ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAÎTRISE EN GÉNIE MÉCANIQUE
M.Ing.

PAR
BEN EL HAJ ALI AMIN

CALCUL DISTRIBUÉ POUR DES PROBLÈMES MULTIPHYSIQUES

MONTRÉAL, LE 19 AOÛT 2002

CE PROJET A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE :

M. Azzeddine Soulaïmani, directeur de mémoire
Département de génie mécanique à l'École de technologie supérieure

M. Tony Wong, président de jury
Département du génie de la production automatisée de l'École de technologie supérieure

M. Yousef Saad, professeur invité
Département d'informatique de l'University of Minnesota

IL A FAIT L'OBJET D'UNE PRÉSENTATION DEVANT JURY
LE 22 JUILLET 2002
À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

CALCUL DISTRIBUÉ POUR DES PROBLÈMES MULTIPHYSIQUES

Ben El Haj Ali Amin

SOMMAIRE

La simulation numérique des écoulements tridimensionnels compressibles est complexe et exige de grandes ressources de calcul. Le grand nombre d'équations à résoudre dans ce genre d'applications engendre l'épuisement des ressources des machines traditionnelles.

Le calcul parallèle a le potentiel de repousser les limites des systèmes existants, en mémoire et en temps de calcul.

L'apport du calcul parallèle devient particulièrement plus ressenti pour les problèmes multiphysiques. En effet, la résolution de ce type de problèmes nécessite un couplage entre les équations gouvernantes intrinsèques à chaque discipline, ce qui engendre un nombre encore plus élevé d'équations à résoudre.

Dans ce travail, un code parallèle de calcul en éléments finis **PFES** a été implémenté. Ce code a été validé pour la résolution d'un problème d'aéroélasticité sur l'aile **Agard 445.6**.

Une étude de performance de ce code a permis de mettre en évidence la valeur ajoutée du calcul parallèle pour la résolution de ce type de problèmes.

DISTRIBUTED COMPUTING FOR MULTI-PHYSICS PROBLEMS

Amin Ben El Haj Ali

ABSTRACT

Numerical simulation of compressible three-dimensional flows is complex and requires massive computational resources. Traditional machines run out of resources facing the huge number of equations to be solved. Parallel computing has the potential to push back the limits of the existing systems, in memory and computing times. Parallel computing is particularly more essential for the multi-physics problems. Indeed, the solution of these problems requires a coupling between the governing equations of each discipline. Computational and software implementation issues become more complex in coupled multi-physics problems. In this work, a parallel-distributed methodology is proposed and implanted in a finite element code called PFES (Parallel Finite Element System). As a case study, the numerical solution of aeroelastic problems is undertaken. The code is validated for the well-known benchmark test of the Agard 445.6 wing. A performance investigation highlights the added value of parallel computing.

REMERCIEMENTS

Je tiens à exprimer mes remerciements les plus vifs et les plus sincères :

- Au professeur Azzeddine Soulaïmani Ing., PhD. mon directeur de recherche. Professeur Soulaïmani a encadré ce travail ce qui m'a permis de profiter de son expérience, de sa réputation dans le domaine, de ses directives et de son soutien aussi bien scientifique que moral. Pour cela et pour bien d'autres choses je tiens à lui exprimer ma profonde gratitude.

- Au professeur Tony Wong PhD., professeur au département du génie de la production automatisée de l'École de Technologie Supérieure.

- Au professeur Yousef Saad PhD, professeur au département d'informatique de l'University of Minnesota.

- À tous les membres du groupe de recherche **Granit** qui m'ont soutenu durant ma maîtrise.

Je dédie ce travail à toute ma famille et à toute personne qui m'est chère.

TABLE DES MATIÈRES

Page

LISTE DES FIGURES

LISTE DES TABLEAUX

INTRODUCTION

La CFD (Computational Fluid Dynamics) est basée sur la résolution d'un ensemble d'équations aux dérivées partielles telles que les équations de Navier-Stokes pour le cas des écoulements visqueux, ou les équations d'Euler pour les écoulements non visqueux. La résolution de ces équations permet de décrire le comportement d'un fluide. Ces équations sont généralement discrétisées en utilisant la méthode de différences finies, des éléments finis ou la méthode de volumes finis.

La simulation numérique des écoulements tridimensionnels compressibles est complexe et exige de grandes ressources de calcul. La discrétisation des équations dans le cas de ces simulations, nécessite généralement un maillage très fin. Le choix du maillage dépend de la complexité du problème et du solveur numérique utilisé. Le grand nombre d'équations à résoudre dans ce genre d'applications engendre l'épuisement des ressources des machines traditionnelles en terme de capacité de mémoire. D'où le vif intérêt du calcul parallèle qui a le potentiel de repousser les limites des systèmes existants, en mémoire et en temps de calcul.

L'apport du calcul parallèle devient particulièrement plus ressenti pour les problèmes multiphysiques par exemple l'interaction fluide-structure. En effet, la résolution de ce type de problèmes nécessite un couplage entre les équations gouvernantes intrinsèques à chaque discipline, ce qui engendre un nombre encore plus élevé d'équations à résoudre.

Outre l'apport presque infini de mémoire, le développement rapide de la technologie des processeurs offre aux systèmes parallèles construits à partir d'ordinateurs personnels interconnectés, une puissance de calcul considérable.

Les objectifs de ce travail peuvent être classés comme suit :

1. L'implémentation d'un code de calcul parallèle, en éléments finis, capable de résoudre des problèmes multiphysiques.
2. Tester les performances et valider le code ainsi obtenu.
3. Résoudre un problème de stabilité aéroélastique et comparer les résultats avec les données expérimentales.

La démarche suivie est la suivante :

1. Une étape de familiarisation avec la version initiale du code PFES « PARALLEL FINITE ELEMENT SYSTEM ».
2. Le code est initialement écrit en langage FORTRAN77, sa conversion en FORTRAN90 lui permet de profiter de l'apport de l'allocation dynamique de mémoire.
3. Une première validation pour un problème classique de simulation d'écoulement sur une aile d'avion est nécessaire.
4. Améliorer le code PFES afin de résoudre des problème multiphysiques.
5. Tester les performances
6. Étude aéroélastique de l'aile Agard et comparaison avec les résultats expérimentaux pour fin de validation

CHAPITRE 1

PARALLÉLISME ET ALLOCATION DYNAMIQUE

1.1 Généralités

Dans la littérature, les termes calcul parallèle et calcul distribué sont généralement évoqués dans différents contextes. Les définitions suivantes sont adoptées par C. Leopold [15].

- En calcul parallèle une application est partagée en plusieurs taches qui seront exécuter en même temps.
- En calcul distribué l'application est partagée en plusieurs taches qui seront accomplies en utilisant plusieurs ressources.

Dans le cadre de ce travail cette nuance n'est pas prise en considération. En effet, les aspects rencontrés :

- Utilisation de plusieurs processeurs.
- Processeurs interconnectés
- Plusieurs activités progressent en même temps et communiquent entre elles pour résoudre un problème global.

Sont des caractéristiques communes.

Ce chapitre survole les objectifs principaux, les problèmes majeurs et les notions de base du calcul parallèle ainsi que l'importance de l'allocation dynamique de mémoire.

1.1.1 Importance du parallélisme

Pour comprendre la raison pour laquelle le calcul parallèle attire de plus en plus les scientifiques, il est intéressant de répondre à la question suivante :
Pourquoi plus d'un processeur ?

Utiliser plusieurs processeurs pour lancer un traitement de texte sonne comme "trop inutile". Dans le cas d'un serveur web, une base de données, un programme de ray-tracing ou un planificateur de projets, l'utilisation de plusieurs processeurs peut améliorer les performances. Mais pour des simulations plus complexes de la dynamique des fluides, des processeurs supplémentaires sont absolument nécessaires.

L'affirmation suivante est souvent évoquée :
"Pourquoi ai-je besoin de deux ou quatre processeurs ? Je n'ai qu'à attendre le méga super rapide processeur de la génération prochaine".
La vitesse des processeurs double tous les 18 mois, alors que celle de la mémoire n'augmente pas aussi rapidement. Les applications reliées au domaine de la CFD ont besoin de plus en plus de mémoire et de vitesse d'accès aux données.

Les prédictions indiquent que la vitesse des processeurs ne continuera pas à doubler tous les 18 mois après l'an 2005. Il y a divers obstacles à surmonter pour maintenir ce rythme. Suivant l'application, la programmation parallèle peut accélérer les signficativement choses. De telles performances ne sont pas disponibles sur un seul processeur. Considérer l'approche parallèle est une solution intelligente et durable pour contourner ces limitations. Comme le calcul parallèle est implémenté selon de nombreuses voies, résoudre un problème en parallèle nécessitera de prendre quelques décisions importantes. Ces décisions peuvent affecter dramatiquement la portabilité, la performance, et le coût de l'application.

Pour atteindre les objectifs recherchés, particulièrement, l'étude aéroélastique d'une aile d'avion, le développement des outils à la hauteur du défi est primordial.
Pour ce faire une machine à base de groupe de Pc (Beowulf) a été conçue et réalisée par le groupe **Granit** pour répondre à nos besoins (Annexe, Configuration de Granit). Avec 22 nœuds de calcul et un serveur très puissant, le système Beowulf est une machine à

haute disponibilité et grande puissance de calcul. Cette machine concurrence en termes de coûts/performances les serveurs traditionnels de calcul appelés superordinateurs.

Toutefois, l'acquisition d'une machine parallèle dépourvue d'un code parallèle, est simplement inutile. Au niveau du logiciel, un code de calcul parallèle appelé **PFES** « PARALLEL FINITE ELEMENT SYSTEM » [6,7] à été conçu, implémenté et validé.

Pour mettre en relief les différents aspects du projet la compréhension de quelques notions de base est extrêmement utile.

1.1.2 Notions de base du calcul parallèle

Le but de cette partie, étant d'introduire des notions et des concepts qui nous seront indispensables dans la suite.
Dans tout ce qui suivra, nous utiliserons le terme tâches (*tasks*) et le terme processus (*process*), indifféremment, pour désigner des activités qui s'exécutent en même temps et/ou par des différents processeurs.
Quelques notions de bases [15] sont présentées ci-après.

Temps de calcul/temps réel

Une notion évidente qui intéresse l'utilisateur, est le temps de calcul réel qui est simplement le temps nécessaire, en secondes, à l'achèvement de la tâche. Toutefois, le temps de calcul est le temps indispensable au calcul pur sans prendre en considération les communications et entrées/sorties. Dans la littérature, on adopte, généralement la notion de temps de calcul (temps CPU ou running time). Cependant, nous considérons plus pertinents de mesurer le temps de calcul réel qui reflète plus concrètement le coût d'exécution.

Speed up :

Une des notions les plus fréquemment rencontrée dans le domaine du calcul parallèle, est le *speed up* qui est le rapport entre le temps d'exécution séquentiel et le temps d'exécution parallèle.

$$speedup\ (N) = \frac{T_s}{T_p(N)} \qquad (1,1)$$

Où **N** est le nombre de processeurs, $\mathbf{T_s}$ le temps d'exécution séquentiel et $\mathbf{T_P(N)}$ est le temps d'exécution parallèle avec les **N** processeurs.

Idéalement, $\mathbf{T_s}$ est le coût d'exécution minimal offert par un programme séquentiel pour résoudre le même problème.

Le *speed up* reflète, par conséquent, le gain qui découle de la parallélisation et montre sa valeur ajoutée globale par rapport au calcul séquentiel.

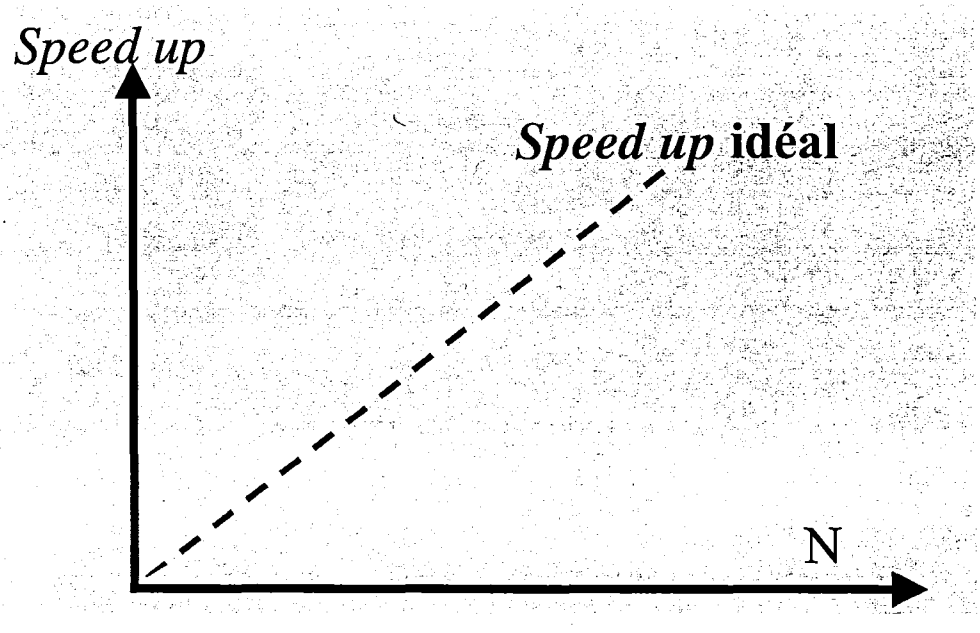


Figure 1 *Speed up* en fonction du nombre des processeurs

Efficacité (effeciency) :

Une autre alternative de mesure, est l'efficacité. Elle est définie par le rapport du *speed up* et du nombre de processeurs :

$$efficacité\ (N) = \frac{speedup(N)}{N} \qquad (1,2)$$

N étant le nombre de processeurs.
Cette mesure reflète la performance de la machine parallèle. Dans le cas idéal, l'efficacité est égale à 1 pour tous les processeurs. Dans des cas extrêmes, on peut même dépasser cette linéarité, pour atteindre la super linéarité. Dans ce dernier cas, les **N** processeurs peuvent résoudre le problème en un temps inférieur à $1/\mathbf{N}^{ième}$ du temps nécessaire en séquentiel.

Complexité de la conception :

La conception et l'optimisation des performances d'un programme parallèle sont plus complexes que celles d'un programme séquentiel. On peut comparer la parallélisation d'un code ou d'un programme à la gestion d'un groupe de travail : répartition des tâches, assignation des travaux, organisation des communications, etc...
Dans la majorité des environnements, le programmeur n'intervient que dans quelques tâches. Le restant est laissé au compilateur, qui trouve des solutions plus ou moins satisfaisantes. Cette répartition dépend de la variété des modèles de programmation.

L'application doit être fractionnée en plusieurs tâches, en général il est conseillé de partager les taches d'une façon équilibrée. Une assignation non équilibrée engendre des temps d'inaction (deadlocks) et donc des pertes de performance de la machine parallèle. Ce partage doit prendre en considération les performances individuelles de chaque processeur.

D'autre part il est bien important d'optimiser la parallélisation en terme de nombre de processeurs. En effet en augmentant le nombre de processeurs, le coût de la communication risque d'augmenter et peut affecter l'efficacité de la machine. L'efficacité est une mesure de cette optimisation [15,20].

Un autre aspect de la conception des programmes parallèles est la gestion des communications ainsi que la synchronisation des processeurs, en un seul mot la coordination. Grâce à une bonne coordination, les différents processeurs coopèrent et échangent les informations nécessaires à la résolution du problème global.

Les aspects présentés dans ce paragraphe ainsi que le choix du model de programmation, l'environnement et l'architecture doivent être optimisés.

La transparence

La transparence est le fait de cacher certains aspects aux utilisateurs (ou programmeurs). Les fonctionnalités invisibles sont alors traitées automatiquement par le système sans intervention du facteur humain.

La portabilité

Un programme est dit portable s'il est capable de s'exécuter sur une variété de machines.

Extensibilité

Plusieurs définitions de l'extensibilité (scalability) sont utilisées. En général, un système (dans notre cas une machine parallèle) est dit extensible, si ses ressources peuvent être augmentées pour résoudre des problèmes plus imposants ou pour augmenter ses performances. Cette amélioration peut être réalisée par l'augmentation du nombre de processeurs ou par le remplacement des processeurs par d'autres plus performants.

1.2 Standard de communication par échange de messages MPI

MPI (Message Passing Interface) [22,32] est une bibliothèque d'échanges de messages pour machines parallèles. Une application **MPI** est un code s'exécutant sur un ensemble de processus exécutant en même temps et communiquant via des appels à des sous-programmes de la bibliothèque **MPI**.

Dans un modèle de parallélisme par échanges de messages, chaque processus exécute éventuellement des parties différentes d'un programme (écrit dans un langage classique comme le Fortran ou le C). Comme ce modèle de programmation parallèle est explicite, la gestion des communications et des synchronisations est donc entièrement à la charge du développeur.

Les variables sont alors privées et résident dans la mémoire locale de chaque processus. Une donnée ne peut être échangée entre deux ou plusieurs processus qu'au travers d'appels à des sous-programmes spécialisés constituant une bibliothèque de routines d'échanges de messages. C'est une encapsulation des données.

MPI (Message Passing Interface) est une bibliothèque standard, dont les applications sont portables sur les machines parallèles actuelles. Un message est constitué de paquets de données transitant du processus émetteur au(x) processus récepteur(s). Il comporte les informations suivantes :

- Les données.
- Le type des données.
- Sa longueur.
- L'identificateur du processus émetteur.
- L'identificateur du processus récepteur.

La bibliothèque **MPI** interprète et gère ces messages. Les fonctions les plus utilisées de **MPI** sont expliquées à l'annexe 2.

1.3 Allocation dynamique

Le logiciel **PFES** est un logiciel de calcul parallèle par éléments finis écrit principalement en Fortran. Ce logiciel est capable de résoudre une grande gamme de problèmes physiques et peut même traiter des problèmes multiphysiques. Parmi les études envisagées, la simulation des écoulements autour des ailes d'avions ainsi que l'étude de la stabilité aéroélastique. Pour atteindre ces objectifs, il est important de surmonter les limites des machines en terme de mémoire et temps de calcul.

Un apport intéressant de la norme **Fortran 90,** est la possibilité d'allouer dynamiquement la mémoire. Pour pouvoir allouer un tableau dynamiquement, on spécifiera l'attribut **ALLOCATABLE** ou **POINTER** au moment de sa déclaration. Un tel tableau s'appelle tableau à « profil différé » (*deffered-shape-array*). Son allocation s'effectuera grâce à l'instruction **ALLOCATE** à laquelle on indiquera le profil désiré.
L'instruction **DEALLOCATE** permet de libérer l'espace mémoire alloué.
De plus, les fonctions intrinsèques, **ALLOCATED** et **ASSOCIATED** permettent d'interroger le système pour savoir si respectivement un tableau ou un pointeur, est alloué ou non.

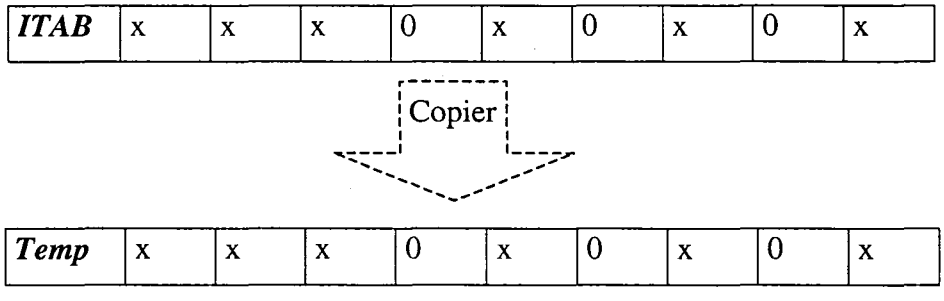
Malheureusement, la norme **Fortran 90** n'a pas intégré la possibilité de faire la ré-allocation dynamique. Puisque les dimensions des tables ne peuvent pas êtres connus d'avance, la re-allocation dynamique est un outil extrêmement utile à **PFES**. Les dimensions des vecteurs sont alors approximées puis si nécessaire corrigés. Pour remédier à cet inconvénient, des routines de ré-allocation ont été programmées.
Ces routines sont : ***Iallocinit*** pour les entiers, ***Sallocinit*** pour les tables simple précision, ***Vallocinit*** pour les tables double précision ainsi que ***Iallocinit2***, ***sallocinit2*** et ***vallocinit2*** pour les tables à deux dimensions.

Dans cette partie les tâches accomplies par ***Iallocinit*** seront détaillées, les autres routines sont similaires.

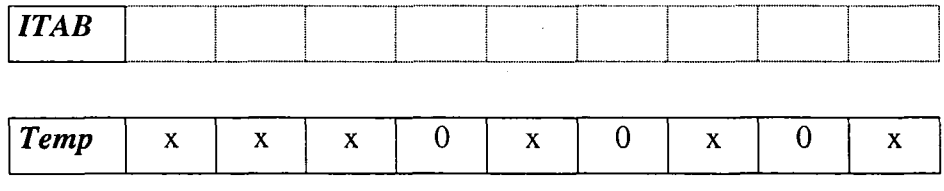
1. Allouer dynamiquement une table ***ITAB*** de type « integer » avec une dimension appelée ***dim.***
2. La gestion des erreurs dues à l'insuffisance de mémoire.
3. Initialiser la table à 0.
4. Si la table est déjà allouée, ré-allouer la table avec la nouvelle dimension. Cette fonction est possible à réaliser par deux manières.
   - Donner la nouvelle dimension ***dim.***
   - Donner un facteur de multiplication ***r.***

Dans les deux cas, une nouvelle dimension **dim2** est calculée. Le principe de la re-allocation est le suivant :

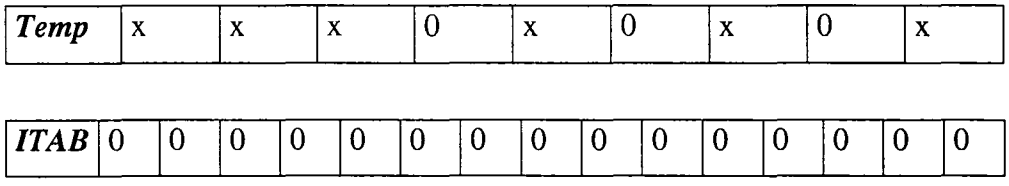
- Stocker les valeurs la table ***ITAB*** dans ***temp***



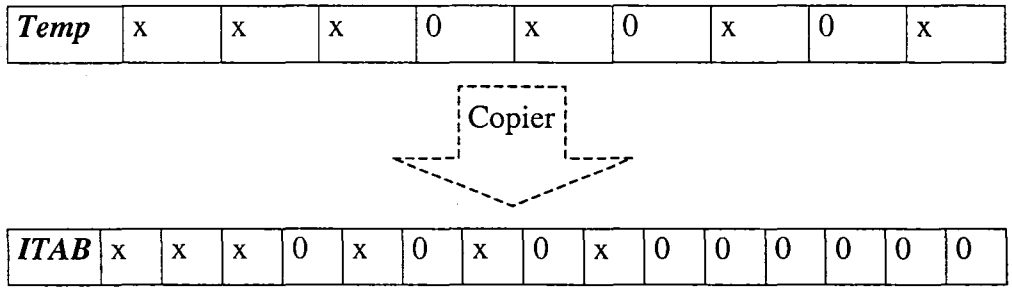
- Libérer la table ***ITAB***



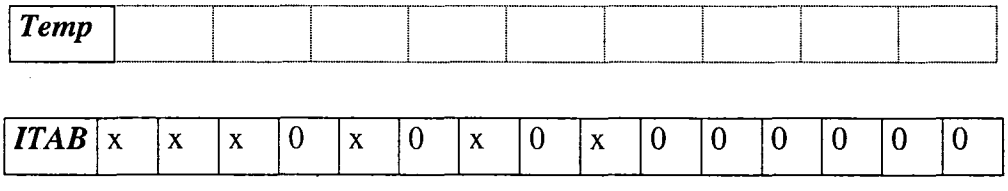
- Allouer et initialiser une nouvelle table ***ITAB*** avec la nouvelle dimension ***dim2***



- Copier le contenu de ***temp*** dans ***ITAB***



- Libérer ***temp***



5. Écrire le journal de l'allocation dans le fichier « ***.log*** ».

CHAPITRE 2

PROBLÉMATIQUE

2.1 Problèmes multiphysiques

Avec le développement et l'avancement des domaines de recherche en calcul scientifique, des problèmes complexes attirent l'attention. De tels problèmes présentent des difficultés, aussi bien aux niveaux analytique que numérique.
Une gamme importante de ces problèmes, incorpore l'aspect multiphysique.
Un problème est dit multiphysique, s'il met en jeu deux ou plusieurs disciplines physiques où plusieurs phénomènes qui interagissent. On peut citer, à titre d'exemple, l'interaction fluide-structure.

Les problèmes multiphysiques présentent des difficultés majeures. D'une part, leur formulation analytique nécessite la collaboration et l'interaction de plusieurs spécialités. La connaissance pertinente de chaque discipline est indispensable pour bien définir, formuler et résoudre le problème global. D'autre part, la formulation numérique de ce type de problème aboutit généralement, à des systèmes algébriques gigantesques. Aussi, les matrices générées sont souvent très mal conditionnées et par conséquent extrêmement difficiles à résoudre.

La résolution directe des problèmes multiphysiques est principalement limitée par le côté informatique. En effet, l'incapacité et l'insuffisance des systèmes informatiques actuels représentent un handicap majeur aussi bien au niveau du stockage qu'au niveau de la puissance de calcul.

Pour pallier ces difficultés mathématiques, numériques et informatiques, intuitivement, une subdivision du problème global en sous-problèmes, pourrait s'imposer. Toutefois,

la qualité et le coût de la solution finale, restent toujours dépendantes du choix de la stratégie de décomposition et de l'algorithme de résolution.
Quelques méthodologies et stratégies de résolution des problèmes multiphysiques seront abordées.

Avant d'entamer les détails de décompositions des problèmes multiphysiques regardons d'une façon générale les différentes stratégies du parallélisme

2.2 Différentes stratégies du parallélisme

Cette section est consacrée à la clarification des différents types de parallélisme [15,20].

2.2.1 Parallélisme de données

Le parallélisme de données utilise le fait que le calcul scientifique travaille sur des données structurées (vecteurs, matrices...) et qu'il est donc possible de faire correspondre à la structure régulière des données du programme une structure régulière de processeurs élémentaires de la machine parallèle.

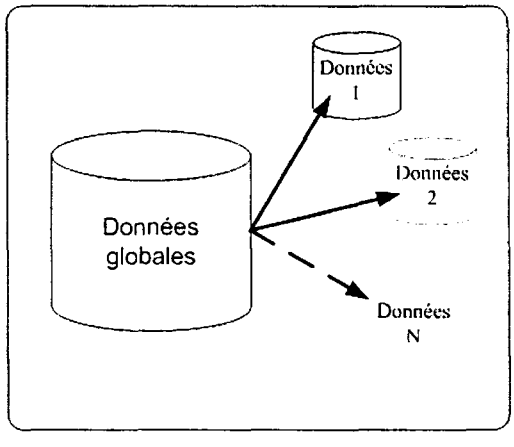


Figure 2 Partitionnement des données

Cette présentation très schématique permet de comprendre facilement les problèmes qui surgissent. La puissance de communication étant très inférieure à la puissance de calcul. Les performances obtenues dépendent très fortement du placement des données dans les mémoires des processeurs, et du nombre, de la fréquence et de la nature des transferts de données. Une caractéristique fondamentale de ce type de parallélisme est son côté fortement synchrone car le traitement se fait généralement simultanément sur les éléments individuels de données structurées.

Le terme SPMD « single program, multiple data » est généralement utilisé pour désigner cette stratégie de parallélisme.

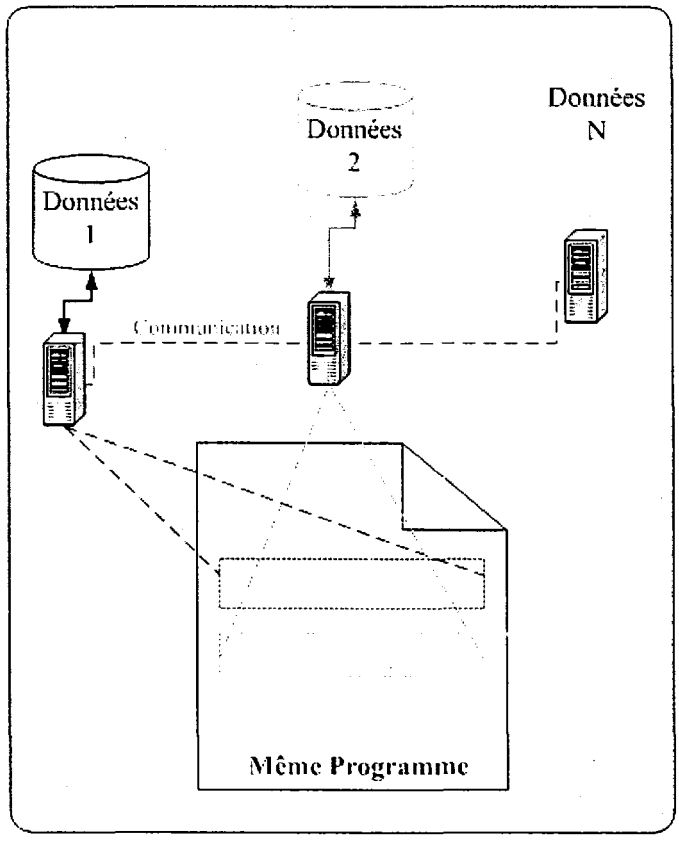


Figure 3 SPMD : Single Program Multiple Data

2.2.2 Parallélisme de processus

Une caractéristique fondamentale est le côté fortement asynchrone de ce type de parallélisme, dès que s'exécutent simultanément des processus différents. La spécification et la validation des programmes concurrents sont très différentes de celle des programmes séquentiels. Là encore, la distinction est faite pour clarifier les problèmes. Les applications réelles du calcul numérique peuvent enchaîner plusieurs étapes, successives ou concurrentes, de traitements organisés sous formes de processus, chacun des traitements mettant en oeuvre le parallélisme de données.

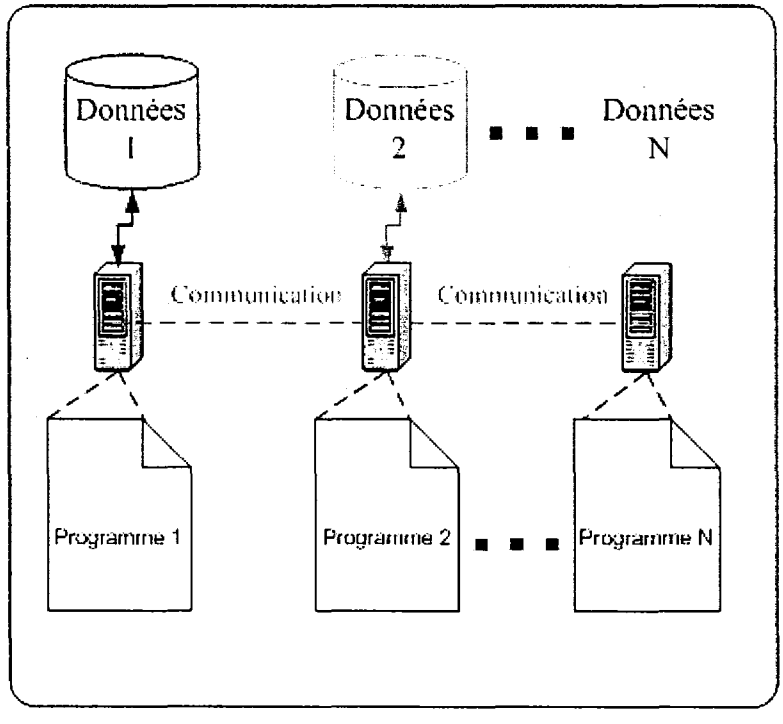


Figure 4 MPMD : **M**ultiple **P**rogram **M**ultiple **D**ata

Contrairement à la SPMD dans cette approche chaque processeur exécute un programme différent.

CHAPITRE 3

PROBLÈMES MULTIPHYSIQUES : ÉTUDE DE CAS

3.1 Stabilité aéroélastique des ailes d'avions

Le couplage entre l'écoulement et les mouvements de structures flexibles peut conduire à des instabilités qui mettent en jeu l'intégrité de la structure.

Le domaine de recherche est orienté vers le couplage aéroélastique, c'est à dire vers l'étude du comportement des structures dont les mouvements au sein de l'écoulement génèrent des efforts induits. Les domaines concernés sont notamment le génie civil (ponts souples, cheminées), les transports (aéronautique, terrestre) et le secteur de l'énergie (tubes d'échangeurs, éoliennes).

Le flottement (flutter) est un phénomène qui se traduit par une vibration importante de la structure induite par un écoulement décollé fortement instationnaire, par exemple:

- aile d'avion volant à grande incidence
- zone décollée dans les culots de lanceurs

Ces vibrations peuvent être la cause de fatigue importante ou même remettre en cause le bon fonctionnement de l'engin (vibration de tuyères). Les études menées ont pour but d'étudier les conditions d'apparition du flottement.

Il est indispensable de s'assurer de l'absence d'instabilité, surtout, dans le domaine de l'aéronautique, d'où l'importance de l'étude de la stabilité aéroélastique des ailes d'avions

3.2 Stratégie de résolution

Différentes stratégies de résolution parallèle des problèmes multiphysiques et en particulier le problème de flottement des ailes d'avions, ont été développées [3,4,5,6,7,11,13,14]. Ces stratégies peuvent être globalement classées selon les deux types de parallélisme précédemment expliquées.

Le problème de stabilité aéroélastique des ailes d'avions est un problème qui met en jeu deux domaines physiques. Un domaine structure constitué par l'aile et un domaine fluide qui l'entoure. Au cour du mouvement, l'interaction entre ces deux domaines peut conduire à des phénomènes qui déstabilisent la structure.
Le but des simulations envisagées est de détecter ces instabilités.
L'algorithme séquentiel suivant illustre la communication entre les domaines fluide et structure :

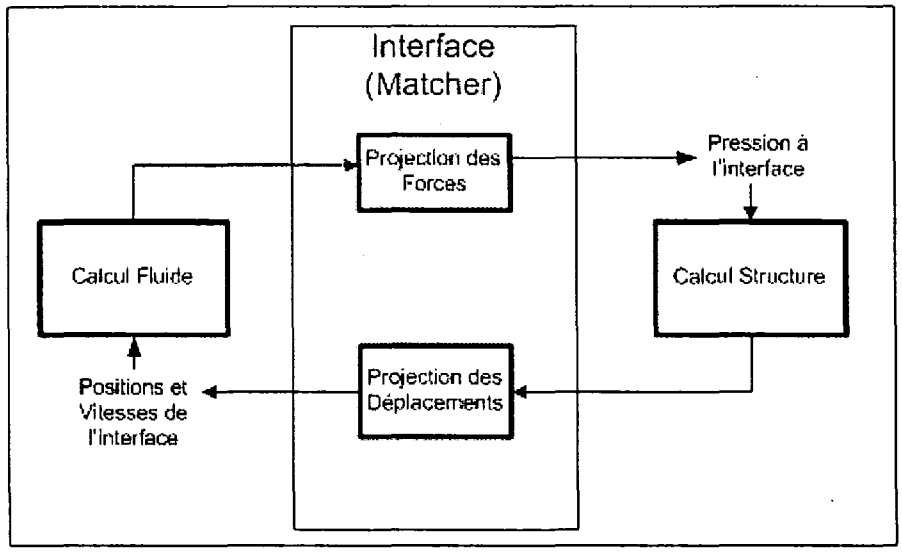


Figure 5 Algorithme séquentiel : interaction fluide-structure

La simulation du problème de flottement, nécessite l'implantation d'une interface entre le domaine fluide et le domaine structure. Cette interface, appelée **matcher**, joue le rôle d'un traducteur ou d'un convertisseur de l'information entre les deux solveurs.

L'algorithme séquentiel est basé sur une alternance entre le solveur CFD et le solveur CSD, intercalé par un appel du « matcher ». Le matcher assure deux fonctions :

- Projeter les pressions calculées par le fluide sur les noeuds de la structure
- Mettre à jour les conditions aux limites du fluide en se basant sur les déplacement de la structure.

Une façon de paralléliser cet algorithme est de décomposer le problème global en plusieurs sous-domaines. Les données sont alors partagées entre les processeurs, le plus équitablement possible.

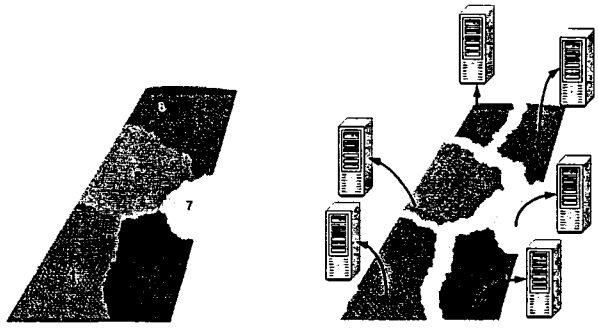


Figure 6 Décomposition du domaine

A part l'ajout de quelques routines de communication, l'algorithme reste inchangeable. Dans ce cas, tous les processeurs passent par les mêmes routines. Cette méthode peut être classée comme étant une approche SPMD.

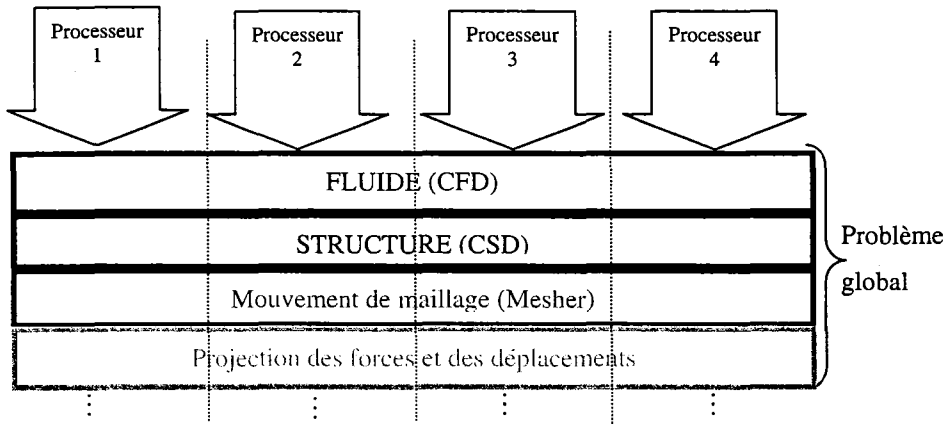


Figure 7 Approche SPMD pour la résolution d'un problème multiphysique

Une deuxième manière de paralléliser l'algorithme séquentiel (Figure 12) est de diviser le problème en deux domaines fonctionnels, un domaine pour le fluide et un autre pour la structure. Ensuite, chaque domaine est divisé en plusieurs sous-domaines. Cette approche est adoptée par le programme PFES.

