Application des méthodes de décomposition de domaines au calcul parallèle

Yves Escaig — Gilbert Touzot

Laboratoire de Mécanique de Rouen
CNRS UPRESA 6104
Place Emile Blondel
BP 8
76131 Mont Saint-Aignan

RÉSUMÉ. Dans cet article, on s'intéresse aux applications du calcul parallèle distribué à la résolution de problèmes linéaires statiques en mécanique des structures. La méthode de résolution utilisée est la méthode des éléments finis. Dans la première partie, on rappelle quelques définitions et concepts du calcul parallèle. Dans la seconde partie, les méthodes de décomposition de domaines sont introduites. Enfin, on étudie une méthode de décomposition de domaines particulière dans laquelle le problème interface est assemblé et résolu de manière directe. Sa parallélisation est présentée et les performances obtenues sur Cray T3D sont analysées.

ABSTRACT. In this article, we study the use of distributed parallel processing for the resolution of linear static problems using the finite element method. In the first part, we briefly present some concepts and definitions of parallel processing. In the second part, the domain decomposition methods are introduced. Finally, we focus on a particular method where the interface problem is explicitely constructed and solved with a direct solver. Its parallelisation is presented and the performances obtained on a Cray T3D are analysed.

MOTS-CLÉS : calcul distribué, méthodes de décomposition de domaines, méthodes directes.
KEY WORDS: distributed computing, domain decomposition methods, direct solvers.

1. Introduction

Le calcul parallèle tient une place prépondérante dans les moyens de calculs à la disposition de la modélisation numérique. Ceci tient, d'une part aux économies qu'il permet de réaliser par rapport aux calculateurs séquentiels. Il est en effet plus facile de fabriquer N composants relativement simples qu'un seul composant très complexe. Les calculateurs parallèles actuels reprennent tous aujourd'hui des composants (processeurs, mémoire, bus,...) standard du marché. Il est même possible, malgré les limites de cette approche, de créer une machine parallèle « virtuelle » avec les réseaux de stations de travail présents dans la plupart des organismes de recherches. D'autre part, les calculateurs parallèles seuls permettent

d'obtenir de très grandes puissances de calcul. Personne n'envisage la machine « Teraflops » en dehors d'architectures multiprocesseurs. Néanmoins, l'utilisation efficace de ces machines, afin d'obtenir de hautes performances, pose de nombreux problèmes, non seulement aux informaticiens, mais aussi aux numériciens, mécaniciens, thermiciens qui développent des algorithmes et des méthodes numériques.

Cet exposé a pour objet de donner quelques éléments de base afin d'aborder le calcul parallèle, puis de montrer une application particulière du calcul parallèle : la méthode des éléments finis. Étant donné l'étendu du sujet, l'exposé ne pourra entrer dans tous les détails mais s'efforcera de donner des références bibliographiques.

Dans la suite de ce texte, la première partie sera consacrée à la présentation des concepts de base de l'architecture des machines parallèles et de leur mode de programmation. Dans une seconde partie, les méthodes de décomposition de domaines et de leur parallélisation seront abordées.

2. Les concepts du calcul parallèle

2.1. *Les composants de base*

Il existe 2 composants de base d'une machine parallèle qui vont influencer l'architecture de celle-ci. Il y a tout d'abord la mémoire qui peut être entièrement accessible à tous les processeurs (on parlera de mémoire partagée), ou être découpée en blocs, chacun étant uniquement accessible à un seul processeur (on parlera de mémoire distribuée). On peut évidemment envisager un certain nombre de solutions hybrides qui combinent les 2 modèles précédents.

Le second composant est le contrôle de l'activité des processeurs : qui va décider des instructions qui vont être exécutées par chaque processeur. Ce contrôle peut être entièrement centralisé ou au contraire distribué, c'est-à-dire que chaque processeur décide seul de ce qu'il fait.

2.2. *Les architectures*

Les architectures des machines parallèles sont généralement classifiées sous la forme d'une matrice à 2 lignes et 2 colonnes (voir figure 1) : les colonnes représentent le type de contrôle (les instructions) et les lignes le type de données sur lesquelles opèrent ces instructions.

On notera qu'aucune machine de type MISD n'a été construite. Pour les architectures de type MIMD, on peut envisager que la mémoire soit partagée ou distribuée.

Dans le cas où la mémoire est partagée, il y a des conflits d'accès à celle-ci lorsque plusieurs processeurs veulent lire ou écrire des données en même temps. Dans ce cas, les requêtes sont traitées les unes après les autres, ce qui entraîne une baisse des performances. En pratique, il n'est pas possible d'obtenir de bonnes performances au-delà de 16 processeurs.

Afin de s'affranchir de ces problèmes de conflits mémoire, la solution consiste à distribuer les mémoires pour que chaque processeur possède sa mémoire propre.

		INSTRUCTIONS	
		SINGLE	MULTIPLE
DATA	SINGLE	SISD (machines sequentielles)	MISD acune implémentation
	MULTIPLE	SIMD à mémoire distribuée - Connection Machine - Maspar	MIMD à mémoire partagée - Cray C90, Convex, SGI Challenge ou distribuée - Intel Paragon, Cray T3D, IBM SP

Figure 1. *Classification des machines parallèles*

Dans ce cas, il n'y a plus de conflits. En revanche, le partage d'informations entre processeurs nécessite d'effectuer des communications. Les gains potentiels par rapport aux mémoires partagées viennent du fait que ces communications n'ont lieu qu'à certains moments, alors que les références mémoires ont lieu en permanence. Dans la suite de ce texte, nous nous intéresserons exclusivement aux machines MIMD.

Dans le cas des machines à mémoire distribuée, la composante essentielle est le réseau d'interconnexion des couples processeurs-mémoire. Trois paramètres principaux conditionnent la qualité et les performances du réseau :

- la largeur : le nombre de processeurs séparant les extrémités du réseau,
- le degré : le nombre de lien par processeur,
- la facilité de construction et d'extension.

Les topologies de réseau les plus utilisées sont :

- les hypercubes, mais de moins en moins à cause de difficultés de mise en œuvre,
- les grilles 2D,
- les tores 3D.

