Gestion automatique de la mémoire

Paul Feautrier

Paul.Feautrier@ens-lyon.fr.

ENS Lyon

Gestion de la mémoire

Deux objectifs :

- Utiliser le moins de mémoire possible.
- Améliorer le fonctionnement des caches.

Trois contextes :

- Calcul séquentiel :
  - Améliorer la localité augmente les performances.
  - Malgré les progrès de la technologie, il existe des applications limitées par la taille mémoire (ex. la chimie quantique).
- Calcul Parallèle : pour avoir de bonnes performances :
  - Choisir le bon algorithme (exemple de la FFT).
  - Optimiser le fonctionnement des caches.
  - Paralléliser en minimisant les échanges entre processeurs.
- Systèmes embarqués :
  - Minimiser le coût de fabrication, donc la taille de la puce, donc la taille de la mémoire.
  - Minimiser la consommation, donc améliorer la localité.

Plan

- Réduction de la taille de la mémoire dans les programmes séquentiels et parallèles.
- Darte
- Lefebvre, Quilleré.
- Amélioration de la localité pour les programmes séquentiels.
- Wolf et Lam
- Cédric Bastoul
- Perspectives

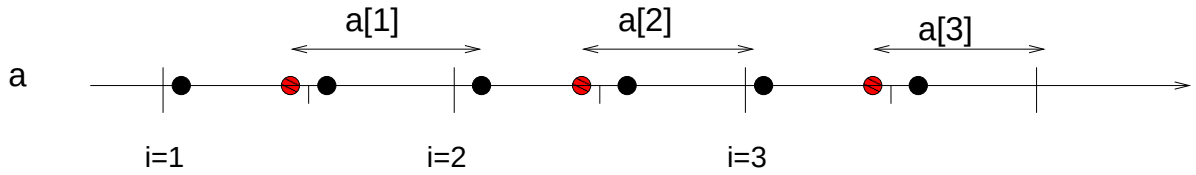
Réduire la taille de la mémoire

Analyse des durées de vie, I

$$e^x = \sum_{i=0}^{\infty} x^i / i!$$

```
for(i=1;i<n;i++}{
   a[i] = a[i-1]*x/i;
   s[i] = s[i-1] + a[i];
}
```

```
for(i=1;i<n;i++}{
   a = a*x/i;
   s = s + a;
}
```

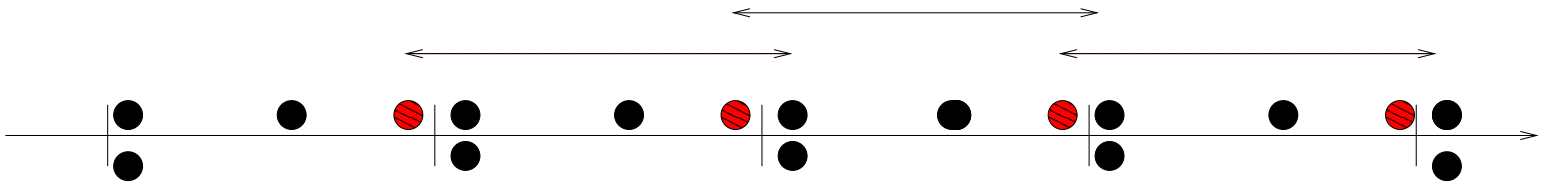


Il n'y a jamais besoin de stocker plus d'une valeur de a ou de s.

Analyse des durées de vie, II

$$\lim_{i \to \infty} s_i \quad s_{i+1} = f(s_i).$$

```
for(i=0; abs(s[i] - s[i-1] > eps); i++)
   s[i+1] = f(s[i]);
```



Il faut deux cellules de mémoire pour s.

```
for(i=0; abs(s[i%2] - s[(i-1)%2]) > eps; i++)
  s[(i+1)%2] = f(s[i%2]);
```

Cas d'un programme parallèle

- Zbigniew Chamski.
- On a construit le programme parallèle à partir d'une version en assignation unique.
- On a fabriqué une transformation d'espace temps pour chaque instruction S :

$$\begin{pmatrix} t \\ p \end{pmatrix} = H_S i,$$

où on suppose H_S unimodulaire.

- On réindexe les tableaux de façon que les nouveaux indices ne dépendent que de t et p.