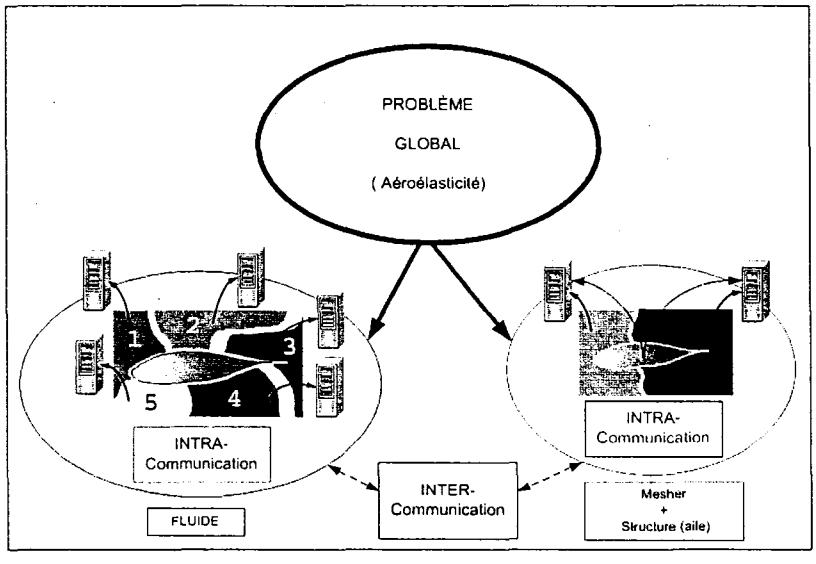


Figure 8 Décomposition fonctionnelle

Cette méthode englobe les deux approche **SPMD** et **MPMD**. La communication entre les processeurs de la même famille est appelée intra-communication. La communication entre les familles est appelée inter-communication.

Plusieurs algorithmes des résolutions peuvent être considérés pour résoudre ce problème ainsi divisé [6,7].

Les communications sont établies en faisant appel à la bibliothèque MPI. Pour chaque type de communication, un communicateur dédié est appelé.
Le programme PFES fait appel à quatre communicateur :
Le communicateur ***MPI_COMM_WORLD*** est le communicateur par défaut de **MPI**. Il permet de communiquer des informations entre tous les processeurs.
Le communicateur ***Family*** est un sous-communicateur de ***MPI_COMM_WORLD.*** ***Family*** permet la communication entre les processeurs de la même famille pour réaliser l'intra-communication.

Le communicateur ***Diplomacy*** permet la communication entre les leaders des familles et donc assure l'inter-communication.
Dans les routines de PSPARSLIB le programme PFES a besoin de passer des informations entre les voisins. Les voisins sont les sous-domaines qui ont une interface commune ou simplement des nœuds en commun. Tous les voisins appartiennent nécessairement au même domaine physique.
Pour éviter les communications point-à-point, un sous communicateur de ***Family*** appelé ***Neighbors*** a été construit.

Contrairement aux autres communicateurs, le communicateur ***Neighbors*** n'est pas un communicateur classique. Ce dernier peut être vu comme une table de communicateurs. Chaque élément de cette table, est utilisé par un processeur bien déterminé pour envoyer des informations à ses voisins. La complexité de l'implémentation de ***Neighbors*** vient

du fait que les différents groupes qui constituent les voisins ne sont pas indépendants. Cela veut dire qu'un processeur peut appartenir à plusieurs groupes en même temps. De ce fait, il est impossible de construire des communicateurs dédiés à chaque groupe.

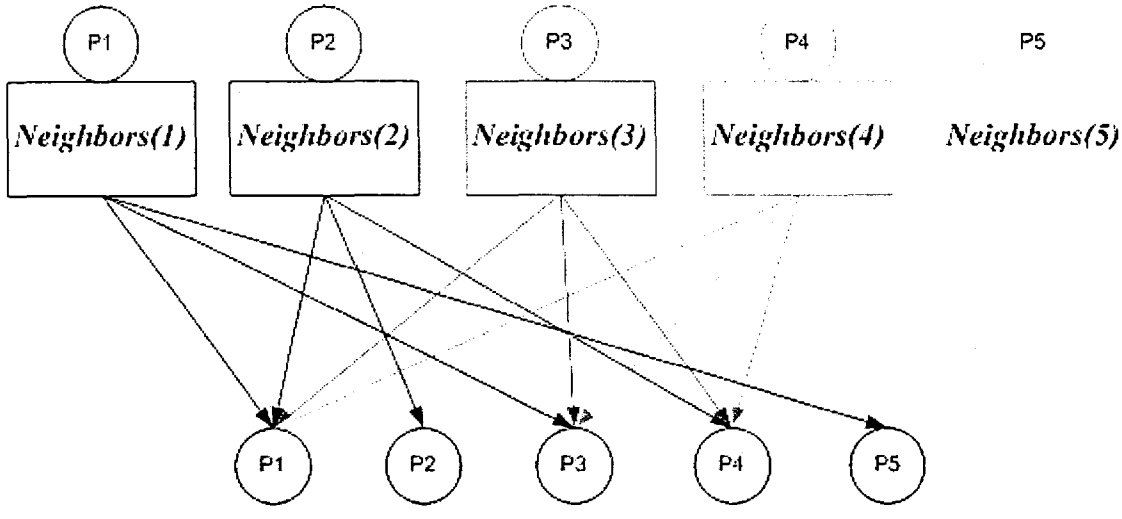


Figure 9 Table des communicateurs inter-voisins ***Neighbors***

Les communicateurs ***Neighbors*** ne peuvent être utilisés correctement qu'avec les routines d'envoi d'informations.

L'algorithme suivant donne un aperçu sur le programme PFES :

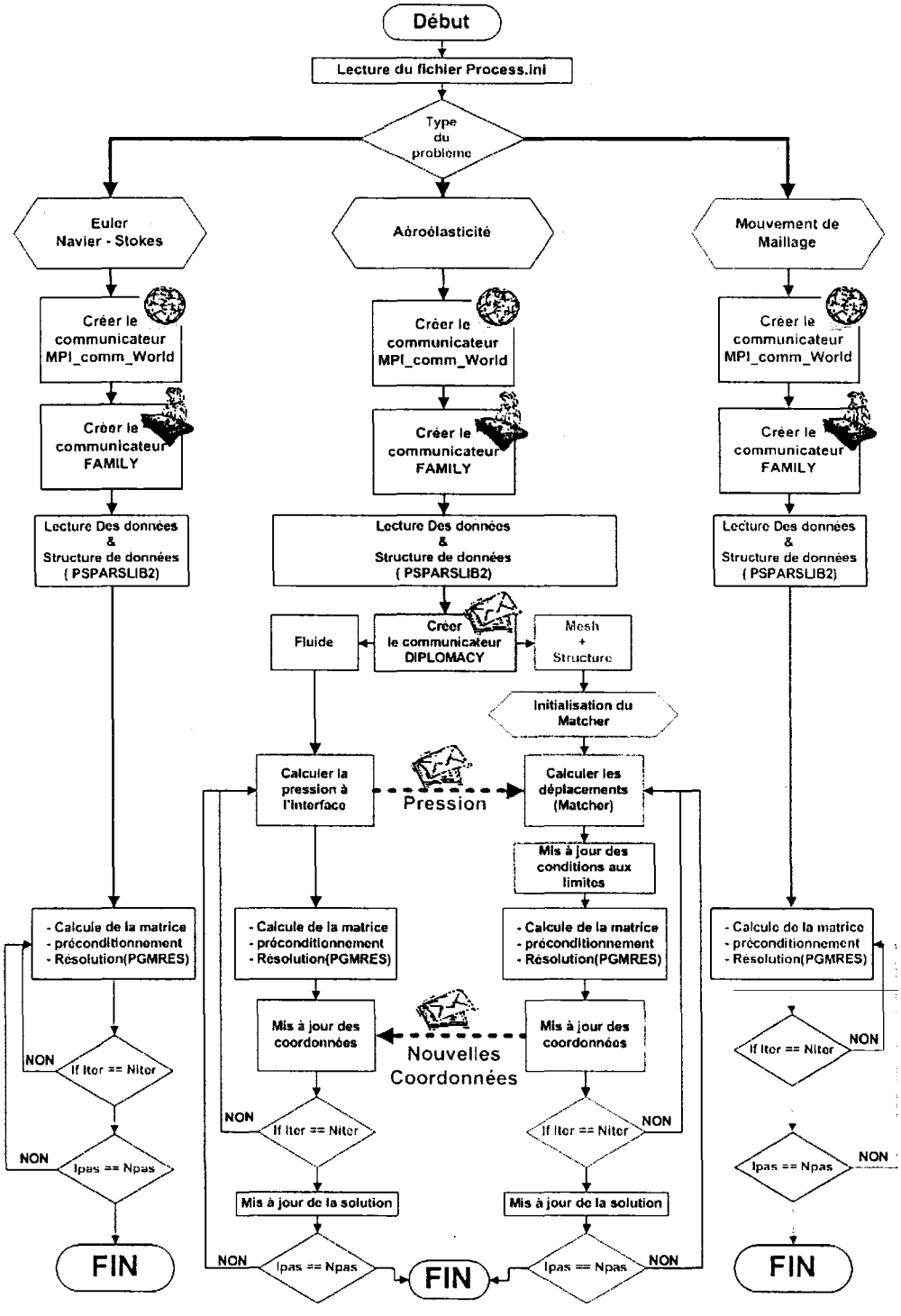


Figure 10 Algorithme général PFES

L'algorithme suivant illustre la communication entre le fluide et la structure :

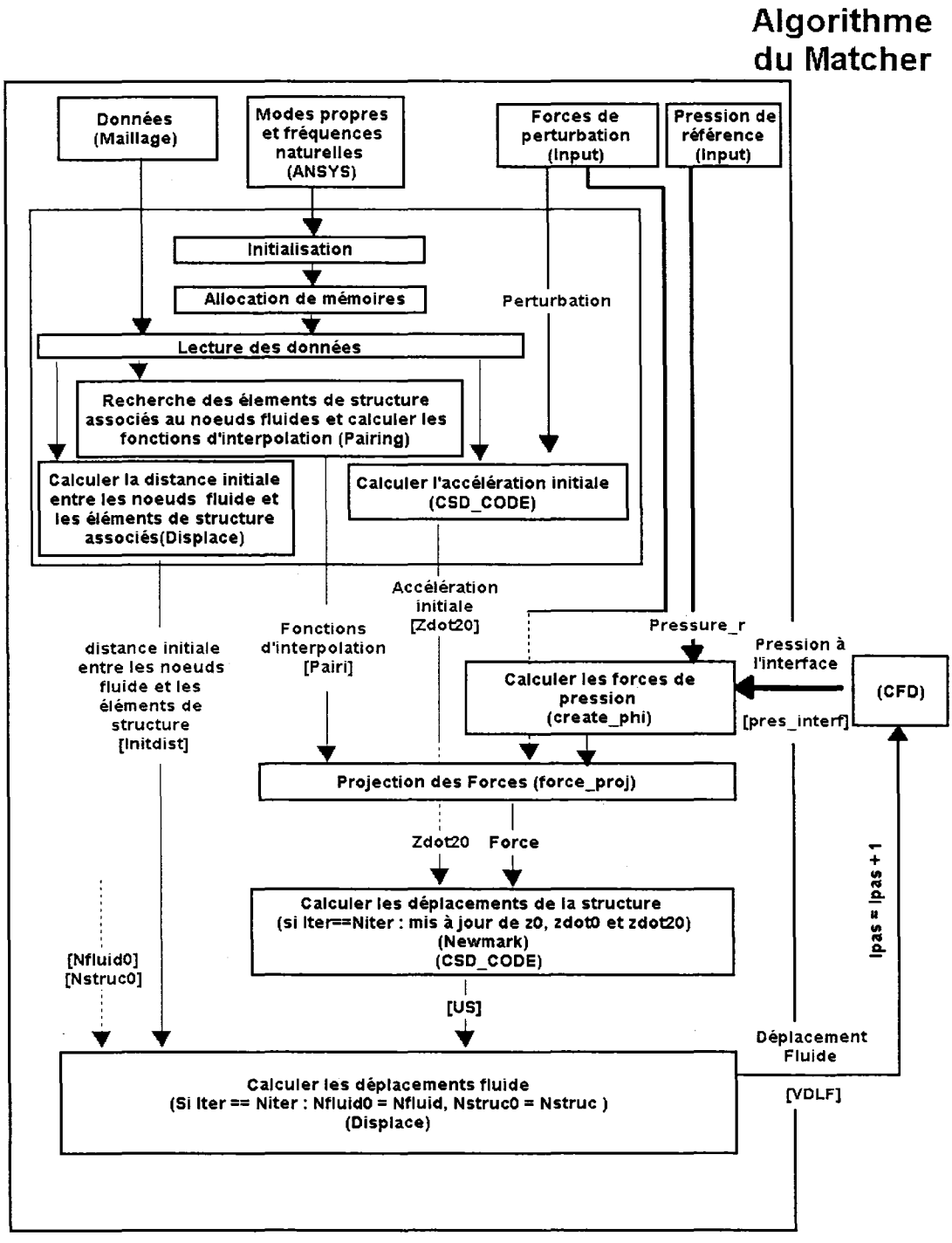


Figure 11 Interaction Fluide/Structure

CHAPITRE 4

STRUCTURE DE DONNEES

Une façon assez simple qui permet de résoudre les systèmes découlant de la discrétisation des équations aux dérivées partielles, est l'approche de décomposition du domaine.

Le domaine est partagé en plusieurs sous-domaines, la solution globale du système est récupérée par l'assemblage des solutions locales indépendantes de chaque sous-domaine.Chaque processeur prend en charge un ou plusieurs sous-domaines. Les solutions partielles sont par la suite combinées, après quelques itérations, pour former une approximation de la solution du système global [5].

La méthode de décomposition est basée sur l'algorithme « additive-Schwarz ». L'algorithme de résolution est basé sur une combinaison des procédures Time-Marching, Inexact-Newton et l'algorithme **GMRES**. (voir section 5.1.2). Le **MPI** (voir section 1.2) est utilisé pour la communication et la bibliothèque **PSPARSLIB** pour la structure de données.

4.1 La bibliothèque PSPARSLIB

Pour implémenter un code basé sur l'approche de décomposition du domaine on a besoin d'outils pour décomposer le domaine, associer chaque sous-domaine à un processeur, préparer les structures de données et enfin d'un algorithme de résolution. PSPARSLIB [3,28,30,31] est une bibliothèque bien adaptée pour effectuer ces taches. La première étape consiste à diviser le domaine en plusieurs sou-domaines avec un outil comme METIS [18].

La bibliothèque PSPARSLIB réalisée par le groupe du professeur Y. Saad au Minnesota est un bel outil pour le calcul distribué spécialement adapté aux solveurs itératifs des matrices creuses. Cet outil est une implémentation parallèle du solveur itératif SPARSKIT [30].

4.1.1 SPARSKIT

SPARSKIT [30] est un outil de base pour le calcul séquentiel avec les matrices creuses (sparse matrices) présenté en langage Fortran77. Cette bibliothèque est le fruit de plusieurs années de travail. La multitude d'interventions et améliorations qui lui ont été apportées tout au long de ces années, l'ont rendue une bibliothèque complète regroupant des outils pertinents pour le développement et l'implémentation des techniques de résolutions des matrices creuses et particulièrement pour les solveurs itératifs.

SPARSKIT offre aussi des accélérateurs et des préconditionneurs qui sont des outils incontournables pour les problèmes de grandes tailles.

4.1.2 Implémentation de PSPARSLIB

La bibliothèque PSPARSLIB [3,28,30,31] peut être considérée comme une parallélisation de la bibliothèque SPARSKIT. Elle offre les outils essentiels pour la résolution des systèmes découlant de la discrétisation des équations aux dérivées partielles, par décomposition du domaine.
Les accélérateurs locaux sont identiques à ceux qui sont fournis avec SPARSKIT. Cette bibliothèque inclut des procédures de préconditionnement global basées sur les algorithmes «Additive- Schwarz», « Multiplicative- Schwarz » et « Schur complement ».

Durant le processus de résolution, chaque processeur (sous-domaine) doit échanger les valeurs calculées (ou imposées) aux nœuds de l'interface avec ses voisins. Pour permettre un tel échange d'une façon efficace, il est important de déterminer la liste des nœuds couplés avec les autres sous-domaines avoisinants.

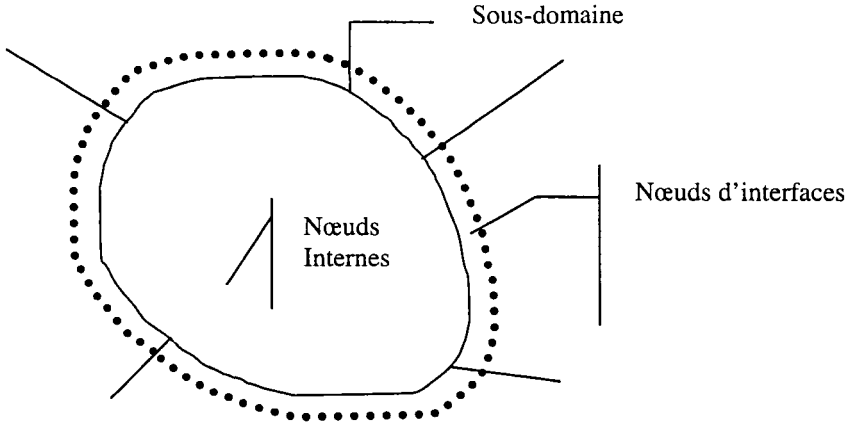


Figure 12 Nœud d'interface entre les sous-domaines

Dans une première phase, le programme procède à l'initialisation des processeurs et à la lecture des fichiers de données. Les données lues sont alors stockées dans des tables allouées dynamiquement au sein du programme. Dans cette partie, seulement l'implémentation de PSPARSLIB sera abordée.

Il est très important durant tout le processus de résolution, de connaître les nœuds d'interface entre les domaines. La routine **nod2dom** permet de construire la table d'appartenance (***map***). Ainsi pour chaque équation i :

Si $map(i)>0$ alors $map(i)$ est le numéro du processeur auquel cette équation appartient.

Si $map(i)<0$ alors $map(i)$ est un pointeur vers la liste des processeurs aux quels cette équation appartient.

PSPARSLIB présente une manière intelligente pour stocker les données. En effet, il est inutile de stocker les valeurs nulles. Pour cela la matrice globale est remplacée par trois vecteurs appelés ***VKGS***, ***Ia*** et ***ja***.

Les figure 19, 20 et 21 donnent une idée plus claire sur ces vecteurs.

VKGS est la matrice sparse stockée en vecteur des coefficients non nuls. Le nombre des coefficients non nuls est ***nnz***.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**...**	**m**
1					0				0
2	0					0			
3	0								0
4	0	0							0
5			0						
6	0					0	0		0
7	0	0	0						0
...									
n					0				0

Figure 13 Matrice globale

Le vecteur ***Ia***(i) fournit le nombre des valeurs non nulles dans la ligne i.

i =	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**...**	**n**
***Ia*(i)**	1	7	13	19	..				

Figure 14 ***Ia*** Vecteur des nombres cumulatifs des valeurs non nulles

JA est un vecteur qui contient les numéros des colonnes dans la matrice globale.

k=	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
***Ja*(k)**	1	2	3	4	6	7	2	3	4	5	7	m	2	3	4	5	6	7	..

Figure 15 ***Ja*** Vecteur des numéros de colonnes des valeurs non nulles

Ja : les numéros de colonne des valeurs non nulles dans la matrice globale.

Pour une ligne i de la matrice globale les numéros des colonnes non nulles sont :
Ja(***Ia***(i) + j) avec j variant de 0 à ***Ia***(*i* + *1*) – ***Ia***(*i*) -1

La table ***map*** ainsi que ***Ja*** et ***Ia*** sont envoyées ensuite à la routine ***Getjamap.*** Cette dernière construit un nouveau vecteur appelé ***mamap***, similaire à ***map***. Ce vecteur indique à quel domaine appartient chaque élément de ***Ja***.
Si $mamap(i) > 0$ alors $mamap(i)$ est le numéro du processeur au quel cet élément appartient.
Si $mamap(i) < 0$ alors $mamap(i)$ est un pointeur vers la liste des processeurs (ou sous-domaines) aux quels cet élément appartient.
Remarque : Localement dans ***getjamap*** la table ***mamap*** est appelée ***jamap.***

Une autre étape très importante dans le processus de préparation de la structure de données, est le stockage des informations concernant les équations relatives aux nœuds situés aux interfaces entre les différents sous-domaines. Cette tâche est accomplie par la routine ***BDRY.*** Cette dernière utilise principalement les vecteurs suivant :

- Le nombre cumulatif des valeurs non-nulles de la matrice globale stockée dans le vecteur ***Ia*** précédemment défini.
- Le vecteur ***riord***, ce vecteur est composé de deux parties : Les ***nb*** premières valeurs représentent la liste des équations locales (c'est une copie du vecteur maploc). Les valeurs allant de ***nb***+ 1 jusqu'à 2**nb*** sont initialisées a zéro.

La routine BDRY a pour sortie :

- nbnd : nbnd –1 représente le nombre d'équations locales internes
- Afin de simplifier le processus de résolution, les équations locales sont réordonnées d'une manière plus intelligente et plus souple. Les équations relatives aux nœuds internes sont stockées dans le vecteur ***riord*** de ***nb***+1 à ***nb+nbnd***-1. Les équations relatives aux nœuds de l'interface sont stockées dans ***riord*** de ***nb+nbnd*** à 2****nb***.
- ***Nproc*** : le nombre de sous-domaine (ou processeurs) voisins.
- ***Proc*** : un vecteur de dimension ***nloc*** qui contient la liste des processeurs voisins.
- ***Ix*** : liste des équations voisines, stockées processeur par processeur.
- ***Ipr :***Ipr (j) pointe vers le début de la liste des équations de l'interface, couplées

Une fois le vecteur ***Ix*** est cré, les différents processeurs échangent alors les informations nécessaires pour résoudre le problème global. Cette tâche est assurée par plusieurs routines dont; **befconsis, Setup1**et **Setup2** qui seront expliquées par la suite.
La routine **befconsis** est appelée pour construire le vecteur ***Jb***. Ce vecteur contient les indices des colonnes associés à la matrice des conditions aux limites externes.

4.2 Adaptation à la méthode des éléments finis

Parmi les améliorations apportées à la bibliothèque PSPARSLIB est l'utilisation de l'allocation dynamique de mémoire. En effet, la compatibilité entre le fortran 77, 90 et 95 a permis une intégration très réussite de l'allocation dynamique de mémoire ainsi que l'utilisation des modules pour gérer plus efficacement les variables globales. (Ces aspects informatiques sont détaillés à la section1.3). Ainsi, nous avons pu profiter de PSPARSLIB sans avoir à la réécrire en C ou toute autre langage qui offre l'allocation dynamique de mémoire.

D'autres ajouts ont été apportés à la bibliothèque PSPARSLIB pour l'adapter à la méthode des éléments finis. PSPARSLIB a été conçue pour la résolution d'un système de type $\mathbf{A}x = b$ où $\mathbf{A}$ est une matrice déjà construite ce qui n'est pas le cas dans notre méthodologie de résolution. En effet, le code PFES construit la trace de $\mathbf{A}$ de chaque sous-domaine (matrice locale) par l'assemblage des matrices élémentaires. La matrice global $\mathbf{A}_{glob}$ n'est jamais construite.

Pour adapter PSPARSLIB quelques routines ont été ajoutées.

✓ La routine **Setup1** assure les tâches suivantes :

Pour la résolution du problème local chaque sous-domaine a besoin des numéros d'équations des nœuds locaux externes.

Les figures suivantes illustrent la façon dont les équations sont stockées.

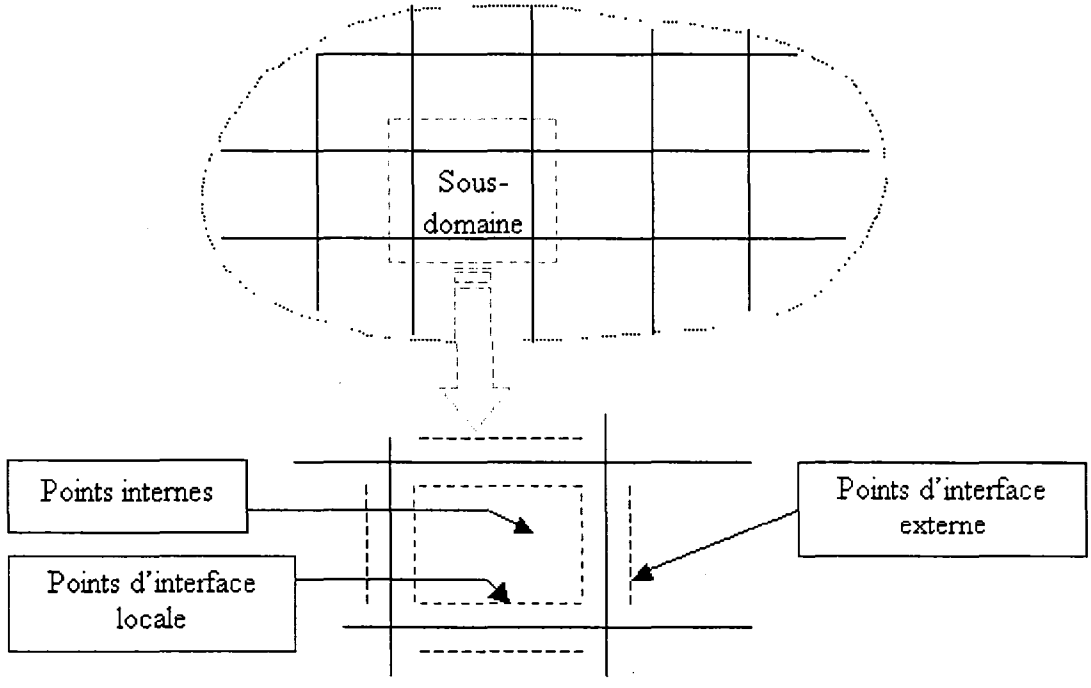


Figure 16 Points d'interface

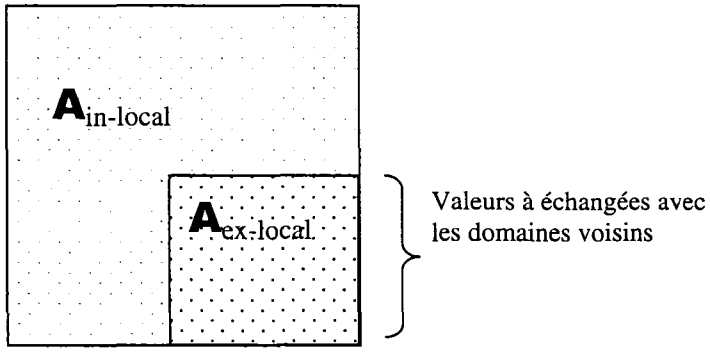


Figure 17 Matrice du domaine : blocs interne et externe

Une première étape consiste à construire les vecteurs **iax** et ***jax*** associés à la matrice locale des équations d'interface $\mathbf{A}_{\text{ex-local}}$. **iax** et ***jax*** sont équivalents à ***ia*** et ***ja*** pour la matrice globale voir les figures 20, 21 et 22.

Chaque processeur envoi à ces voisins le nombre d'équations sur leurs frontières communes, et vis-versa chaque processeur reçoit cette information de ses voisins, ainsi il la stocke dans une table temporaire.

Remarquons à ce stade que cet échange se fait exclusivement entre les voisins dans le sein du même domaine physique, appelé dans le programme famille et ceci grâce au communicateur appelé ***Neighbors***. Certaines informations pourraient être communiquées entre tous les processeurs de la famille à travers le communicateur ***Family***, ou entre les domaines physiques à travers le communicateur ***Diplomacy***. Généralement, la résolution des problèmes multiphysiques nécessite une interface entre les domaines. (ces aspects sont plus détaillés à la section 3.2).

Une fois l'étape précédente de la communication est achevée, chaque processeur sait combien il doit recevoir de ses voisins. Commence alors l'échange des numéros des équations des frontières pour former ***iax*** et ***jax.***

✓ Les vecteurs ***iax*** et ***jax*** sont des vecteurs locaux le processus de résolution nécessite leur assemblage, c'est le rôle de La routine **Setup2.** Les vecteurs globaux ***iaxglob*** et ***jaxglob*** jouent le même rôle que ***ia*** et ***ja*** mais cette fois pour les équations aux nœuds d'interfaces externes.

✓ La routine setup3 est appelée chaque itération pour assembler les matrices $\mathbf{A}_{\text{ex-local}}$ appelées ***vax*** pour construire la matrice ***vaxglob*** du domaine. Cet assemblage est nécessaire pour assurer la fabrication de la matrice locale (Algorithme de Schwarz).

CHAPITRE 5

ALGORITHMES PAR SOUS DOMAINES

5.1.1 Méthodes de décomposition du domaine

Les méthodes de résolution des systèmes sur machines parallèles sont en pleine évolution. En 1870 **Schwarz** [1,10,29] a proposé la procédure suivante : Le domaine Ω est divisé en plusieurs sous-domaines $\{\Omega_1,\Omega_2, \ldots\Omega_n\}$.

D'une manière générale la méthode consiste à parcourir les sous-domaines Ω_i et de résoudre le système local en utilisant des conditions aux limites basées sur la solution la plus récente des autres sous-domaines. La solution globale du problème est trouvée par assemblage des solutions locales. Dans cette section, deux algorithmes de **Schwarz** seront présentés.

5.1.1.1 Schwarz multiplicative

Soit un domaine Ω, la procédure consiste à diviser ce domaine en $\boldsymbol{n}$ sous-domaines Ω_i. Avec $\boldsymbol{i}$ allant de 1 à $\boldsymbol{n}$. Les frontières internes des différents sous-domaines sont notées par $\Gamma_{i,j}$. $\Gamma_{i,j}$ est la frontière interne à Ω_i avec le domaine Ω_j.
Les $\Gamma_{i,0}$ sont les traces de la frontière du domaine Ω au niveau des sous-domaines Ω_i.

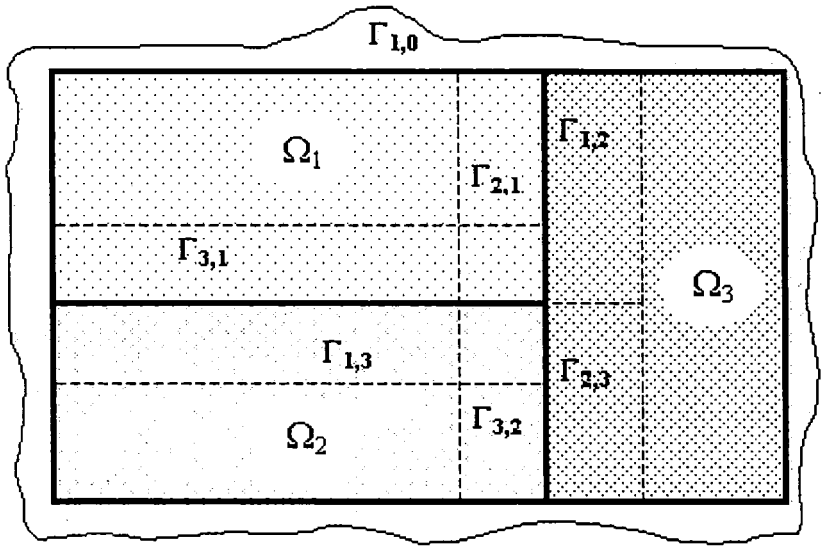


Figure 18 Sous-domaines et frontières

L'algorithme de Schwarz multiplicative peut alors s'écrire comme suit :

1. Choisir une solution initiale u_0.
2. Répéter jusqu'à satisfaction du critère de convergence :
3. de i=1,.....,s :
4. calculer u_i^{k+1} dans le domaine Ωi, avec $u_i^{k+1}=u_j^k$ sur Γi,j
5. mettre à jour u sur la frontière Γi,j, $\forall$ j
6. Fin de la boucle i sur les domaines
7. convergence

La figure suivante illustre le calcul de U dans Ω1 à l'itération k+1, ligne 4 de l'algorithme de **Schwarz multiplicative**.

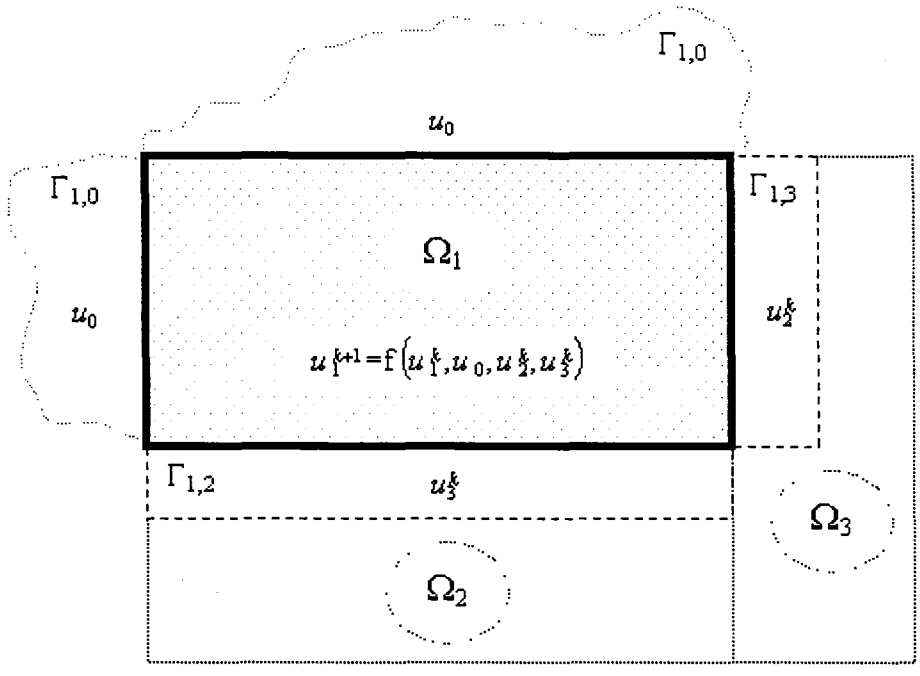


Figure 19 Algorithme de Schwarz multiplicative

5.1.1.2 Schwarz Additive

Cette deuxième variante de l'algorithme de **Schwarz** est différente de la procédure de **Schwarz multiplicative** par le fait que la solution n'est mise à jour qu'à la fin de la boucle sur tous les domaines.

L'algorithme Schwarz additive peut s'écrire comme suit :

1. **For** i = ,....s :
2. calculer $\delta_i = \mathbf{R}_i^T \mathbf{A}_i^{-1} \mathbf{R}_i (b - \mathbf{A}x)$
3. **Fin** de la boucle sur les domaines
4. $x_{new} = x + \sum_{i=1}^{s} \delta_i$

où s est le nombre de domaines, $\mathbf{R}_i$ est l'opérateur de restriction associé au sous-domaine i et $\mathbf{A}_i = \mathbf{R}_i \mathbf{A} \mathbf{R}_i^T$ la matrice locale associée au sous-domaine.

$\mathbf{R}_i$ est une matrice de dimension $n_i x\, n$ qui ne contient que des 0 et des 1.
Pour plus de détailles voir [29].

5.1.2 Algorithmes de résolution GMRES

La résolution de systèmes linéaires creux de grande dimension est un problème courant en calcul scientifique. Parmi les méthodes itératives de résolution par projection, un algorithme bien adapté est **GMRES** « Generalized Minimal Residual » développé par Saad et Shultz [27].

5.1.2.1 GMRES standard

La résolution des problèmes physiques, particulièrement en mécanique des fluides, abouti généralement à un système d'équations non-linéaires. Après discrétisation le problème peut être écrit sous la forme :

$$F(x)=0 \quad avec \quad F: \Re^n \rightarrow \Re^n \tag{5.1}$$

La méthode de Newton peut être utilisée pour la résolution de ce problème, Considérons une solution approximative initiale $x_0 \in \Re^n$, à chaque itération i la solution peut être écrite de la façon suivante :

$$\mathbf{A\,x = b} \tag{5.2}$$

Où **x** est le vecteur inconnu, **A** est une matrice non-symétrique de grande taille, et **b** le second membre.

$$x = x_0 + z \tag{5.3}$$

où x_0 est une solution initiale, et z est un élément du sous espace de Krylov $\mathbf{K_k}$, de dimension k, associé au résidu $r_0=\mathbf{b}-\mathbf{A}x_0$ et la matrice jacobienne **A** :

$$K_k = span\{r_0, \mathbf{A}\, r_0, ..., \mathbf{A}^{k-1} r_0\} \tag{5.4}$$

L'algorithme **GMRES** emploie une base orthonormée de l'espace de Krylov obtenue par la méthode d'orthogonalisation de **Gram-Schmidt** modifié.

L'algorithme **GMRES** détermine z de façon telle que la norme $\|\mathbf{b}-\mathbf{A}(x_0-z)\|$ soit minimisée.

On construit, via le procédé d'**Arnoldi**, une base orthonormée $\mathbf{U_k}$, de l'espace de **Krylov** de dimension k. Le procédé d'**Arnoldi** est basé sur la méthode d'orthogonalisation de **Gram-Schmidt**, qui peut être présenté comme suit :

Procédé d'orthogonolisation de Gram-Schmidt

$$u_1 = \frac{r_0}{\| r_0 \|}$$

Pour $i = 1, k$

  $u_{i+1} = \mathbf{A}\, u_i$

  Pour $j = 1, i$

    $\beta_{i+1,j} = \left(u_{i+1}, u_j\right)$

    $u_{i+1} \leftarrow u_{i+1} - \beta_{i+1,j}\, u_j$

    $u_{i+1} = \frac{u_{i+1}}{\| u_{i+1} \|}$

  Fin de la boucle sur j

Fin de la boucle sur i

où u_k est le kème vecteur de la base orthogonale de l'espace de **Krylov** :

$$\mathbf{U_k} = \{u_1, u_2, ..., u_k\} \tag{5.5}$$

Le fait que U_k soit une base orthonormée de l'espace de Krylov, on peut montrer que la relation suivante est satisfaite :

$$\mathbf{A\,U_k} = \mathbf{U_{k+1}\,H_k} \tag{5.6}$$

où $\mathbf{H}_k$ est la matrice de **Hessenberg** supérieure de dimension (k+1)×k

$$\mathbf{H_k} = \begin{bmatrix} \beta_{2,1} & \beta_{3,1} & \cdots & \beta_{k,1} & \beta_{k+1,1} \\ \|u_2\| & \beta_{3,2} & \cdots & \beta_{k,2} & \beta_{k+1,2} \\ 0 & \|u_3\| & \cdots & : & : \\ : & 0 & & : & : \\ : & : & \cdots & \beta_{k,k-1} & : \\ 0 & 0 & \cdots & \|u_k\| & \beta_{k+1,k} \\ 0 & 0 & \cdots & 0 & \|u_{k+1}\| \end{bmatrix} \tag{5.7}$$

Si on écrit z comme combinaison linéaire des $\boldsymbol{u_j}$:

$$z = \sum_{j=1}^{k} y_i\, u_j \qquad y \in \mathbf{R}^k \tag{5.8}$$

et

$$r_0 = U_{k+1}\, e \tag{5.9}$$

où $e = \{\|r_0\|, 0, ..., 0\}^T$, possédant k+1 éléments.