Un autre point qui influence les performances des communications est la méthode de transmission des messages entre 2 processeurs non directement connectés. La première méthode, dite « store and forward » consiste à sauvegarder le message entier sur chaque processeur intermédiaire avant de le transmettre au

processeur suivant. Cette méthode ne nécessite aucun matériel particulier, mais elle utilise peu efficacement le réseau puisqu'un seul lien est actif à un moment donné. Il faut de plus prévoir des mémoires tampon suffisantes sur chaque processeur pour sauvegarder le message entier. L'autre méthode, dite « cut-through » consiste à découper le message en sous-parties, puis à faire transiter celles-ci sur le réseau à la manière d'un pipeline. Sur les processeurs intermédiaires, les sous-parties des messages ne sont pas stockées, mais directement réémises. Cette solution, beaucoup plus efficace, nécessite du matériel spécialisé, en général sous la forme d'un processeur de communication.

Sur les architectures des machines multiprocesseurs, on peut consulter [HWA 84] [HWA 93] [STO 93] [HOC 92] [COS 93].

2.3. *Les modèles de programmation*

On peut définir un modèle de programmation comme l'ensemble des mécanismes permettant l'organisation des tâches d'un programme et des communications entre les processeurs. A l'origine, les modèles de programmation étaient très liés aux architectures des machines.

Aujourd'hui, ils sont complètement indépendants. On distingue trois modèles principaux :

- échanges explicites de messages : les tâches s'exécutant sur les différents processeurs sont indépendantes ; elles communiquent des informations et se synchronisent à l'aide d'échange de messages. Ces messages sont programmés directement par l'utilisateur. Ce modèle est celui qui permet la réalisation des applications les plus performantes, mais il est celui qui demande le plus d'efforts à l'utilisateur.

- parallélisme de données : dans ce modèle, la structure séquentielle du programme n'est pas modifiée. Chaque processeur exécute le même programme, mais sur des données différentes. L'utilisateur spécifie uniquement la distribution des données sur les processeurs, et c'est le système (en général un compilateur) qui génère les messages entre les processeurs. Ce modèle est plus simple, entraîne moins d'erreurs, mais n'est applicable qu'à une classe particulière de problèmes où les données sont régulières (matrices pleines).

- mémoire distribuée virtuellement partagée : dans ce modèle, la distribution des mémoires est masquée par le système : l'utilisateur voit un espace d'adressage unique. La programmation est effectuée comme dans le cas d'une machine à mémoire partagée.

L'utilisation de ces différents modèles est illustrée sur l'exemple suivant :

```
      REAL X(N,N), Y(N,N)

      DO J = 2, N-1
        DO I = 2, N-1
          X(I,J)=0.25*(Y(I-1,J)+Y(I+1,J)+Y(I,J-1)+Y(I,J+1))
        END DO
      END DO
```

Figure 2-a. *Exemple de boucle à paralléliser*

```
      REAL X(N,N_OVER_P), Y(N,0:N_OVER_P+1)
      IF (MY_ID .NE. 0) THEN
        SEND( Y(1:N,1), MY_ID-1 )
      END IF
      IF (MY_ID .NE. P-1) THEN
        SEND( Y(1:N,N_OVER_P), MY_ID+1 )
      END IF
      IF (MY_ID .NE. P-1) THEN
        RECEIVE( Y(1:N,N_OVER_P+1), MY_ID+1 )
      END IF
      IF (MY_ID .NE. 0) THEN
        RECEIVE( X(1:N,0), MY_ID-1 )
      END IF
      LOW = 1
      IF (MY_ID .EQ. 0) LOW=2
      HIGH = N_OVER_P
      IF (MY_ID .EQ. P-1) HIGH=N_OVER_P-1
      DO J = LOW,HIGH
        DO I = 2, N-1
          X(I,J) = 0.25*(Y(I-1,J)+Y(I+1,J)+Y(I,J-1)+Y(I,J+1))
        END DO
      END DO
```

Figure 2-b. *Exemple de programmation avec le modèle « échanges explicites de messages »*

```
      REAL X(N,N), Y(N,N)
!HPF$ DISTRIBUTE (*,BLOCK)16:23 PM:X,Y
      FORALL ( J = 2 : N-1 )
        FORALL ( I = 2 : N-1 )
          X(I,J) = 0.25*(Y(I-1,J)+Y(I+1,J)+Y(I,J-1)+Y(I,J+1))
        END FORALL
      END FORALL
```

Figure 2-c. *Exemple de programmation avec le modèle « parallélisme de données »*

```
      REAL X(N,N), Y(N,N)
CMIC$ DO ALL SHARED(X,Y,N) PRIVATE(J,I) NUMCHUNK=8
      DO J = 2, N-1
        DO I = 2, N-1
          X(I,J) = 0.25*(Y(I-1,J)+Y(I+1,J)+Y(I,J-1)+Y(I,J+1))
        END DO
      END DO
```

Figure 2-d. *Exemple de programmation avec le modèle « mémoire virtuellement partagée » (syntaxe de l'Autotasking de Cray)*

Etant donné le type d'applications visées (modélisation par éléments finis), nous nous intéresserons uniquement au modèle d'échanges explicites de messages.

Sur les modèles de programmation, et plus généralement la programmation des machines parallèles, on peut consulter [LEW 92][LEW 93][BRA 93][COS 93] [KUM 94] [FOS 95].

2.4. *Analyse des performances*

Le but des machines parallèles étant d'accélérer l'exécution d'applications, l'analyse des performances est essentielle. Le principal critère d'analyse des performances est le facteur d'accélération. C'est celui que l'on étudiera ici. Néanmoins, il est nécessaire de prendre compte, lors de l'évaluation de l'intérêt d'une parallélisation, l'augmentation de la taille des problèmes qui peuvent être résolus grâce aux machines parallèles.