Réindexation, I

- Soit A un tableau figurant à gauche dans une affectation. Puisqu'on est en assignation unique :

$$A[i] = \cdots.$$

- On veut que dans le nouveau programme on ait :

$$A'[t, p] = ...$$

  donc $A'[H_S i] = A[i]$.

- Soit B un tableau utilisé à droite dans S, et soit F sa matrice d'indexation : $B[Fi]$. Il est défini par une instruction T :

$$B[j] = \cdots;$$

Réindexation, II

- Soit H_T la transformation d'espace temps de T. B a été remplacé par un tableau $B'[H_T j] = B[j]$. Dans S, on doit donc remplacer $B[Fi]$ par :

$$B'[H_T F i] = B'[H_T F H_S^{-1}(t,p)].$$

- Comme la transformation d'espace temps doit respecter les dépendances, il faut que t soit supérieur à la première composante de $H_T F H_S^{-1}(t,p)$.
- Il est très fréquent que cette composante soit de la forme $t - d$ où d est un petit entier. Il est facile de voir que la première dimension de B' peut être réduite à $d+1$ en prenant tous les indices modulo $d+1$. Dans certains cas, on peut se contenter de d cellules de mémoire, mais la détection de ces cas est complexe.

Réindexation, exemple

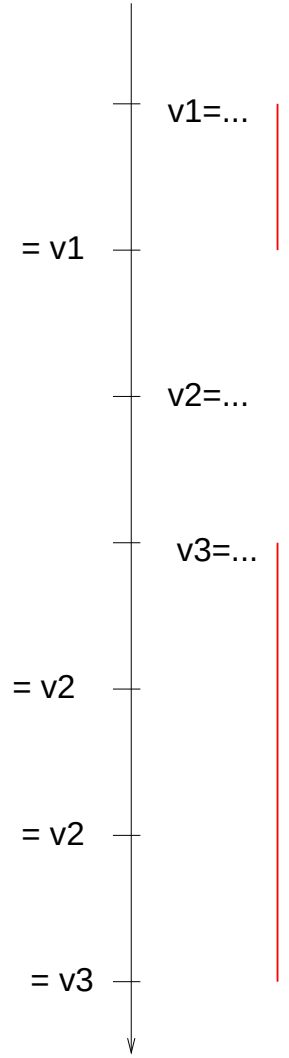
```
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    c[i][j] = c[i][j-1] + a[i][j] * b[j];
```

- L'ordonnancement est $\theta(i,j) = j$ et donc la matrice de la transformation espace temps est $H = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ en coordonnées homogènes. Cette matrice a pour déterminant -1 et elle est son propre inverse.
- La matrice d'indexation est $F = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix}$.
- Pour le produit HFH on trouve $F' = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ ce qui correspond au code :

```
for(t=0; t<n; t++)
  for(p=0; p<n; p++)
    c'[t][p] = c'[t-1][p] + a[p][t]*b[t];
```

et on peut réduire à 2 (en fait à 1) la première dimension de c'.

Disgression : allocation de registres



- On considère un bloc de base en forme SSA (chaque variable n'est affectée qu'une fois). Problème, allouer les variables au nombre minimum de registres.
- Pour chaque variable il est possible de déterminer un intervalle de vivacité. Deux variables interfèrent si leurs intervalles de vivacité ont une intersection non vide.
- On peut tracer un graphe d'interférence. On le colore avec le nombre minimum de couleurs. Chaque couleur représente un registre.
- Voir l'article original de Chaitin.
- Pour l'exemple, il faut deux couleurs, donc deux registres.

Darte, I

- On cherche à imiter la méthode précédente. On suppose que l'on a ordonnancé un programme en assignation unique. Soit θ l'ordonnancement.
- Soit A un tableau. Pour tout indice i, il existe au plus une opération unique u qui crée $A[i]$ à l'instant $\mathbf{First}(i) = \theta(u)$.
- On peut déterminer également $\mathbf{Last}(i) = \max\{\theta(v) \mid v \text{ lit } A[i]\}$.
- Deux indices i et j interfèrent ssi :

$$\max(\mathbf{First}(i), \mathbf{First}(j)) \leq \min(\mathbf{Last}(i), \mathbf{Last}(j)).$$

Si i et j interfèrent on pose $(i, j) \in \mathcal{I}$. On introduit également $\mathcal{D} = \{j - i \mid i, j \in \mathcal{I}\}$.