Avec ces notations on a :

$$\| \mathbf{b} - \mathbf{A}(x_0 - z) \| = \left\| r_0 - \mathbf{A}\left(\sum_{j=1}^{k} y_j u_j \right) \right\| = \| r_0 - \mathbf{A}\,\mathbf{U_k}\, y \| = \| \mathbf{U_{k+1}}(e - \mathbf{H_k}\, y) \|$$
$$= \| e - \mathbf{H_k}\, y \| \qquad (5.10)$$

Donc, le problème de minimisation dans l'espace de **Krylov** K_k peut être écrit comme suit :

Trouver $y \in \mathbf{R}^k$ tel que :

$$\min_{z \in k} \| \mathbf{b} - \mathbf{A}(x_0 - z) \| = \min_{y \in \mathbf{R}^k} \| e - \mathbf{H_k}\, y \| \qquad (5.11)$$

Algorithme GMRES

1. calculer $r_0 = b - \mathbf{A}x_0$, $\beta = \| r_0 \|^2$ et $v_1 = \frac{r_0}{\beta}$
2. définir la m(m+1) matrice $\overline{H_m}\{h_{i,j}\}_{1 \le i \le m+1,\ 1 \le j \le m}$, Initialiser $\overline{H_m} = 0$
3. **Pour** j = 1 à m
4. calculer $w_j = \mathbf{A}\, v_j$
5. **Pour** i = 1 à j
6. $h_{ij} = (w_j, v_i)$
7. $w_j = w_j - h_{ij}\, v_i$

   Fin de la boucle sur i
8. $h_{j+1,j} = \| w_j \|^2$, si $h_{j+1,j} = 0$ alors m = j, **GOTO** ligne **10**
9. $v_{j+1} = \frac{w_j}{h_{j+1,j}}$

   Fin de la boucle sur j

10. calculer y_m de façon à minimiser $\left\| \beta e_1 - \overline{H}_m y \right\|^2$
11. $x_m = x_0 + V_m y_m$
12. Si la condition de convergence est satisfaite **STOP**
    sinon $x_0 = x_m$, **GOTO** ligne **1**

5.1.2.1.1 Préconditionnement

Pour augmenter la performance de la résolution itérative (taux de convergence) un préconditionneur de type **ILUT** [2] est utilisé. Ce préconditionneur est très bien adapté pour la résolution des systèmes linéaires. Pour le cas des systèmes non-linéaires d'autres techniques sont utilisées. Dans cette section quelques algorithme de **GMRES** préconditionné seront présenter. Pour plus de détails sur ces techniques voir [2,3,5].

5.1.2.2 GMRES préconditionné

L'algorithme de résolution utilisé et basé sur une procédure de marche dans le temps (time-marching) combinée à un algorithme de **Newton** et des variantes de **GMRES** [29]. Les méthodes de stabilisation généralement utilisées, introduisent des non-linéarités additionnelles aux problèmes. Dans certains cas l'algorithme de **GMRES** préconditionné standard peut alors stagner. Dans ces cas d'autres variantes et combinaisons sont envisagées. En particulier une combinaison très réussie de la méthode de décomposition de domaine **Additive Schwarz** (ou **Multiplicative Schwarz**), le préconditionneur **ILUT** [2] et une version non-linéaire du solveur GMRES.

Les algorithmes de **GMRES** standard et **FGMRES** [2] préconditionnés sont présentés dans cette section.

5.1.2.2.1 GMRES préconditionné standard

L'algorithme présenté dans cette section est l'algorithme de GMRES préconditionné à droite [29], « Right preconditioned GMRES » cet algorithme transforme le système (4.) en :

$$\mathbf{A}\mathbf{M}^{-1}\,\mathbf{M}x=\mathbf{b} \tag{5.12}$$

où **M** est une matrice de préconditionnement, cette matrice est généralement très similaire à **A** mais facilement inversible, la résolution est alors plus rapide. L'algorithme a besoin aussi d'une solution initiale x_0.

Algorithme GMRES préconditionné standard:

« Right preconditioned GMRES »

5. calculer le résidu initial $r_0 = b - \mathbf{A}x_0$

calculer $\beta = \| r_0 \|^2$

calculer $v_1 = \frac{r_0}{\beta}$

6. définir la m(m+1) matrice $\overline{H_m}\{h_{i,j}\}_{1 \le i \le m+1,\ 1 \le j \le m}$

Initialiser $\overline{H_m} = 0$

7. **Pour** j = 1 à m
8. calculer $z_j = \mathbf{M}^{-1} v_j$
9. calculer $w_j = \mathbf{A}\, z_j$
10. **Pour** i = 1 à j
11. $h_{ij} = (w_j, v_i)$
12. $w_j = w_j - h_{ij}\, v_i$

Fin de la boucle sur i

13. $h_{j+1,j} = \left\| w_j \right\|^2$

14. si $h_{j+1,j} = 0$ alors m = j, **GOTO** ligne **16**

15. $v_{j+1} = \frac{w_j}{h_{j+1,j}}$

Fin de la boucle sur j

16. définir $\mathbf{V}_m = \left[v_1, v_2, ..., v_m \right]$ et $\overline{H_m} \left\{ h_{i,j} \right\}_{1 \le i \le m+1,\ 1 \le j \le m}$

17. calculer y_m de façon à minimiser $\left\| \beta e_1 - \overline{H}_m y \right\|^2$

18. $x_m = x_0 + \mathbf{M}^{-1} V_m y_m$

19. Si la condition de convergence est satisfaite **STOP**
    sinon $x_0 = x_m$, **GOTO** ligne **1**

5.1.2.2.2 GMRES préconditionné non-linéaire

La version dite non-linéaire de GMRES et similaire à la version standard sauf que dans le cas non linéaire le jacobien **A** n'est pas calculé analytiquement. Il est à noter dans ce cas que **A** représente le jacobien d'une fonction ***F*** non-linéaire. Il n'est toujours pas possible de calculer cette matrice **A**. Ou simplement le calcule du jacobien de telle fonction coûte trop cher ou trop compliqué analytiquement. Il est alors préférable d'approximer la matrice jacobienne A. Cette dernière est utilisée pour construire la matrice de préconditionnement **M**.

Comme dans l'algorithme GMRES on a besoin de calculer le produit $w_j = \mathbf{A} z_j$. Celui-ci peut être approximé parla différence :

$$\frac{F(x_0 + \varepsilon z_j) - F(x_0)}{\varepsilon} \tag{5.13}$$

où ε et nombre approprié très petit. La résolution des problèmes non-linéaires est basée sur une version non-linéaire de **GMRES**. Cette approche générale est appelée Newton-inexacte. Cette appellation vient du fait que le jacobien est calculé approximativement.

5.1.2.2.3 FGMRES

Le rôle du préconditionneur M est de permettre une résolution approximative et rapide du système :

$$\mathbf{A}\,x = \mathbf{b} \tag{5.14}$$

Le choix du préconditionneur est une étape extrêmement importante dans le processus de résolution. En effet, si le préconditionneur est très proche de la matrice **A** le programme peut converger très rapidement, et même, dans certain cas en une seule itération. Dans ce cas le préconditionneur coûtera très cher en terme de temps de calcule et en mémoire. À l'inverse, on peut utiliser un préconditionneur moins coûteux comme celui donné par **ILUT**(0). Dans ce dernier cas il est fort possible de rencontrer des problèmes de convergence. Une des variantes de **GMRES** consiste à varier le préconditionneur à chaque itération $z_j = \mathbf{M}_j^{-1} v_j$; ligne 4 de l'Algorithme GMRES préconditionné standard est appelée **FGMRES** [5].

Similairement au **GMRES** standard une version de FGMRES non linéaire peut être utiliser en utilisant le calcul par différence-finie (5.13) pour approximer $w_j = \mathbf{A}\,z_j$.

FGMRES

1. calculer le résidu initial $r_0 = b - \mathbf{A}x_0$

   calculer $\beta = \|r_0\|^2$

   calculer $v_1 = \frac{r_0}{\beta}$

2. définir la m(m+1) matrice $\overline{H_m}\{h_{i,j}\}_{1\le i\le m+1,\ 1\le j\le m}$

   Initialiser $\overline{H_m}=0$

3. **Pour** j = 1 à m

4. calculer $z_j=\mathbf{M}_j^{-1}v_j$

5. calculer $w_j=\mathbf{A}\,z_j$

6. **Pour** i = 1 à j

7. $h_{ij}=(w_j,v_i)$

8. $w_j=w_j-h_{ij}\,v_i$

   Fin de la boucle sur i

9. $h_{j+1,j}=\|w_j\|^2$

10. si $h_{j+1,j}=0$ alors m = j, **GOTO** ligne **12**

11. $v_{j+1}=\dfrac{w_j}{h_{j+1,j}}$

    Fin de la boucle sur j

12. définir $\mathbf{Z}_m=[z_1,z_2,.....,z_m]$ et $\overline{H_m}\{h_{i,j}\}_{1\le i\le m+1,\ 1\le j\le m}$

13. calculer y_m de façon à minimiser $\left\|\beta e_1-\overline{H}_m y\right\|^2$

14. $x_m = x_0 + \mathbf{M}^{-1}V_m y_m$

15. Si la condition de convergence est satisfaite STOP

16. sinon $x_0 = x_m$, **GOTO** ligne **1**

5.1.2.3 GMRES parallélisé

Les méthodes de décomposition de domaine, sont des méthodes générales, simples et bien adaptées à la résolution des équations à dérivées partielles en calcul parallèle. Le domaine est alors divisé en plusieurs sous-domaines. La solution globale est construite par assemblage des solutions locales. Dans cette section, une méthodologie de résolution parallèle des systèmes non-linéaires, basé sur l'algorithme de **GMRES**, sera détaillée.

La version parallèle de GMRES, appelée **PGMRES**, regroupe les mêmes étapes vues dans l'algorithme séquentiel (section 5.1.2.2). Cependant, quelques opérations, tel que le produit matrice-vecteur, ne peuvent être réalisées avec la même simplicité. Le fait que les données soient partagées entre les processeurs, impose des étapes de communications supplémentaires.

Dans cette partie, les techniques utilisées pour résoudre ces difficultés seront abordées.

Gardons à l'esprit que la décomposition n'est qu'un moyen pour surmonter les limites des machines. L'objectif principal étant de résoudre le problème global, une convergence globale doit être assurée à la fin du procédé de résolution.

En d'autres mots, le test de convergence de la version parallèle de GMRES doit refléter la qualité de la solution globale et non pas la qualité des solutions locales.
Un calcul du résidu global s'impose.

$$Res_{glob} = \mathbf{A}\,x - \mathbf{b} \tag{5.15}$$

Le résidu global Res_{glob} ne peut être calculé directement. Un assemblage des résidus locaux est alors indispensable.

$$\mathrm{Res}_{glob} = \text{Assemblage des } \mathrm{Res}_{\mathrm{loc}} = \Re(\mathrm{Res}_{\mathrm{loc}}) \tag{5.16}$$

Soit R_i l'opérateur de restriction associé au sous-domaine i.

on sait que

$$\mathbf{A}_i = R_i \, \mathbf{A} \, R_i^T \tag{5.17}$$

et $R_i \; z_{glob} = z_i$

$R_i^T \; z_i = z_{glob}$

On peut écrire alors le résidu global (5.15) comme suit:

$$\mathrm{Res}_{glob} = \sum_{i=1}^{m} R_i^T \mathrm{Res}_{loc} \tag{5.18}$$

La version parallèle de GMRES nécessite aussi le calcul du carré de la norme du résidu global.

$$\| \mathrm{Res} \|_{Glob}^2 = \sum_{i=1}^{m} \| \mathrm{Res} \|_{Loc\,i}^2 \tag{5.19}$$

de même pour le produit scalaire (w,v) qui doit être calculé correctement.

$$(w,v)_{Glob} = \sum_{i=1}^{m} (R_i, w_i)_{Loc} \, (R_i, v_i)_{Loc} \tag{5.20}$$

Pour assurer la continuité de la solution aux interfaces entre les différents sous-domaines, les degrés de liberté partagés doivent être moyennés. Cette opération, ainsi que l'assemblage du résidu, sont facilement implémentées grâce à la structure de données générée par la bibliothèque PSPARSLIB.

Le calcul de la norme du résidu est effectué par la routine ***MPI_Allreduce***. Pour assembler et moyenner les DDL aux interfaces des sous-domaines, les routines ***consis_count*** et ***consis_av*** sont utilisées.

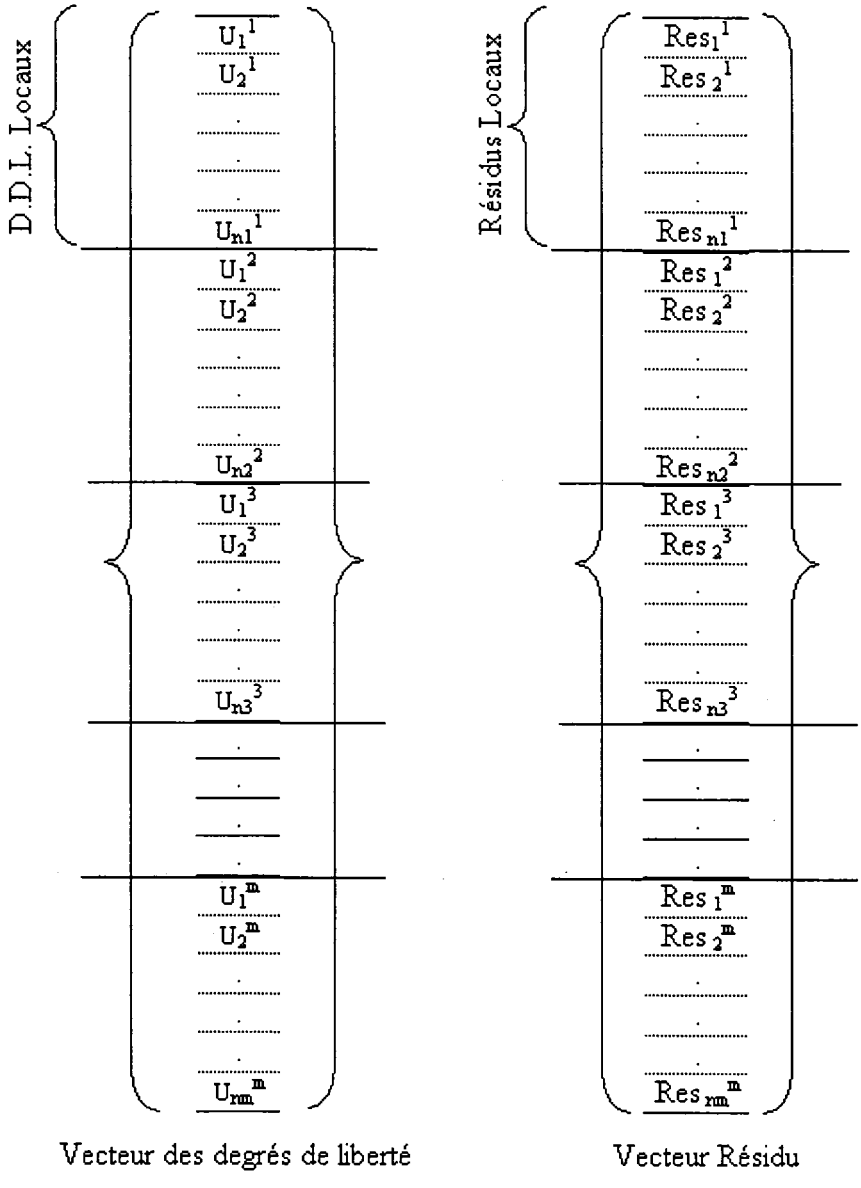


Figure 20 Vecteur des degrés de liberté et vecteur résidu

L'algorithme de PGMRES est détaillé au paragraphe suivant. Pour plus de détails la routine ***pgmres*** est présentée. (Annexe 4, Pgmres).

GMRES PARALLELE

« Parallel Newton-GMRES »

1. Décomposer le maillage à l'aide du code Metis [18].
2. Préparer la structure de donnée.
3. **Itération** sur le pas de temps, de Ipas = 1, Npas :
4. Calculer et factoriser la matrice du préconditionnement locale M chaque N pas de temps.
5. **Itérations** de Newton, de ***Iter*** = 1, ***Niter***
6. Calculer le résidu initial r0 = F(x0), $\beta = \| r_0 \|^2$ et.
7. Définir la m(m+1) matrice $\overline{H_m}\{h_{i,j}\}_{1 \le i \le m+1,\ 1 \le j \le m}$ et initialiser.
8. **Pour** j = 1 jusqu'à m :
9. Calculer $z_j = \mathbf{M}^{-1} v_j$
10. Calculer la représentation locale de la solution perturbée, communiquer les valeurs aux interfaces pour construire la représentation globale en moyennant ces valeurs. Et calculer $\mathbf{F}(x_0 + \varepsilon z_j)$.
11. Calculer $w_j = \frac{F(x_0+\varepsilon z_j) - F(x_0)}{\varepsilon}$
12. **Pour** i = 1 jusqu'à j :
13. Calculer localement $h_{ij} = (w_j, v_i)$ est sommer toutes les valeurs locales.
14. calculer $w_j = w_j - h_{ij} v_i$

    Fin boucle sur i
15. Calculer localement $h_{j+1,j} = \| w_j \|^2$ est faire la somme sur tous les domaines.

    si $h_{j+1,j} = 0$ alors m = j, **GOTO** ligne **17**

16. $v_{j+1} = \frac{w_j}{h_{j+1,j}}$

    Fin boucle sur j

17. définir $\mathbf{Z}_m = [z_1, z_2, ..., z_m]$ et $\overline{H_m}\{h_{i,j}\}_{1 \le i \le m+1,\ 1 \le j \le m}$

18. calculer y_m de façon à minimiser $\left\| \beta e_1 - \overline{H}_m y \right\|^2$

    calculer localement $x_m = x_0 + \mathbf{M}^{-1} V_m y_m$ est communiquer les valeurs aux interfaces entre les sous-domaines, pour calculer la moyenne. Cette opération garantie la continuité de la solution globale entre les sous-domaines.

19. Si la condition de convergence est satisfaite **STOP**
    sinon $x_0 = x_m$, **GOTO** ligne **6**

20. **Fin** des itérations de Newton

21. **Fin** de la boucle sur le temps

CHAPITRE 6

APPLICATION A L'INTERACTION FLUIDE-STRUCTURE

Ce chapitre présente, les équations gouvernantes dans chaque domaine physique, les algorithmes de résolutions adoptés, les tests de performance et les résultats des simulations.

6.1 Rappel du code CFD/CSD/Mesh

Dans cette section, les équations, ainsi que la méthodologie de résolution du problème dans chaque domaine physique, sont brièvement rappelées.

6.1.1 Fluide

Cette section est consacrée à la présentation des équations gouvernantes du domaine fluide ainsi que la méthode de stabilisation **SUPG** [5] « Streamline Upwinding Petrov-Galerkin ».

Dans le domaine fluide, une formulation basée sur la méthode des éléments finis stabilisée, est utilisée pour la discrétisation spatiale tridimensionnelle des équations d'**Euler** et de **Navier-Stokes**. La méthode **SUPG** est utilisée pour la stabilisation. La résolution est basée sur un solveur parallèle implicite. La méthode de décomposition de Schwarz et une version parallèle non-linéaire de GMRES dite **PGMRES** (voir section 5.1.2.3), sont utilisées pour la résolution.

6.1.1.1 Équations de Navier-stokes

Les équations de Navier-Stokes adimensionnelles pour un fluide compressible, exprimées en fonction de (ρ, **U**, **E**), s'écrivent comme suit :

$$\frac{\partial \rho}{\partial t} + \nabla.\mathbf{U} = 0 \tag{6.1}$$

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla.(\mathbf{U} \otimes \mathbf{u}) + \nabla p - \nabla.\boldsymbol{\sigma} = \rho \mathbf{f} \tag{6.2}$$

$$\frac{\partial e}{\partial t} + \nabla.[(e+p)\mathbf{u}] - \nabla.(\boldsymbol{\sigma}.\mathbf{u}) - \nabla.\mathbf{q} = \rho r + \mathbf{f}.\mathbf{U} \tag{6.3}$$

Où ρ est la densité, **U** le moment par unité de volume, u la vitesse, σ le tenseur des contraintes visqueuses, q le flux de chaleur, f les forces volumiques par unité de masse, et **E** l'énergie totale par unité de volume.

Les équations supplémentaires suivantes sont utilisées pour la résolution du système:

$$u = \frac{U}{\rho},$$

$$T = \frac{E}{\rho} + \frac{|U|^2}{2\rho^2},$$

$$p = (\gamma - 1)\rho T,$$

$$q = -\frac{\mu \gamma}{\mathrm{Re}\, p_r} \nabla T,$$

$$\sigma = \frac{\mu}{\mathrm{Re}} \left[\nabla u + (\nabla u)^t - \frac{2}{3} (\nabla.u)\, \mathbf{I} \right].$$

Où R_e est le nombre de Reynolds, P_r le nombre de Prandtl, **I** le tenseur identité et μ la viscosité laminaire adimensionnelle

6.1.1.2 Méthode de stabilisation SUPG

Le système d'équations global, sur lequel on veut appliquer les méthodes de stabilisation, peut s'écrire sous la forme vectorielle suivante :

$$\mathbf{V}_{,t} + \mathbf{F}_{i,i}^{conv}(\mathbf{V}) = \mathbf{F}_{i,i}^{diff}(\mathbf{V}) + \Im \tag{6.4}$$

ou sous la forme quasi-linéaire:

$$\mathbf{V}_{,t} + \mathbf{A}_i \mathbf{V}_{,i} = \left(\mathbf{K}_{ij} \mathbf{V}_{,j} \right)_{,i} + \Im \tag{6.5}$$

où $\mathbf{A}_i$ sont les matrices jacobiennes de transformation du vecteur flux de convection telles que:

$$\mathbf{A}_i = \mathbf{F}_{i,\mathbf{V}}^{conv} = \frac{\partial \mathbf{F}_i^{conv}}{\partial \mathbf{V}} \tag{6.6}$$

et $\mathbf{K}_{ij}$ sont les coéfficients des matrices de diffusion, définies par :

$$\mathbf{K}_{ij} \mathbf{V}_{,j} = \mathbf{F}_i^{diff} \tag{6.7}$$

Selon la méthode **SUPG**, la forme vectorielle (6.4) et la forme quasilinéaire (6.5) sont remplacées respectivement par les formulations variationnelles faibles sous forme vectorielle et quasilinéaire suivantes:

$$\int_{\Omega} \left\{ \mathbf{W} . \left[\mathbf{A}_0 \mathbf{V}_{,t} + \mathbf{F}_{i,i}^{conv}(\mathbf{V}) - \Im \right] + \mathbf{W}_{,i} . \left[\mathbf{F}_{i,i}^{diff}(\mathbf{V}) \right] \right\} d\Omega$$

$$+ \sum_e \int_{\Omega e} \left\{ \left(\mathbf{A}_i^t . \mathbf{W}_{,i} \right) \tau \left[\mathbf{A}_0 \mathbf{V}_{,t} + \mathbf{F}_{i,i}^{conv}(\mathbf{V}) - \mathbf{F}_{i,i}^{diff}(\mathbf{V}) - \Im \right] \right\} d\Omega e$$

$$= \oint_{\Gamma} \left\{ \mathbf{W} . \left[\mathbf{F}_{i,i}^{diff}(\mathbf{V}) . \mathbf{n}_i \right] \right\} d\Gamma \tag{6.8}$$

$$\int_{\Omega} \left\{ \mathbf{W} \cdot \left[\mathbf{A}_0 \mathbf{V}_{,t} + \mathbf{A}_i \mathbf{V}_{,i} - \Im \right] + \mathbf{W}_{,i} \cdot \left[\mathbf{K}_{ij} \mathbf{V}_{,j} \right] \right\} \, d\Omega$$

$$+ \sum_{e} \int_{\Omega e} \left\{ \left(\mathbf{A}_i^t \cdot \mathbf{W}_{,i} \right) \tau \left[\mathbf{A}_0 \mathbf{V}_{,t} + \mathbf{A}_i \mathbf{V}_{,i} - \left(\mathbf{K}_{ij} \mathbf{V}_{,j} \right) \mathbf{V}_{,j} - \Im \right] \right\} \, d\Omega e$$

$$= \oint_{\Gamma} \left\{ \mathbf{W} \cdot \left[\left(\mathbf{K}_{ij} \mathbf{V}_{,j} \right) \cdot \mathbf{n}_i \right] \right\} \, d\Gamma \qquad (6.9)$$

avec

$$\underline{\underline{\tilde{\tau}}} = \mathbf{A}_0^{-1} \underline{\underline{\tau}} \qquad (6.10)$$

Dans le cas des variables conservatives, $\mathbf{V} = (\rho, \mathbf{U}, E)_t$ et la matrice $\mathbf{A}_0$ devient l'identité. Par conséquent, la matrice $\underline{\underline{\tilde{\tau}}}$ se réduit à la matrice $\underline{\underline{\tau}}$ correspondant aux variables conservatives $\mathbf{V}$ [5].

La formulation variationnelle ci-dessus se distingue par deux propriétés importantes :

- C'est une méthode de résidus pondérés au sens qu'une solution exacte régulière du problème physique original reste encore une solution du problème variationnel. Ceci assure, non seulement une bonne précision de l'approximation mais aussi, une stabilité spatio-temporelle.

- La stabilité est assurée grâce au terme elliptique :

$$\sum_{e} \int_{\Omega e} d\Omega e \, \left\{ \left(\mathbf{A}_i^t \cdot \mathbf{W}_{,i} \right) \underline{\underline{\tilde{\tau}}} \, \mathbf{F}_i^{conv}(\mathbf{V}) \right\} \qquad (6.11)$$

en écriture vectorielle ou

$$\sum_{e} \int_{\Omega e} \left\{ \left(\mathbf{A}_i^t \cdot \mathbf{W}_{,i} \right) \tilde{\tilde{\tau}} \left[\mathbf{A}_i \, \mathbf{V}_i \right] \right\} d\Omega e \tag{6.12}$$

en écriture quasi-linéaire.

6.1.2 Mouvement de maillage (Mesh)

Plusieurs choix peuvent être considérés pour désigner l'opérateur de distribution du mouvement du domaine mobile à l'interface fluide. Dans le programme PFES, le mouvement de maillage est défini par les équations d'élasticité.
À l'instant t :

$$\rho_m \mathbf{x}_{,tt} - \mathrm{div}(\mathbf{P}(\mathbf{x})) = \mathbf{b} \quad (\text{ dans le domaine } \Omega(0)\,) \tag{6.13}$$

où ρ_m est la densité fictive, P le tenseur des contraintes (PK1) et b le vecteur des forces volumiques [6,7].

La résolution de l'équation (6.13) permet de trouver les déplacements du maillage **x**.
Les conditions cinématiques sur les frontières mobiles sont :

$$\mathbf{w}.\mathrm{n} = \mathbf{u}.\mathrm{n} \tag{6.14}$$

où **u** désigne la vitesse du fluide et **w** la vitesse du maillage.

La vitesse **w** est calculée à l'aide d'un schéma de différence finie pour l'équation différentielle $\mathbf{w} = \mathbf{x}_{,t}$.
Généralement, ρ_m et **b** sont considérés nuls.

Au cours du mouvement, il n'est pas toujours garanti que les éléments de maillage gardent une taille acceptable pour des calculs CFD assez précis. En effet, les petits éléments peuvent subir de grandes distorsions. Afin d'éviter les volumes négatifs et les grandes distorsions, des lois adéquates ont été appliqués [6,7].

6.1.3 Structure

Dans le domaine de la mécanique des structures, les codes commerciaux existants, permettent des analyses linéaires très satisfaisantes. La méthode des éléments finis, bien adaptée à la résolution de ces problèmes, est largement utilisée.
Une façon intelligente de procéder, est de créer une interface pour extraire les modes propres et les fréquences naturelles, à partir d'un code commercial existant.
Dans le domaine structure, le modèle linéaire est décrit comme suit :

$$[\mathbf{M}]\{\ddot{q}(t)\}+[\mathbf{D}]\{\dot{q}(t)\}+[\mathbf{K}]\{q(t)\}=\{r(t)\} \tag{6.15}$$

où $\{\ddot{q}(t)\}$, $\{\dot{q}(t)\}$ et $\{q(t)\}$ sont respectivement l'accélération, la vitesse et les déplacements de la structure.
$\{r(t)\}$ est le vecteur des forces appliquées sur la structure.
La matrice masse **[M]**, la matrice d'amortissement **[D]** et la matrice de rigidité **[K]** sont considérées constantes positives.
En utilisant le changement de variable suivant :

$$\{q(t)\} = [\mathbf{P}_0]\{Z(t)\} \tag{6.16}$$

avec $\{Z(t)\}$ le vecteur des cordonnées modales et $[\mathbf{P}_0]$ la matrice modale.

L'équation (6.15) devient :

$$[\mathbf{M}][\mathbf{P}_0]\{\ddot{z}(t)\}+[\mathbf{D}][\mathbf{P}_0]\{\dot{z}(t)\}+[\mathbf{K}][\mathbf{P}_0]\{z(t)\}=\{S(t)\} \tag{6.17}$$

$\{S(t)\}=[\mathbf{P}_0]^T\{r(t)\}$ représente le vecteur des forces modales.

Pour chaque mode *i,* on peut, alors, écrire l'équation différentielle suivante :

$$\ddot{Z}_i(t)+2\,\eta_i\,\omega_i\,\dot{Z}_i(t)+\omega_i^2 Z_i(t)=S_{0i}(t) \tag{6.18}$$

ω_i et η_i sont respectivement la fréquence naturelle et l'amortissement du $i^{ième}$ mode.
L'algorithme de **Newmark** [19] est utilisé pour l'intégration temporelle.

L'algorithme suivant représente les différentes étapes du programme PFES.

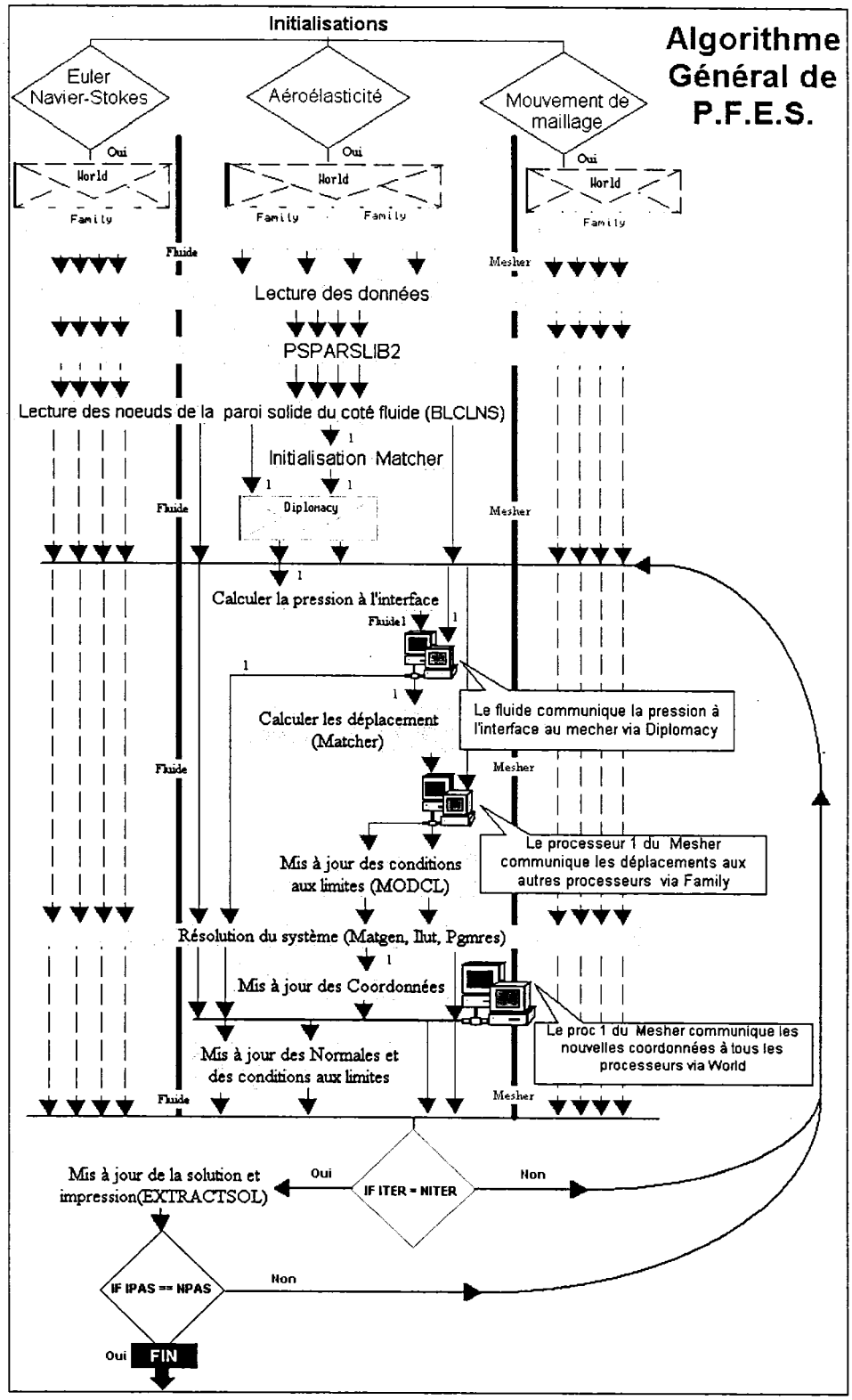


Figure 21 Algorithme général P.F.E.S.

L'algorithme suivant présente la méthodologie de résolution standard.

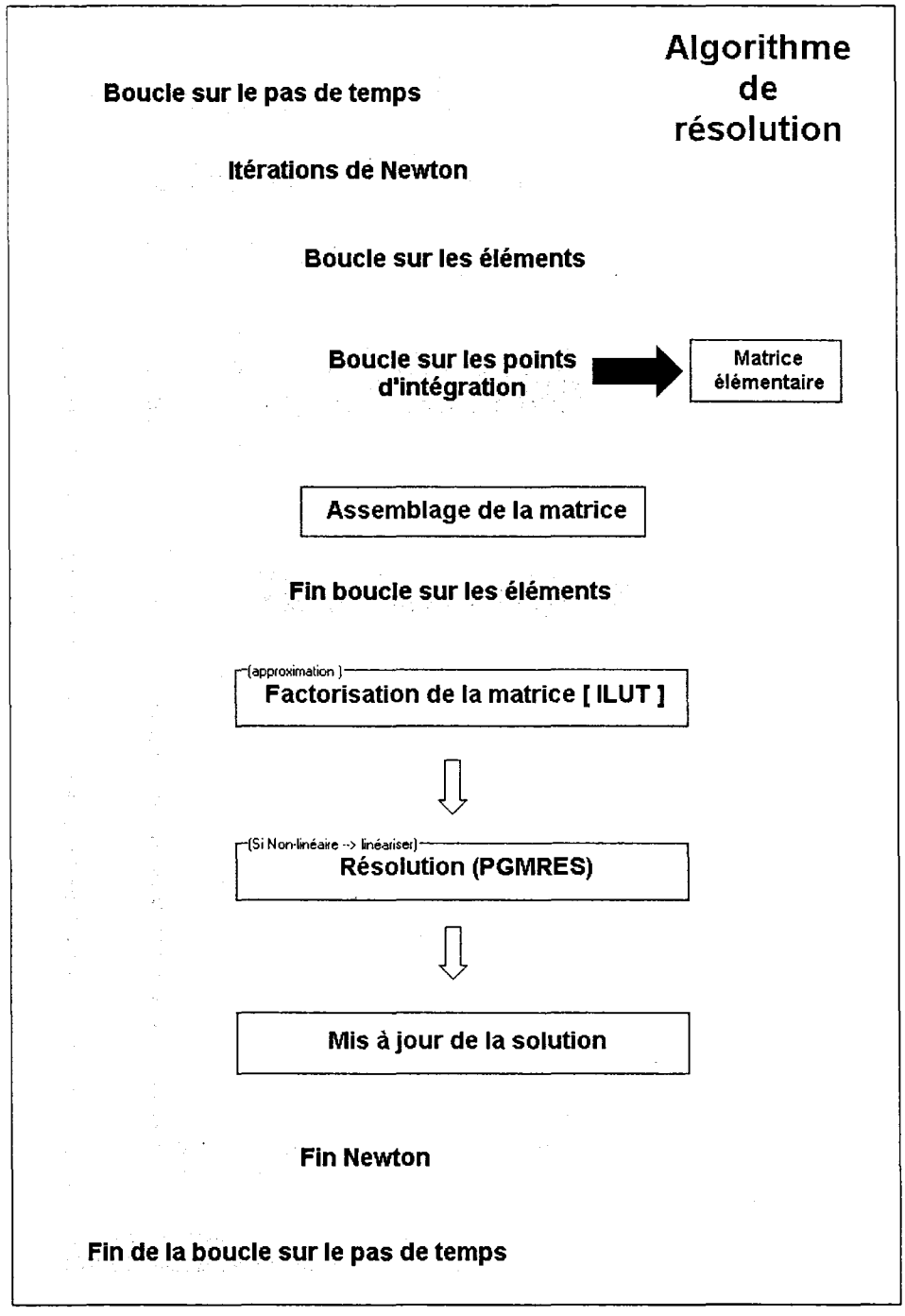


Figure 22 Algorithme de résolution et méthode des éléments finis standard

L'algorithme sus présenté est très bien adapté aux problèmes de petite et moyenne tailles. Cependant, dans la résolution des systèmes de grandes tailles tel que les problèmes multiphysiques, cet algorithme s'avère assez pesant en terme de mémoire et temps de calcul. En effet, dans ces cas, les opérations de factorisation, de produit matriciel et de résolution, sont de plus en plus coûteuses. Pour remédier à ces inconvénients, les modifications suivantes sont apportées à l'algorithme standard :

- Factorisation incomplète **ILUT**: Les matrice résultantes de la modélisation des problèmes étudiés, sont souvent très mal conditionnées. De ce fait, leur préconditionnement est inéluctable. Toutefois, l'obtention d'un préconditionneur convenable, passe inévitablement par une factorisation de la matrice. Une des alternatives envisageables est la méthode **LU**. Dans notre cas, l'application de la factorisation **LU** sous sa version standard dite complète, est très coûteuse. La méthode finalement adoptée est une factorisation approximative. Il s'agit d'une méthode basée sur une factorisation **LU** incomplète communément connue sous le nom **ILUT** [2].

ILUT

1. Do i = 1, n
2. Do k = 1, i-1
3. $A_{i,k} = A_{i,k} / A_{k,k}$
4. If $| A_{i,k} |$ is not too small then:
5. Do j = K + 1 , n
6. $A_{i,j} = A_{i,j} - A_{i,k} * A_{k,j}$
7. ENDDO
8. END IF
9. ENDDO
10. Drop small element in row $A_{i,*}$
11. ENDDO

- Calcul de résidu : un des fardeaux de l'algorithme présenté ci-dessus, est le coût du produit matrice-vecteur. Néanmoins, ce problème est contourné grâce à une méthodologie de résolution basée sur le calcul de résidu [voir 5.1.2.3].
- Parallélisation : en dépit des améliorations apportées par l'adoption de la factorisation incomplète ILUT et le calcul du résidu, la parallélisation de l'algorithme demeure inévitable vu la grande taille des problèmes traités. Outre le gain quantitatif apporté par la subdivision du domaine(temps/mémoire), la parallélisation offre une grande flexibilité de programmation. Mieux encore, cette subdivision offre l'avantage de refléter l'aspect multiphysiques des problèmes abordés.

