin S\}$ (resp.$\gamma(S) = \{(u, v) \in E : u \in S, v \in S\}$). On notera $\delta(v)$ au lieu de $\delta(\{v\})$ pour $v \in V$.

On notera $\mathbb{R}^E$ l'ensemble des vecteurs indexés par E, c.à.d. que les composantes d'un vecteur de $\mathbb{R}^E$ sont en bijection avec les éléments de E. Pour $E^\star \subset E$ et $x \in \mathbb{R}^E$, alors $x(E^\star)$ représente $\sum_{e \in E^\star} x_e$. Soit $x^\star \in \mathbb{R}^E$, avec $x^\star_e$ vu comme une *capacité* de l'arête e; pour $S \subset V$ on appelle $x^\star(\delta(S))$, la *valeur* de la coupe définie par S.

3.2 Le problème du voyageur de Commerce vu comme un programme linéaire en nombres entiers

Il y a plusieurs façons de modéliser le problème du voyageur de commerce comme un programme linéaire en nombres entiers. Une seule a été pour le moment utilisée comme base d'un processus de résolution, celle connue sous le nom de *formulation à deux indices*. A chaque arête $e \in E$ on associe une variable x_e qui vaudra 1 si e est dans la solution optimale et 0 sinon. Le problème du voyageur

de Commerce peut alors se formuler comme le programme linéaire en nombres entiers suivant:

$$\text{(IP(STSP))} \qquad \min \sum_{e \in E} c_e x_e \tag{3.1}$$

subject to

$$x(\delta(v)) = 2 \quad \text{for} \quad v \in V \tag{3.2}$$

$$x(\delta(S)) \geq 2 \quad \text{for} \quad 3 \leq |S| \leq |V|/2 \tag{3.3}$$

$$0 \leq x_e \leq 1 \quad \text{for} \quad e \in E \tag{3.4}$$

$$x_e \text{ integer} \quad \text{for} \quad e \in E \tag{3.5}$$

Equations (3.2) disent qu'exactement deux arêtes sont incidentes à un sommet. Inequalities (3.3) empêchent la création de cycles qui ne contiendraient pas tous les sommets et sont par conséquent appelées *inéquations d'élimination de sous tours*. Notez que le nombre de ces inéquations est exponentiel et interdit donc la résolution du programme linéaire obtenu en relâchant les contraintes de valeurs entières sur les variables, par des méthodes classiques. Cependant ce programme linéaire ne pose pas trop de problèmes du fait que trouver une inéquation de sous-tour non vérifiée est un problème bien résolu, à savoir un problème de coupe de capacité minimum.

On répète le processus suivant: On part du programme linéaire sans les contraintes de sous tour, on résout par un logiciel de programmation linéaire, on obtient une solution x^*, en utilisant les valeurs x_e^* comme des capacités on résout le problème de la coupe minimum, si celle-ci est de valeur 2, aucune inéquation de sous tour n'est violée, sinon on a trouvé une telle inéquation violée, on la rajoute au programme linéaire et on itère jusqu'à ne plus trouver de telle inéquation. On n'a aucune garantie de terminer rapidement, cependant par expérience, c'est le cas pour toutes les instances connues.

3.3 Résolution du programme linéaire en nombres entiers

A partir de là on pourrait entrer dans une méthode classique de Branch and Bound. Malheureusement cette méthode ne donne aucun résultat même pour des instances de taille modeste.

Il faut renforcer la formulation. On sait que n'importe quel programme linéaire en nombres entiers peut se transformer en programme linéaire par l'ajout d'un nombre fini de contraintes. Pour le Problème du voyageur de commerce il est peu probable qu'un jour on connaisse l'ensemble de ces contraintes. On ne les connaît que jusqu'à 10 villes. Pour donner une idée de la complexité voici pour les premières valeur de $n = |V|$ le nombre minimum de contraintes nécessaires. Dans le tableau suivant deux inéquations sont dans la même classe si les coefficient s'obtiennent par renumérotation des sommets. Ayez en tête que l'on résout relativement facilement aujourd'hui des instances de 2000 villes.