On définit le facteur d'accélération S d'un programme par :

$$S = \frac{T_{séquentiel}}{T_{parallèle}}$$

Ce facteur mesure l'efficacité de la parallélisation, de l'algorithme, mais n'indique pas les gains absolus obtenus car l'algorithme sélectionné pour la parallélisation peut être moins efficace que le meilleur algorithme séquentiel connu. Il faut donc définir le facteur d'accélération par :

$$S = \frac{T_{du\ meilleur\ algorithme\ séquentiel}}{T_{parallèle}}$$

En plus du facteur d'accélération, on définit l'efficacité E d'un programme parallèle par :

$$E = \frac{S}{p}$$ (p=nombre de processeurs).

On peut noter que S peut augmenter avec P alors que E diminue. Les performances d'une application parallèle sont aussi fortement influencées par sa granularité G définie par :

$$G = \frac{T_{calcul}}{T_{communication}}$$

Dans le cas idéal, l'utilisation de P processeurs permettra un facteur d'accélération de P. Dans la réalité, trois facteurs principaux viennent faire baisser cette valeur.

Il y a tout d'abord la fraction séquentielle du programme. Les gains dus à la parallélisation ne sont obtenus, et c'est une évidence, que sur la partie parallèle du programme. La fameuse loi d'Amdahl permet de quantifier cette perte :

$$T_{parallèle} = (1 - f_p).T_{seq} + \frac{f_p T_{seq}}{p}$$

où f_p est la fraction parallèle du programme. On obtient donc :

$$S = \frac{T_{seq}}{T_{parallèle}} = \frac{1}{1 - f_p + \frac{f_p}{p}}$$

Ensuite, le déséquilibre entre les temps d'exécution des différentes tâches parallèles augmente le temps d'exécution. (voir figure 3).

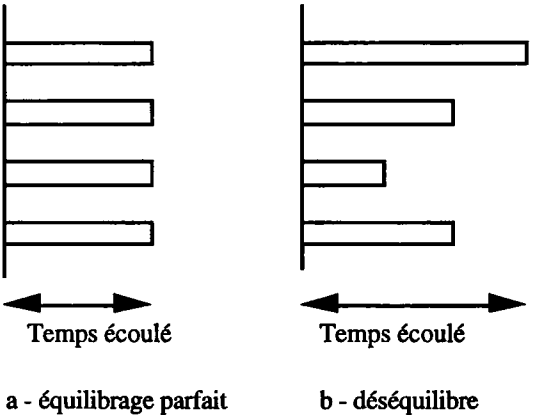


Figure 3. *Exemple de déséquilibre de charge d'une boucle parallèle*

Enfin, le troisième facteur est le temps consacré aux communications entre processeurs. Généralement, le temps de communication d'un message d'une longueur de « n » octets peut être modélisé par :

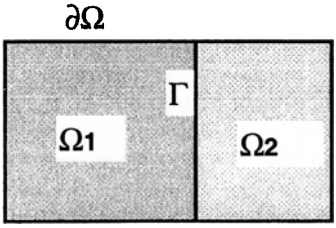
$T_{com} = \alpha + \beta.n$

où α est le temps de démarrage (latence) et β le temps de transmission d'un octet. Sur la plupart des machines, $\alpha >> \beta$. Il est donc nécessaire de limiter non seulement le volume des communications, mais aussi le nombre de messages.

3. Parallélisation d'une méthode de décomposition de domaines

3.1. *Introduction aux méthodes de décomposition de domaines*

On considère le domaine Ω suivant, découpé en deux sous-domaines Ω_1 et Ω_2 sans recouvrement.



On se propose de résoudre le problème suivant :

$$\begin{cases} L\,u = f & \text{sur } \Omega \\ u = u_0 & \text{sur } \partial\Omega \end{cases} \qquad (3.1)$$

où L est un opérateur différentiel elliptique du 2nd ordre. Résoudre (3.1) revient à résoudre, si on découpe Ω en deux sous-domaines Ω_1 et Ω_2, des problèmes de la forme

$$L\,u_1 = f \text{ dans } \Omega_1 \quad \text{et} \quad L\,u_2 = f \text{ dans } \Omega_2$$

avec des conditions de raccordement adéquates sur l'interface, du type

$$\begin{aligned} &\Phi(u_1) = \Phi(u_2) \quad \text{sur} \quad \Gamma' \text{ avec} \quad \Gamma' \subseteq \Gamma \\ &\Psi(u_1) = \Psi(u_2) \quad \text{sur} \quad \Gamma'' \text{ avec} \quad \Gamma'' \subseteq \Gamma \end{aligned} \qquad (\text{avec } \Gamma = \Gamma' \cup \Gamma'')$$

Dans le cas du Laplacien, $\Phi(u) = u$, $\Psi(u) = \dfrac{u}{n}$, $\Gamma = \Gamma' = \Gamma''$

3.1.1. *Méthodes de résolution*

Les algorithmes de résolution par décomposition en sous-domaines reviennent à résoudre les sous-problèmes indépendamment les uns des autres en tenant compte des conditions de raccord. Et pour tenir compte de ces conditions de raccordement, il y a trois stratégies envisageables.

La première stratégie consiste à obtenir comme résultat d'un calcul (soit direct soit itératif) la condition

$$\Psi(u_1) = \Psi(u_2) \qquad \left(\frac{\partial u_1}{\partial n_1} + \frac{\partial u_2}{\partial n_2} = 0 \text{ dans le cas du Laplacien}\right)$$

la solution cherchée vérifiant par hypothèse $\Phi(u_1) = \Phi(u_2)$ ($u_1 = u_2$ dans le cas du Laplacien). C'est ce que l'on appelle la méthode du complément de Schur.

La deuxième stratégie est duale de la première. On cherche à satisfaire la relation $\Phi(u_1) = \Phi(u_2)$ en s'imposant au départ $\Psi(u_1) = \Psi(u_2)$. La contrainte $\Phi(u_1) = \Phi(u_2)$ est traitée au moyen d'un multiplicateur de Lagrange. Cette méthode est appelée méthode hybride ou méthode des multiplicateurs de Lagrange [ROU 89].