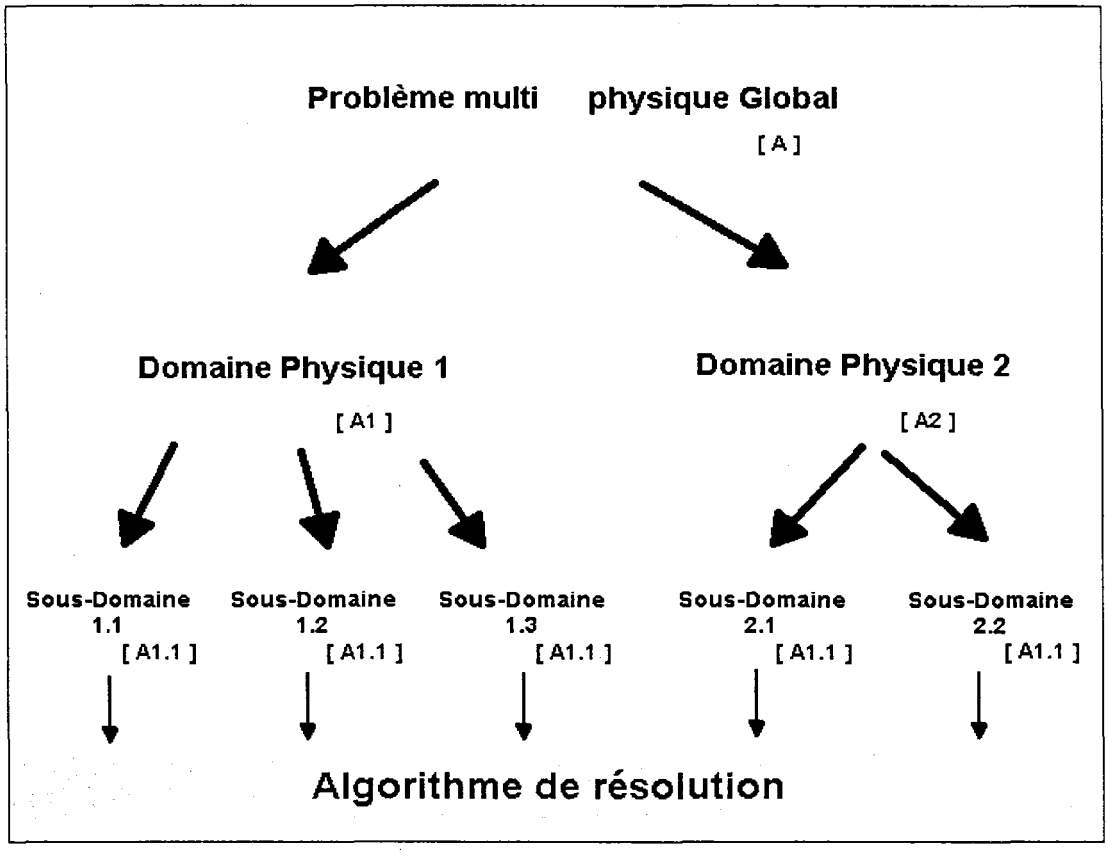


Figure 23 Parallélisation des problèmes multiphysiques

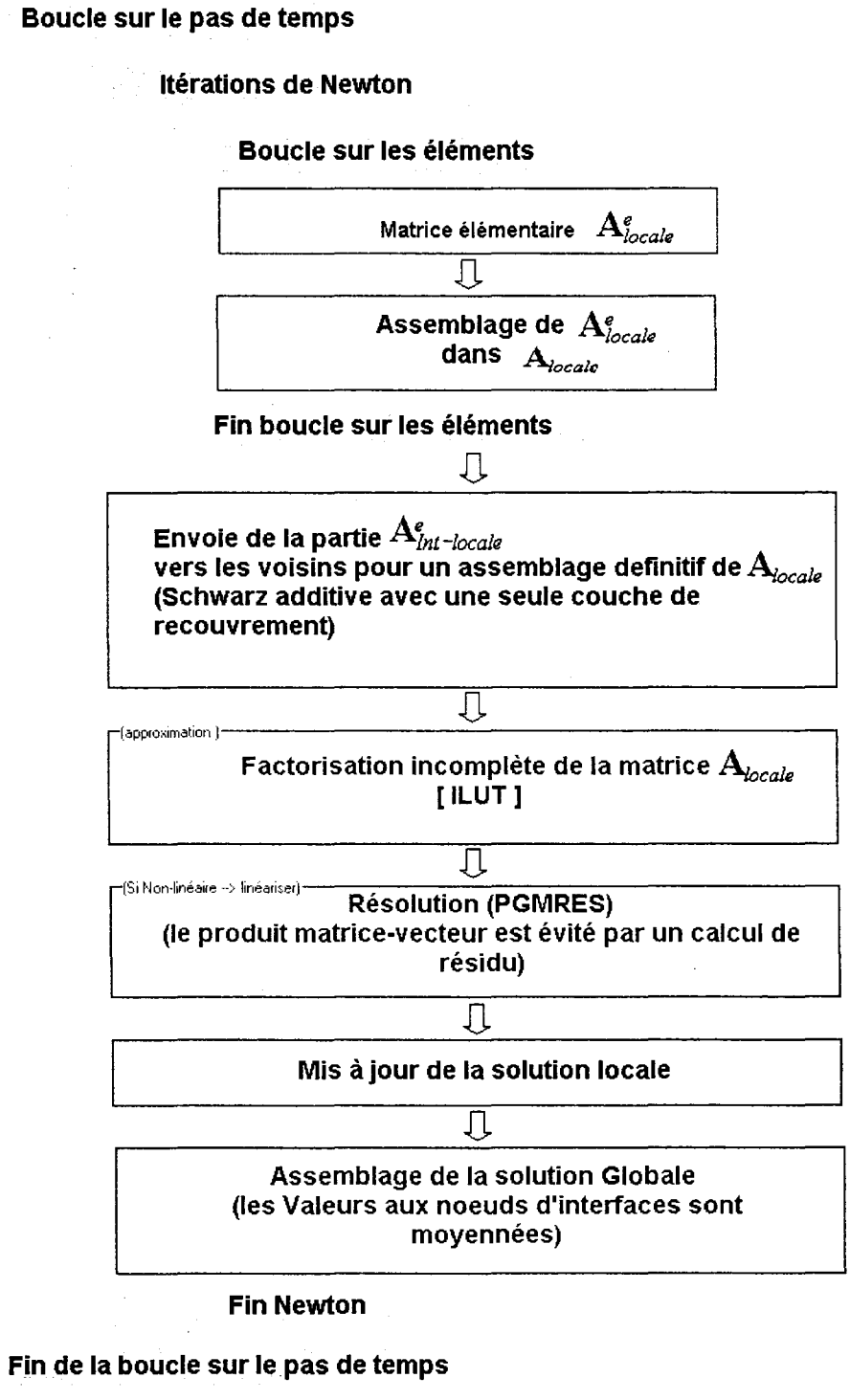


Figure 24 Algorithme de résolution adopté

6.2 Approche par couplage fonctionnel et interfaces graphiques

Le programme PFES est un programme général capable de résoudre une grande variété de problèmes physiques et multiphysiques. Pour le cas d'un problème de simulation d'interaction fluide-structure, l'algorithme de résolution est globalement basé sur les approches suivantes :

o Fluide :

- La méthode des éléments finis pour la discrétisation des équations gouvernantes.
- L'algorithme de décomposition de **Schwarz** et **PGMRES** pour la résolution.

o Mesh :

- Une formulation cinématique arbitraire Eulérienne-lagrangienne est utilisée pour adapter le mouvement de maillage aux déplacements de la structure.

o Structure :

- Les déplacements de la structure sont calculés à l'aide d'un schéma classique de **Newmark** [19].

o Couplage Fluide-Structure :

- La résolution du problème CFD-CSD_Mesh couplé, est basé sur un algorithme **Newton-PGMRES** global. Un préconditionnement par bloc est utilisé.

6.2.1 Interfaces graphiques

En plus du choix du type du problème, la forme des fichiers de sorties, les paramètres physiques et les paramètres de résolution, **PFES** offre aux utilisateurs une variété de choix de méthodes de stabilisation, d'interpolation et de capture de choc (shock capturing).

Ces paramètres et choix, ainsi que l'emplacement des fichiers de données sont fournis à PFES en forme de deux fichiers d'entrée. Vu la complexité des problèmes et le grand nombre de paramètres, ces fichiers généralement conçus en forme de cartes de données sont difficilement déchiffrables par les nouveaux utilisateurs.

Pour simplifier l'utilisation de PFES une interface graphique à été conçue et réalisée. Appelée « PFES INPUT WIZARD » cette interface permet de créer assez facilement les fichiers de paramètres.

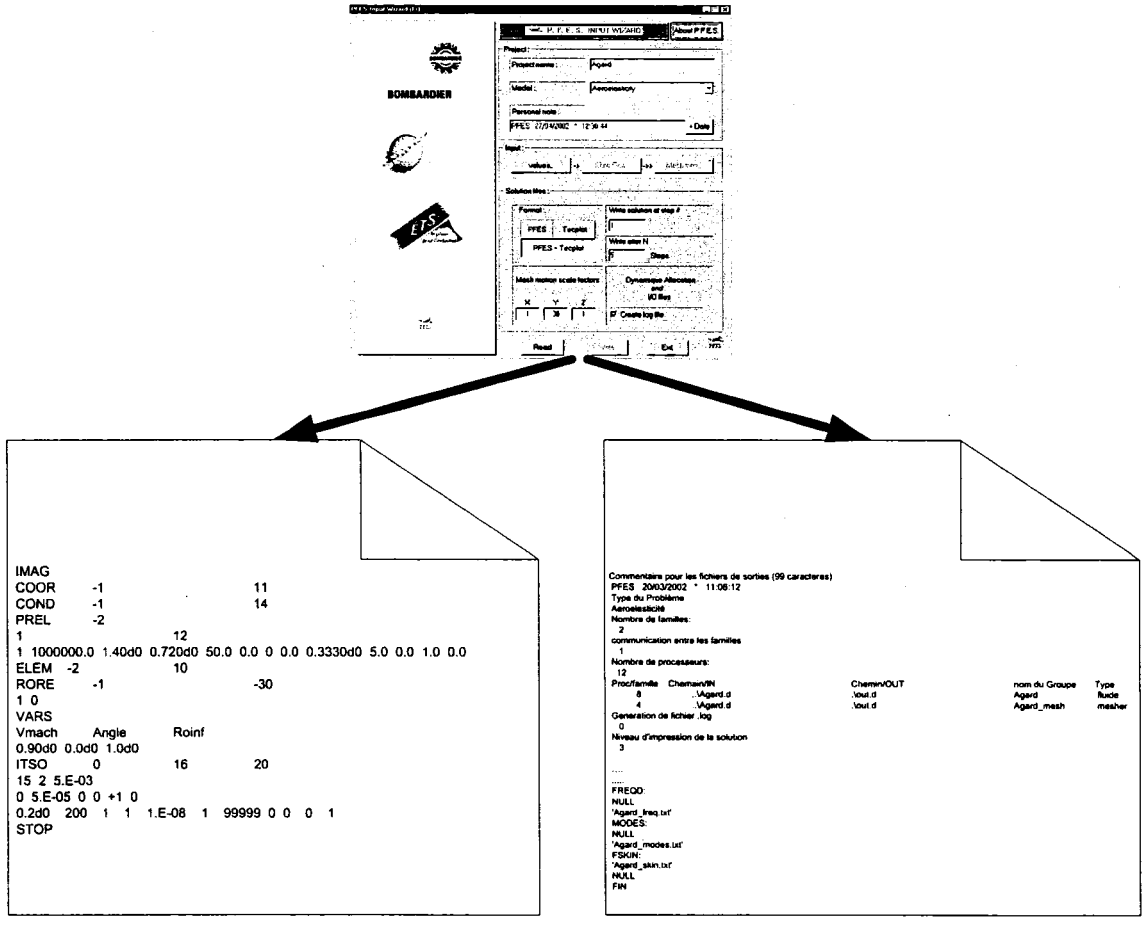


Figure 25 Création des fichiers par l'interface PFES INPUT WIZARD

Le rôle de cette interface est de créer les deux fichiers de paramètres, « process.ini » et « *.inp » respectivement de la gauche vers la droite. L'interface est aussi capable de lire des fichiers existants afin que l'utilisateur puisse changer un ou plusieurs paramètres.

Cette section explique brièvement l'utilisation de l'interface graphique.

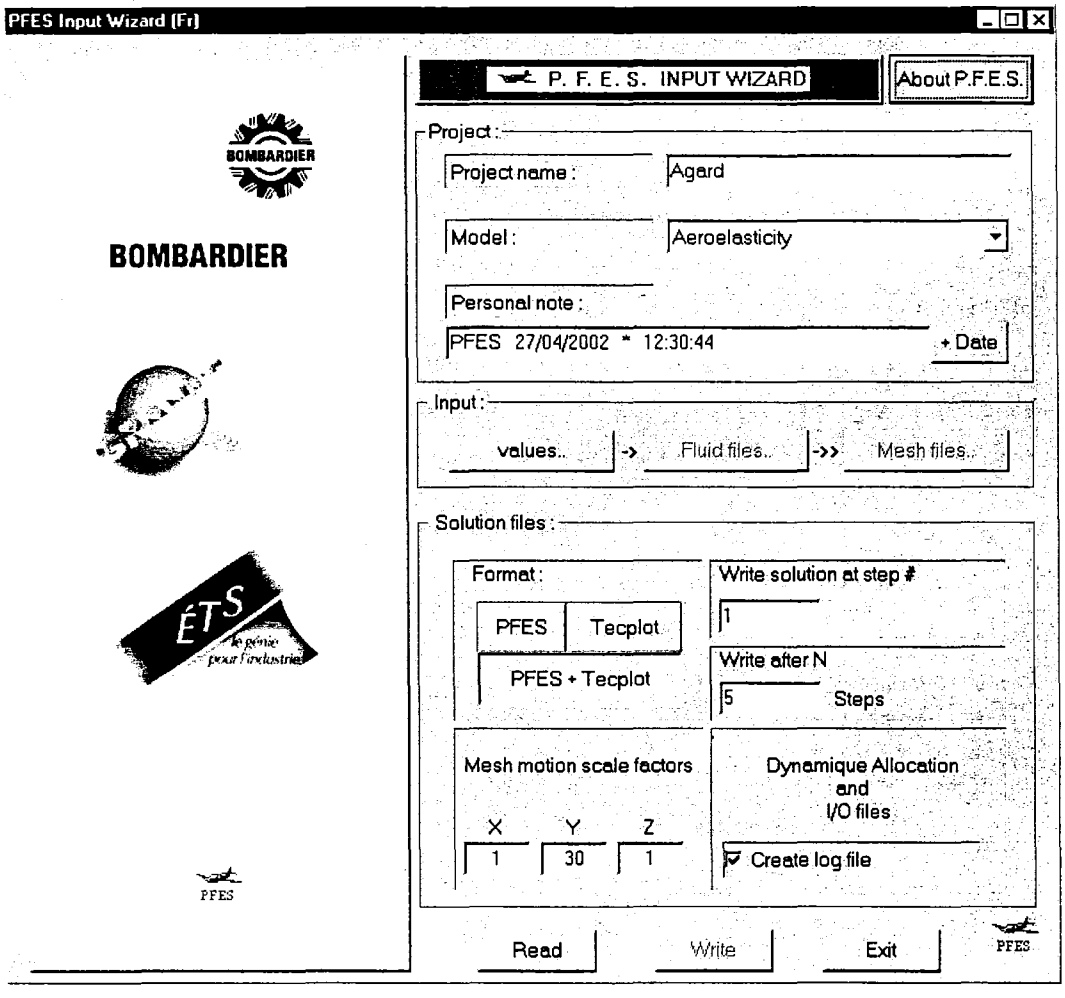


Figure 26 Interface graphique PFES INPUT WIZARD

Cette première fenêtre permet de sélectionner le type de problème à résoudre, tel que Euler, Navier stokes, mouvement de maillage ou encore aéroélasticité...

Les paramètres de sorties figurent aussi sur cette fenêtre. Ces paramètres sont :

- Le format des fichiers de sortie
- La fréquence d'écriture de la solution (Ex chaque 3 pas de temps ou chaque 10 pas de temps..)
- Le pas de début d'impression
- L'échelle d'impression
- La création ou non du fichier « *.log », fichier de l'historique des ouvertures des fichiers et de l'allocation dynamique de mémoire.

Pour changer un ou plusieurs paramètres dans des fichiers existants, il suffit d'appuyer sur le bouton **lire**. Les informations stockées dans les fichiers seront alors chargées par l'interface. L'utilisateur n'a qu'à changer les valeurs désirées et réimprimer les fichiers de nouveau.

Une fois l'utilisateur choisit le type de problème, le bouton « Values » devient actif. Ce bouton permet d'ouvrir la fenêtre des paramètres physiques.

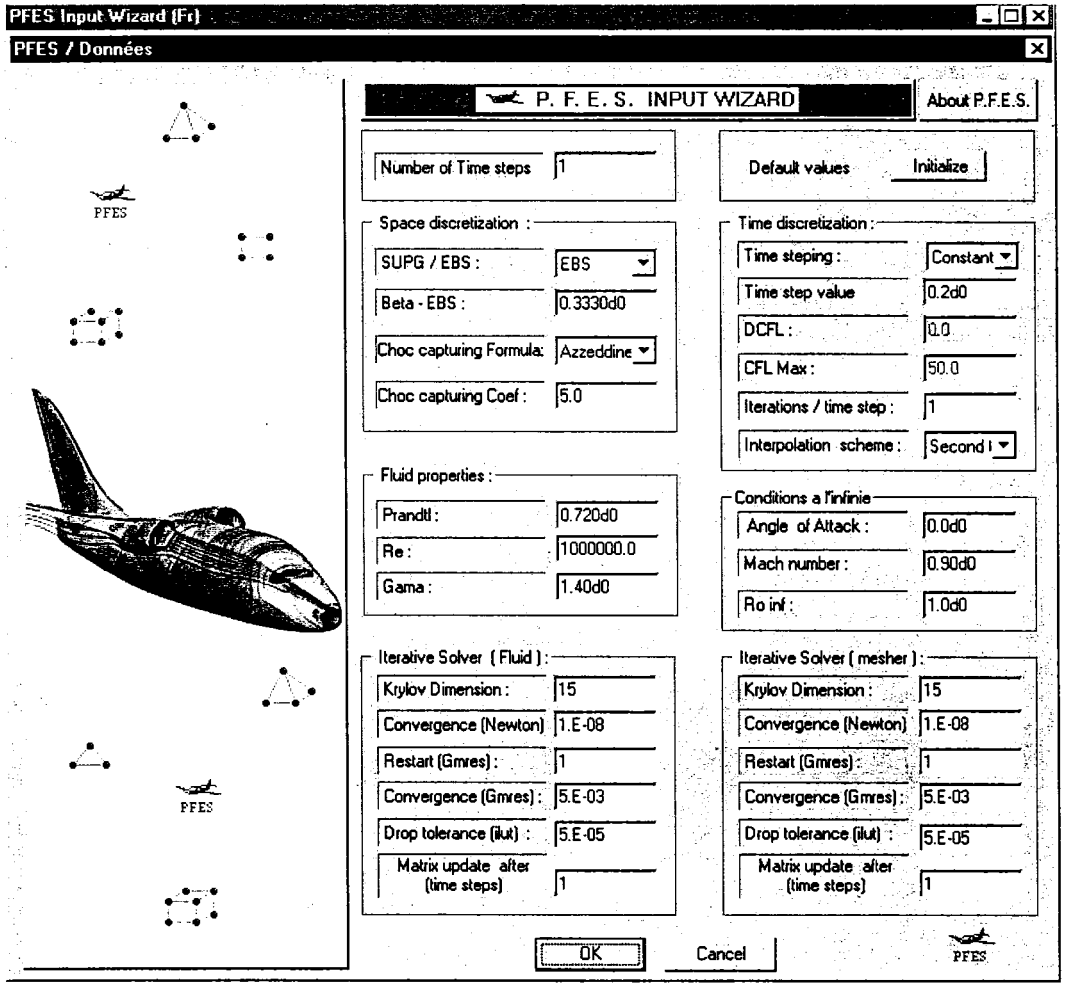


Figure 27 Fenêtre de saisie des paramètres physiques

La fenêtre des paramètres physiques permet la saisie et la modification du nombre de pas de temps, la discrétisation spatiale, la discrétisation temporelle, les propriétés du fluide, les conditions à l'infini et les paramètres de résolution.

Dans la zone « discrétisation spatiale » l'utilisateur peut sélectionner la méthode SUPG ou la méthode EBS pour la stabilisation et choisir entre la méthode de Azzeddine et la méthode de Behr pour le shock capturing. Pour la discrétisation temporelle, l'utilisateur

a le choix entre un pas de temps fixe ou variable, et une interpolation de 1er ou 2^{ième} ordre.

Une fois l'utilisateur valide ses choix, la première page réapparaît et le bouton « Fichiers Fluide » devient actif.

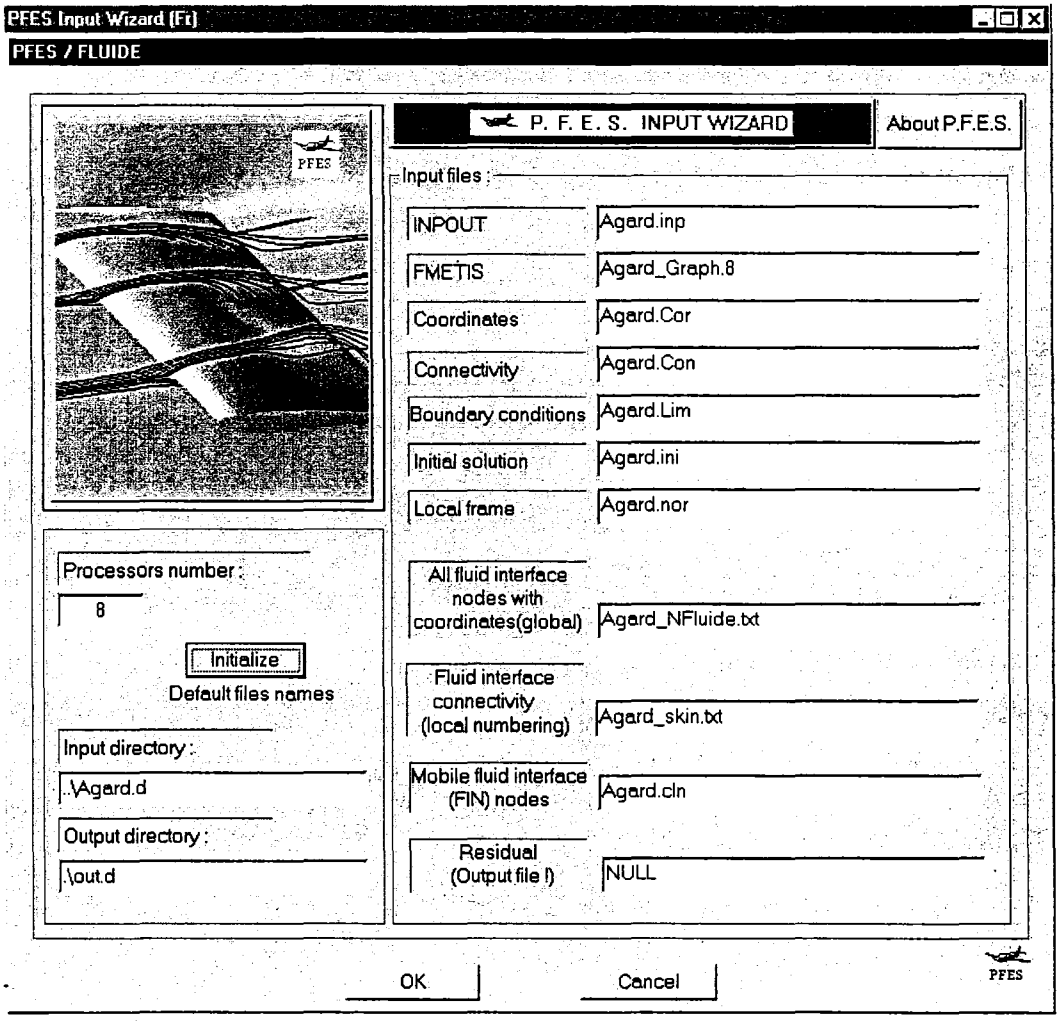


Figure 28 Fenêtre de saisie des noms de fichiers de données du domaine fluide

La fenêtre de saisie des noms de fichiers de données propres au domaine fluide permet à l'utilisateur d'expliciter le nombre de processeurs qu'il désire assigner à la famille en

question, d'indiquer les chemins des fichiers d'entrées et de sorties. Ces informations servent à la création du fichier « process.ini », le fichier d'initialisation du code PFES.

Dépendamment du type de problème, d'autres fenêtres de saisie des chemins de fichiers peuvent apparaître. La figure suivante représente la fenêtre dédiée aux domaines Mesh et structure.

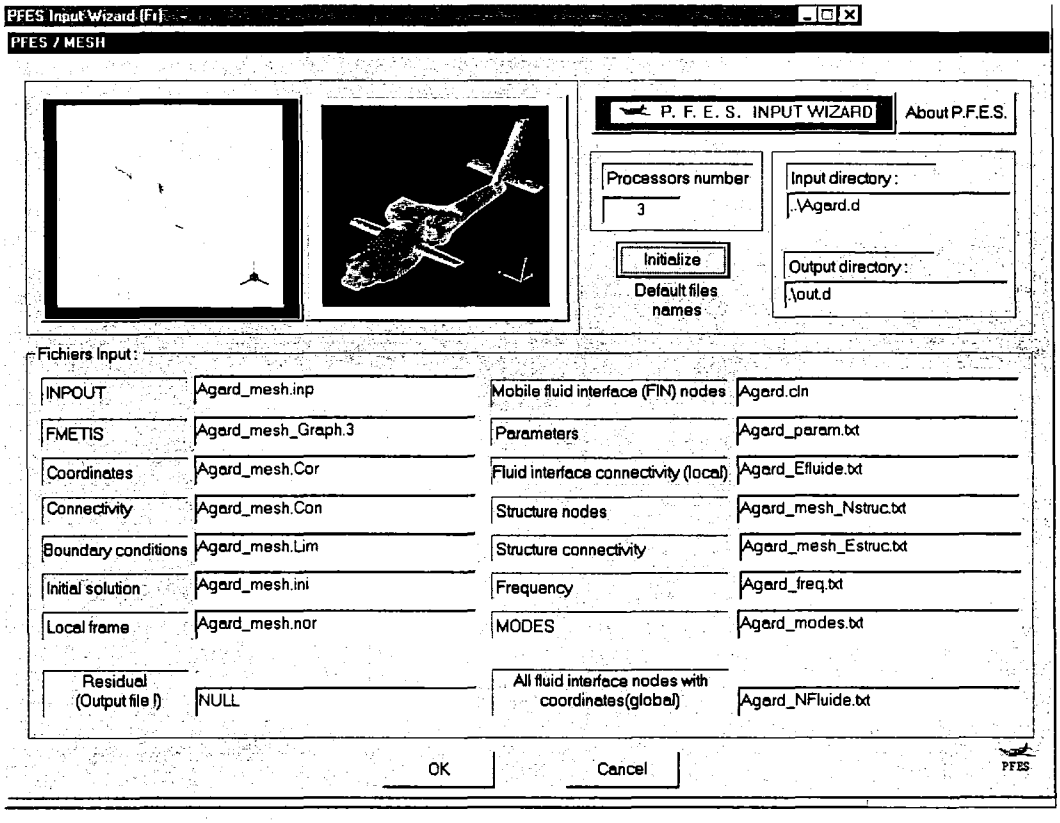


Figure 29 Saisie des noms de fichiers de données des domaines Mesh et structure

Cette fenêtre présente les mêmes fonctions que celles de la fenêtre précédente. Les zones de textes s'activent et se désactivent dépendamment du type du problème.

6.3 Simulations et résultats

Dans cette section les résultats des études de performance ainsi que des résultats des simulations en mécanique des fluides et en aéroélasticité seront présentées et discutées.

6.3.1 Tests de performance

Plusieurs tests de performances ont été réalisés. Le but est de mettre en évidence l'importance du parallélisme, et d'étudier les différents facteurs qui entrent en jeu afin d'optimiser au maximum le rendement du programme.

6.3.1.1 Problème non couplé

Le premier test consiste à étudier le speed-up du programme PFES sur la machine SUN **Enterprise 6000** dans le cas d'un problème de simulation d'écoulement sur l'aile Onera.

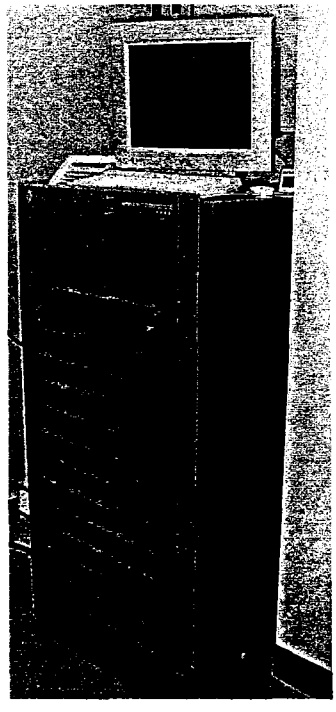


Figure 30 Enterprise 6000

Tableau I

Les caractéristiques de la machine Enterprise

Fonction	Serveur de calcul
Type de machine	Enterprise 6000
CPU	14 x UltraSPARC 167 MHz
Mémoire	1.7 Go
Capacité de Disque	4.2 Go
Système d'exploitation	Sun Solaris 7
Périphériques	CDROM, Tape backup

Tout au long de ce test, on a utilisé le même maillage de l'aile Onera (15K nœuds, donc environ 75K équations). Le nombre de pas de temps est de 150.

Une variation du nombre de processeurs de deux à douze, a donné les temps de calcul suivants

Tableau II

Temps de calcul sur la machine Enterprise

Machine : Enterprise 6000 Système d'exploitation : Solaris (Onera, 150 pas)		
Nb. Proc.	***Temps (h)***	***temps (s)***
2	6,303	22689,3936
3	4,568	16443,7952
4	3,261	11739,32018
6	2,130	7669
8	1,698	6112,760087
10	1,374	4946,126237
12	1,186	4270,004697

Le graphique suivant illustre l'importance de l'augmentation du nombre de processeurs sur le temps d'exécution :

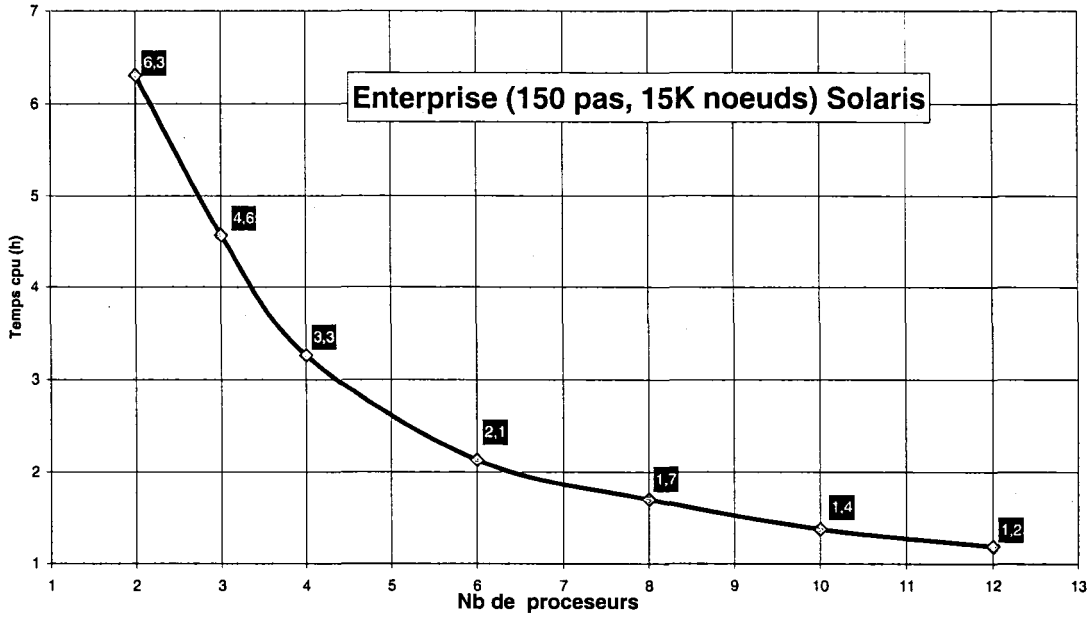


Figure 31 Temps de calcul sur Enterprise

On remarque qu'en augmentant le nombre des processeurs, le gain en terme de temps de calcul est de moins en moins important. Cela est dû à la taille du problème. Plus le problème est de grande taille, plus il est intéressant de le deviser et d'utiliser un nombre croissant de processeurs. Ceci s'explique par l'augmentation du nombre de nœuds aux interfaces et le coût élevé de la communication conséquente.

L'exemple suivant illustre la problématique sus-indiquée :
Soit le maillage structuré de 25 nœuds.

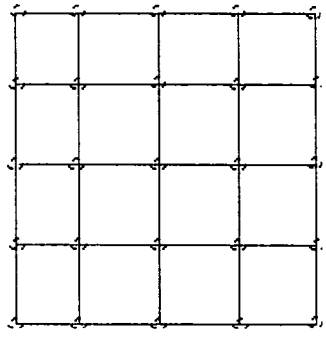


Figure 32 Maillage structuré de 25 nœuds

Si on décide de deviser ce domaine en deux sous-domaines d'une manière très simple:

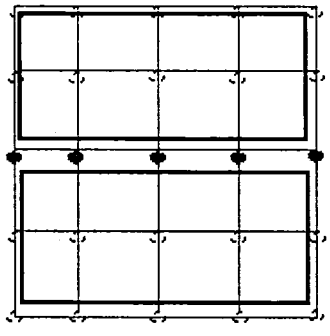


Figure 33 Nœuds à l'interface de deux sous-domaines

On a alors 5 nœuds sur l'interface entre les deux sous-domaines, ce qui représente le $1/5^{ième}$ du nombre de nœuds total. Ce qui va générer des coûts de communication.

Maintenant si on décide de deviser le domaine en 4 :

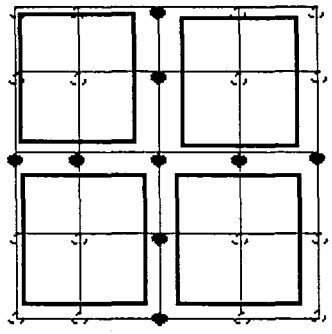


Figure 34 Nœuds aux interfaces des quatre sous-domaines

Le nombre des nœuds aux interfaces augmente pour atteindre neuf dans ce cas. Ce qui représente environ le 1/3 du nombre total, et donc le coût de la communication et de plus en plus important. En supposant que PFES a une efficacité de 100% avec 2 processeurs, on peut tracer la courbe de speed-up [15] (Voir la section 1.1.2) suivante :

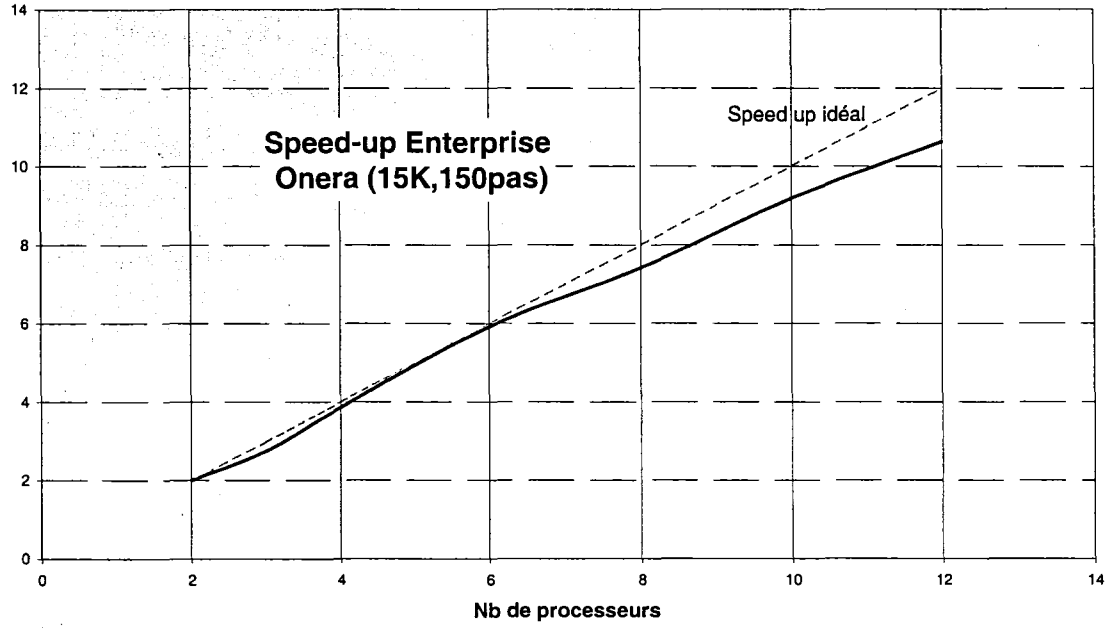


Figure 35 *Speed up* sur la machine Enterprise

L'efficacité des programmes parallèles est influée aussi, par l'architecture de la machine et le système d'exploitation utilisé. Des tests ont été réalisés pour étudier l'influence de ces facteurs sur le temps de calcul. Pour ce faire le même test a été lancé sur la machine **Enterprise** et sur un cluster appelé **Andalous** conçu et réalisé par le groupe **Granit** (Annexe 1).

La figure suivante illustre les performances de **PFES** sur différentes architectures et systèmes d'exploitation

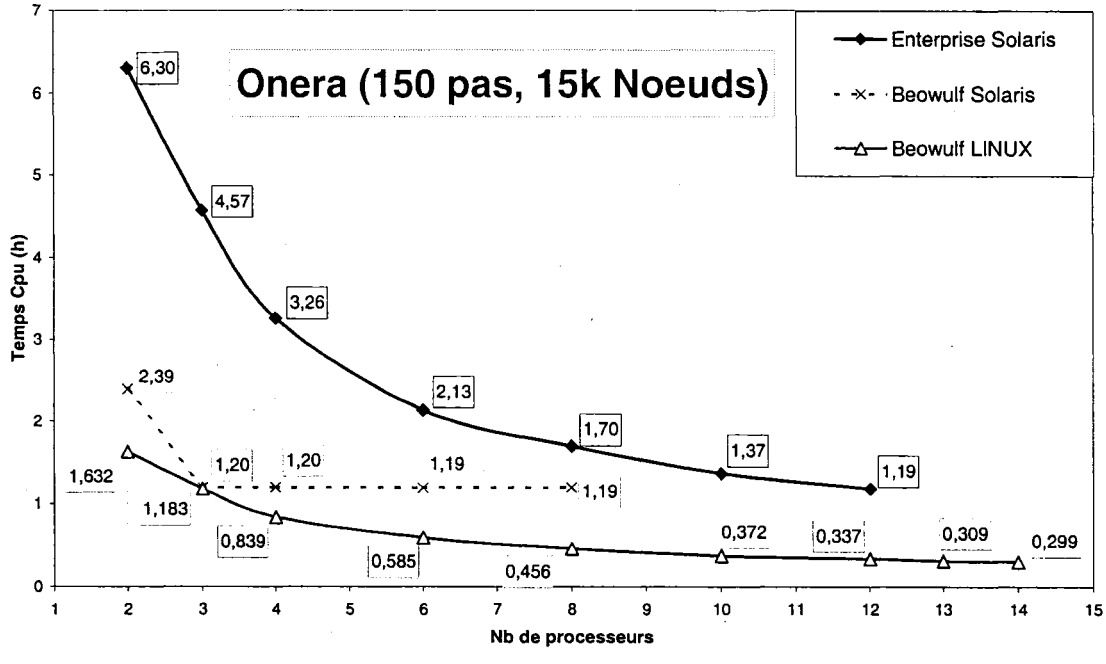


Figure 36 Temps de calcul en fonction du nombre des processeurs

Bien que, le coût de communication sur le cluster soit élevé, le gain apporté par la vitesse des processeurs réduit considérablement le temps de calcul. Les performances de la machine Beowulf sont alors, nettement supérieures à celles des autres machines qui sont à notre disposition.