On donnera quelque exemples d'inéquations permettant de renforcer la formulation.

Références

[1] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Girija J. Narlikar. Space efficient scheduling of Parallelism with Synchronization Variables. In *Proceedings of the 9th Symposium on Parallel Algorithms and Architectures*. ACM Press, June 1997.

[2] A. Borodin. On relating time and space to size and depth. *SIAM Journal of Computing*, 6:733–744, 1977.

n	# tours	# contraintes différentes	# classes
3	1	0	0
4	3	3	1
5	12	20	2
6	60	100	4
7	360	3 437	6
8	2 520	194 187	24
9	20 160	42 104 442	192
10	181 440	$\geq$ 51 043 900 866	$\geq$ 15 379

Tableau 1: Statistiques

[3] A. Borodin and I. Munro. *The Computational Complexity of Algebraic and Numeric Problems* . Elsevier, New-York, 1975.

[4] R.P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21:201–206, 1974.

[5] E.G. Coffman and P.J. Denning. *Operating Systems Theory.* Prentice-Hall, 1973.

[6] S.A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[7] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, June 1998.

[8] J. von zur Gathen. Parallel arithmetic computations: a survey. In *Proc. 12th Int. Symp. Math. Found. Comput. Sci., Bratislava*, pages 93–112. Springer-Verlag LNCS 233, 1986.

[9] A.M. Gibbons and W. Rytter. *Efficient parallel algorithms.* Cambridge University Press, 1988.

[10] L.M. Goldschlager. The monoton and planar circuit value problems are log space complete for P. *SIGACT News*, 9:25–29, 1977.

[11] R.L. Graham. Bounds for Certain Multiprocessor Anomalies. *Bell System Tech J.*, 45:1563–1581, 1966.

[12] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–426, 1969.

[13] H.J. Hoover, M.M. Klawe, and N.J. Pippenger. Bounding fan-out in logical networks. *J. ACM*, 31:13–18, 1984.

[14] E. Kaltofen, G.L. Miller, and V. Ramachandran. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM J. Computing*, 17:687–695, 1988.

[15] R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leuwen, editor, *Algorithms and Complexity*, pages 869–932. Elsevier, 1990.

[16] Aicha Kerfali, Jean-Louis Roch, and El Mostafa Daoudi. Algorithmes parallèles à grain adaptatif - Application à la parallélisation de gzip. In *RENPAR'15*, pages 18–26, Nice, France, octobre 2003.

[17] Jean-Claude Konig and Jean-Louis Roch. Machines virtuelles et techniques d'ordonnancement. In *Proceedings école d'hiver de PRS ICARE'97*, Auusois, France, dec 1997. http://www-apache.imag.fr/ jlroch/ps/97-sched-aussois.ps.gz.

[18] Eric Mohr, David A. Kranz, and Jr Robert H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):263–280, July 1991.

[19] Girija J. Narlikar and Guy E. Blelloch. Space-efficient implementation of nested parallelism. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–36, June 1997.

[20] Victor Y. Pan and Franco P. Preparata. Work-preserving speed-up of parallel matrix computations. *SIAM Journal on Computing*, 24(4), 1995.

[21] N. Pippenger. On simultaneous ressource bounds. In *20 th. Annual IEEE Foundations of Computer Science*, pages 307–311, 1979.

[22] N. Revol. Evaluation parallèle de circuits arithmétiques. Technical Report Pré-rapport de thèse, IMAG, 1993.

[23] J.-L. Roch, T. Gautier, and R. Revire. Athapascan: Api for asynchronous parallel programming. Technical Report RT-0276, INRIA Rhône-Alpes, projet APACHE, February 2003.

[24] W.L. Ruzzo. On uniform circuit complexity. *J. Computer and System Sciences*, 22, 3:365–383, 1981.

[25] D. B. Shmoys, J. Wein, and P. Williamson. Scheduling parallel machines on-line. *SIAM Journal on Computing*, 24(6):1313–1331, 1995.