Enfin, la troisième possibilité consiste à mélanger les deux stratégies précédentes. Sur un sous-domaine, on impose la condition $\Phi(u_1) = \Phi(u_2)$ pour en déduire la valeur de $\Psi(u_1)$, alors que sur l'autre sous-domaine, on impose la condition $\Psi(u_2) = \Psi(u_1)$ pour en déduire la valeur de $\Phi(u_2)$. C'est la méthode développée par Quarteroni [QUA 90].

3.1.2. *La méthode du complément de Schur en éléments finis*

Soit un domaine Ω découpé en deux sous-domaines Ω_1 et Ω_2. Soit Γ leur interface. Afin de discrétiser le problème, on utilise une méthode d'éléments finis. A partir du problème variationnel discrétisé, en numérotant d'abord les degrés de liberté(d.d.l.) de Ω_1, puis ceux de Ω_2, puis enfin ceux de Γ, on obtient ainsi un système linéaire final à résoudre de la forme :

$$\begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix}$$

A_{11} : bloc correspondant aux d.d.l. situés à l'intérieur de Ω_1

A_{22} : bloc correspondant aux d.d.l. situés à l'intérieur de Ω_1

A_{33} : bloc correspondant aux d.d.l. situés sur l'interface Γ

A_{i3} : bloc correspondant au couplage entre les d.d.l. de Ω_i et les d.d.l. de Γ

Dans le cas d'un opérateur symétrique, ce système est symétrique, et on a $A_{3i} = [A_{i3}]^t$.

En utilisant les deux premières équations, on peut éliminer les inconnues u_1 et u_2 de la 3^e équation et on obtient le système suivant :

$$\begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ 0 & 0 & S \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \tilde{f}_3 \end{pmatrix}$$

$$S = A_{33} - A_{31} A_{11}^{-1} A_{13} - A_{32} A_{22}^{-1} A_{23}$$

$$\tilde{f}_3 = f_3 - A_{31} A_{11}^{-1} f_1 - A_{32} A_{22}^{-1} f_2$$

Dans le cas d'un opérateur coercif, les blocs A_{ii} sont inversibles car ce sont les matrices associées au problème de départ avec des conditions aux limites de Dirichlet homogènes sur l'interface.

La matrice S est appelée matrice du complément de Schur. Elle correspond à la discrétisation d'un opérateur de Steklov-Poincaré sur l'interface Γ [AGO 87]. Cet opérateur permet de passer, sur l'interface, de $\Phi(u)$ à $\Psi(u)$. Si l'opérateur associé au problème de départ est symétrique, la matrice S est symétrique. Si la matrice A de départ est définie positive, alors la matrice S l'est également. En effet, on a la relation matricielle suivante :

$$\Delta = P^t A P \quad avec \quad A = \begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ [A_{13}]^t & [A_{23}]^t & A_{33} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \tilde{f}_3 \end{pmatrix} \quad \Delta = \begin{pmatrix} A_{11} & 0 & 0 \\ 0 & A_{22} & 0 \\ 0 & 0 & S \end{pmatrix}$$

$$P = \begin{pmatrix} I & 0 & -[A_{11}]^{-1}[A_{13}] \\ 0 & I & -[A_{22}]^{-1}[A_{23}] \\ 0 & 0 & I \end{pmatrix}$$

qui représente un changement de base pour la matrice A, et montre donc que Δ et S sont également définies positives si A l'est.

Le problème à résoudre sur l'interface, appelé problème global, s'écrit donc :

$$S\, u_3 = \tilde{f}_3 \text{ sur } \Gamma \tag{3.2}$$

Pour résoudre ce problème global, on peut procéder de deux manières.

La première manière consiste à construire explicitement la matrice S et le second membre $\tilde{f}_3$, puis à résoudre de manière directe le système linéaire ainsi obtenu.

C'est la stratégie qui est employée dans la méthode des sous-structures, bien connue en mécanique des solides. Cette même stratégie est également employée dans la méthode multifrontale, qui associe méthode frontale [IRO 69] et méthode des sous-structures.
Cependant, la construction explicite de la matrice S peut s'avérer très coûteuse car elle nécessite l'inversion des matrices A_{ii}. D'où l'idée d'utiliser, pour résoudre le problème (3.2), des méthodes itératives qui ne nécessitent pas le calcul explicite de la matrice S, mais uniquement celui du produit de cette matrice par des vecteurs. Le calcul d'un produit S.v, nécessaire dans les méthodes itératives, se ramène alors à l'évaluation de $(A_{3i}\ A_{ii}^{-1}\ A_{i3})\ v$ qui se déroule en trois étapes :

1. $w = A_{i3}.v$ *(produit matrice - vecteur)*
2. $x = A_{ii}^{-1}.w$ *(résolution du problème $A_{ii}x = w$ sur le sous domaine Ω_i)*
3. $y = A_{3i}.x$ *(produit matrice - vecteur)*

Afin de réaliser l'étape (2), on a ici encore le choix entre méthodes itératives et méthodes directes. Le meilleur choix semble être celui des méthodes directes [ROU 89]. En effet, les problèmes locaux devant être résolus de nombreuses fois durant le processus itératif global, le coût de la factorisation des matrices locales a tendance à être amorti. De plus, ce coût de factorisation dépend de la largeur de bande des matrices. Or, le découpage en sous-domaines favorise une numérotation des nœuds qui diminue cette largeur de bande. Il est cependant souvent nécessaire de renuméroter les nœuds dans chaque sous-domaine après le découpage dans le cas d'une approche transparente. Enfin la résolution itérative, donc inexacte, des problèmes locaux dégrade la précision des quantités calculées et le conditionnement du problème global, ce qui induit un plus grand nombre d'itérations au niveau global.

Comme algorithme itératif de résolution du problème (3.2), dans le cas où la matrice S est symétrique définie positive, on peut choisir l'algorithme du gradient conjugué. Or, à chaque itération, il est nécessaire de calculer des produits matrice-vecteur de type Sv, ce qui revient à résoudre un problème sur chaque sous-domaine. Pour être compétitive, cette méthode doit nécessiter beaucoup moins d'itérations que d'inconnues d'interface, afin de réduire le nombre de ces résolutions. Pour cela, il faut trouver un bon préconditionneur de la matrice du complément de Schur S. Un tel préconditionneur doit essayer de satisfaire les conditions suivantes :

- rendre la résolution du problème à l'interface
  - indépendante de h, pas de discrétisation à l'intérieur des sous-domaines,
  - indépendante de H, dimension globale des sous-domaines,
  - indépendante des coefficients de l'équation à résoudre,
  - indépendante de la géométrie.