En se basant sur les résultats évoqués ci-dessus, le Beowulf se révèle comme étant l'outil optimal qui permet d'arriver au bout de ce projet.

Un deuxième maillage plus fin (**135K** nœuds, **614k** éléments) de l'aile Onera à été utilisé pour l'étude de performance sur la machine **Andalous** (Annexe1, machiunes).
Les résultats de ces tests sont les suivants :

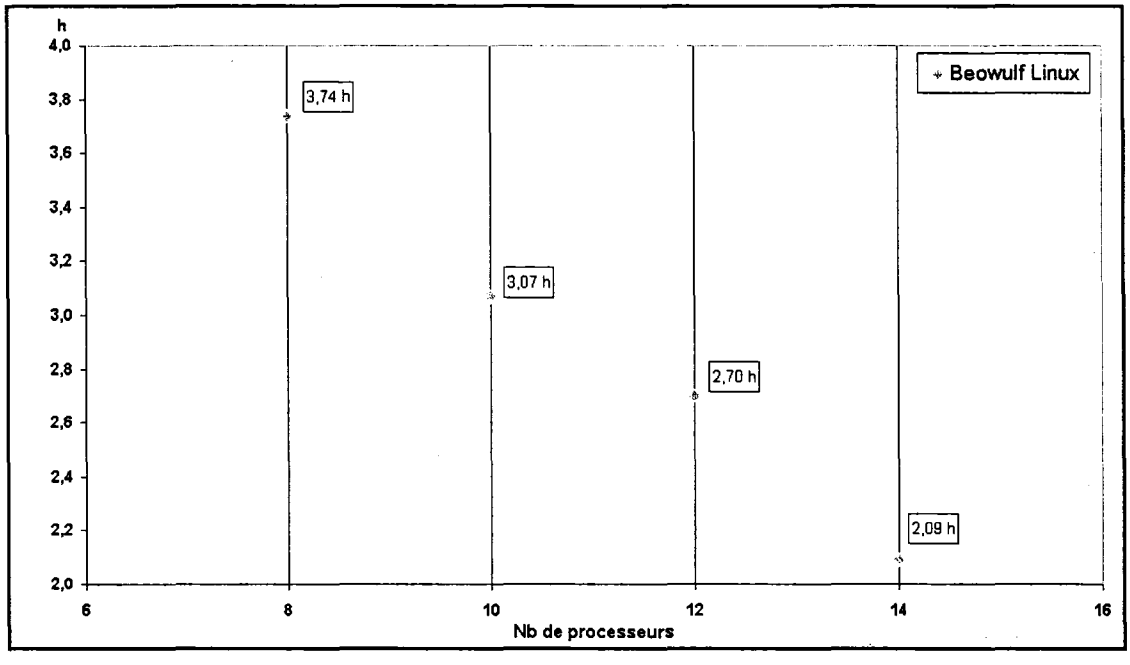


Figure 37 Temps de calcul en fonction du nombre de processeur pour un maillage de 135K noeuds

La courbe donnée par la figure ci-dessus montre une forte décroissance linéaire du temps de calcul avec l'augmentation du nombre de processeurs utilisés. Ce qui voudrait dire que pour la taille du problème considéré, le nombre de processeur optimal n'est pas atteint.

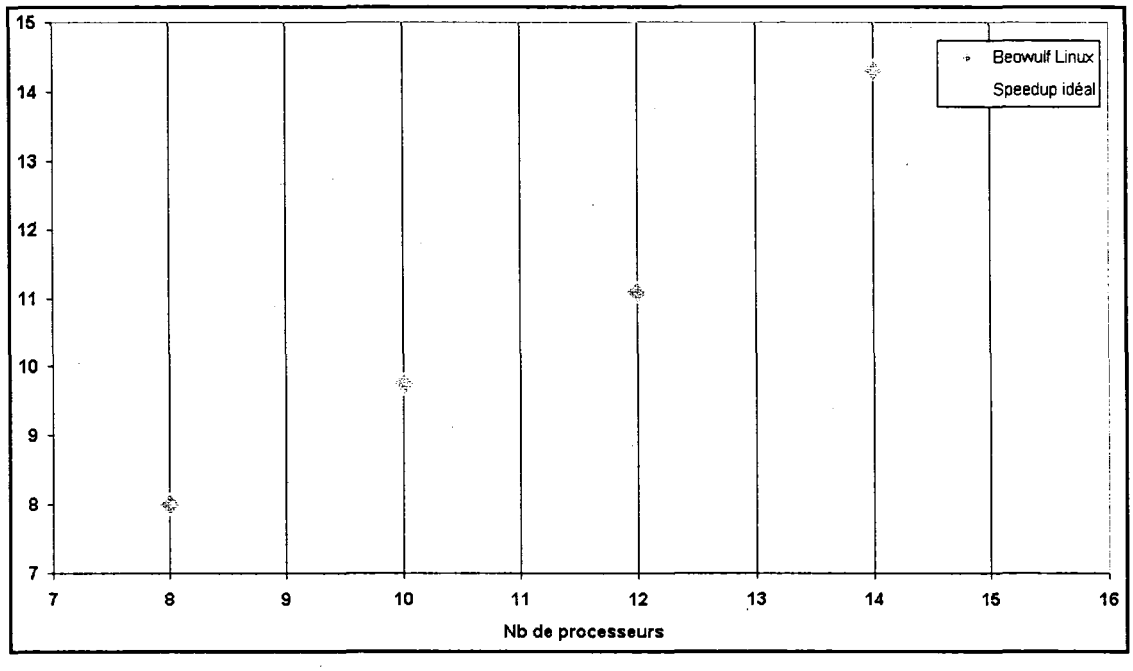


Figure 38 Speedup dans le cas d'un maillage de 135K noeuds

La courbe donnée par la figure ci-dessus, montre que l'évolution du Speedup en fonction de nombre de processeurs est assez proche du comportement théorique. Le comportement faiblement non linéaire peut être expliqué par la légère différence du nombre d'éléments par processeur.

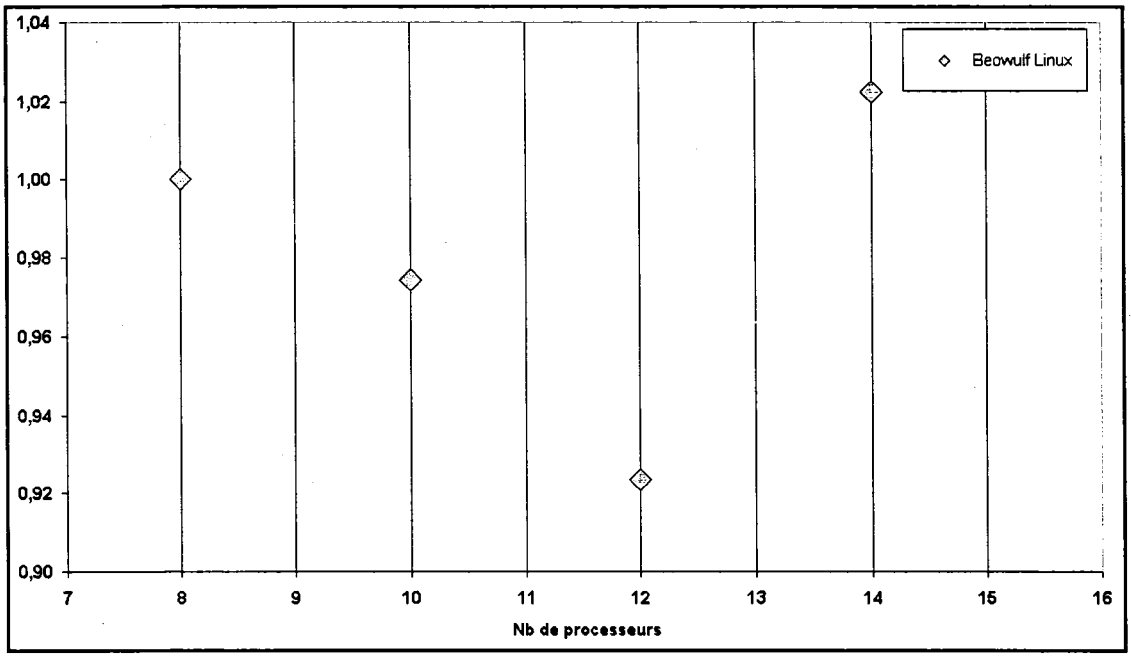


Figure 39 Efficacité dans le cas d'un un maillage de 135K noeuds

La courbe d'efficacité en fonction du nombre de processeurs montre que pour les nombres de processeurs utilisés l'efficacité est supérieure à 92% et atteint son maximum pour 14 processeurs.

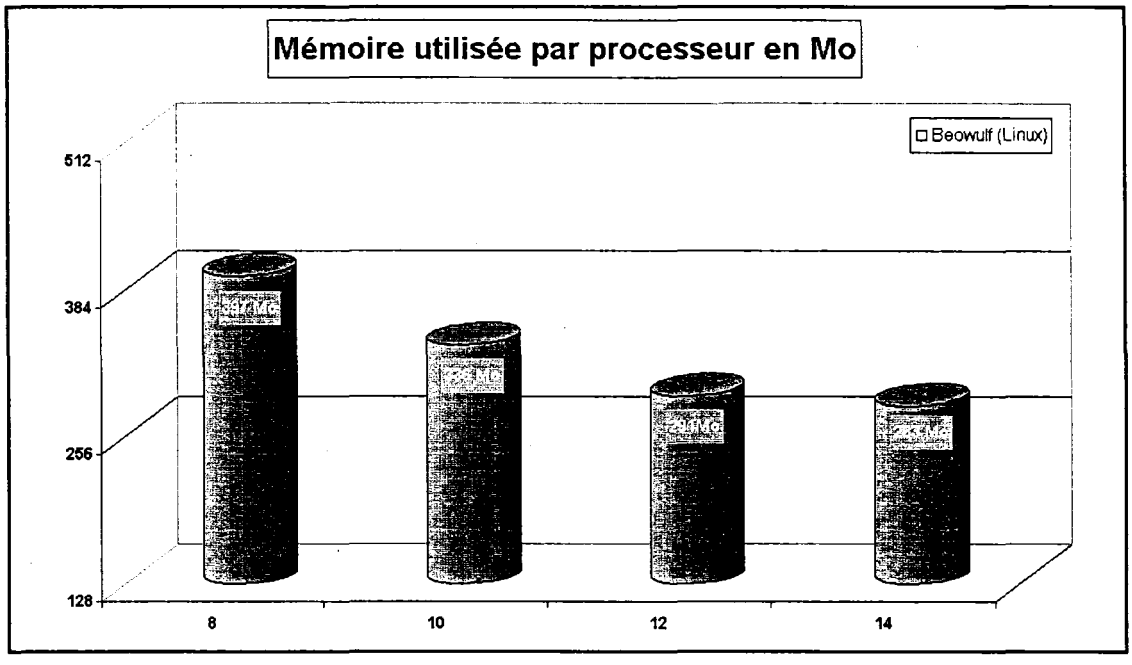


Figure 40 Mémoire utilisée par processeur

la décroissance de la mémoire utilisée par processeur est de moins en moins spectaculaire quand le nombre de processeurs augmente. Ceci est du à l'augmentation du nombre de nœuds aux interfaces.

Une deuxième série de tests de performance ont été menés sur un nouveau cluster appelé **Thunderbird** (Annexe1, Machines) qui a servi pour l'étude du phénomène de flottement sur l'aile Agard.

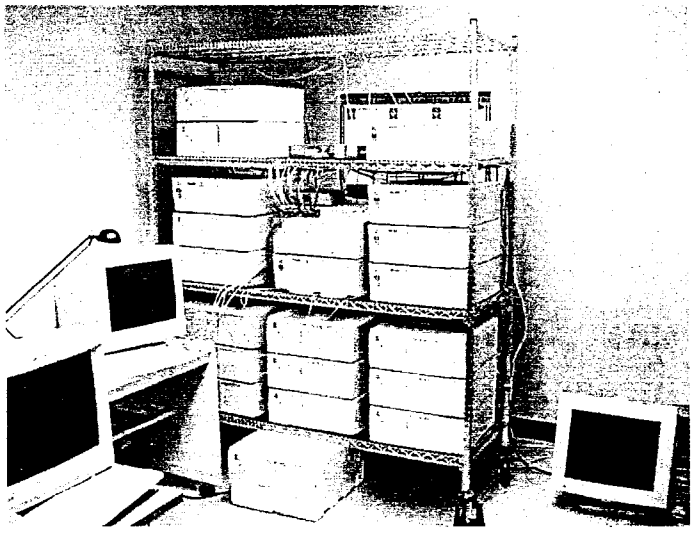


Figure 41 Le cluster Granit composé de Andalous et Thunderbird

Comme le prouve la série de tests précédente, l'importance de parallélisme est d'autant plus perceptible que la taille du problème est imposante.

Un problème d'écoulement autour d'une aile d'avion englobant un maillage de 40 K nœuds est traité. Ce maillage nous garantit des résultats en Euler assez précis en vue de comparaison. La grosseur du problème permet de mettre en épreuve la stabilité de la machine Thunderbird ainsi que la robustesse du code PFES en lui-même.

Les résultats des tests sont illustrés sur les graphiques, pour plus de détails sur les mesures voir (Annexe 3, Temps de calcul).

Dans une première phase, un problème de CFD classique a été traité pour mesurer les temps d'exécution CPU et les temps réels. Ces mesures ont été prises pour différentes décompositions du domaine (de 3 à 16 sous-domaines).
Le temps d'exécution CPU est le temps écoulé pendant les calculs sans prendre en considération, ni le temps de la communication, ni le temps d'attente appelé aussi le *temps mort.*

Le temps réel est le temps qui sépare l'instant où le code a été lancé, à l'instant où la tache demandée est accomplie. En d'autres mots, le temps réel est la période pendant laquelle l'utilisateur doit attendre pour avoir les résultats désirés.

Le test consiste à simuler l'écoulement sur l'aile Agard dans les conditions suivantes :

- Nombre de pas de temps : 100
- Type du pas de temps : fixe
- Nombre d'itérations de Newton par pas : 1
- La matrice est calculée et factorisée à chaque pas de temps
- Un maillage de 37965 Nœuds, 177042 éléments.

Seul le nombre de processeurs utilisé change d'un test à un autre.

Les éléments sont partagés entre les processeurs le plus équitablement possible. Voici le nombre d'éléments attribués à chaque processeur dans les différents cas :

Tableau III

Répartition des éléments

Nombre de Processeurs	**Nombre d'éléments par processeur**
3	59221, 58857, 58964
4	45562, 43173, 44344, 43963
5	35639, 36042, 35390, 35591, 34380
6	29534, 29531, 29706, 29356, 30230, 28685
7	25995, 25065, 24895, 25352, 24555, 25242, 25938
8	22649, 21712, 21883, 22075, 21999, 22669, 22329, 21726
9	19420, 20244, 20242, 19236, 19323, 19914, 19216, 19222, 20225
10	17244, 17188, 18148, 18207, 17743, 17876, 18172, 17645, 17447, 17372
11	16070, 15658, 16447, 15991, 16374, 15975, 16033, 16577, 15808, 15698, 16411
12	14956, 15121, 14696, 14631, 15105, 14337, 14585, 15136, 14379, 14812, 14491, 14793
13	13369, 14026, 13251, 13872, 13862, 13289, 13463, 14026, 13563, 13536, 13528, 13231, 14026
14	12415, 12302, 12719, 12721, 12786, 13021, 12308, 13024, 12281, 12329, 13006, 13019, 12413, 12698
15	11467, 11526, 11611, 11664, 11910, 11552, 11581, 12156, 12134, 12142, 11489, 12074, 12126, 12134, 11476
16	11226, 10792, 10747, 11365, 10862, 10981, 11360, 11106, 11346, 10864, 11363, 10840, 10913, 10923, 10986, 11368

La figure suivante illustre l'attribution équitable des nombres d'éléments pour différentes décompositions :

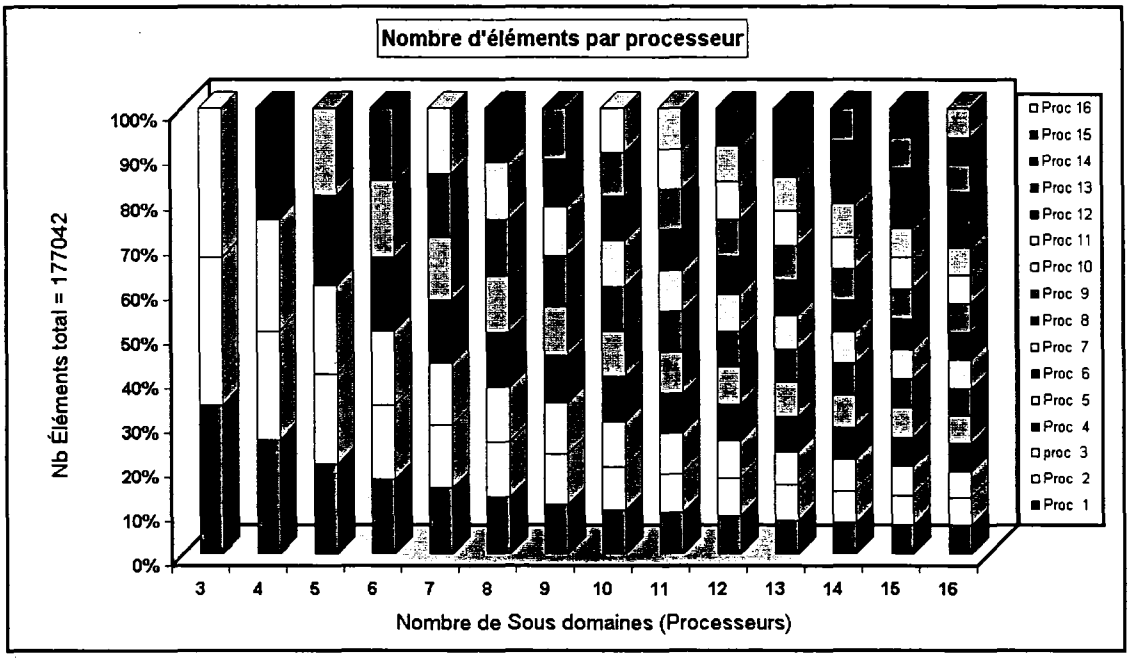


Figure 42 Distribution des éléments entre les processeurs

À remarquer que pour chaque décomposition, la somme des éléments est toujours égale à 177042. Par contre, la somme des nombres de nœuds locaux change d'une décomposition à une autre. Cette variation est causée par les nœuds d'interfaces entre les sous domaines qui sont comptés plus qu'une fois.

Tableau IV

Nombre Global de Noeuds

NB SOUS Domaines	$\sum$ DES NOEUDS LOCAUX	NB SOUS Domaines	$\sum$ DES NOEUDS LOCAUX
3	39034	10	40526
4	39319	11	40700
5	39460	12	40760
6	39781	13	41230
7	39941	14	41411
8	40131	15	41543
9	40430	16	41585

La figure suivante illustre cette augmentation fictive du nombre de nœuds, qui se traduit par une augmentation du nombre d'équations à résoudre. Jusqu'à un certain nombre de sous domaines, le coût en terme de temps de calcul engendré par cette augmentation reste négligeable devant le gain apporté par l'utilisation de multiples processeurs.

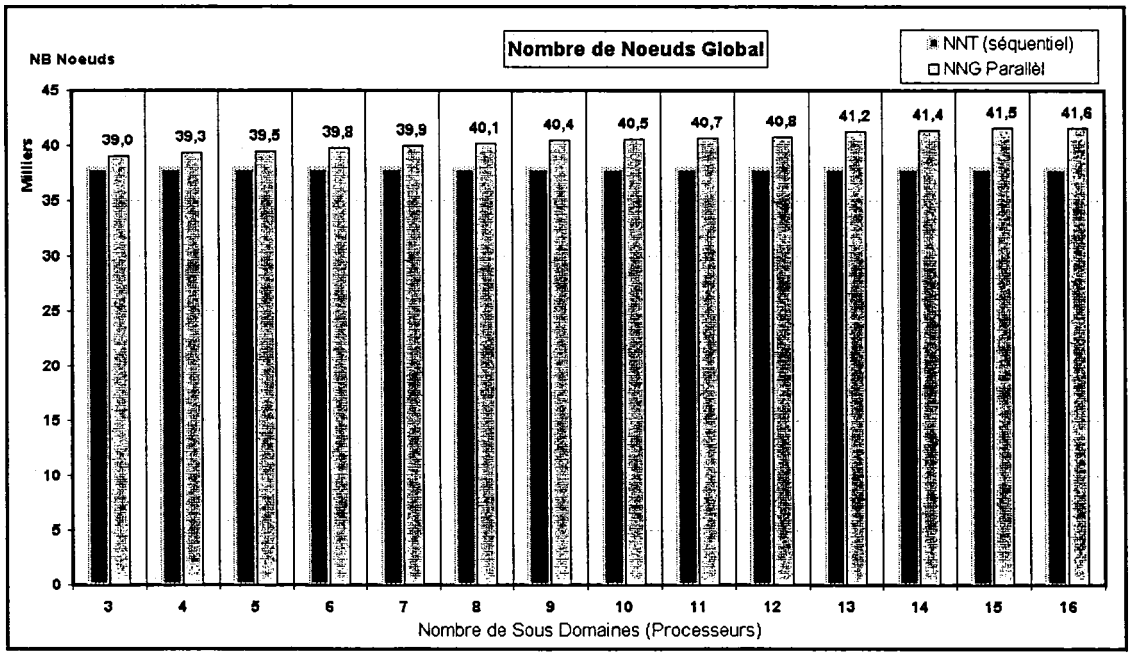


Figure 43 Nombre de nœuds aux interfaces des sous domaines

Le nombre de nœuds aux interfaces augmente avec le nombre de sous-domaines ce qui est totalement logique. Cette augmentation reflète aussi une augmentation des données communiquées. L'utilisation d'un très grand nombre de processeurs (dépendemment de la finesse du maillage), engendrait une augmentation excessive du nombre de nœuds aux interfaces et de la quantité de valeurs échangées. Dans ce cas, le temps de calcul et le temps de communication provoqués par cette augmentation peuvent influer dramatiquement les performances. Toutefois, pour un maillage de 38K nœuds un nombre de sous-domaines de l'ordre de 16 est loin de poser un tel problème.

La figure suivante montre qualitativement la diminution du temps de calcul CPU des processeurs avec le nombre des sous-domaines :

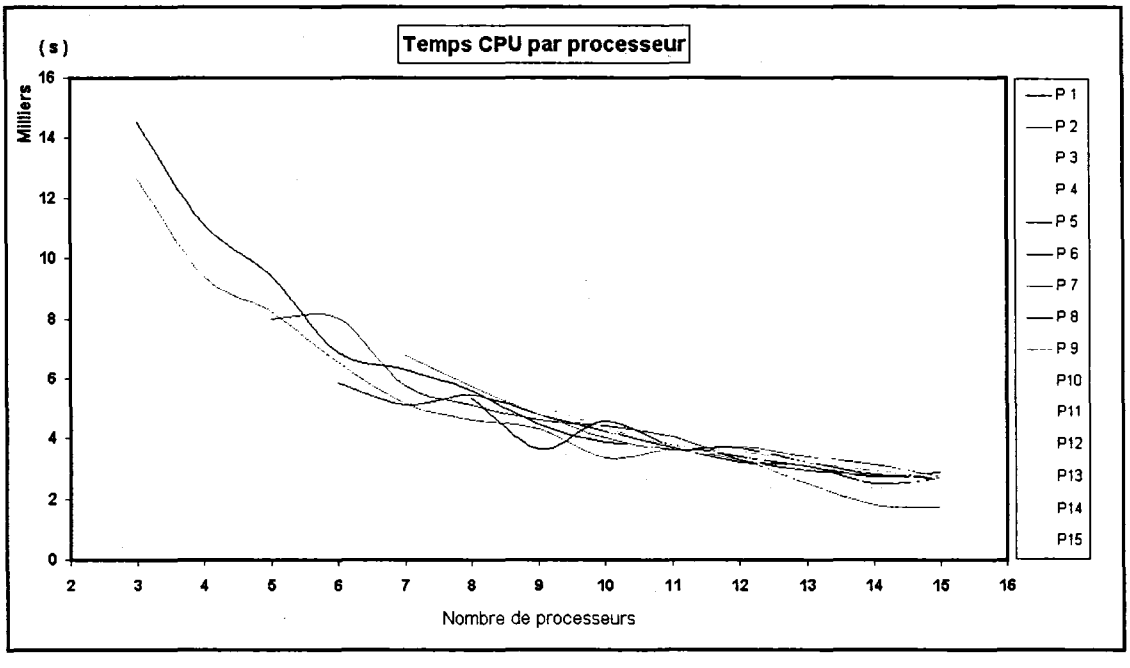


Figure 44 Temps de calcul CPU en fonction du nombre des sous-domaines

Idéalement, pour une décomposition donnée, les temps CPU des processeurs doivent êtres identiques. Cependant, on remarque une légère différence entre les performances individuelles des processeurs. Cette différence peut être expliquée par la variation du

nombre des nœuds locaux selon les processeurs. En plus, l'écriture dans les fichiers de sortie ralentisse le processeur numéro 1, appelé leader du groupe.

Le but des tests est de mesurer le *speed up* et l'efficacité du duo, la machine Thunderbird et le code PFES. Une façon de voir le *speed up* CPU global et de calculer ce dernier à partir du plus grand temps CPU des différents processeurs. En effet à cause des communications les processeurs doivent êtres synchronisés, le temps de calcul CPU global est alors déterminé par le processeur le moins rapide.
La figure suivante présente le temps CPU ainsi calculé :

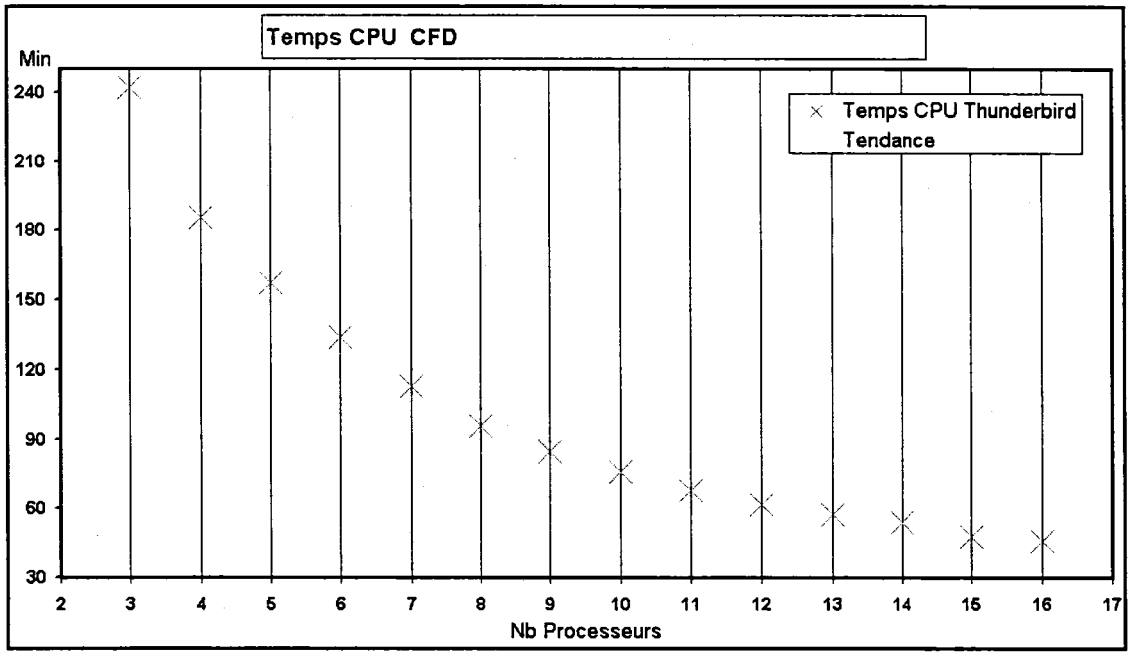


Figure 45 Temps CPU pour un problème non couplé

La figure ci-dessus illustre clairement le gain en terme de temps CPU. En effet, pour 100 pas, le temps de calcul avec 3 processeurs est de 240 minutes, alors qu'avec 16 processeurs il est réduit à moins de 50 minutes. Sachant que, pour la recherche du point

neutre de la stabilité aéroélastique, 1800 pas de temps sont généralement nécessaires. En peut alors imaginer le gain apporté par la parallèlisation pour ce genre de simulations. Chaque processeur de la machine Thunderbird dispose de 512 Mo de mémoire vive. Cette quantité de mémoire limite le nombre minimal de sous-domaines à 3. Si uniquement 2 processeurs sont utilisés, le temps de calcul sera affecté par la lecture/écriture sur le disque dur (swap).

La figures suivante donne une idée sur la mémoire utilisée, par processeur, pour chaque décomposition.

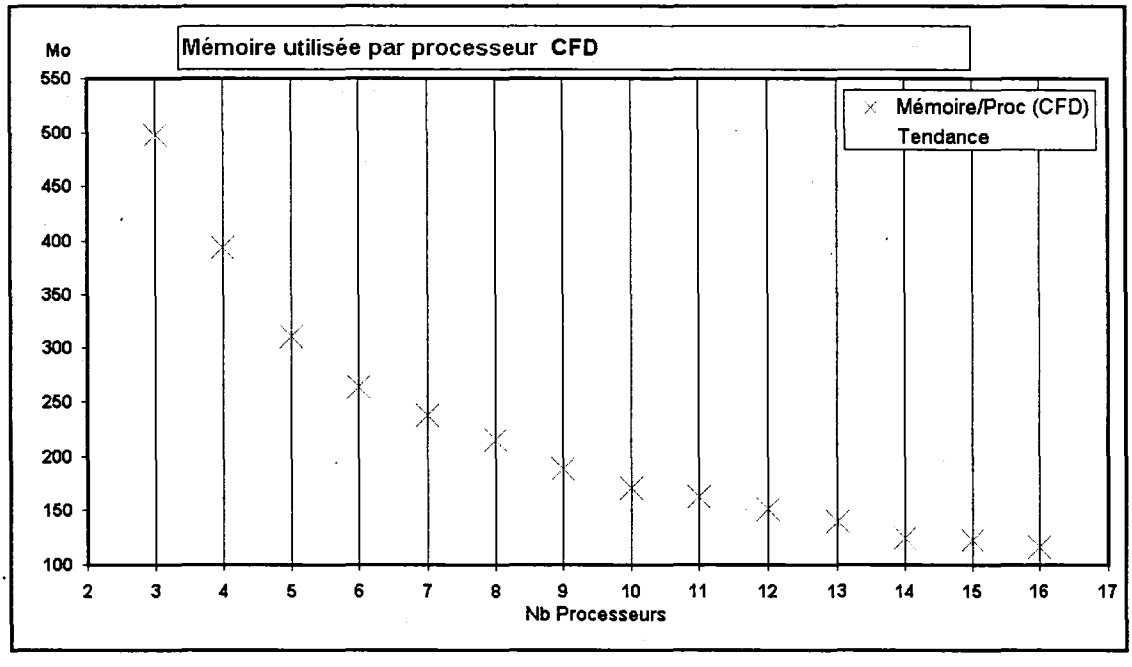


Figure 46 Mémoire utilisée par processeur en fonction de nombre des sous-domaines

La mémoire localement utilisée est nettement réduite par l'utilisation de multiples processeurs. Outre la rapidité des processeurs, cette réduction représente un avantage majeur des clusters par rapport aux machines parallèles à mémoire partagée. En effet, sur le cluster, chaque processeur dispose de sa propre mémoire.

Regardons maintenant, le comportement de la mémoire globale utilisée en fonction du nombre de sous-domaines.

Les processeurs n'utilisent nécessairement pas la même quantité de mémoire. Mais cette variation est assez petite. La mémoire totale utilisée est alors approximée par la mémoire utilisée par l'un des processeurs multipliée par le nombre des sous-domaines.

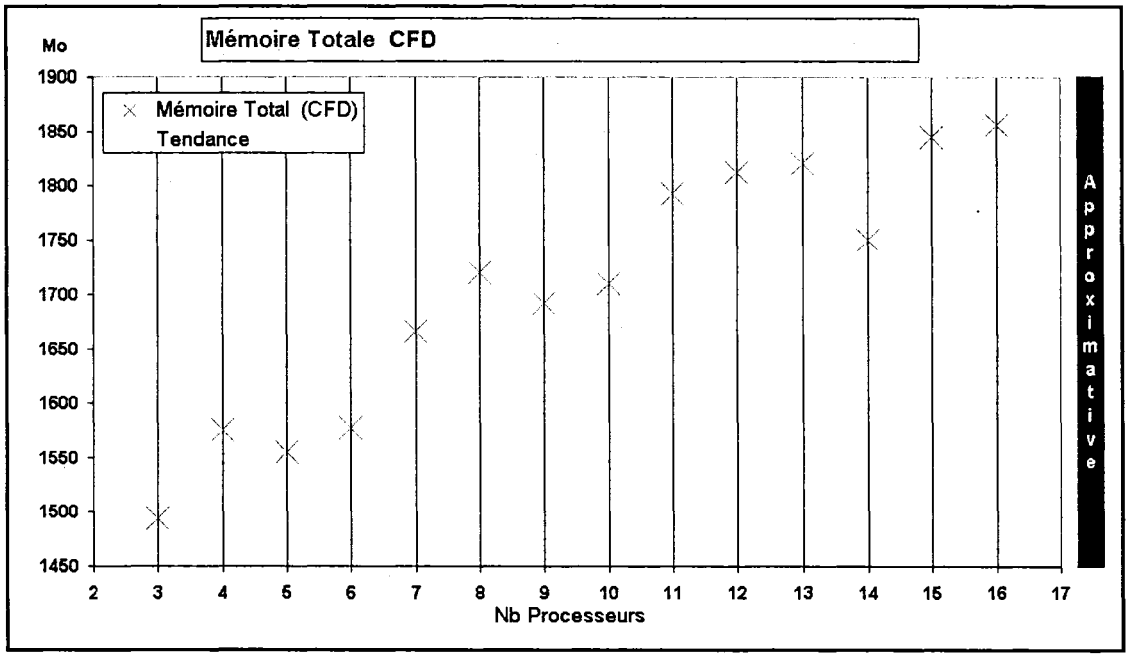


Figure 47 Mémoire totale en fonction de la décomposition

Contrairement à la mémoire locale, la mémoire globale utilisée, augmente avec le nombre des processeurs. L'augmentation est engendrée par le nombre croissant des nœuds d'interfaces entre les sous-domaines.

Pour une machine à mémoire distribuée, comme Thunderbird, une augmentation du nombre de participants est automatiquement accompagnée d'une augmentation de la mémoire totale disponible. Pour une machine à mémoire partagée, l'utilisation de multiples processeurs peut être limitée à cause de la mémoire.

Par exemple, si on utilise 3 processeurs sur Thunderbird, on a plus que 1,5 Go de mémoire au total, si on utilise 16 on a alors plus de 8 Go. Mais sur une machine à mémoire partagée qui possède par exemple 16 processeurs et 1,5 Go de mémoire, seulement 3 processeurs peuvent êtres utilisés à cause du manque de mémoire.(Figure 52).

Une manière plus claire de voir la performance, est le calcul du *speed up*. Le *Speed up* est fonction du temps de calcul du code séquentiel le plus rapide possible T_s.

$$speedup\ (N) = \frac{T_s}{T_P(N)} \tag{6.19}$$

La mémoire d'un seul processeur est limitée à 512Mo. Cette quantité ne permet pas de calculer le temps de calcul T_s. Une façon de l'estimer est de considérer que le *speed up* de 3 processeurs est égale à 3.

La figure suivante représente le *speed up* en fonction du nombre de sous-domaines.

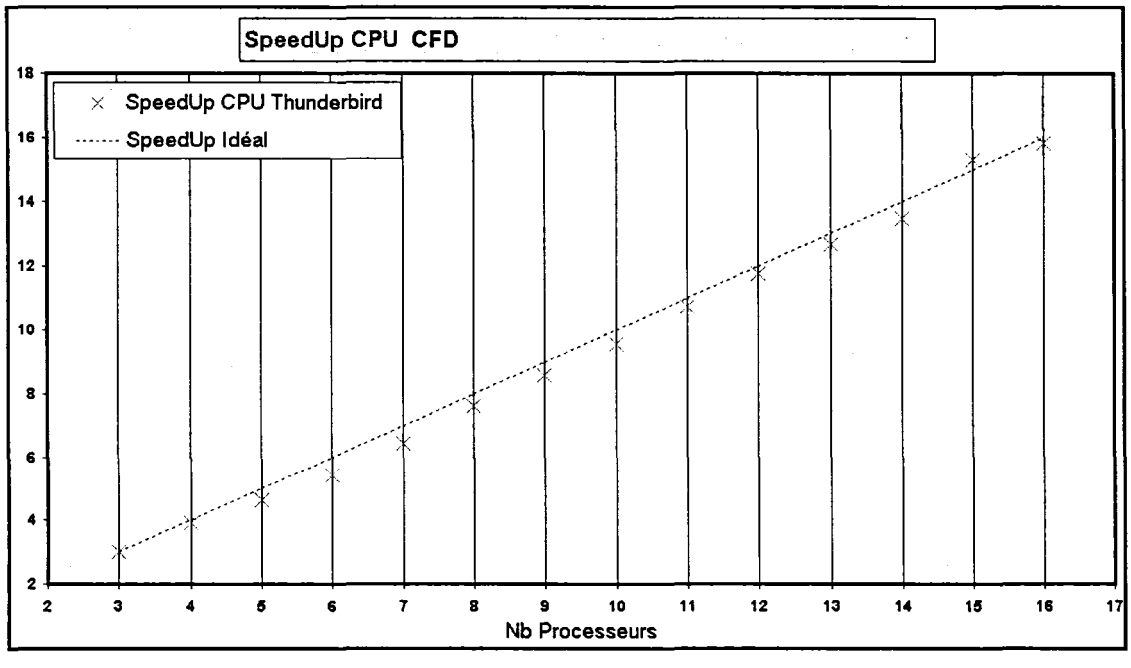


Figure 48 *Speed up* CPU dans le cas de problème non couplé

Le *speed up* ainsi calculé est très proche du *Speed Up* idéal ce qui est signe de haute performance. Un *speed up* idéal correspond à une réduction du temps de calcul de n fois, si n processeurs sont utilisés. Le *speed up* idéal n'est pas une limite absolue. Dans certains cas, il peut être atteint ou même dépassé, en particulier dans le cas des problèmes multiphysiques où la décomposition fonctionnelle est intelligemment combinée avec la parallèlisation des données. Plus encore, dans certains cas, la convergence du code parallèle peut être plus rapide que celle du code séquentiel ce qui améliore le *speed up*.