Darte, II

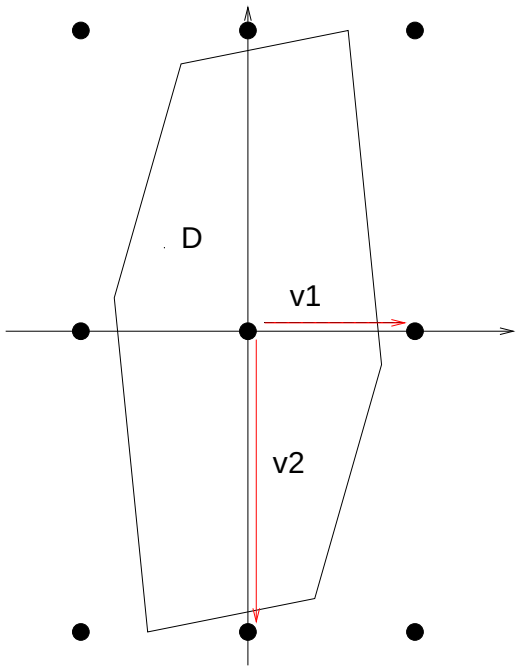
- On décide de remplacer A par un tableau contracté A' suivant la formule $A[i] \rightarrow A'[Mi \bmod b]$ ou M est une matrice entière et b un vecteur entier. La taille de A' est le produit des composantes de b.
- Pour que cette contraction soit légitime, il faut que :

$$i, i' \in \mathcal{I}, Mi \bmod b = Mi' \bmod b \Rightarrow i = i',$$

ou encore :

$$i \in \mathcal{D}, Mi \bmod b = 0 \Rightarrow i = 0.$$

Réseau Critique



- Les points tels que $Mi \bmod b = 0$ forment un réseau, qui ne doit toucher $\mathcal{D}$ qu'en l'origine.

$$\begin{aligned} Mi \bmod b &= 0, \\ Mi' \bmod b &= 0, \\ M(i+i') \bmod b &= 0. \end{aligned}$$

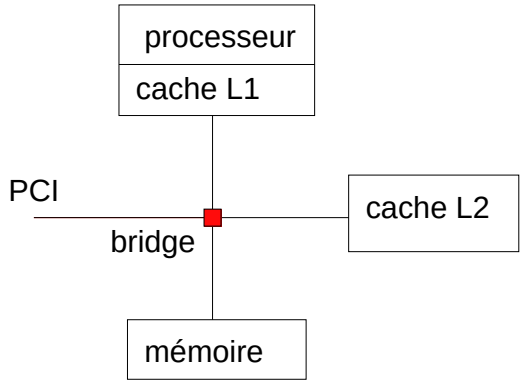
- Un réseau est *admissible* s'il n'a aucun point commun avec $\mathcal{D}$.
- Le réseau *critique* est celui qui a le plus petit déterminant, donc utilise le moins de mémoire.

Recherche du réseau critique

- On peut construire une méthode énumérative, dont la complexité est exponentielle avec la dimension du tableau.
- On peut déduire des bornes inférieures et supérieures à partir de résultats classiques de géométrie des nombres.
- Les méthodes de Lefebvre et de Quilleré sont des heuristiques pour trouver le réseau critique.