- être applicable à une large classe de problèmes,

- être peu coûteux et parallélisable.

Le développement de tels préconditionneurs constitue une part importante de la recherche sur la décomposition de domaines. On trouve une bibliographie fournie sur le sujet, dont quelques exemples sont [BRA 89] [LET 90] [LET 94]. Sur les méthodes de décomposition de domaines, on peut consulter en particulier [FAR 94].

Dans la suite de ce texte, on s'intéresse aux méthodes avec résolution directe du problème interface.

3.2. *Méthode directe de décomposition de domaines*

L'algorithme de la méthode du complément de Schur avec résolution directe du problème interface peut être décrit de la manière suivante :

- étape 1. Renumérotation des équations de chaque sous-domaine
- étape 2. Élimination des degrés de liberté (ddl) internes : condensation
- étape 3. Assemblage du problème interface
- étape 4. Résolution du problème interface
- étape 5. Restitution des résultats du problème interface dans les sous-domaines

Cet algorithme a été implanté dans le logiciel de modélisation en thermique et calcul des Structures SIC (Système Interactif de Conception) [AUN 90] [BRE 91] [TOU 89].

Les étapes 2 et 4 sont les plus consommatrices en temps CPU. L'analyse des performances séquentielles (voir figure 4) montre que :

- le temps total d'exécution dépend du nombre de sous-domaines,
- le temps de l'exécution de l'étape 2 est dominante pour un petit nombre de sous-domaines (jusqu'à 16) et décroît rapidement avec le nombre de sous-domaines,
- le temps d'exécution de l'étape 4 augmente régulièrement avec le nombre de sous-domaines, puisque la taille du problème interface augmente.

On peut expliquer la baisse du temps d'exécution par le fait que le découpage permet de réduire la largeur de bande de chaque sous-domaine. Dans cet exemple, on peut constater que la méthode de décomposition de domaines est plus efficace qu'une méthode de résolution sans découpage dans le cas où la matrice globale est stockée de manière ligne de ciel (*skyline*), et est comparable (et quelquefois meilleure) dans le cas d'un stockage creux (*sparse*) [ESC 96].

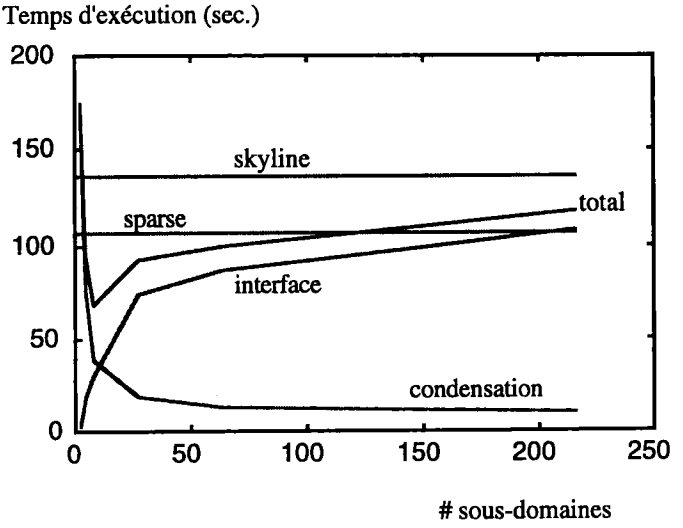


Figure 4. *Performances en séquentiel de la méthode de décomposition de domaines directe*

3.3. *Parallélisation de la méthode*

La condensation de chaque sous-domaine (étape 2) est indépendante et peut donc être effectuée en parallèle. Cette étape, qui ne nécessite aucune communication, pourra conduire à des performances maximum. Pour l'étape 4, la matrice du problème interface est distribuée sur tous les processeurs, la factorisation et la résolution ont lieu en parallèle, avec les performances connues à ce type d'algorithme : celles-ci sont bonnes lorsque la largeur de bande du système est suffisamment importante par rapport au nombre de processeurs [FAR 94] [ORT 89] [GAL 94] [DON 91].

Le choix le plus important du point de vue des performances est celui du nombre de processeurs affecté à chaque sous-domaine. Le choix le plus naturel consiste à affecter un processeur par sous-domaine.

Ce choix permet d'utiliser l'ensemble des processeurs tout au long de l'algorithme, tout en limitant le nombre de sous-domaines, donc, la taille du problème interface. Mais ce choix pose 2 problèmes : il implique généralement un mauvais équilibrage de la charge lors de la phase de condensation puisqu'il n'existe aucun degré de liberté pour la répartition des tâches. D'autre part, lorsque le nombre de processeurs devient important (>64) les performances de l'algorithme se rapprochent beaucoup de celles d'une résolution parallèle d'un système d'équations linéaires, limitant donc l'intérêt de la méthode de décomposition de domaines. Pour améliorer l'équilibrage des tâches, on peut augmenter le nombre de sous-domaines du nombre de processeurs, mais alors, on augmente également la taille du problème interface. Il est possible au contraire, d'affecter plusieurs processeurs par sous-domaines. Cela permet de découpler le nombre de sous-domaines du nombre de processeurs disponibles, donc de donner plus de liberté dans la définition des sous-

domaines. On peut aussi améliorer l'équilibrage des tâches en affectant un nombre différent de processeurs par sous-domaine. Mais cela oblige à paralléliser la phase de condensation. Dans la suite de ce texte, nous ferons toujours le choix d'affecter un processeur par sous-domaine.

L'algorithme distribué peut s'écrire de la manière suivante :

- distribution de données éléments finis de chaque sous-domaine,
- condensation des sous-domaines en parallèle,
- échange global de toutes les matrices condensées à tous les processeurs, puis assemblage de ces matrices,
- factorisation et résolution parallèle du problème interface,
- restitution des résultats du problème interface en parallèle,
- regroupement des résultats de tous les sous-domaines sur un processeur.