Une autre alternative de mesure, est l'efficacité. Elle est définie par le rapport du *speed up* et du nombre de processeurs :

$$efficacité\ (N) = \frac{speedup(N)}{N} \tag{6.20}$$

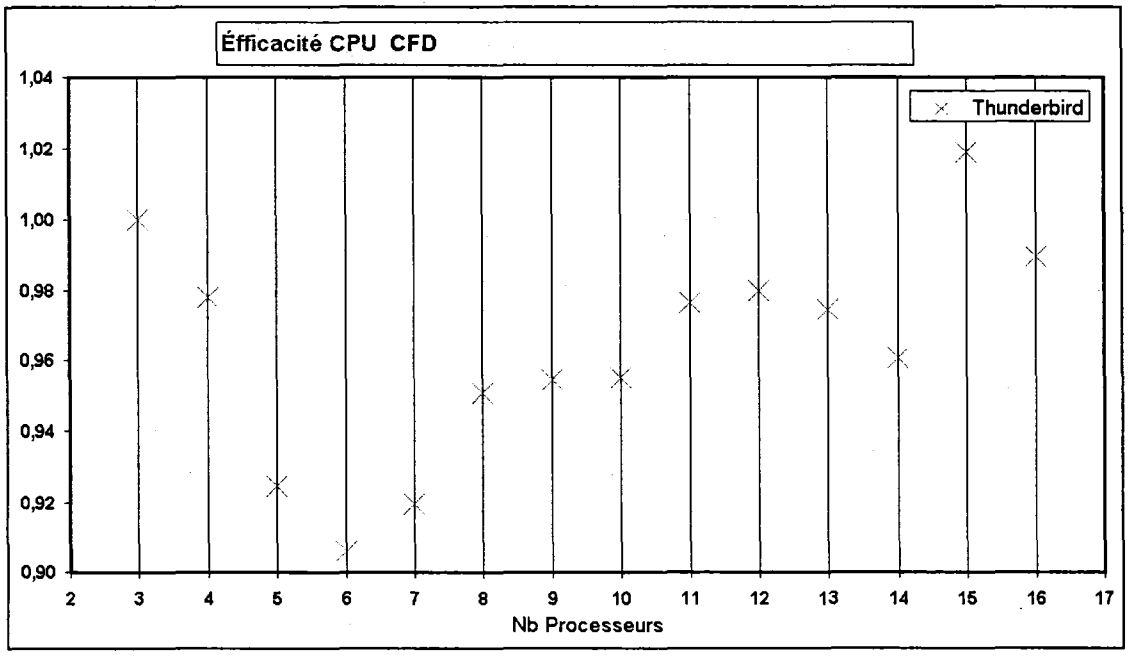


Figure 49 Efficacité dans le cas d'un problème non couplé

Une efficacité de 1 correspond à un *speed up* idéal. Pour toutes les décompositions testées, l'efficacité est supérieure à 0.9, ce qui est en concordance avec la courbe du *Speed up*.

Les figures précédentes illustrent sans doute, les excellentes performances du couple PFES-Thunderbird, en terme de temps de calcul CPU. Cependant, ce qui préoccupe plus l'utilisateur est le temps réel d'exécution. L'étude de l'efficacité en terme de temps réel, prend en considération le calcul, la communication et l'attente. Contrairement au temps CPU, le temps d'exécution total est le même pour tous les processeurs.

Sur la figure suivante le temps d'exécution réel est donné en fonction du nombre de processeurs.

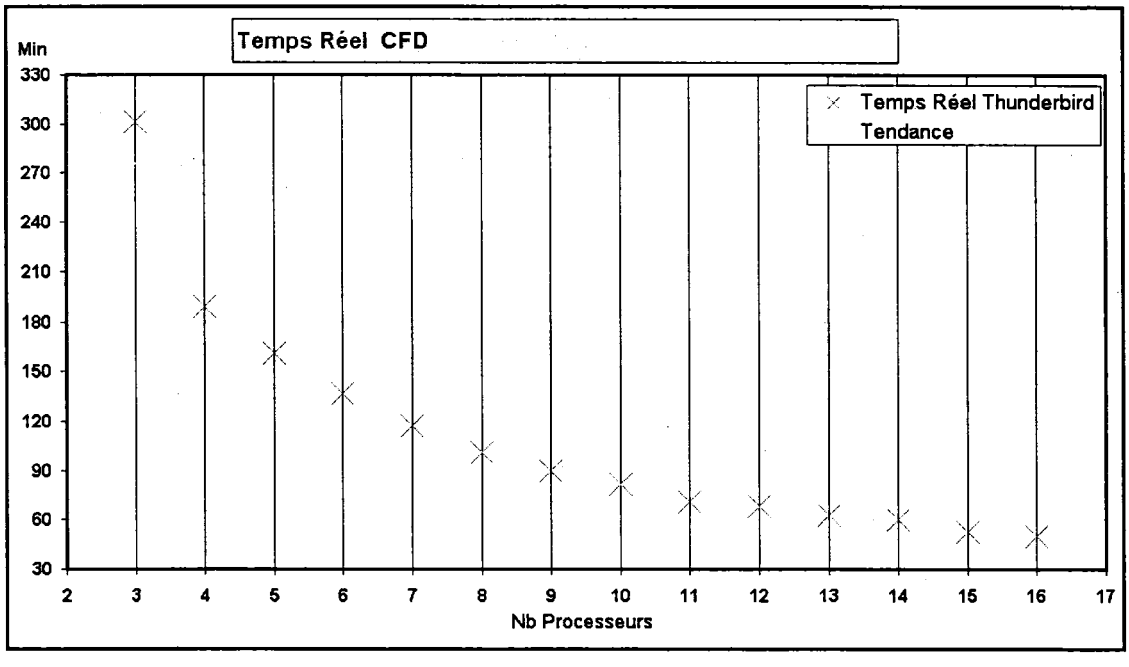


Figure 50 Temps d'exécution réel pour un problème non couplé

Similairement au temps CPU, le temps d'exécution réel est significativement réduit par l'utilisation de multiples processeurs.

Une comparaison qualitative entre la courbe du temps CPU et la courbe du temps réel donne une idée sur le coût de la communication et de l'attente :

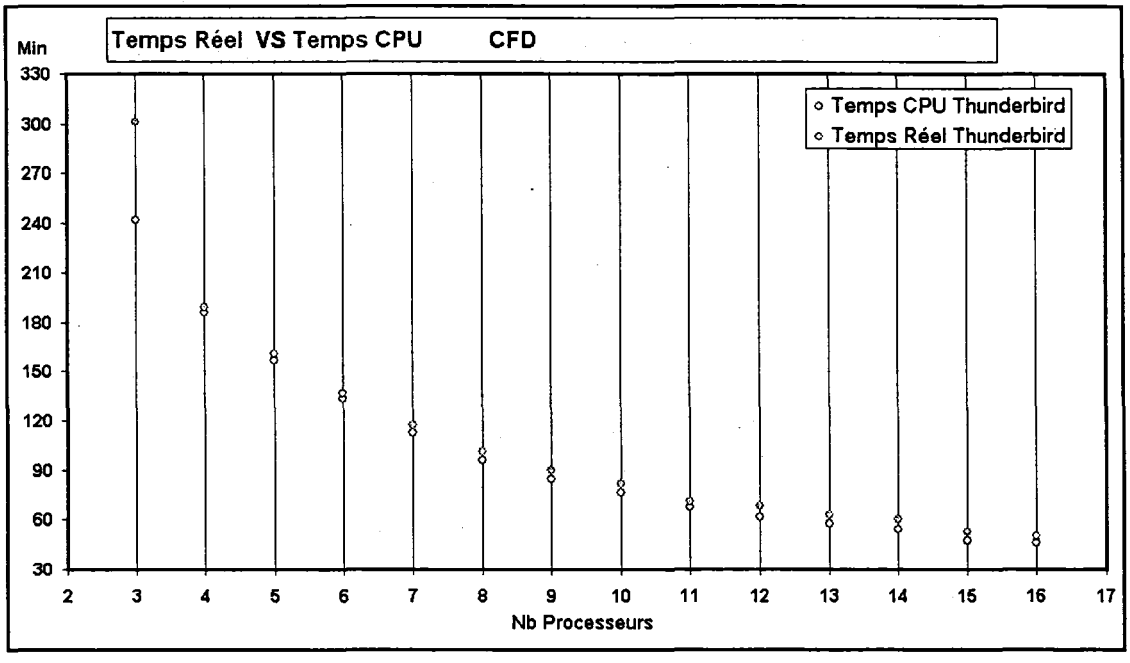


Figure 51 Temps CPU et temps réel en fonction du nombre des sous-domaines

D'après cette figure on remarque que le coût de communication reste assez faible par rapport au temps total ce qui explique les bonnes performances. Plusieurs modes de communications offertes par la bibliothèque MPI ont été essayées. **Les routines de communications collectives se sont avérées plus rapides, stables et portables**. Les routines de communication point-à-point sont particulièrement instables sur les machines **SGI**.

Pour mieux quantifier le coût de la communication, la figure suivante présente le pourcentage du temps de communication et d'attente en fonction du nombre de processeurs.

Le pourcentage est calculé de la façon suivante :

$$\frac{Temps\ Réel - Temps\ CPU}{Temps\ Réel} \tag{6.21}$$

Pour chaque décomposition, le temps CPU maximum est utilisé.

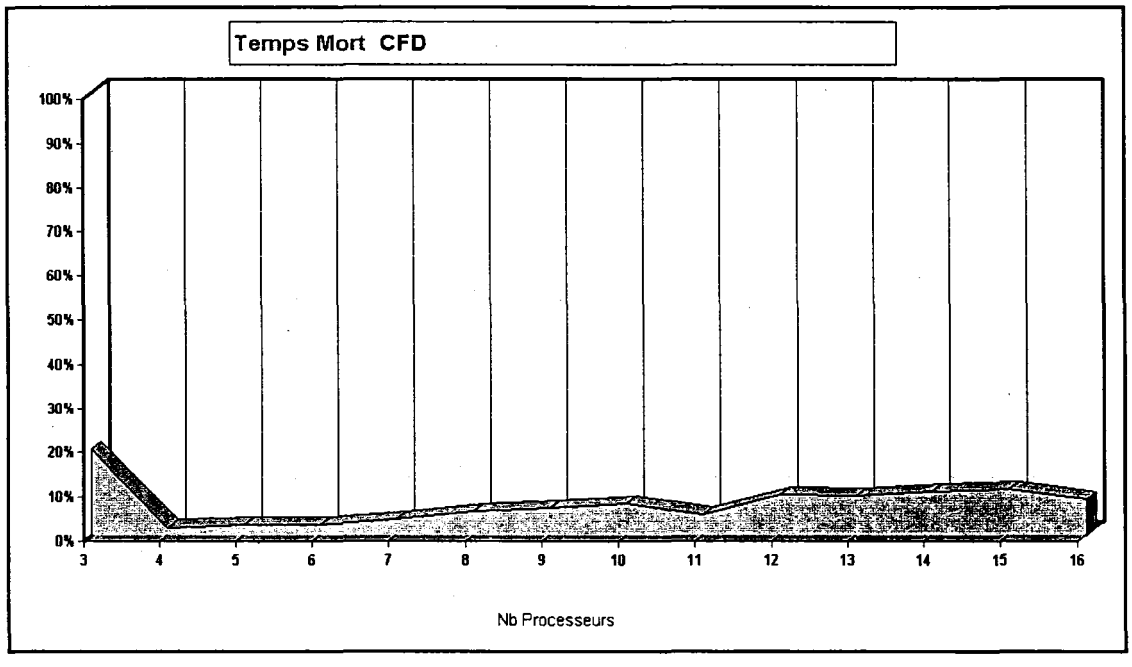


Figure 52 Taux du temps de communication en fonction du nombre de sous-domaines

Le rapport entre le temps des communications-attentes et le temps réel augmente légèrement avec le nombre de sous-domaines. cette augmentation est le résultat de l'accroissement du nombre des nœuds aux interfaces. En effet, plus le problème est grand plus il est intéressant de le subdiviser sans subir, significativement, l'effet de la communication.

La valeur exceptionnellement élevée enregistrée pour le cas de trois processeurs est probablement dûe au manque de mémoire, et donc l'écriture sur le disque. D'ailleurs la mémoire utilisée dans ce cas (500Mo) est très proche de la limite de 512 Mo.

Le *speed up* en terme de temps de calcul CPU donne une idée sur les performances individuelles des processeurs. Une mesure plus importante est le *speed up* réel qui reflète la performance globale. Cette dernière mesure est plus parlante pour les utilisateurs.

La figure suivante donne le *speed up* en fonction du nombre des sous domaines.

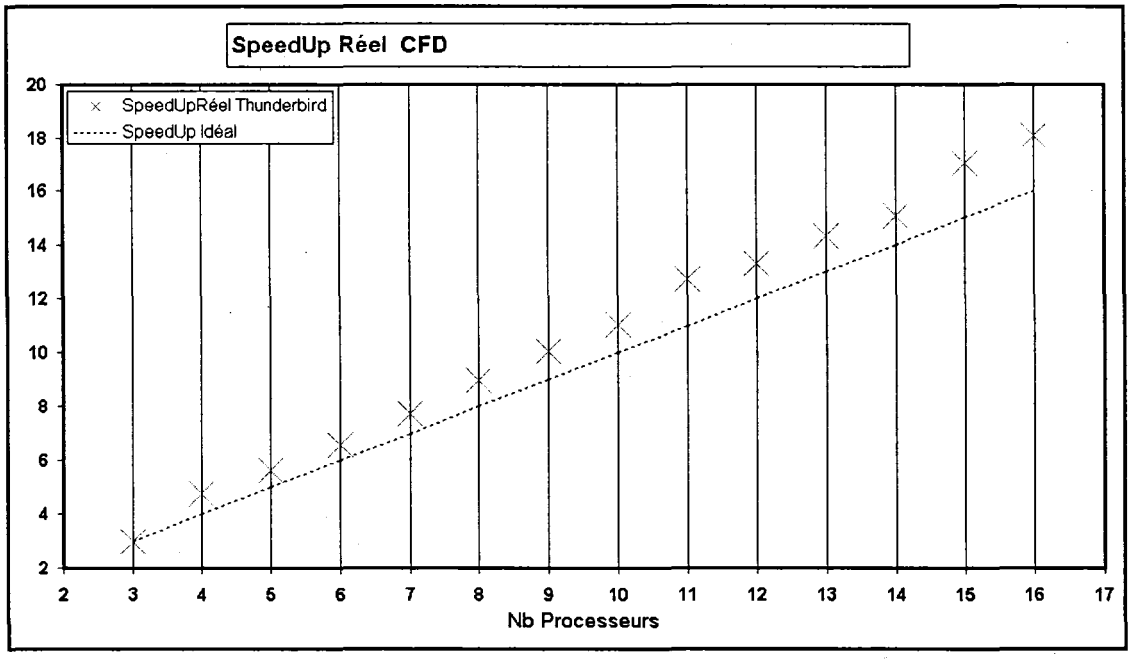


Figure 53 *Speed Up* réel en fonction du nombre de processeurs

Outre le gain apporté par l'utilisation de multiple processeur, la vitesse de convergence du code parallèle PFES a généreusement contribué à la réduction du temps d'exécution. En effet, la décomposition des données et le moyennage des valeurs aux interfaces n'ont pas affecté sensiblement la convergence du code parallèle. Le nombre d'itération de PGMRES a enregistré une très légère augmentation.

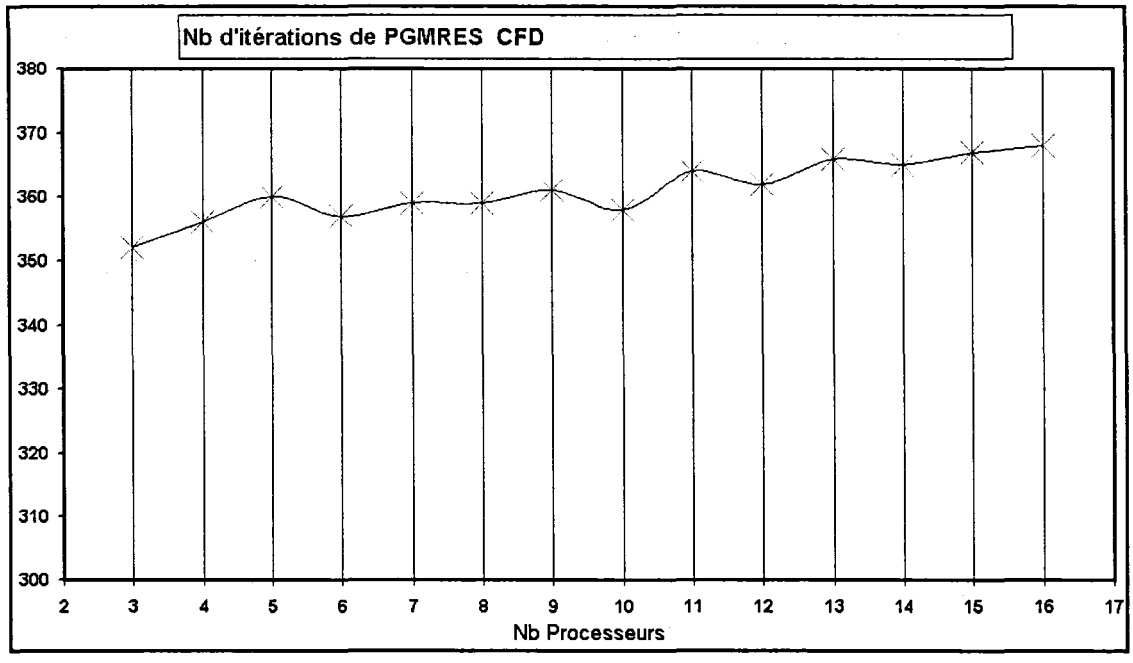


Figure 54 Nombre d'itérations total en fonction du nombre de processeurs

Cette augmentation reste négligeable devant le gain en terme de temps de calcul surtout lors de la factorisation de la matrice et le calcul du préconditionneur. À noter que ces deux dernières opérations coûtent beaucoup plus cher que les itérations de PGMRES.

Les valeurs du *speed up* enregistrées démontrent l'amélioration due au parallélisme. la courbe suivante traduit ces performances en terme d'efficacité :

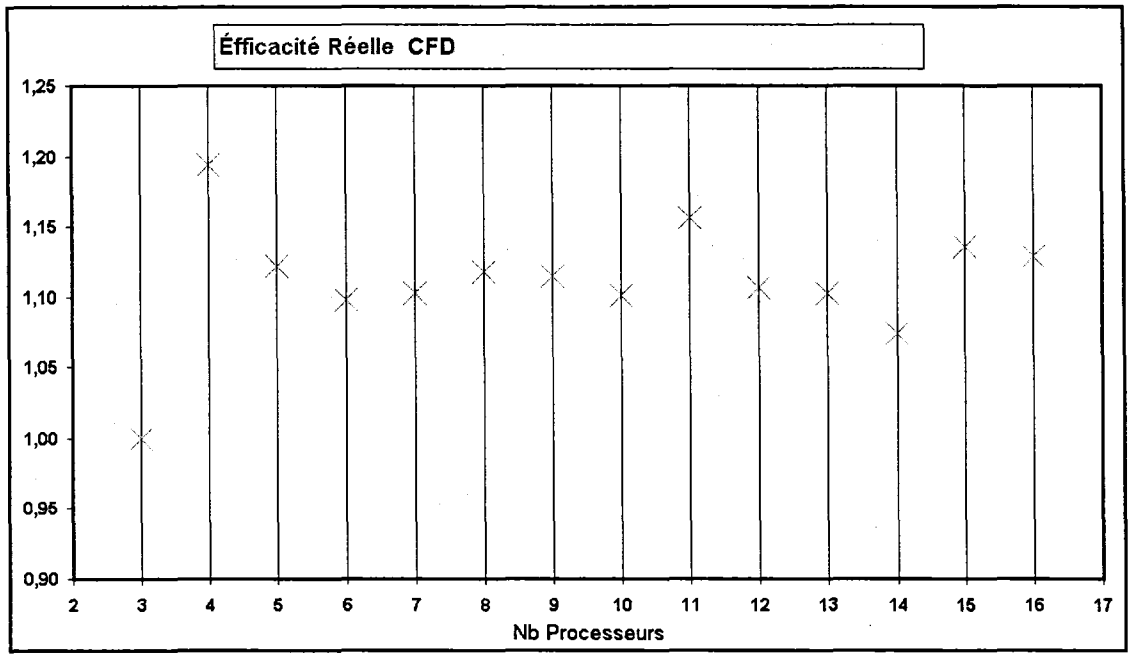


Figure 55 Efficacité réelle en fonction du nombre de sous-domaines

Les valeurs d'efficacité sus-indiquées reflètent les hautes performances de la machine et du code PFES dans le cas d'un problème classique de CFD.

6.3.1.2 Problème d'aéroélasticité

D'autres tests ont été réalisés en vue d'une étude de performance dans le cas des problèmes multiphysiques. Le même maillage de 38K nœuds de l'aile Agard ainsi que la machine Thunderbird dotée de 16 processeurs, ont été utilisés.

L'algorithme de résolution des problèmes multiphysiques adopté par PFES prévoit en plus de la décomposition du domaine physique, une décomposition fonctionnelle basée sur l'aspect multiphysique lui-même. Pour le cas du problème d'aéroélasticité qui met en jeux un domaine dit fluide et un domaine dit structure, les processeurs sont divisés en deux familles. Plusieurs combinaisons de nombre de processeurs attribués sont alors envisageables.

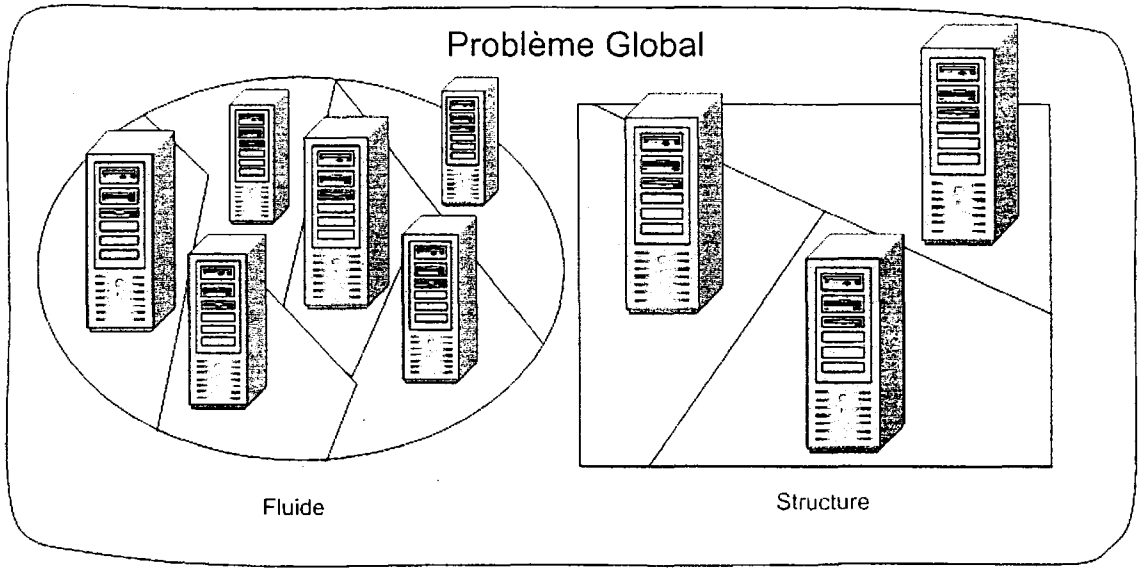


Figure 56 Parallélisation des données et décomposition fonctionnelle

Outre l'étude des performances, le but de cette deuxième série de tests est de trouver les combinaisons optimales pour chaque nombre de processeurs disponibles.

Il est à remarquer que, le nombre d'équations à résoudre est trés élevé dans le domaine fluide par rapport à celui du domaine structure. Les combinaisons qui assignent un grand nombre de processeurs à la structure sont alors évitées. Seules les combinaisons intéressantes sont étudiées.

Le code **PFES** est une version parallèle du code séquentiel. Dans cette version le nombre de processeurs par famille ne peut être inférieur à 2.
Le tableau des résultats (Annexe 3, temps de calcul) présente les différentes combinaisons étudiées ainsi que les résultats trouvés.

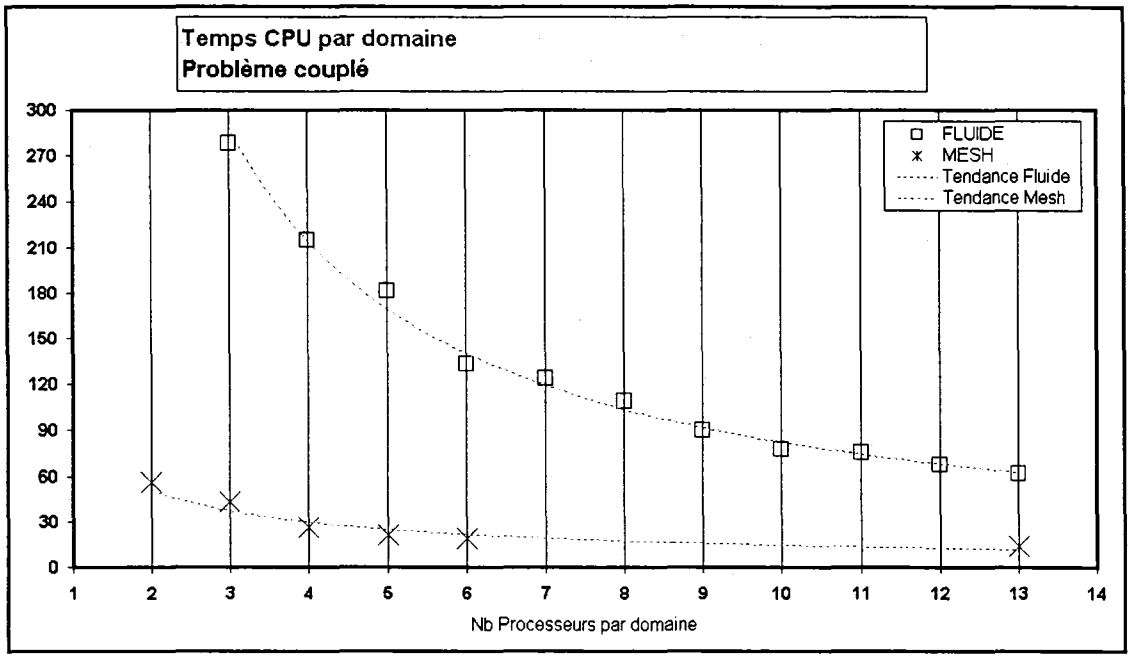


Figure 57 Temps CPU par domaine pour un problème multiphysique

Dans le cas des problèmes multiphysiques, l'étude du temps de calcul pur est aussi important que l'étude du temps de calcul réel. La courbe du temps CPU par domaine permet à l'utilisateur de prendre une décision assez précise dans le choix du nombre de processeurs à allouer à chaque famille.

Pour améliorer le temps de calcul réel, le temps d'attente doit être minimisé, voir réduit à 0. Pour ce fait, l'assignation des processeurs doit se faire d'une manière qui permet d'avoir des temps CPU les plus proches que possible dans tout les sous-domaines.
Selon la courbe précédente, pour n'importe quelle décomposition du domaine fluide de 3 à 13, le temps CPU fluide est supérieur à celui de la structure. Par conséquent, il est inutile d'assigner plus que 2 processeurs à cette dernière famille.

Remarquons que, dans le cas d'une décomposition du domaine fluide en 13 sous-domaines, le temps CPU du fluide est très proche de celui de la structure décomposée

en deux. Si la machine Thunderbird disposait d'un nombre plus grand de processeurs, l'assignation de plus de 13 processeurs au domaine fluide engendrerait une réduction du temps CPU qui franchirait probablement la barre de 60 minutes. Dans ce cas, une attribution de 2 processeurs à la structure ne sera plus un choix optimal car les processeurs fluide seront plus rapide, l'attribution de 3 processeurs pour la structure s'imposera.

La figure suivante illustre les performances enregistrées en terme de *Speed up* CPU pour chaque domaine physique.

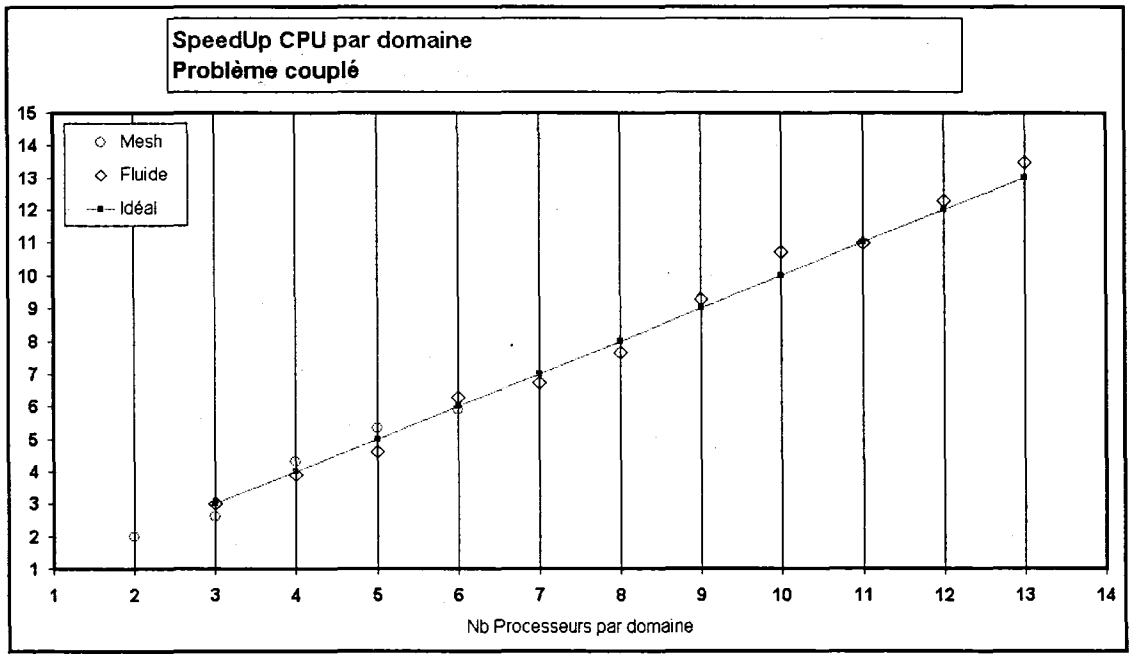


Figure 58 *Speed up* CPU par domaine pour un problème multiphysique

Les résultats trouvés révèlent des valeurs de *Speed up* très proches de l'idéal. En effet, l'étude de *Speed up* CPU ne prend pas en charge les communications et les attentes. En terme de CPU, les deux domaines peuvent êtres alors considérés indépendants. Les résultats ainsi trouvés sont alors identiques au cas non couplé (Figure 53).

Bien que l'étude du temps CPU permet d'optimiser l'assignation des processeurs aux domaines physiques, l'étude de *Speed up* CPU ne reflète pas les performances du problème couplé.

Une étude basée sur le temps de calcul réel s'impose. La figure suivante donne le temps d'exécution pour différentes décomposions étudiées.

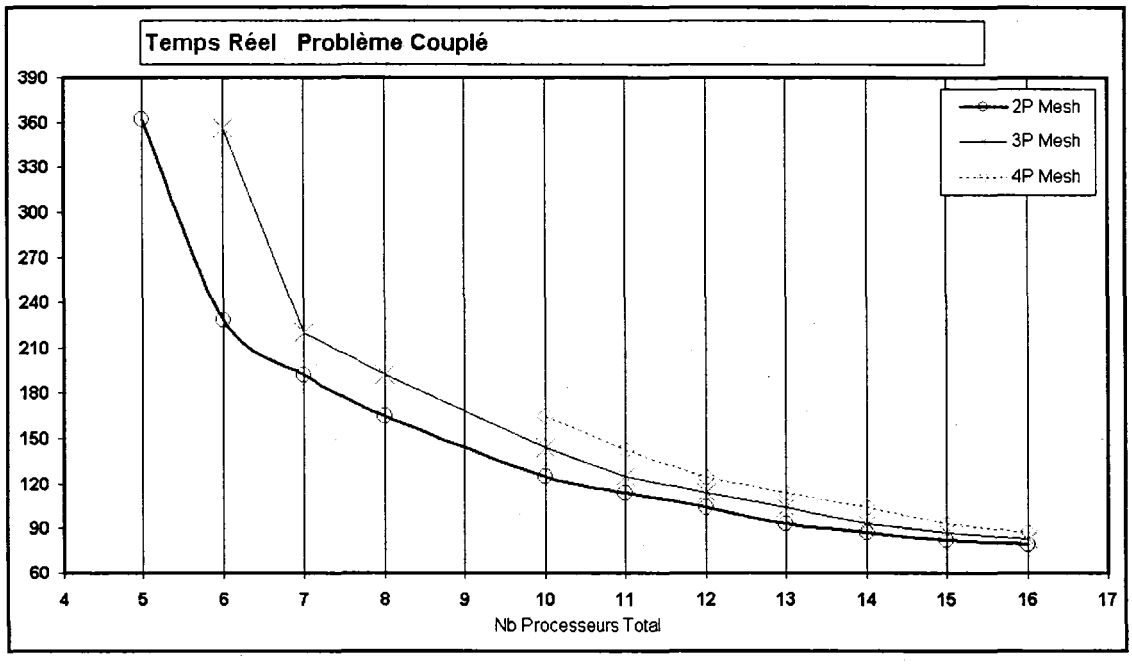


Figure 59 Temps réels pour différentes combinaisons

L'axe des abscisses représente le nombre total *Nbproc* de processeurs utilisés. Une courbe appelée « *n*P Mesh » représente le temps réel dans le cas d'une décomposition où *n* processeurs sont assignés au Mesh. Le nombre assigné au fluide est alors *Nbproc* – *n*. Globalement, l'utilisation de multiples processeurs engendre une réduction remarquable du temps d'exécution. Mais dépendant du nombre de processeurs attribués au Mesh, cette réduction est plus ou moins importante. Comme prévu par l'étude du temps CPU,

l'utilisation de 2 processeurs pour le Mesh est optimale jusqu'à un nombre total de 16 processeurs.

La courbe de *Speed up* suivante confirme d'une part, le choix d'une attribution de 2 processeurs à la structure et reflète d'une autre part, les performances du code et de la machine dans le cas d'un problème multiphysique.

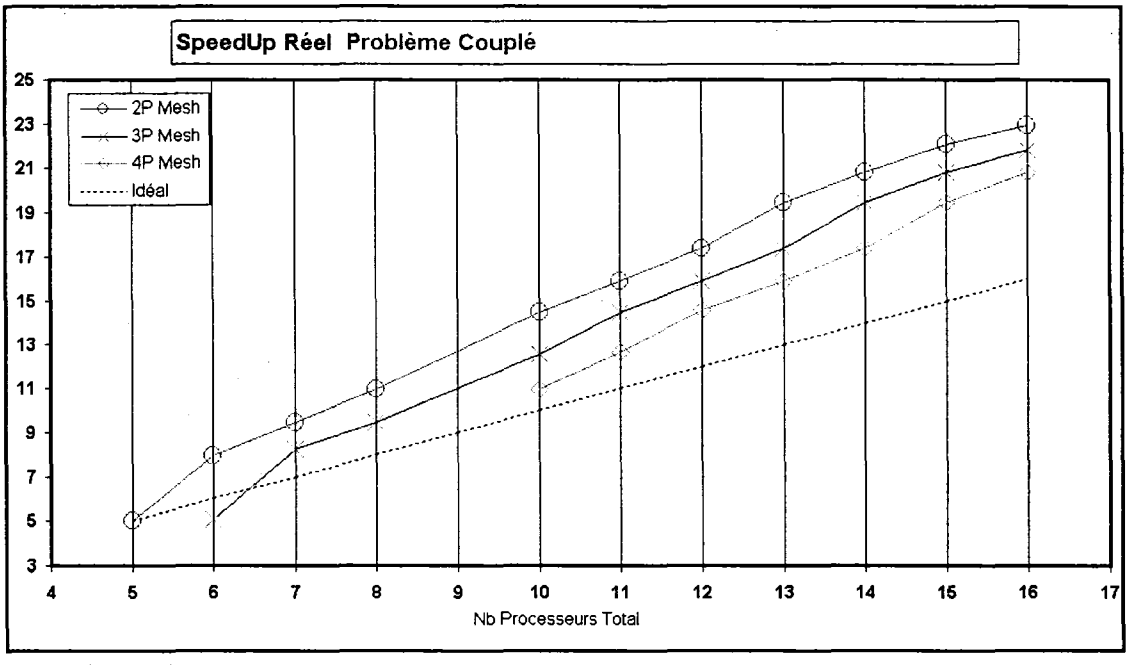


Figure 60 *Speed up* réel pour différentes combinaisons

Le *Speed up* augmente significativement avec le nombre total des processeurs. L'utilisation de 16 processeurs engendre une réduction du temps d'exécution compris entre 21 et 23 fois selon la répartition des processeurs entre les familles. L'assignation de deux processeurs à la structure donne les meilleures performances.

L'augmentation du *Speed up* d'une façon linéaire montre que la taille du problème permet d'utiliser un nombre de processeurs supérieur à 16, sans toutefois perdre de performance.

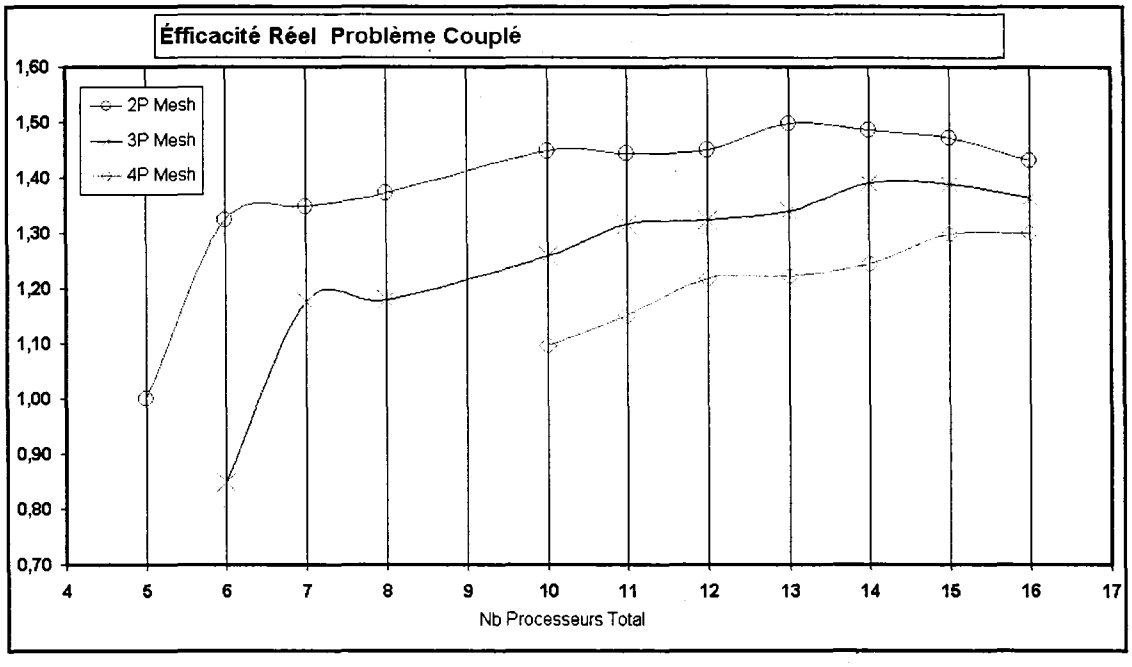


Figure 61 Efficacité réelle pour différentes combinaisons

L'efficacité a atteint une valeur de 1,5 pour 13 processeurs, ce qui reflète une très haute performance. Les valeurs d'efficacité confirment les points discutés sur la courbe du *Speed up*.