Amélioration de la localité

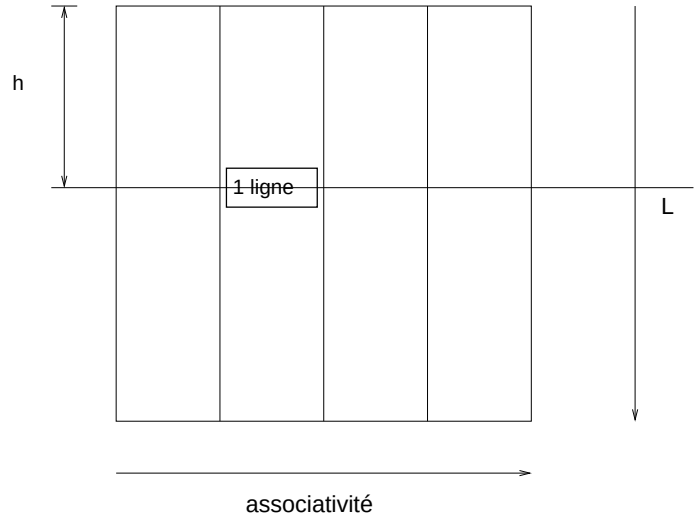
Amélioration de la localité



- Le processeur cherche d'abord une information dans le cache (*hit*) et sinon en mémoire (*miss*).
- En cas de miss, l'information demandée est notée dans le cache.
- S'il n'y a pas de place libre, on choisit une *victime* qui est renvoyée en mémoire.

- Il y a plusieurs algorithmes de *remplacement* : LRU, FIFO, RANDOM avec tabou. Leurs performances ne sont pas très différentes.
- On souhaite avoir le moins de miss possibles, pour des raisons de performance et d'économie d'énergie.

Structure d'un cache



- Une donnée d'adresse x va dans le *bucket*

$$h = \frac{x}{c} \bmod L.$$

c est la taille de la ligne de cache (8 à 32 octets) et a est l'associativité.

- $a = 1$: cache *direct mapped*.
- Une valeur usuelle est 4. Capacité totale $C = Lca$.

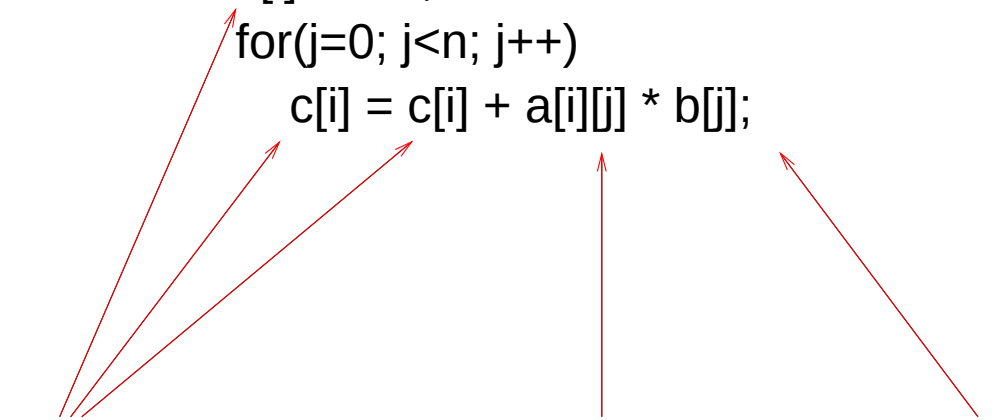
- Il y a interférence quand le nombre d'entrées qui ont la même valeur de h est plus grand que a.
- La situation idéale : $L = 1$ et $a = C/c$. On parle d'un cache totalement associatif. Il n'y a plus d'interférences que quand le cache est saturé.

Les localités

- « Une donnée qui vient d'arriver dans le cache a de grandes chances d'être réutilisée rapidement ».
- Localité temporelle. L'algorithme de remplacement essaie de garder une donnée en cache le plus longtemps possible.
- « Une donnée voisine d'une donnée en cache a de grandes chances d'être utilisée dans le futur ».
- Localité spatiale. On choisit une taille de ligne supérieure à la taille d'un mot mémoire.

Un exemple

```
for(i=0; i<n; i++)
    c[i] = 0.0;
    for(j=0; j<n; j++)
        c[i] = c[i] + a[i][j] * b[j];
```



localité temporelle

localité spatiale en C mais pas en Fortran

localité spatiale

Méthodes exactes

- Il est possible de construire des polyèdres dont l'union décrit les miss.
- Il est ensuite possible de compter les points de ces polyèdres par des méthodes mathématiques (polynomes d'Ehrhardt, automates fi nis).
- On peut ainsi prédire d'avance le nombre de miss (aux interférences du système près).
- Les formules obtenues peuvent être paramétriques par rapport à la taille du calcul (par exemple n ci-dessus) mais pas par rapport à la structure du programme.
- On ne peut donc ensuite procéder que par tatonnements. Ces méthodes n'ont été utilisées que dans des cas très particuliers (produit de matrices) et sans donner de résultats probants.
- Voir essentiellement les travaux de Sid Chatterjee et la thèse d'Adam Slowik.