3.4. *Analyse des performances parallèles*

Pour analyser les performances, on considère le problème test en calcul des structures d'un cube, encastré à sa base, et soumis à un chargement constant sur sa face supérieure. Ce cube est découpé en sous-cubes réguliers pour constituer les sous-domaines. Ce problème est représentatif de la plupart des structures tridimensionnelles massives.

Comme cela fut décrit précédemment, on cherche à quantifier le facteur d'accélération de la méthode. Le calcul de ce facteur fait intervenir le temps d'exécution séquentiel. Or, les problèmes traités sur P processeurs ne peuvent généralement pas être résolus sur un processeur. Pour cette raison, on introduira la notion de facteur d'accélération référencé, où le temps d'exécution séquentiel est mesuré sur une machine séquentielle différente de la machine parallèle. On peut noter que pour les grands problèmes, il n'existe pas de machine séquentielle ayant une mémoire suffisante pour permettre la résolution. Dans ce cas, les performances ne peuvent qu'être extrapolées. Les performances parallèles ayant souvent tendance à s'améliorer lorsque la taille des problèmes augmente, on introduira la notion de facteur d'accélération mis à l'échelle, où pour chaque nombre de processeurs, on donne les performances obtenues lors de la résolution du plus grand problème possible. Les résultats présentés ont été obtenus sur le CRAY T3D du CEA à Grenoble. Cette machine possède 128 processeurs, 64 Mo de mémoire par processeurs (48 Mo sont utilisables en pratique par une application). Chaque processeur fournit une puissance d'environ 15 Mflops. Les facteurs α et β des communications ont pour valeur approximative 5.10^{-6} s et 1.10^{-8} s.

3.4.1. *Évolution avec le nombre de processeurs*

On commence par analyser l'évolution des différentes étapes de l'algorithme distribué en fonction du nombre de processeurs, pour différentes tailles de problèmes (figure 5). Pour un petit nombre de processeurs (4, 8, 16), l'étape de condensation est largement dominante. Pour 32 processeurs, l'étape de factorisation du problème interface devient prépondérante. On remarque également l'importance prise par l'étape d'échange des matrices condensées et par l'étape de résolution du problème interface.

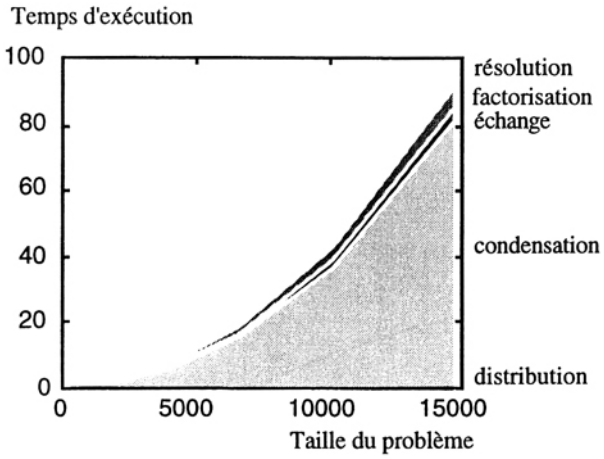


Figure 5-a. *4 processeurs*

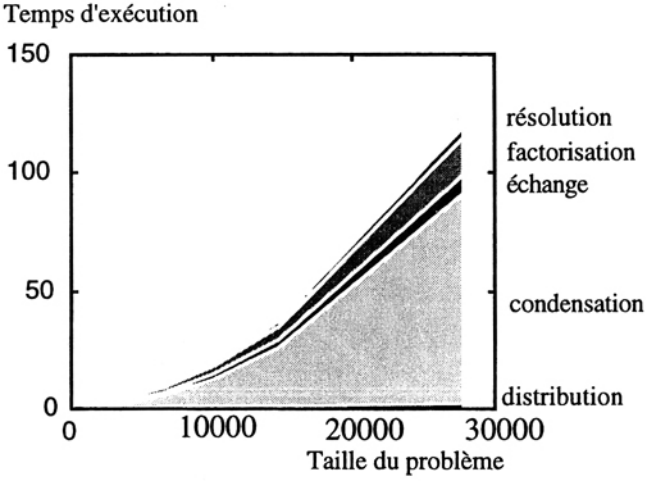


Figure 5-b. *8 processeurs*

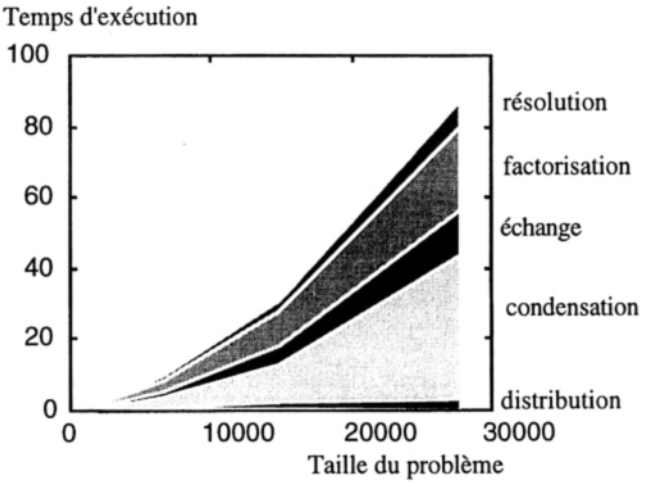


Figure 5-c. *16 processeurs*

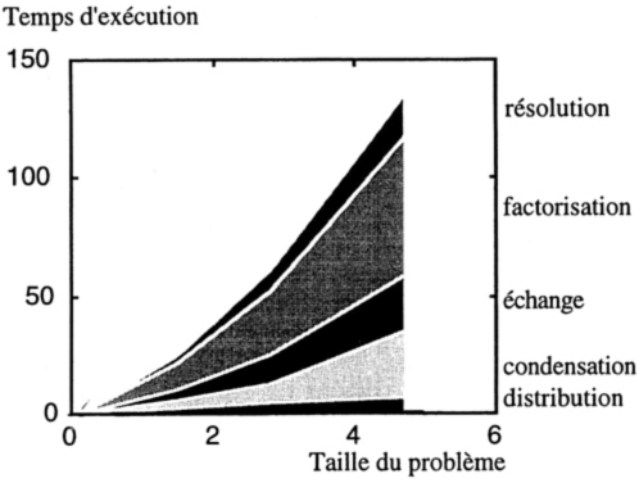


Figure 5-d. *32 processeurs*

Figure 5. *Evolution des temps d'exécution des différentes étapes de l'algorithme distribué en fonction du nombre de processeurs et de la taille des problèmes*