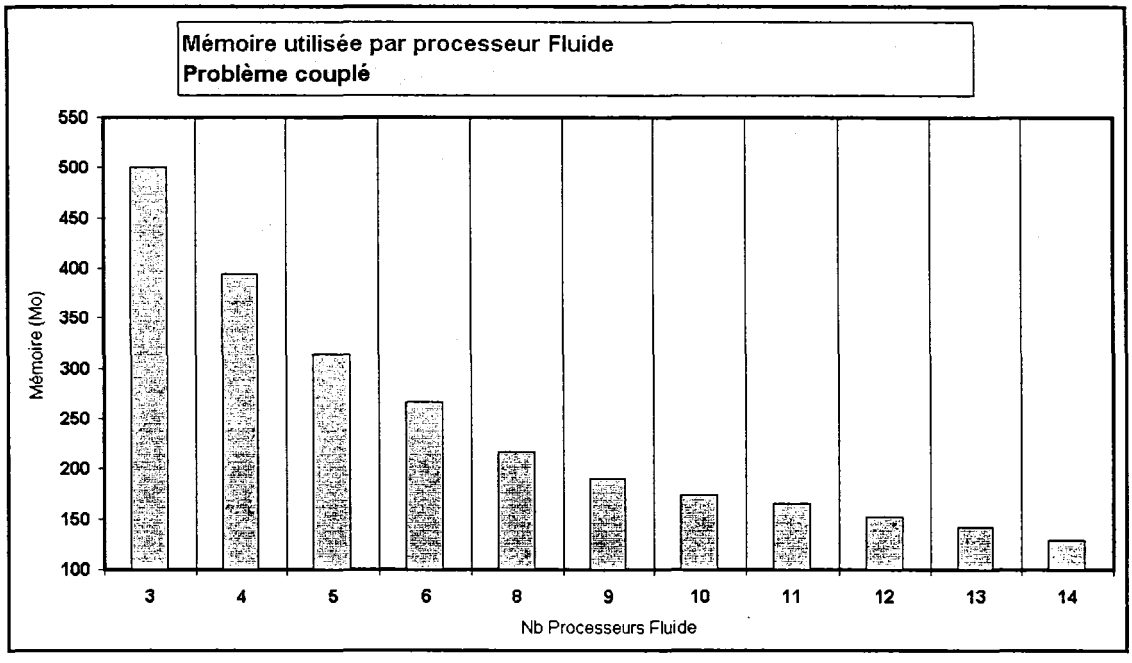


Figure 62 Mémoire utilisée par processeur fluide
en fonction du nombre des sous-domaines

L'utilisation de multiples processeurs a réduit sensiblement l'utilisation de mémoire par chaque processeur. Dans le cas des problèmes multiphysiques, des tables supplémentaires sont utilisées pour stocker les valeurs communiquées entre les familles. Une comparaison avec le cas du problème non couplé permet de voir l'ampleur de l'augmentation de mémoire utilisée.

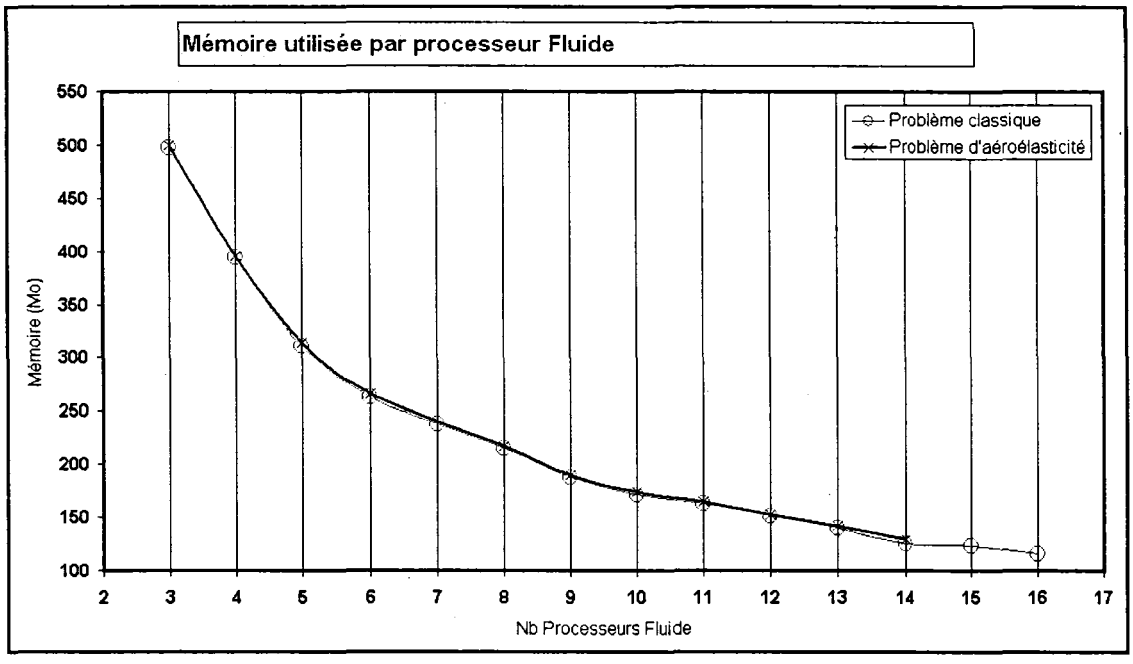


Figure 63 Mémoire utilisée dans le cas d'un problème de CFD et un problème d'aéroélasticité

Une superposition des courbes de mémoire utilisée par un processeur fluide, du problème couplé et non couplé, montre une très légère augmentation de la mémoire.

La mémoire nécessaire au processeur fluide est beaucoup plus importante que celle utilisée par un processeur structure. C'est pourquoi seule la mémoire utilisée par les processeurs assignés au domaine fluide est évoquée.

6.3.2 Résultats des simulation

Dans le paragraphe suivant, le résultat de l'étude aéroélastique sur une aile flexible est présenté.

L'aile en question est l'Agard 445.6 [17]. Le profil est le NACA65A004. Le modèle considéré est le model 3.

Pour la résolution des équations d'Euler, un maillage de 37965 nœuds et 177042 éléments a été utilisé. Le nombre d'équations couplées générées est 388464. Concernant la structure, le maillage utilisé est constitué de 1250 nœuds et 1176 éléments coques quadrilatérals.

Les 5 premiers modes ont été obtenus à partir du code commercial ANSYS. Les fréquences propres sont : 9.6 Hz, 39.42 Hz, 49.60 Hz, 96.095 Hz et 126.30 Hz.

La courbe suivante presente l'indice de flottement en fonction du nombre de Mach :

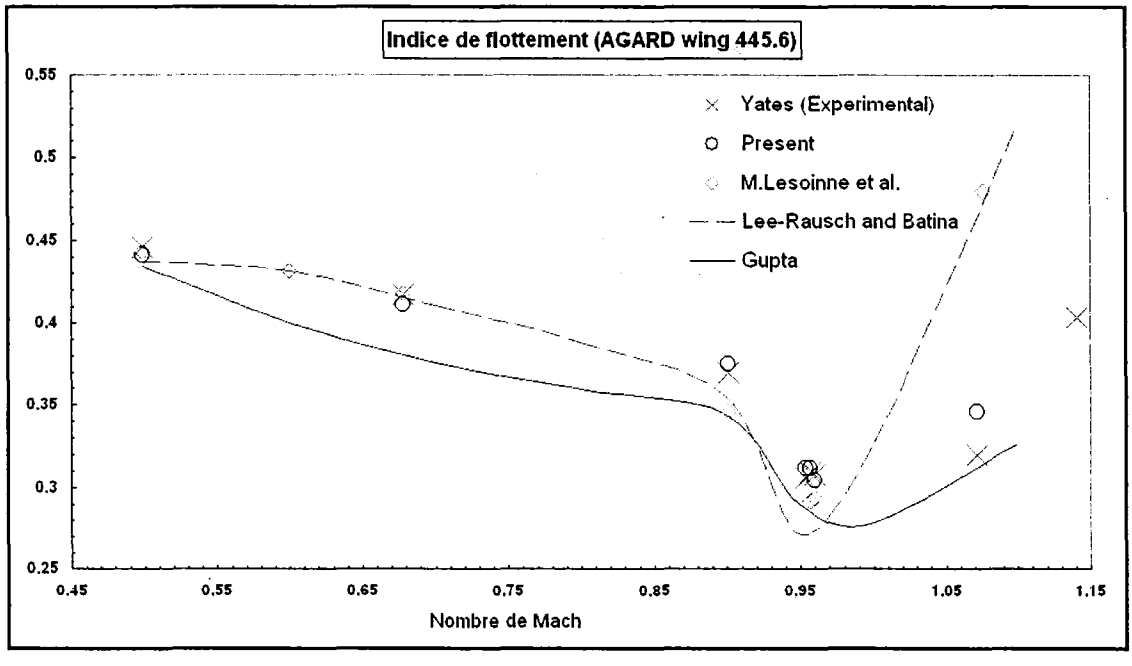


Figure 64 Indice de flottement de l'aile Agard 445.6

La figure 69 présente une comparaison des indices de flottement obtenus par le code PFES avec les résultats expérimentaux [17] et avec quelques résultats numériques reportés dans la littérature [6].

CONCLUSION

L'objectif de ce projet est le développement d'un outil de calcul capable de résoudre des problèmes multiphysiques.

La complexité et la taille des problèmes multiphysiques engendrent l'épuisement des ressources des machines traditionnelles en terme de capacité de mémoire. Pour surmonter cette difficulté, un code de calcul parallèle a été conçu et implémenté.

Le code **PFES** « PARALLEL FINITE ELEMENT SYSTEM » est un code de calcul parallèle, en éléments finis, capable de résoudre des problèmes multiphysiques.

L'un des points forts du code est l'utilisation de l'allocation dynamique de la mémoire offerte par la norme **Fortran 90**. Cet avantage nous a permis d'utiliser la bibliothèque **PSPARSLIB** ecrite en Fortran 77 sans avoir besoin de la réécrire dans un autre langage. Ainsi, le code obtenu est écrit en un seul langage, simple et bien apprécié par les scientifiques.

En plus d'une parallèlisation de données, indispensable pour la résolution des problèmes de très grandes tailles, une parallèlisation fonctionnelle basée sur une approche **MPMD** « Multiple Program, Multiple Data » et qui respecte l'aspect multiphysique, a été utilisée.

D'autre part, la parallèlisation du code a réduit considérablement le temps de calcul. Les études menées dans le cas d'une simulation d'un écoulement autour de l'aile Agard et dans le cas d'un problème d'aéroélasticité ont révélé des performances très satisfaisantes. En effet, à titre d'exemple, pour un nombre de processeurs de16 on a atteint un Speedup de 23 ce qui représente une efficacité supérieure à 140 %.

Les résultats de l'étude aéroélastique sur l'aile **Agard 445.6** montrent que le code **PFES** a réussi à reproduire fidèlement la courbe expérimentale d'indice de flottement.

Finalement, le code élaboré durant ce projet est robuste et bien structuré. Basé sur des modules séparés, **PFES** présente l'avantage d'être clair et flexible et ce pour une éventuelle amélioration future.

ANNEXES 1

Les machines

Configuration du Cluster (Granit):

Le Cluster Granit est composé de deux sous clusters (Andalous et Thunderbird).

Les deux Clusters sont configurés de telle façon qu'ils peuvent opérer séparément ou ensembles.

Les nœuds de calculs sont cabales d'opérer sous la commande de l'un des deux Clusters au choix.

Le Cluster Andalous :

Est composé d'un serveur et 8 nœuds de calcul reliés entre eux par un réseau Ethernet 10/100 mb/s.

Le Cluster Thunderbird :

Est composé d'un serveur et 14 machines de calcul, tous des PC's de haute gamme.

L'ensemble est relié par un Switch intelligent 10/100 mb/s.

Les composantes :

Andalous		**Thunderbird**	
Machines	Pentium III	Machines	Pentium IV
CPU	Pentium III (600 Mhz) à 512K	CPU	Pentium IV (1700Mhz) à 512K
Mémoire	500 Meg SDRAM Pc(133Mhz)	Mémoire	500 Meg RDIMM (800Mhz) ECC

Le cluster a une capacité totale de 12 Go de mémoire. Il faut mentionner que tous ces PC's sont construits de composantes normales qu'on peut trouver dans n'importe quel PC de bureau.

ANNEXES 2

Fonctions de MPI

LES FONCTIONS DE MPI
LES PLUS UTILISÉES DANS LE PROGRAMME P.F.E.S.

La bibliothèque **MPI** offre plusieurs sous-programmes qui permettent d'établir la communication entre les processeurs. Les messages échangés sont interprétés et gérés par un environnement qui peut être comparé à la téléphonie ou encore à une messagerie électronique.

Ces sous-programmes peuvent être classés en trois catégories :

1 Environnement

- Le sous-programme, *MPI_INIT( )*, permet d'initialiser l'environnement nécessaire

  integer, intent(out) :: ierr

  call MPI_INIT(ierr)
- Réciproquement, le sous-programme *MPI_FINALIZE( )* désactive cet environnement :

  call MPI_FINALIZE(ierr)
- *MPI_COMM_RANK( )* permet d'obtenir le rang d'un processus.

  integer, intent(out) :: rang, ierr

  call MPI_COMM_RANK(MPI_COMM_WORLD, rang, ierr)
- *MPI_Comm_size ( )* permet d'obtenir le nombre de processus.

  integer, intent(out) :: nproc, ierr

  MPI_Comm_size (MPI_COMM_WORLD, nproc, ierr)

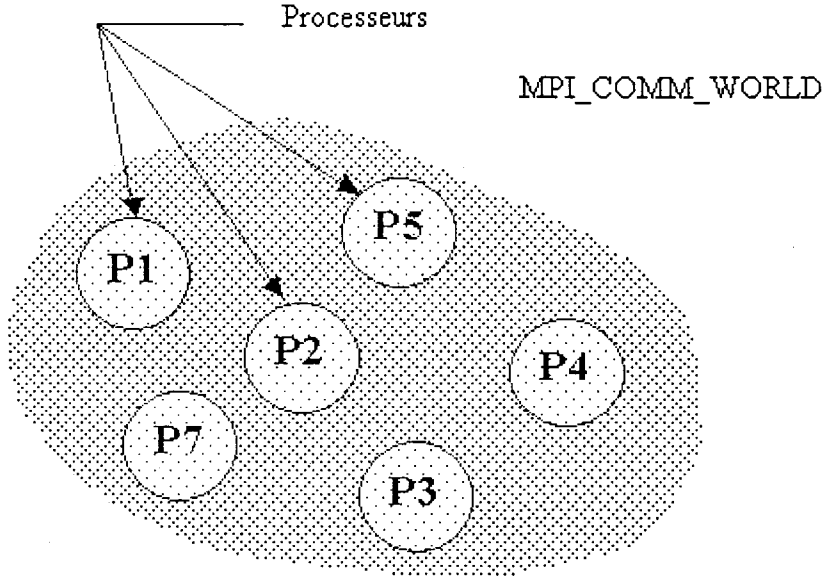


Figure 65 Communicateur MPI_COMM_WORLD

- *MPI_Comm_split ( )* permet de créer un nouveau communicateur, basé sur les couleurs, à partir d'un communicateur existant.
  MPI_Comm_split (comm_existant, color, key, comm._new, ierr)
- *MPI_Comm_group ( )* permet de connaître le groupe associé au communicateur
  MPI_Comm_group (comm, group, ierr)
- *MPI_GROUP_INCL ( )* permet de créer un nouveau groupe à partir d'un autre.
  MPI_Group_incl (group_existant, nbproc, ranks, group_out)
- *MPI_COMM_CREATE ( )* permet de créer un communicateur au sein d'un groupe de processeurs.
  MPI_COMM_CREATE(old_comm, procs, new_comm, ierr)

2 Communications point à point

Une communication est dite « point à point » si elle s'effectue entre deux processus, l'un appelé processus émetteur et l'autre processus destinataire.

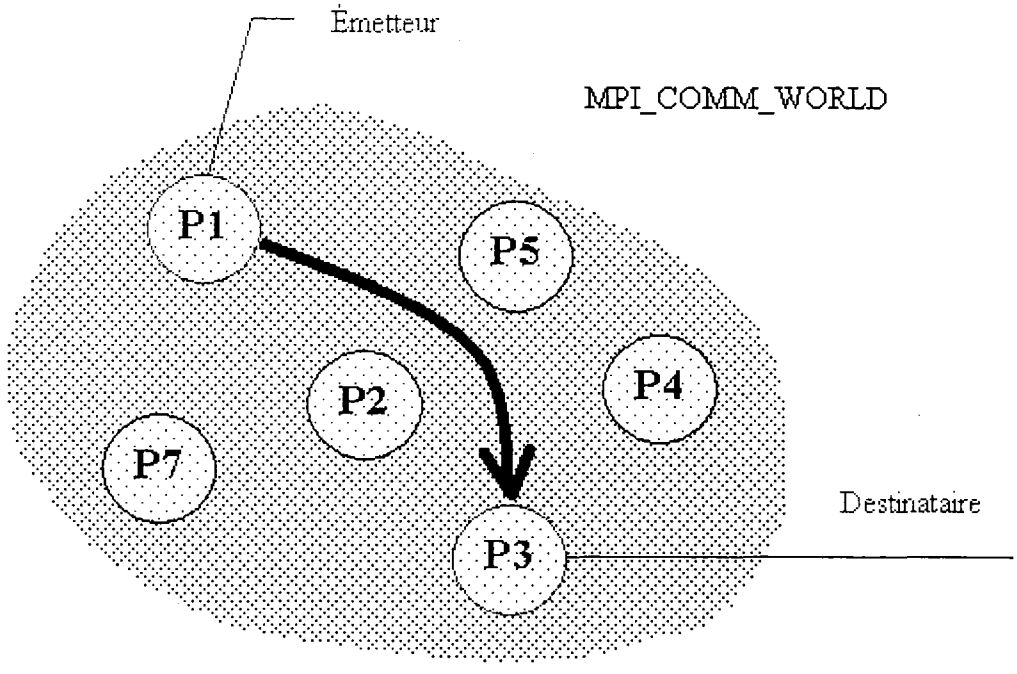


Figure 66 Communications point à point

Plusieurs variantes de communication point à point peuvent être établies, toutefois le principe reste le même; Un processeur envoie, l'autre reçoit.

Les fonctions de **MPI** qui permettent l'envoi et la réception sont les suivantes :

2.1 Les communications bloquantes

mpi_send*(buf, count, datatype, dest, tag, comm, ierror)*

buf : Les données à envoyer

count : Le nombre de données à envoyer

datatype : indique le type de donnée

dest : définit le destinataire du message

tag : définit un tag pour le message

comm : définit un communicateur.

Ierror : retourne l'erreur

Remarquons que les données à envoyer (buf) sont copiées dans le buffer du processeur émetteur. Ce buffer est géré par le communicateur. Ce n'est seulement qu'une fois le message parti (même s'il n'est pas encore arrivé), que le processeur émetteur peut passer aux opérations suivantes.

mpi_recv*(buf,count,datatype,source,tag,comm,status,ierror)*

buf : Les données à recevoir

count : Le nombre de données à recevoir

datatype : indique le type de donnée

source : définit la source du message

tag : définit le tag du message

comm : définit un communicateur.

Status : status de la réception du message (reçu, en attente,...)

Ierror : retourne l'erreur

Une primitive *receive* bloquante, bloque l'exécution jusqu'à ce qu'un message soit reçu. Une fois le message arrivé, le processeur destinataire le place dans son buffer interne.

Notes additionnelles sur les communications bloquantes

La correspondance entre les types MPI et les types Fortran est fournie dans les tableaux suivants :

Tableau V

Correspondance entre les types MPI et les types Fortran

Type MPI	Type Fortran
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION

MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)

Il existe 3 modes pour un *send* bloquant:

- Mode *Buffer*

Dans ce mode, le *send* peut être déclenché indépendamment de l'opération *receive*. C'est un mode local. S'il n'y a pas de *receive*, MPI garde le *send* dans son *buffer*.
Syntaxe: ***mpi_Bsend****(buf,count,datatype,dest,tag,comm,ierror)*

- Mode Synchrone

Dans ce mode, le *send* peut être déclenché indépendamment de l'opération *receive*. Néanmoins, le *send* synchrone ne se termine que si un *receive* est posté. Ainsi un *send*, terminé en mode synchrone, indique que le *buffer* du *send* est vide et peut donc être utilisé à nouveau. C'est également une opération locale.
Syntaxe: ***mpi_Ssend****(buf, count, datatype, dest, tag, comm, ierror)*

- Mode Ready

Le *send* ne commence que si et seulement si une opération *receive* a été postée. Le message est envoyé dès que possible. Il a la même syntaxe que le *send* normal mais peut être précisé.
Syntaxe *:* ***mpi_Rsend****(buf,count,datatype,dest,tag,comm,ierror)*

2.2 Les communications non bloquantes

Les communications non bloquantes nécessitent un argument supplémentaire appelé *request*, qui permet d'identifier les opérations de communication et de faire correspondre l'opération qui initialise la communication avec celle qui la réalise effectivement.

MPI_ISEND (don, taille, dtype, dest, tag, comm, request, irc)

don : donnée (adresse initiale) à envoyer,

taille : nombre d'éléments à envoyer,

dtype : type MPI de chaque élément à envoyer,

dest : rang du processus destinataire,

tag : étiquette du message,

comm : communicateur,

request : identifiant de l'envoi,

irc : code de retour.

Les envois non bloquants utilisent les mêmes quatre modes que les envois bloquants et ont le même sens. Dans tous les cas, l'initialisation de l'envoi est locale, le sous-programme rend immédiatement la main. Enfin un envoi non bloquant peut être mis en correspondance avec une réception bloquante et vice-versa.
La syntaxe d'un envoi non bloquant standard est la suivante :

MPI_ISEND (don, taille, dtype, dest, tag, comm, request, irc)

don : donnée (adresse initiale) à envoyer,

taille : nombre d'éléments à envoyer,

dtype : type MPI de chaque élément à envoyer,

dest : rang du processus destinataire,

tag : étiquette du message,

comm : communicateur,

request : identifiant de l'envoi,

irc : code de retour.

Dans les cas de communications bloquantes, il faut attendre que le send/receive soit effectué pour pouvoir passer à la ligne suivante (ou en tous cas, que la variable envoyée

soit copiée dans un buffer). A fin de gagner du temps, on utilise les communications non bloquantes. Il s'agit de poster des communications point à point (*send* ou *receive*) avec une étiquette et de continuer à travailler.

3 Les communications collectives

Plutôt que de communiquer seulement de point à point entre deux processus, il est de temps en temps nécessaire de communiquer globalement. Par exemple:

- pour partager la valeur d'une variable connue par seulement un processeur,
- pour faire une réduction (somme, max...) de données distribuées sur les différents processus et rapatrier cette valeur sur un processus,
- pour distribuer les valeurs contenues dans un tableau sur un processus vers tous les autres,
- pour récupérer des données distribuées dans un tableau sur un processus
- ...

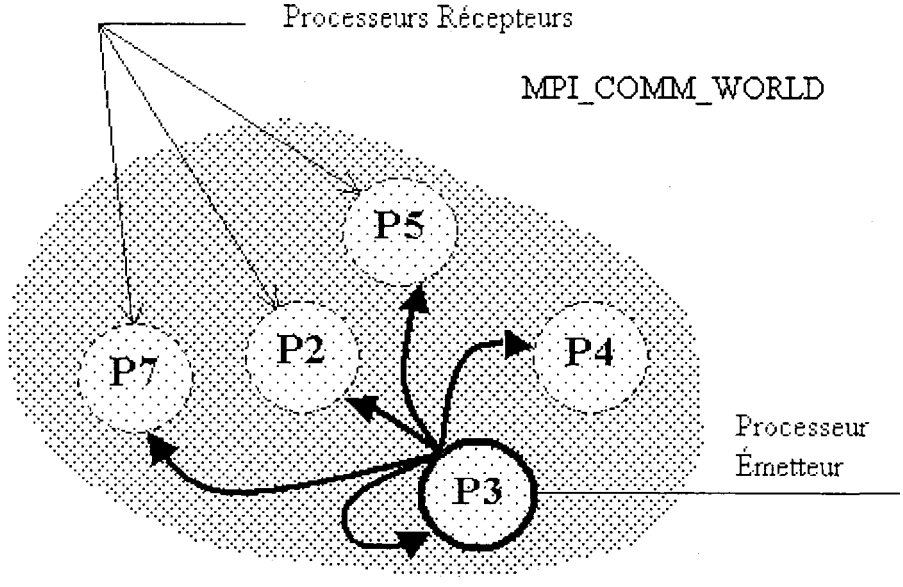


Figure 67 Exemple de Communication collective

Les principales primitives utilisées sont:

- ***broadcast*** : pour envoyer une information d'une tâche vers toutes les autres.
- ***Reduce*** : une information sur une tâche est réduite vers une autre tâche.
- ***scatter*** : répartit un vecteur d'une tâche entre les autres processeurs.
- ***gather*** : rassemble les informations de toutes les tâches vers une seule.

3.1 Broadcast

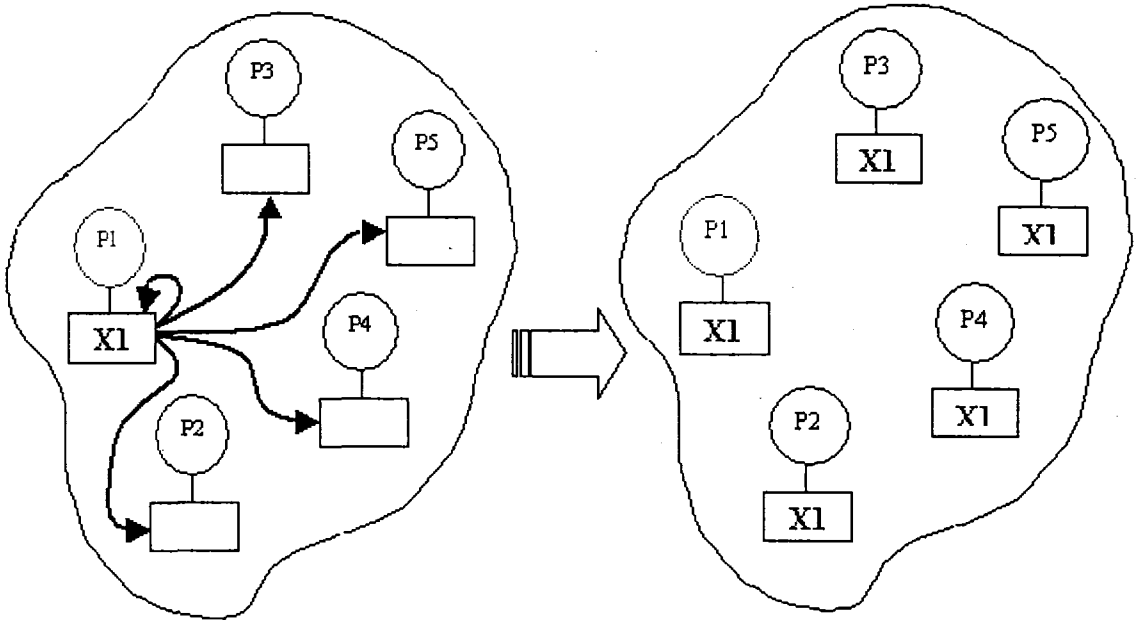


Figure 68 MPI_Bcast

mpi_bcast(buf, count, datatype, root, comm, ierror)

3.2 Reduce

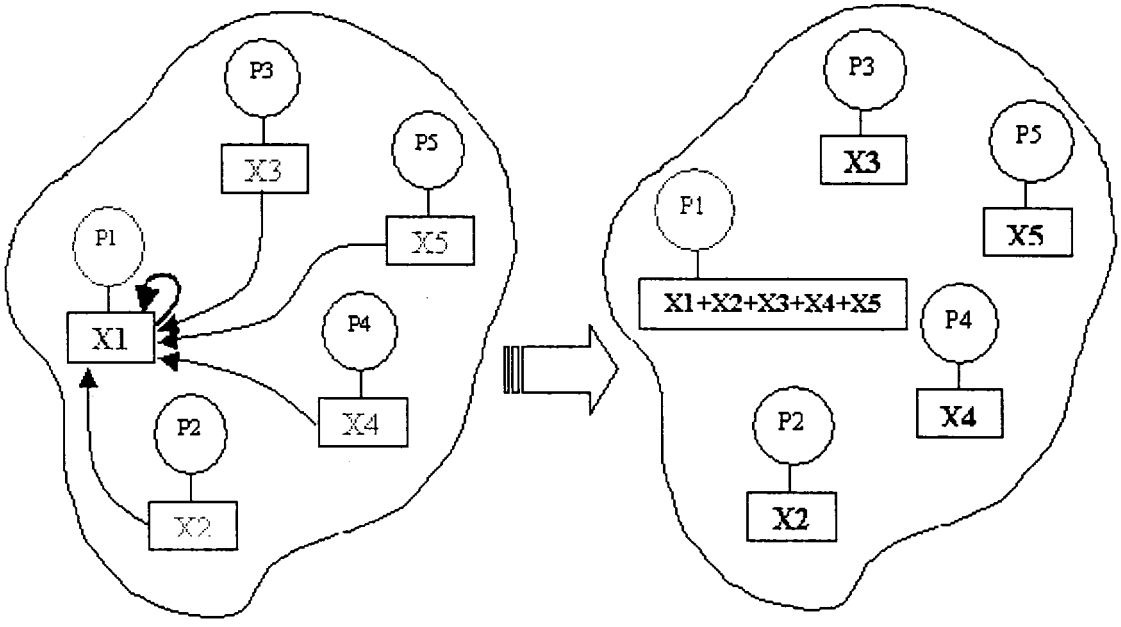


Figure 69 MPI_REDUCE

mpi_reduce (sendbuf, recvbuf, count, datatype, op, root, comm, ierror)

La variable *root* est le rang de la tâche qui reçoit la synthèse des données via l'opérateur *op*. *op* peut en principe être n'importe quelle opération commutative et associative.

Pour que toutes les taches reçoivent la synthèse des données, le sous-programme *MPI_Allreduce (sendbuf, recvbuf, count, datatype, op, comm),* peut être utilisé.

Voici la liste des opérateurs :

Tableau VI

Opérations de MPI

Nom	Définition
MPI_MAX	Maximum
MPI_MIN	Minimum

MPI_SUM	Somme
MPI_PROD	Produit
MPI_LAND	.AND. logique
MPI_BAND	.AND. bit à bit
MPI_LOR	.OR. logique
MPI_BOR	.OR. bit à bit
MPI_LXOR	.XOR. logique
MPI_BXOR	.XOR. bit à bit
MPI_MAXLOC	maximum et indice
MPI_MINLOC	minimum et indice

3.3 scatter

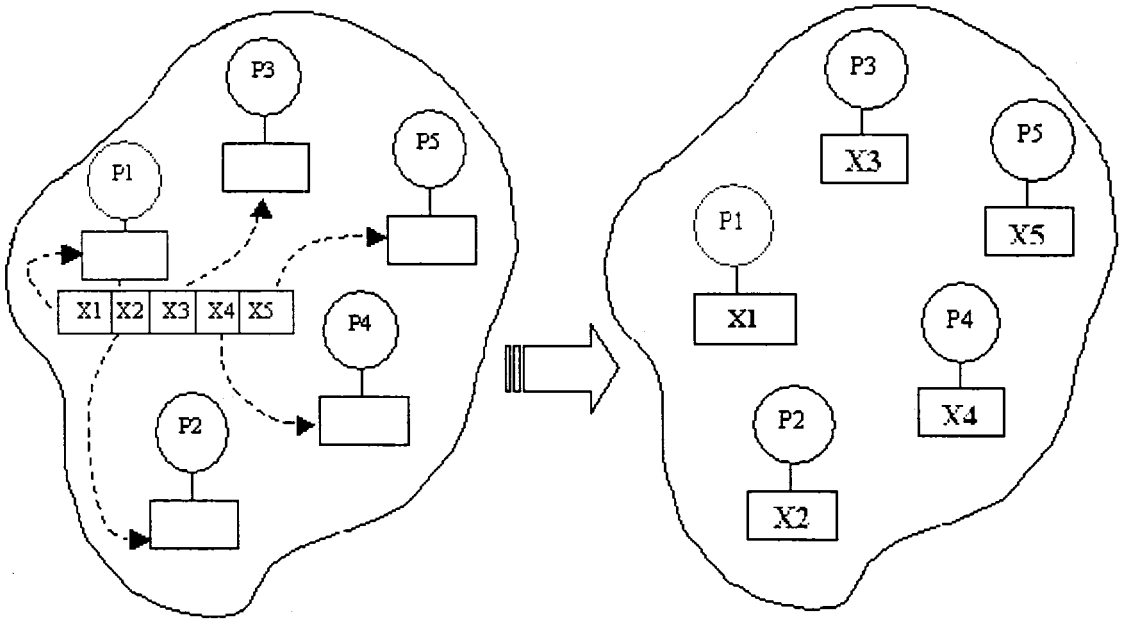


Figure 70 MPI_SCATTER

mpi_scatter *(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm,ierr)*

Partage *sendbuf* (de *sendcount*mp_size* éléments de type *sendtype*) du processus *root* entre les autres tâches. Le processus 0 recevra la première tranche de *sendcount*

éléments, le processus 1 recevra la deuxième. Le résultat est stocké dans *recvbuf*. Dans la plupart des cas, *sendcount=recvcount, sendtype=recvtype*.

3.4 Gather

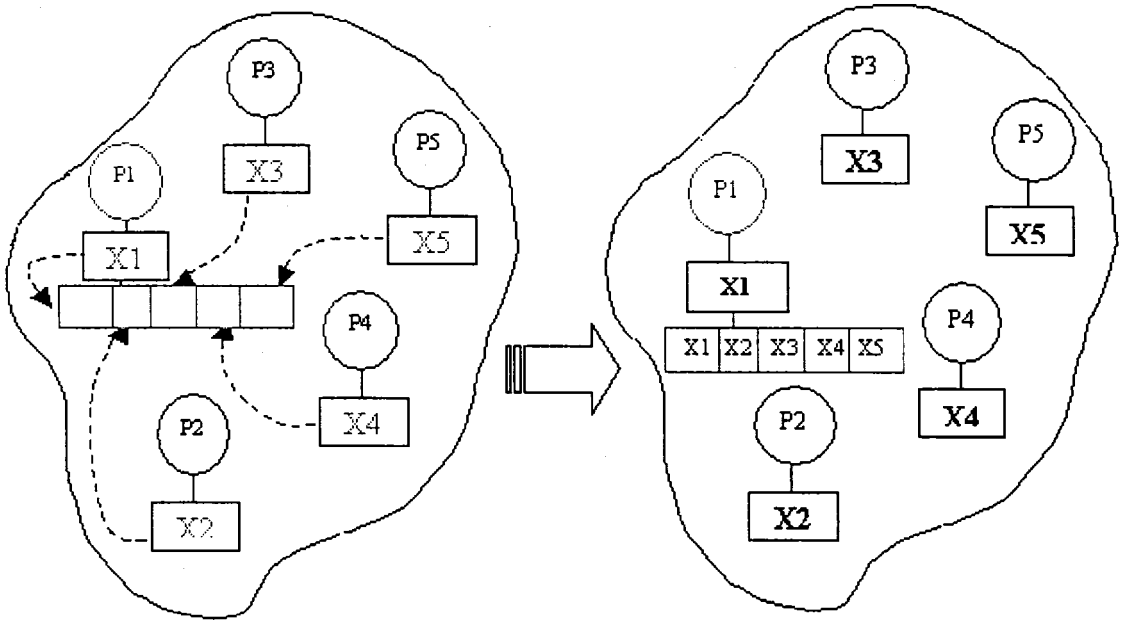


Figure 71 MPI_Gather

mpi_gather *(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,root,comm,ierr)*

Le process *root* reçoit dans *recvbuf* (un tableau de *sendcount*mp_size* éléments) des données (de *sendcount* éléments chacune) envoyées par chaque tâche. C'est la procédure exactement inverse à ***scatter***.

3.5 Allgather

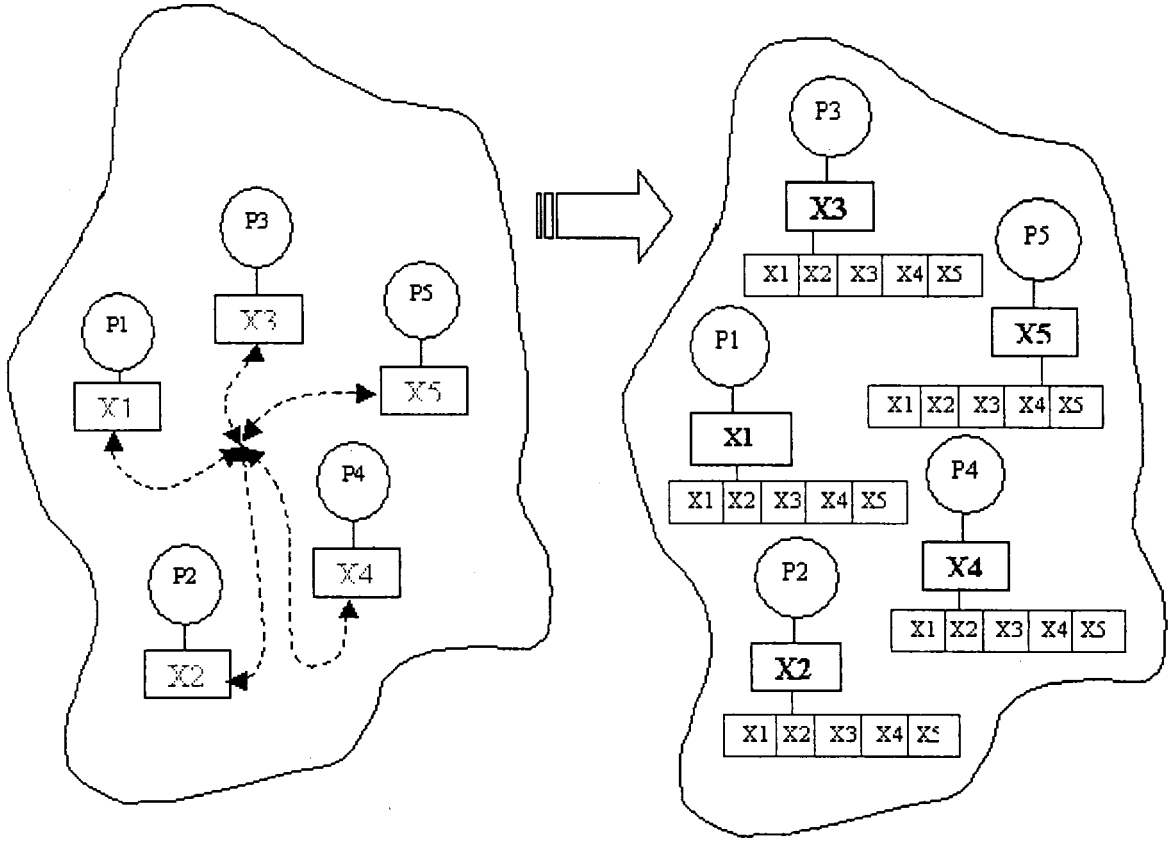


Figure 72 MPI_ALLGather

MPI_Allgather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

Pour communiquer des tables, *MPI_Gatherv* et *MPI_Scatterv* peuvent être utilisés.