Wolf et Lam

- Ne s'applique qu'à un nid de boucles parfait.
- On peut alors défi nir des distances de dépendance :

$$D = \{j - i \mid i \ \delta \ j\}.$$

- Tous les vecteurs de D sont lexicopositifs.
- Comme D est trop compliqué, on le résume par une famille de vecteurs de direction : $\{[a_1, b_1], \ldots [a_d, b_d]\}$ où d est la profondeur du nid. Cas particuliers : $a_i = -\infty, b_i = \infty, a_i = b_i$.

Nids totalement permutables

- Une transformation unimodulaire de matrice T est légale si

$$\forall d \in D : Td \gg 0.$$

- Un nid est *totalement permutable* ssi toutes les transformations qui sont des permutations sont légales. Il faut pour celà que toutes les coordonnées des vecteurs d soient positives.
- Il existe un algorithme qui étant donné un nid de boucle, le met sous la forme d'un nid séquentiel contenant un nid totalement permutable (qui peut être vide).

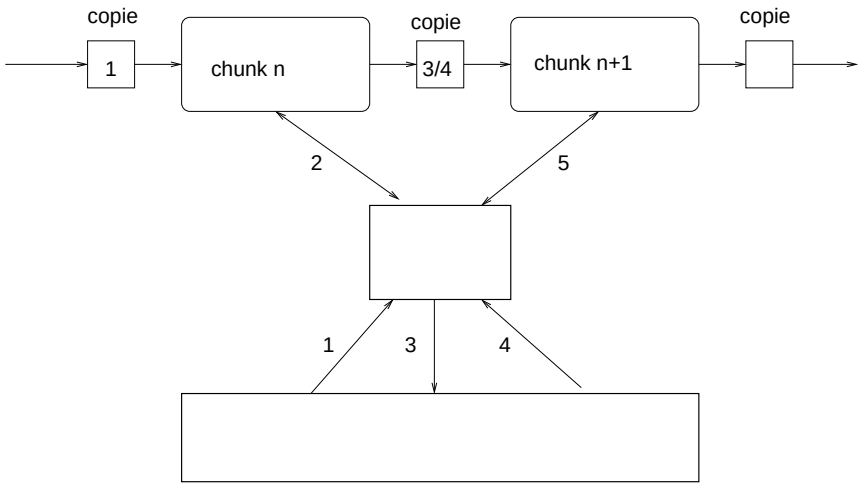
Vecteurs de réutilisation

- Il y a réutilisation si deux opérations accèdent à la même cellule de mémoire. (C'est une généralisation des dépendances).
- **Intuition** Une réutilisation ne peut être exploitée que si elle se produit dans la boucle la plus interne.
- On représente chaque réutilisation par un sous-espace vectoriel.
  1. **Self-Temporel** l'ensemble de réutilisation de $A[Fi + d]$ est $\ker F$.
  2. **Self-Spatial** pour le même accès, l'ensemble de réutilisation est $\ker F_S$. F_S se déduit de F en omettant la dernière ligne (en C) ou la première (en Fortran). Les réutilisations spatiales sont un sur-ensemble des réutilisations temporelles.
  3. **Groupe** On ne considère que les réutilisations uniformément engendrées : $A[Fi + d_1], \ldots, A[Fi + d_p]$. L'espace de réutilisation est $\ker F + \{c_2, \ldots, c_n\}$ où c_i est solution de $Fc_i = d_i - d_1$.

Esquisse de l’algorithme

- Trouver la transformation unimodulaire qui crée le plus possible de boucles totalement permutables.
- Calculer les sous-espaces de réutilisation.
- Trouver la permutation qui place le plus possible de réutilisations dans la boucle la plus interne.

Cédric Bastoul, Chunking



- On émule le fonctionnement d'une mémoire *scratch-pad*.
- Un *chunk* s'exécute depuis la mémoire scratch-pad, sauf pour les données qui ne sont pas réutilisées.

- Si on travaille avec un cache, les copies et recopies sont automatiques, mais il ne doit pas y avoir de miss pendant l'exécution du chunk.
- On associe à chaque chunk un numéro par une fonction θ. Les chunks sont exécutés dans l'ordre de leurs numéros, puis séquentiellement.
- Pour respecter les dépendances : $u \ \delta \ v \Rightarrow \theta(u) \leq \theta(v)$.
- On choisit une fonction de chunking affi ne, $\theta(i) = Ti + d$.