3.4.2. *Facteurs d'accélération mesurés*

Pour les cas qui peuvent être traités par un processeur, on peut calculer le facteur d'accélération réel, calcul basé sur le temps d'exécution écoulé (figure 6). Sur ces courbes, on extrapole les résultats pour des problèmes que l'on ne peut pas traiter sur un processeur. On constate des performances proches du maximum. Cela est dû au faible nombre de processeurs et au découpage en sous-domaines très réguliers qui favorise l'équilibrage des tâches.

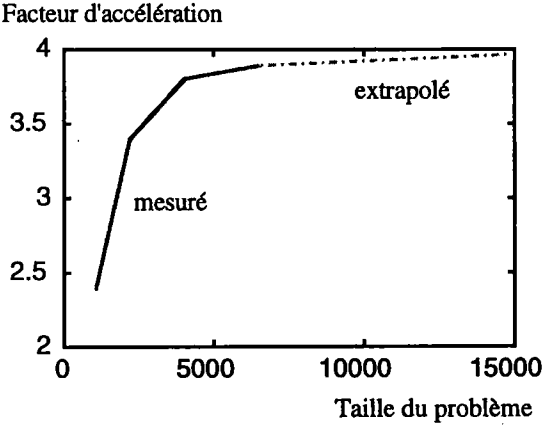


Figure 6-a. *Facteurs d'accélération mesurés pour 4 PE*

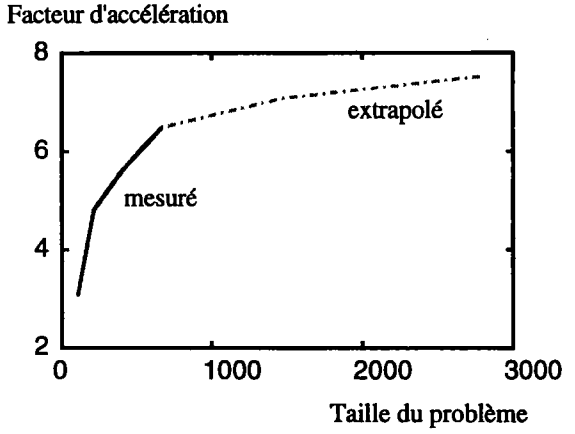


Figure 6-b. *Facteurs d'accélération mesurés pour 8 PE*

3.4.3. *Facteurs d'accélération référencés*

Pour obtenir une valeur approchée du facteur d'accélération pour 16 et 32 processeurs, on utilise en référence des temps d'exécution séquentiels obtenus sur un Cray J916 possédant 1 Go de mémoire (Cray de l'ENSAM à Paris). Le calcul de l'estimation du facteur d'accélération se déroule en 2 étapes. On commence à calculer le ratio R pour P processeurs :

$$R(p) = \frac{T_{J916}}{T_{p\ processeurs\ T3D}}$$

Dans un second temps, on calcule le facteur d'accélération estimé par rapport au facteur d'accélération réellement mesuré pour un nombre plus faible de processeurs :

$$S_{estimé}(p) = S_{mesuré}(4)\frac{R(p)}{R(4)}$$

En utilisant cette formule, on obtient les facteurs d'accélération estimés décrits sur la figure 7.

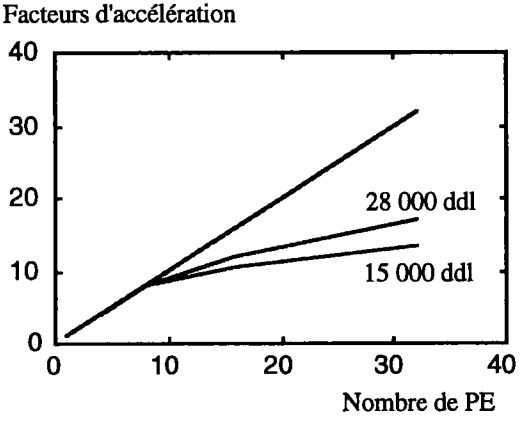


Figure 7. *Facteurs d'accélération référencés*

3.4.4. *Facteurs d'accélération mis à l'échelle*

Sur la figure précédente, on constate, pour 32 processeurs par exemple, que les performances augmentent avec la taille des problèmes. Il est donc intéressant de calculer les facteurs d'accélération (estimés) mis à l'échelle (figure 8).

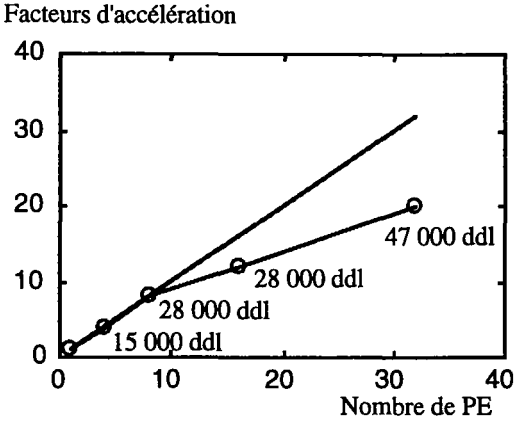


Figure 8. *Facteurs d'accélération mis à l'échelle*

Pour 32 processeurs, l'efficacité est encore de 63 %.

Une autre manière de présenter les résultats, et les gains obtenus grâce au calcul parallèle, consiste à indiquer directement les temps d'exécution mesurés. Ceci permet une analyse plus qualitative que quantitative (figure 9).