Estimations asymptotiques

- Pour une référence $A[Fi + b]$ dans une instruction S, on montre que la taille de l'empreinte (*footprint*) est :

$$F(t) = \mathbf{Card}(\{Fi \mid \theta(i) = t, i \in D_S\}).$$

Si les boucles sont de taille n, la valeur asymptotique est $O(n^\ell)$, avec $\ell = \mathbf{rank} \begin{pmatrix} F \\ T \end{pmatrix} - \mathbf{rank}\, T$.

- De même, le trafi c total est :

$$U = \mathbf{Card}(\{Fi, \theta(i) \mid i \in D_S\}),$$

dont la valeur asymptotique est $O(n^k), k = \mathbf{rank} \begin{pmatrix} F \\ T \end{pmatrix}$.

Algorithme

- On doit garantir que l'empreinte tient dans le cache et trouver le trafi c minimum. On choisit c tel que $n^c \leq C$, la capacité du cache.
- Les rangs de T et de $\begin{pmatrix} F \\ T \end{pmatrix}$ ne sont pas indépendants. On cherche par *bactracking* une combinaison qui donne des empreintes de taille au plus égale à n^c et le trafi c minimum.
- On construit la matrice T par complétion.
- On applique des transformations qui ne changent pas le rang jusqu'à satisfaire les dépendances.

Généralisation

- S'il y a plusieurs références dans la même instruction, on doit construire, toujours par complétion, une matrice T qui satisfait plusieurs conditions : $\begin{pmatrix} F_i \\ T \end{pmatrix} = r_i, i = 1, \ldots, p$.
- La localité temporelle se traite en enlevant la première (ou la dernière) ligne de F.
- Une méthode analogue permet de traiter la réutilisation de groupe.
- Il manque encore le tuilage.

Digression, Cloog

- Pour obtenir le programme objet, il faut réordonner les itérations suivant la fonction de chunking. Il s'agit d'une technique de parcours d'une union de polyèdre.
- Cloog est une extension d'une méthode de Quilleré. Il accepte une fonction de réordonnancement quelconque (ordonnancement, placement, chunking, etc).
- Il essaie de générer le moins de gardes possibles. On peut contrôler le degré d'expansion du code.
- Il calcule les pas des boucles, le cas échéant.
- Il est librement accessible `www.prism.uvsq.fr/~cedb/bastools`.

Autres Applications

- Le *chunking* fonctionne bien pour les caches, mais il a été conçu pour les mémoires scratch-pad.
- Important pour les applications embarquées, parce qu'on peut mieux prédire les performances et réduire la consommation.
- Ce qui manque :
  1. Allouer les morceaux de l'empreinte en mémoire locale.
  2. Ecrire les codes de copie et de recopie.
  3. Modifi er le code engendré par Cloog pour accéder à la mémoire scratch-pad.
- Trouver un processeur muni d'une mémoire scratch-pad (FPGA ?).
- Autre application : le calcul *out-of-core*. La mémoire joue le role du cache et les disques le role de la mémoire.

Perspectives

Localité dans les programmes parallèles

- Deux sortes de localité :
  - localité inter-processeur ; réduire le nombre de messages.
  - localité intra-processeur ; classique.
- En général, la parallélisation par ordonnancement détruit la localité :

```
for(t=0; t<L; y++)
   doall{F(t)}
```

  Le mécanisme des boucles parallèles distribue les itérations sans prendre en compte le contenu du cache.
- Est-il possible de faire mieux en parallélisant par placement (thèse de Yosr Slama) ?
- Ou en appliquant la méthode de Cédric Bastoul à des caches multiples ?

Ordonnancement sous contrainte de mémoire

Etant donné un programme P, trouver un programme P' :

- équivalent (qui respecte les dépendances).
- qui n'utilise pas plus de M mots de mémoire (de travail).
- qui va le plus vite possible (ajuster le nombre de processeurs).

Ce problème est ouvert. Il a de nombreuses variantes (on peut transformer une contrainte en fonction objectif et réciproquement).