Pour améliorer les performances de cet algorithme, il est possible d'utiliser les particularités de la bibliothèque de communication SHMEM du Cray T3D, afin d'améliorer les vitesses de communication, donc l'efficacité des étapes d'échange et de factorisation.

3.5. *Conclusion*

L'algorithme distribué présenté précédemment offre une très grande efficacité pour un faible nombre de processeurs, et une efficacité moyenne pour un plus grand nombre de processeurs. Les performances s'améliorent lorsque la taille du problème augmente. Il faut donc augmenter la mémoire disponible sur chaque processeur en même temps que l'on augmente le nombre de processeurs.

Un certain nombre d'optimisations peuvent être effectuées pour améliorer les performances. Mais la principale modification consiste à utiliser plusieurs processeurs par sous-domaines afin de réduire la taille du problème interface.

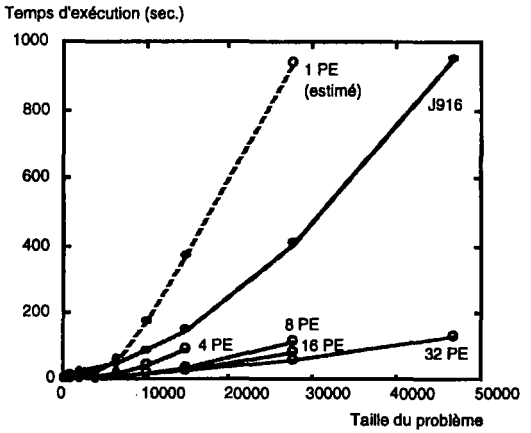


Figure 9. *Comparaison des temps d'exécution*

Remerciements

Les tests présentés ici ont été effectués sur le CRAY T3D du CEA de Grenoble et sur le Cray J916 de l'ENSAM à Paris.

4. Bibliographie

[AGO 87] AGOSHKOV V.I., « Poincaré-Steklov operators and domain decomposition methods », *Proceedings of the first international symposium on domain decomposition methods for partial differential equations*, Paris, 1987, Glowinski, Golub, Meurant, Periaux, eds, SIAM, Philadelphia (1988).

[AUN 90] AUNAY S., Architecture de logiciel de modélisation et traitements distribués. Thèse de l'Université de Technologie de Compiègne, 1990.

[BRA 89] BRAMBLE J.H., PASCIAK J.E., SCHATZ A.H., « The construction of preconditioners for elliptic problems by substructuring IV », *Mathematics of computation, vol. 53, num. 187*, 1989.

[BRA 93] BRÄUNL T., *Parallel programming*, Prentice-Hall International Editions, 1993.

[BRE 91] BREITKOPF P., KNOPF-LENOIR C., MORANCAY L., TOUZOT G., VAYSSADE M., *New Software Architecture: SIC*, Communication à STRUCOM, novembre 1991.

[COS 93] COSNARD M., TRYSTRAM D., *Algorithmes et architectures parallèles*, InterEditions, 1993.

[DON 91] DONGARRA J., DUFF I., SORENSEN D., VAN DER VORST H., « Solving linear systems on vector and shared memory computers », SIAM.

[ESC 96] ESCAIG Y., TOUZOT G., « Optimization od direct domain decomposition methods », *Computer Assisted Mechanics and Engineering Sciences, 3,* 1996.

[FAR 94] FARHAT C., ROUX F-X., « Implicit parallel processing in strctural mechanics », *Computational Mechanics Advances,* 1994.

[FOS 95] FOSTER I., *Designing and building parallel programs,* Addison-Wesley, 1995.

[GAL 94] GALLIVAN K. & al., « Parallel algorithms for matrix computations », SIAM, 1994.

[HOC 92] HOCKNEY R., JESSHOPE C., *Parallel Computers 2,* Institut of Physics Publishing, 1992.

[HWA 84] HWANG K., BRIGGS F., *Computer Architecture and Parallel Processing,* McGRAW-HILL , 1984.

[HWA 93] HWANG K., *Advanced Computer Architecture,* McGRAW-HILL, 1993.

[IRO 69] IRONS B.M., « A frontal solution program for finite element analysis », *Int. Jour. Num. Meth. Eng., Vol* 2, 1970, pp 5-32.

[KUM 94] KUMAR V., GRAMA A., GUPTA A., KARYPIS G., *Introduction to parallel computing,* The Benjamin/Cummings Publishing Company, 1994.

[LET 90] LE TALLEC P., DE ROECK Y-H., VIDRASCU M., « Domain decomposition methods in computational mechanics », *Computational Mechanics Advances, Vol 1,* 2 North-Holland, 121-220.

[LET 94] LE TALLEC P., Domain decomposition methods for large linearly elliptic three dimensional problems. Rapport de recherche de l'INRIA n°1182, 1990.

[LEW 92] LEWIS T., EL-REWINI H., *Introduction to parallel computing,* Prentice-Hall International Editions, 1992.

[LEW 93] LEWIS T., *Fundations of parallel programming,* IEEE Computer Society Press, 1993.

[LIO 87] LIONS P.L., *Poincaré-Steklov operators and domain decomposition methods,* Proceedings of the first international symposium on domain decomposition methods for partial differential equations, Paris, 1987, Glowinski, Golub, Meurant, Periaux, eds, SIAM, Philadelphia, 1988.

[ORT 89] ORTEGA J., *Introduction to parallel and vector solution of linear systems,* Plenum Press, 1989.

[QUA 90] QUARTERONI A., Domain decomposition method for the numerical solution of partial differential equations. University of Minesota Supercomputing Institute Research Report UMSI 90/246, 1990.

[ROU 89] ROUX F-X., Méthode de décomposition de domaine à l'aide de multiplicateurs de Lagrange et application à la résolution en parallèle des équations de l'élasticité linéaire. Thèse de l'Université de Paris 6, 1989.

[STO 93] STONE H., *High-Performance Computer Architecture,* Addison-Wesley, 1993.

[TOU 89] TOUZOT G., VAYSSADE M., Un système de modélisation interactive. Rapport final, Contrat MRT 86.E.0265.01., juin 1989.