4 Barrière de synchronisation

Pour synchroniser des tâches, il est possible de se servir de barrières:

*call **mpi_barrier** (mpi_comm_world,ierr)*

ANNEXES 3

Publication

Nonlinear Computational Aeroelasticity: Formulations and Solution Algorithms

A. Soulaïmani, A. BenElhajAli and Z. Feng
Département de génie mécanique, École de technologie supérieure, 1100 Notre-Dame Ouest, Montréal, Québec, Canada, H3C 1K3. E-mail : asoulaimani@mec.etsmtl.ca

Abstract

This paper highlights some technical features of an analysis methodology being developed for nonlinear computational aeroelasticity. A conservative finite element formulation for the coupled fluid/mesh interaction problem is proposed. Fluid-structure coupling algorithms are then discussed with some emphasis on distributed computing strategies. Numerical results are finally shown for the Agard 445.6 wing.

Introduction

Aeroelasticity is the study of the mutual interactions between aerodynamic, inertial and elastic forces on flexible structures, such as aircraft. The aerodynamic forces induced by the flow on an aircraft depend on the geometric configuration of the structure. On the other hand, the aerodynamic forces cause elastic deformations and displacements of the structure. Accurate prediction of aeroelastic phenomena such as static divergence and flutter is essential to the *design and control* of high performing and *safe* aircraft. Transport aircraft of the future are expected to become much more complex. With the advanced subsonic and the transonic civil aircraft, it is becoming increasingly important to perform static and dynamic aeroelastic analysis using highly accurate fluid and structural computational models. In general, classical aeroelasticity often leads to oversized aircraft design. Thus, more accurate computational capabilities for aeroelascitity analysis are desired. In general, aeroelasticity analysis treats static and dynamic aspects. Static analysis is usually associated with performance. In dynamic analysis, the concern is focused on safety through *stability*, and *dynamic response* studies. Instability problems occur when the structure sustains energy from the fluid that exceeds the capacity of elastic potential energy. Thus, there exists a critical flight speed beyond which instabilities take place that are characterized by high amplitude oscillations. Wing *flutter* is an example of such instabilities. *Buffeting*, which is the unsteady response of a structure caused by the fluctuations in the incoming flow, is another example. In aeroelastic response problems, one looks for the deformation and stress states in the structure as a response to turbulence or any unsteadiness in the flow. When the response of the structure is finite, the structure is stable. The structure flutters when its response to any finite disturbance is highly amplified.

We present an integrated CFD-CSD simulation methodology for flutter calculations based on distributed-parallel finite elements solvers. The main technical features of the proposed approach are:

- **Flow solver:** Stabilized Finite Element Formulations are being used for space discretization of the three-dimensional Euler and Navier-Stokes equations. An implicit parallel solver based on Schwarz Domain Decomposition methods and on the nonlinear version of the GMRES algorithm is developed.
- **Mesh solver:** Arbitrary-Lagrangian-Eulerian kinematics formulation is used in order to adapt the grid motion with the structure displacement. An ALE formulation is developed where, at any instant, the mesh configuration verifies the discrete form of the Geometric Conservation Law (DGCL). Furthermore, the dynamic mesh is modeled as a linear elastic material which undergoes large displacements.

- **Structure solver:** Finite Element commercial models for linear elasticity are used to perform the eigenvalue analysis. The time dependent structural displacement field is then computed using a classical Newmark scheme.
- **Distributed/Parallel computations:** Aeroelastic analyses are computationally intensive. Therefore, they can benefit from parallel processing technology. Intra parallelism (i.e., within each field) and inter parallelism (i.e., to couple the three fields fluid-mesh-structure) is performed using the message-passing paradigm. The computer code developed is designed to run on shared memory machines as well as on distributed machines such as Beowulf clusters. A Beowulf cluster recently developed at ETS containing 24 PCs and 12 GB of RAM is used presently as the distributed computing platform.
- **Fluid-Structure coupling:** Implicit solution algorithms are proposed to solve the CFD-CSD-Mesh coupled problem. It is based on the use of the Newton-GMRES algorithm for the entire problem along with a block preconditioning technique. Each block corresponds to *a functional* domain.
- **Matcher:** A fourth module is built in order to perform the tasks of load transfer from the fluid to the structure and the exchange of structure motion to the fluid.

The computational fluid dynamics code

PFES is our finite element code for the numerical solution of some multi-physics problems. For Euler and Navier-Stokes equations, a new formulation referred to as EBS (Edge Based Stabilized finite element formulation) is developed. This method combines in some way the best features of respectively the Galerkin finite element method (provides high order schemes, easy to implement on unstructured grids, etc.) and the Finite Volume based methods (e.g. ease to construct monotone upwind schemes on unstructured meshes). It was numerically demonstrated [1-3] that EBS can be less diffusive than SUPG [4-6] and the standard Finite Volume schemes. Accuracy is critical for solving shocks and separation regions present in the transonic regime. The implicit solver used is based on the Flexible GMRES algorithm preconditioned by the incomplete factorization ILUT [7]. The parallel version of the code makes use of the domain decomposition approach.

The Euler compressible equations in fixed meshes are written in a compact form and in terms of the conservation variables are written in a vector form as

$$V_{,t}+F_{i,i}^{adv}(V)=F_{i,i}^{diff}(V)+F_s \qquad (1)$$

where **V** is the vector which components are the specific momentum, density and specific total energy. F_i^{adv} and F_i^{diff} are respectively the convective and diffusive fluxes in the ith-space direction, and $\mathbf{F_s}$ is the source vector. Lower commas denote partial differentiation and repeated indices indicate summation. Diagonalizable Jacobian matrices can represent the convective fluxes $A_i=F_i^{adv},_V$. Recall that any linear combination of these matrices has real eigenvalues and a complete set of eigenvectors. It is well known that the standard Galerkin finite element formulation often leads to numerical instabilities for convective dominated flows. In the Galerkin-Least-Squares method (or the generalized Streamline Upwind Petrove Galerkin method), the Galerkin variational formulation is modified to include and integral form depending on the local residual **R(V)** of equation (1), i.e. $R(V)=V_{,t}+F_{i,i}^{adv}(V)-F_{i,i}^{diff}(V)-F_s$, which is identical to zero for the exact solution. The SUPG formulation reads: find **V** such that for all weighting functions **W**,

$$\sum_e \int_{\Omega_e}\left[W.(V_{,t}+F_{i,i}^{adv}-F_s)+W_{,i}F_i^{diff}\right]d\Omega \qquad (2)$$
$$-\int_\Gamma W\cdot F_i^{diff} n_i\, d\Gamma-\int_\Gamma W.A_{\tilde{n}}\,(V-V_\infty)\,d\Gamma+\int_\Omega \mu_c \nabla W.\nabla V d\Omega+\sum_e\int_{\Omega e} A_i^t.W_{,i}\;\tau\cdot R(V)\,d\Omega=0$$

In this formulation, the matrix τ is referred to as *the matrix of time scales*. The SUPG formulation is built as a combination of the standard Galerkin integral form and a perturbation-like integral form depending on the local residual vector. The third integral term in (2) takes into account the far field boundary conditions and the forth integral is a stabilizing term for the shocks. Recently, a new method referred to as the Edge-Based-Stabilized finite element method (EBS) was introduced to stabilize the standard Galerkin method while considering the real characteristics of the flow as computed on the normal direction of element edges. This

method has been proven to be stable and accurate for solving viscous and inviscid compressible flows, but more time consuming as compared to SUPG. Let us now briefly present the EBS formulation.

Consider the eigen-decomposition of $A_n=\sum A_i n_i, A_n=S_n \Lambda_n S_n^{-1}$.

Let $Pe_i=\lambda_i h/2\nu$ be the local Peclet number for the eigenvalue λ_i, h a measure of the element size on the element boundary, ν the physical viscosity and β_i = min (Pe_i/3, 1.0). We define the matrix $\boldsymbol{B_n}$ by

$$\boldsymbol{B_n}=\boldsymbol{S_n L S_n^{-1}} \tag{3}$$

where L is a diagonal matrix whose entries are given by $\boldsymbol{L_i} = (1 + \beta_i)$ if $\lambda_i > 0$; $\boldsymbol{L_i} = - (1 - \beta_i)$ if $\lambda_i < 0$ and L_i = 0 if $\lambda_i = 0$. The proposed EBS formulation is similar to (2) but the last integral term is replaced by:

$$+\sum_e \int_{\Gamma_e} \boldsymbol{W}\cdot\boldsymbol{\tau_n^{ed}}\cdot\boldsymbol{R(V)}d\Gamma=0 \tag{4}$$

with τ_n^{ed} the matrix of *intrinsic length scales* given by $\boldsymbol{\tau_n^{ed}}=\frac{h}{2}\cdot\boldsymbol{B_n}$ (5)

Geometrically conservative ALE formulation

One of the considerations in the mathematical formulation of conservation laws is the type of the kinematic description used for the material particles. A Eulerian description is very often used in fluid dynamics, while a Lagrangian is common in solid mechanics. When the material body contains moving boundaries, a mixed description, partially Lagrangian and partially Eulerian (also called Arbitrary Lagrangian-Eulerian), is more convenient [8-9]. This occurs for any flexible structure surrounded by a flow. It therefore becomes necessary to solve the fluid equations on a moving grid in order to match the fluid and the structure boundaries. Thus, besides the fluid and the structure material fields, there is a third field constituted by the moving mesh, which can be viewed as a material body having its own motion and dynamics. The three-field coupled problem is constituted by a set of partial differential equations, which are coupled through boundary conditions. A class of solution procedures called partitioned or segregated has been advocated to solve this coupled problem [10-14]. Given the displacements of the nodes on the wing, i.e. after solving the structure displacement field, a differential elliptic operator is designed to distribute the boundary motion inside the domain in order to avoid nodes collapsing or elements degenerating. After updating the mesh configuration, the flow field is solved on this mesh. It is shown [10] that the algorithm constructed for updating the dynamic mesh must obey a discrete Geometric Conservation Law (DGCL). A physical interpretation of the GCL is that the motion of the nodes must be compatible with the fact that the volume swept by the edges of the control volume or the element should be exactly equal to the variation of its volume (i.e. volume preserving). Our interpretation of the GCL goes as follows. The DGCL is equivalent to satisfying the kinematic Euler equation for the mesh, i.e.

$$\frac{\partial J}{\partial t}=J\,div\ \boldsymbol{w} \tag{6}$$

where $\boldsymbol{J}$ is the determinant of the geometric gradient tensor $\boldsymbol{F}$ from the current mesh configuration to the reference one and **w** is the velocity of a point of the moving domain. Equation (6) holds for the continuous medium. However, when applying discretization methods in space (FE, FD or FVM) and in time, it is not a priori satisfied due to time and space discretization errors. Those errors should not spoil the accuracy of the coupled problem. In [10,12,13], special time integration schemes satisfying the DGCL for first and second order time accuracy have been proposed in the context of FV methods. Substantial modifications to the original code, which has been developed for fixed meshes, are then required. Given the above interpretation, we propose to investigate the DGCL in another way. The mesh is not updated unless it satisfies equation (6) in a discrete form. Then, a finite element formulation can be easily constructed to solve the operator for distributing the boundary displacement inside the domain constrained to satisfying, in a weak form, the Piola-Kirchoff equation. By doing this, standard time integration procedures usually used for fixed meshes are accurate and still valid for dynamic meshes. Thus, CFD codes built for rigid meshes could be still valid for dynamic meshes. This idea is detailed in the following.

The conservation equations (1) in moving meshes are rewritten as

$$(JV)_{,t}+J\left(F_i^{adv}-w_i V\right)_{,i}=J\left(F_{i,i}^{diff}+F_s\right) \tag{7}$$

In the above equation, space differentiations are done with respect to the actual coordinates at the current time. Using simple differentiation operations, (7) is transformed into

$$(J_{,t} - J\,div\,w)V + J\,(V_{,t} + F_{i,i}^{adv} - w_i V_{,i} - F_{i,i}^{diff} - F_s) = 0 \tag{8}$$

and the corresponding classical Galerkin variational form is :

$$\int_{\Omega_0} W\cdot(J_{,t} - J\,div\,w)V\,d\Omega + \int_{\Omega} W\cdot(V_{,t} - F_{i,i}^{adv} - w_i\cdot V_{,i} - F_s)d\Omega + \int_{\Omega} W_{,i}F_i^{diff}\,d\Omega - \int_{\Gamma} W\cdot F_i^{diff}\,n_i\,d\Gamma = 0 \tag{9}$$

where Ω_0 is the configuration of the mesh at a reference time t_0 and Ω is the current configuration of the mesh at time t. For the continuous solution, Euler identity (6) is satisfied at any instant and for every point of the domain so that the first integral in (9) is identically zero. Thus, the variational formulation of the conversation equations is similar to the case of fixed mesh (up to the additional convective term $-w_i V_{,i}$ or in other words the advective matrices A_i are replaced in the case of moving domains by $A_i - w_i I$ with I the identity matrix):

$$\int_{\Omega} W\cdot(V_{,t} - F_{i,i}^{adv} - w_i\cdot V_{,i} - F_s)d\Omega + \int_{\Omega} W_{,i}F_i^{diff}\,d\Omega - \int_{\Gamma} W\cdot F_i^{diff}\,n_i\,d\Gamma = 0 \tag{10}$$

For a discrete numerical solution Euler identity is not necessarily satisfied. Thus the first integral term in (9) could not, in principle, be neglected. The usual DGCL condition states that for a dynamic mesh and for any arbitrary constant flow field Vc and for the case of $\mathbf{F_s} = 0$, the solution of the discrete problem should be exactly Vc , thus (9) gives

$$\int_{\Omega 0} \phi\,(J_{,t} - J\,div\,w)\,d\Omega = 0 \tag{11}$$

where ϕ is any weighting scalar function. Equation (11) is simply the variational form corresponding to the constraint (6). Note that (10) actually satisfies the usual DGCL condition discussed above. More generally, if it is desired for any purpose to use the non-conservative variational formulation (10) while satisfying explicitly the DGCL condition, the mesh motion could be subjected to the constraint (11). In the finite element methodology one usually interpolates **w** by continuous piece-wise functions, so that the mesh velocity field is continuous in the continuous medium (i.e. the mesh). Then, equation (6) is satisfied for the exact time differentiation of J. However, applying a time discretization scheme, similar to that used for the fluid, to the mesh coordinates to obtain **w** and to $\frac{\partial J}{\partial t}$ yields a truncation error $\frac{\partial J}{\partial t} - J\,div\,w = o(\Delta t^p)$ which is consistent with the truncation error obtained for the discrete (fluid) conservation equations. Thus, one can adopt formulation (10) along with an appropriate time discretization scheme for the evaluation of **w** , and the stability and convergence in time are expected to be obtained. On the other hand, recent theoretical studies by Letallec and his group [15] showed the impact of DGCL on the conservation of energy of the coupled fluid-structure system considered as a unique continuous medium. Energy conservation is a key point in studying fluid-structure interactions. In particular, the evolution of the kinetic energy must be controlled. Most time integration schemes do violate this principle of energy conservation when dealing with deformable domains. More precisely, for fully coupled schemes using conservative formulations and non-volume preserving grid configuration (DGCL), a small pollution term appears in the kinetic energy principle, which may grow exponentially in time. More specifically, it can be shown that using the first order Euler time discretization, the scheme is volume preserving if the fluid equations are integrated over a configuration Ω which is located at the mid distance between two successive configurations of the mesh [10] .

Mesh motion

Many choices can be considered in designing the operator that distributes the fluid-interface motion inside the moving domain. We consider that the mesh motion is defined by the elasto-static equations defined in the mesh configuration at time t :

$$\rho_m \mathbf{X}_{,tt} - div(\mathbf{P}(\mathbf{x})) = b \tag{12}$$

where ρ_m, **P** and **b** are fictitious density, the PK1 stress tensor and the body force. Equation (12) is solved for the mesh displacements **x** with the kinematics condition at the moving boundary $\mathbf{w} \cdot \mathbf{n} = \mathbf{u} \cdot \mathbf{n}$ for inviscid flows, with **u** the fluid velocity. The mesh velocity **w** is computed using a finite difference scheme (similar to that used for the fluid) to the differential equation $w = x_{,t}$. We usually take ρ_m and b equal to zero.

As the mesh moves, it is not always guaranteed that all elements keep an acceptable shape for accurate CFD computations. Especially, small elements are prompt to large distortions. In order to preclude negative volumes or large distortions for small elements, a suitable constitutive law for the mesh medium should be designed.

We consider the mesh as an elastic material undergoing small strains and large rotations. Thus, the PK2 stress tensor $S = P.\ F^{-t}$, (with F the deformation tensor) is linear with the Green strain tensor $E = (F^t F - I)/2$, thus:

$$S = C\,E \tag{13}$$

where C is the fictitious elastic modulo tensor which entries are of order $0(h^{-3})$ with h the element size.

In summary, equations (12) are solved for the finite displacements **x** between the mesh configurations at time t and time t+Δt along with the constitutive relation (13) and the boundary conditions. The non-zero boundary conditions for the mesh equations are actually the imposed displacements at the moving boundary which are computed by the CSD module. Note that the mesh motion problem can generate a large system of nonlinear equations. These are solved, at every time step, using a preconditioned Krylov algorithm (CG or even GMRES).

The CSD analysis

The finite element method is well established for solid and structure computations. In industry, commercial codes are very often used for linear structure analysis. In this work, we consider a classical linear model for the structure which can be described by modal equations as

$$\ddot{z}_i(t) + 2\eta_i\,\omega\,\dot{z}_i(t) + \omega_i^2 z_i(t) = s_{0i}(t) \tag{14}$$

with z_i the generalized normal mode displacement, η_i and ω_i are respectively the damping and the natural frequency of the i^{th} mode. Newmark's algorithm is used to integrate (14).

Coupling algorithms and distributed computing

In dynamic response problems, one looks for the successive flow and structure behaviours for a given set of initial conditions, such as a perturbation in the flow. In linear theory, the flutter speed of an aircraft can be obtained directly from the solution of an eigenvalue problem. In the nonlinear theory, for a given set of flight conditions, predicting whether an aircraft will flutter or not is much more complex and computationally intensive. There are two possible approaches (a) Starting from a deformed state of the structure the fluid-structure coupled solution is computed in the time domain [10-13], (b) Starting around an initial equilibrium state an eigenvalue problem is established by linearizing the coupled dynamic system [14]. The first approach is simpler to implement and enables to capture all the nonlinearities in the fluid-structure system.

Implicit time marching schemes enable the use of large time steps for the structure as well as for the mesh and the fluid fields. The conventional partitioned procedure commonly used in fluid-structure interactions is illustrated in Figure 1. It is based on the following steps:

1. Update the fluid grid to conform to the structural boundary at time t_n.
2. Advance the flow field using the new boundary conditions.
3. Update the surface load on the structure based on the fluid solution at time t_{n+1}
4. Advance the structure using the new fluid surface load.

For parallel computing, one can use an inter-field partitioned approach as illustrated in Figure 2.

1. The fluid grid is updated to conform to the structural boundary at time t_n and the fluid is advanced using the structural boundary conditions at time t_n
2. The structure is advanced using the fluid surface load at time t_n.

With this procedure, the CFD and CSD solvers can run in parallel during the time interval $[t_n , t_{n+1}]$. Inter-field communication and I/O transfer is needed only at the beginning of each time interval. As time progress, there may be a lag between the fluid and the structure so that a spurious energy exchange at the interface may generates undesirable instabilities.

In the following, an implicit iterative scheme is proposed to enhance the coupling between the mesh, the structure and the flow fields. After time and space discretizations of the fluid, structure and mesh equations, an algebraic system of equations for the unknown variables $\mathbf{S= (V, x, q)^t}$ (i.e. flow quantities and structure-mesh displacements) is obtained, which can be formally written as

$$G(S)=0 \tag{15}$$

This non-linear system can be solved using Newton's method as follows:

1. Given an initial structure and mesh configurations and a flow field for the current time step t_n;
2. Do n=1, maximum number of time steps;
3. Do i= 1, maximum number of iterations;
4. Find the correction $\Delta \mathbf{S}^i$ solution of :

$$H\,\Delta S^i=-G\left(S^{i-1}\right)$$

where **H** is the Jacobian matrix associated to **G**;

5. Check the convergence. If satisfied go to 6;
6. EndDo;
7. Update the global solution $\mathbf{S}^i= \mathbf{S}^{i-1}+\Delta \mathbf{S}^i$
8. EndDo.

In this algorithm, the Jacobian matrix **H** is needed. While it is difficult to develop its analytical expression, it is possible to compute an approximation. On the other hand, using Krylov-based iterative methods, such as the GMRES algorithm, to solve the linear system in step (4), one actually only need to compute the matrix-vector product of H and a direction vector z. A good approximation can be computed using a finite-difference formula like

$$Hz=\frac{G(V_0+\sigma z)-G(V_0)}{\sigma} \tag{16}$$

where σ is a small scalar.

To be efficient, Krylov methods need to be appended by a preconditioned, which is in principle a good approximation of **H**. A straightforward choice would be to use a *block-diagonal* matrix, which entries are the approximate Jacobians associated with respectively the flow, the structure and the mesh fields. In other words, the coupled problem is seen as a set of non-linear equations obtained by discretizing a continuous medium. These equations are solved iteratively using preconditioned Newton-GMRES method. While the global residual vector **G(S)** is needed, its main three components are computed by calling the corresponding modules (CFD solver, mesh solver or CSD solver). Obviously, these computations can be performed in parallel and inter-field communications are needed. This decomposition is referred to as the *functional decomposition*. On the other hand, the residuals of the flow field and the mesh motion are computed using a classical *domain decomposition approach*. For CFD, as well as for the mesh motion, we use a robust parallel iterative solver (i.e. intra-parallelism) based on Schwarz-Newton-Krylov techniques. Thus, the available processors are divided into groups of processors; each group is assigned to a specific field. Since CFD computations are more demanding, we assign more processors for the CFD domain than for the mesh solver.
Note that repeated updates of the geometry, for different flow conditions, are needed during the Newton-GMRES iterations. Thus, the flow field continuously drives the geometry so that a better conservation of the energy at the fluid-structure interface can be obtained. Since the main part of the computations is consumed within the CFD solver, the additional communications will not increase the overall simulation cost significantly, provided the number of time steps and non-linear CFD-iterations remains unchanged.
This algorithm is illustrated in Figure 3.

1. Given a geometry and a flow field at time t_n.
2. Compute the initial residual vector $\mathbf{R_0 = G(S_0)}$.
3. Perform Newton–GMRES iterations: Compute the residual for a perturbed solution in a direction **z**: $\mathbf{G(S_0 + \sigma z)}$; this has three major components each one is computed by its corresponding solver.
4. Perform inter-field communications.
5. Test for convergence and update the global solution for time step t_{n+1}.
6. Go to next step.
7.

Note that in the above algorithm one has to perform inter-filed communications in order to compute Krylov-directions for the global problem.
Several variants of this algorithm can be thought of. In its simplest form, one can perform a number of global iterations in which inter-filed communications occur only at the beginning of every iteration (i.e. this is nothing but the fixed-point algorithm). For a reasonable time step, one can observe the convergence for the fluid force transmitted to the structure and for the generalized coordinates after few global iterations.

Inter-field communications

Fluid-structure interactions involve the transfer of loads from the fluid mesh to the structure and the transfer of mesh structure motion to the moving mesh boundary. Since the CFD mesh is much finer than that used by the structure, the traces of these two meshes at the fluid-structure interface do not necessarily match. Then, load and displacements transfer cannot be done in a trivial way. The importance of conservative load transfer in fluid-structure interaction problems has recently been addressed in [16]. We adopt here the algorithm proposed in [16] which main steps consist of:

a) Pairing each fluid grid point S_j on the fluid interface Γ_f with the closet wet structural element $\Omega_S^{(e)} \in \Gamma_S$ (see Figure 4);
b) Determining the natural coordinates χ_j in $\Omega_S^{(e)}$ of the fluid point S_j (or its projection onto $\Omega_S^{(e)}$);
c) Interpolating the displacement of fluid nodes x_f inside $\Omega_S^{(e)}$ using the structure shape functions N_i^s ;
d) Projecting the generalized fluid force to the structure as:

$$f_i = \sum_{j=1}^{j=j_F} \Phi_j N_i^s(\chi_j)$$

with, $\Phi_j = \int_{\Gamma_f} (-pn + \sigma_f \cdot n) N_j^F d\gamma$

The generalized force associated to the fluid node S_j. It is proved that this algorithm preserves load and energy conservation at the fluid-structure interface. On the other hand, in aeroelasticity, the structure is often represented by plate, shell and beam elements which results in geometric discrepancies between the fluid mesh skin Γ_F and the structure boundary Γ_S (as illustrated in Figure 4). Thus, the above transfer algorithm is preceded by a projection step of the fluid nodes **M** onto the structure elements to locate the point **P** and to compute the gap vector **PM**. The motion of P is found using the above interpolation procedure and the motion of M is obtained by considering that the vector **PM** rotates with the structural element as it is done in classical plate theory.

Numerical results

Numerical tests have been carried out for the popular AGARD-445.6 [17] wing. The AGARD-445.6 is a thin swept-back and tapered wing with a symmetrical NACA 65A004 airfoil section. The weakened model-3 is considered here. A coarse unstructured grid is employed for Euler computations that has 37965 nodes, 177042 elements and generates 388464 coupled equations. For the structure, a mesh of 1176 quadrilateral shell elements and 1250 nodes. Using the commercial software Ansys, the firs five dry modes have been computed. There respective frequencies are the following: 9.6 Hz, 39.42Hz, 49.60 Hz, 96.095Hz and 126.30 Hz. The shape modes compare well with those of Yates (figure 5). A flow at a free-stream Mach number of 0.96 and zero angle of attack is computed first. At t=0, a Dirac force is imposed on the tip of the wing. The response

of the wet structure is computed for different free stream pressures q. We use a second order time differentiation scheme with a non dimensional time step $\Delta t = 0.2$ and three global iterations. The experimental flutter pressure at M= 0.96 is q = 61.3 lb/sq ft [17]. Figure 6 shows that at the computed conditions (M= 0.96 is q = 62.0 lb/sq ft) the structure is neutrally stable and the first two modes are in coalescence. Figure 7 shows a comparison of two models for the mesh motion (with q= 71.3), a nonlinear model as described previously (used the above computations) and a linear one where **F= I** and where the second order terms in **E** are dropped. In the linear model the mesh becomes distorted as the motion amplifies until it collapses at time step 290 (negative jacobians). With the nonlinear model, computations run for 900 time steps. Finally, figure 8 shows a comparison of the flutter boundary obtained with our code with the experimental observations [17] and with some numerical results reported in the literature[18 , 19 and 20].

Conclusion

In this paper we have presented a CFD-based aeroelastic model. A suitable finite element formulation is used for all computational fields (fluid, mesh and structure). A functional decomposition approach is used for the solution of the coupled problem. For every physical field a parallel GMRES algorithm is employed to solve the corresponding discrete system. Inter-filed communications occur during global quasi-Newton coupling iterations. Numerical tests on the Agard 445.6 aeroelastic wing show that the inter-filed communications strongly enhance the numerical stability of the time marching procedure. The flutter dip is well produced when three global coupling iterations are used. On the other hand, a nonlinear model for the moving mesh is proposed. Numerical tests show that this model improves the robustness of the aeroelastic code. Finally, an accurate flutter boundary is obtained for the Agard445.6 wing aeroelastic test case.

Acknowledgments

This research has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), PSIR research program of ETS and La Fondation J. Armand Bombardier.

References

[1] A. Soulaïmani and A. Rebaine. "An edge based stabilized finite element method for solving compressible flows", the 14th AIAA Computational Fluid Dynamics Conference, paper AIAA-99-3269.

[2] A. Soulaïmani and C. Farhat. "On a finite element method for solving compressible flows", ICES98, Atlanta, 1998.

[3] A. Soulaïmani, A. Rebaine and Y. Saad. "An edge based stabilized finite element method for solving compressible flows: formulation and parallel implementation", Comput. Methods Appl. Mech. Engrg, vol. 190, pp.6735-6761 (2001).

[4] T.J.R. Hughes and Mallet. "A new finite element formulation for computational fluid dynamics III. The generalized streamline operator for multidimensional advective-diffusive systems", Comput. Methods Appl. Mech. Engrg, 58, 305-328, 1986.

[5] A. Soulaïmani and M. Fortin. "Finite element solution of compressible viscous flows using conservative variables", Comput. Methods Appl. Mech. Engrg., 118, 319-350, 1994.

[6] N.E. Elkadri, A. Soulaïmani and C. Deschenes. "A finite element formulation of compressible flows using various sets of independent variables", Comput. Methods Appl. Mech. Engrg., 181, 161-189, 2000.

[7] Y. Saad. "Iterative Methods for Sparse Linear Systems", *Numerical Linear Algebra with Applications*, 1:387-402, 1994.

[8] J. Donea. "An arbitrary Lagrangian-Eulerian finite element method for transient fluid structure interactions", Comput. Meths. Appl. Mech. Engrg., 33, 689-723, 1982.

[9] A. Soulaïmani and Y. Saad: "An Arbitrary Lagrangian Eulerian finite element formulation for solving three-dimensional free surface flows", 162, 79-106, 1998.

[10] C. Farhat and M. Lesoinne. "On the accuracy, stability, and performance of the solution of three-dimensional nonlinear transient aeroelastic problems by partitioned procedures", AIAA-96-1388, 1996.

[11] R. Löhner et al. "Fluid-Structure Interaction Using a Loose Coupling Algorithm and Adaptive Unstructured Grids", AIAA Paper 95-2259.

[12] M. Lesoinne and C. Farhat. "Geometric conservation laws for aeroelastic computations using unstructured dynamic meshes", AIAA Paper 95-1709.

[13] C. Farhat and M. Lesoinne. "Two efficient staggered algorithms for the serial and parallel solution of three-dimensional nonlinear transient aeroelastic problems", Comput. Maths. Appl. Mech. Engrg, 182, 499-515, 2000.

[14] M. Lesoinne and C. Farhat. "CFD-based aeroelastic eigensolver for the subsonic, transonic and supersonic regimes", Journal of Aircraft, vol 38, no 4, 2001.

[15] T. Fanion, M. Fernandez and P. LeTallec: "Deriving adequate formulations for fluid-structure interaction problems : from ALE to transpiration". Revue Européenne des éléments finis. Volume 9-6, pp. 681-70, 2000.

[16] C. Frahat, M. Lesoinne and P. LeTallec . "Load and motion transfer algorithms for fluid/structure interaction problems with non-matching discrete interfaces: Momentum and energy conservation, optimal discretization and application to aeroelasticity". Comput. Maths. Appl. Mech. Engrg, vol. 157, pp. 95-114, 1998.

[17] E.C. Yates. "AGARD Standard Aeroelastic Configuration for Dynamics Response, Candidat Configuration I.-Wing 445.6", NASA TM 100492, 1987.

[18] E.M. Lee-Rausch, J.T. Batina. "Wing-flutter boundary prediction using unsteady Euler equations aerodynamic methd". AIAA Paper No. 93-1422, 1993.

[19] K.K. Gupta. "Development of a finite elemnt aeroelastic analysis capability". J. Aircraft 33 (1996) 995-1002.

[20] M. Lesoinne, M. Sarkis, U. Hetmaniuk and C. Farhat. "A linearized method for the frequency analysis of three-dimensional fluid/structure interaction problems in all flow regimes". Comput. Maths. Appl. Mech. Engrg. 190 (2001) 3121-3146.

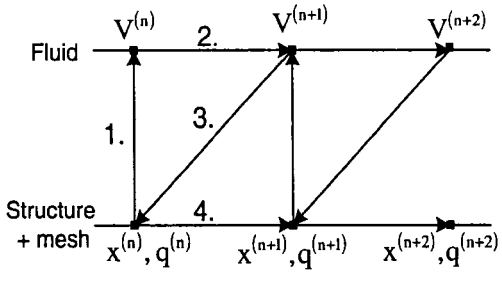


Figure 1. Conventional partitioned procedure

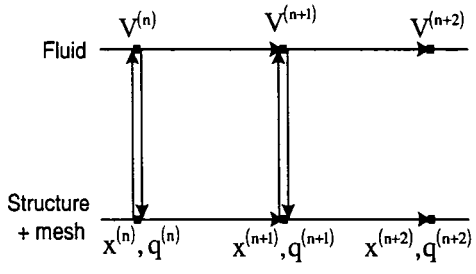


Figure 2. Inter-field partitioned procedure

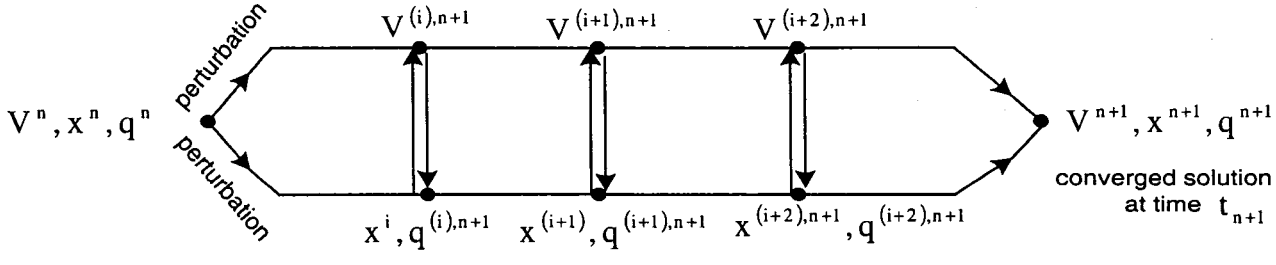


Figure 3. A fully implicit coupled procedure

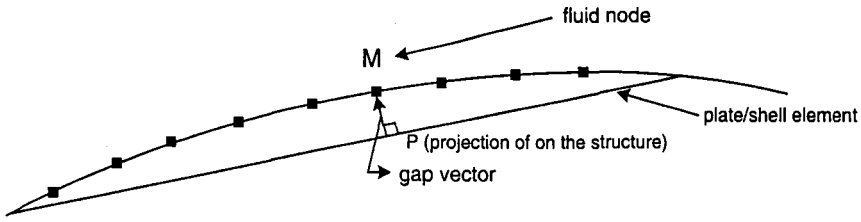


Figure 4. Fluid-solid interface moves according to plate theory

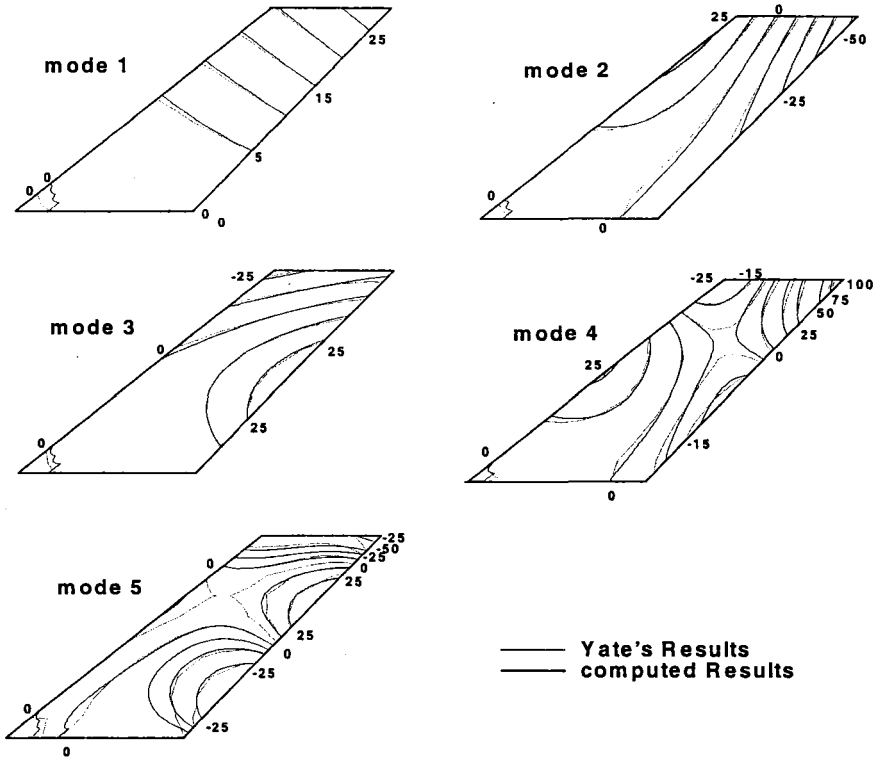


Figure 5. Agard 445.6 structure shape modes

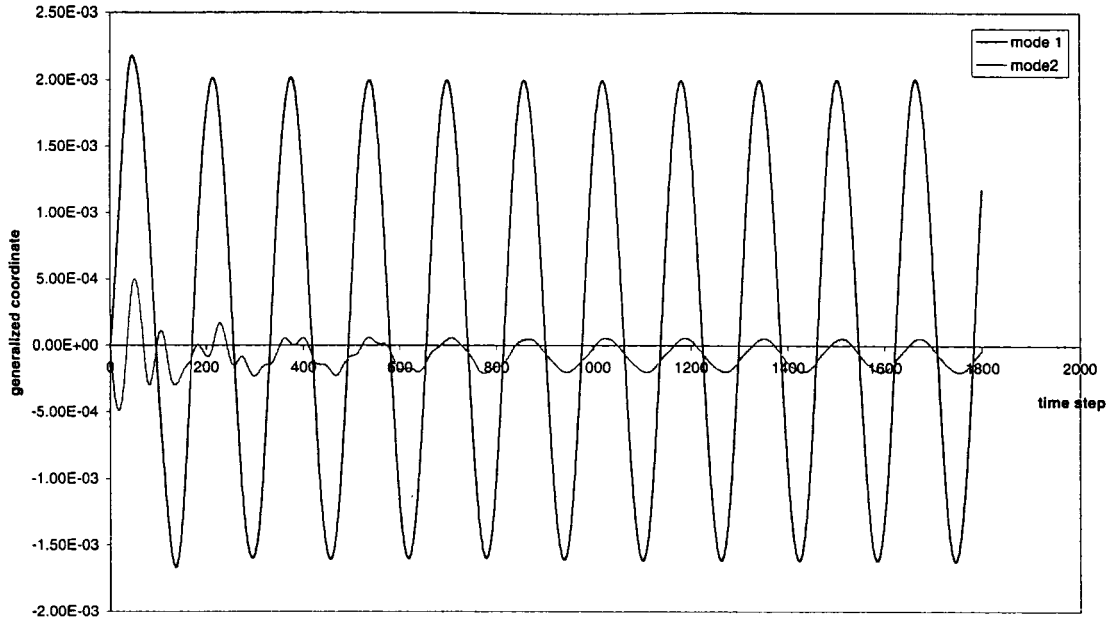


Figure 6. Time history for the first and second modes for Mach= 0.96 and q= 62 lb /sq ft and using three coupling iterations

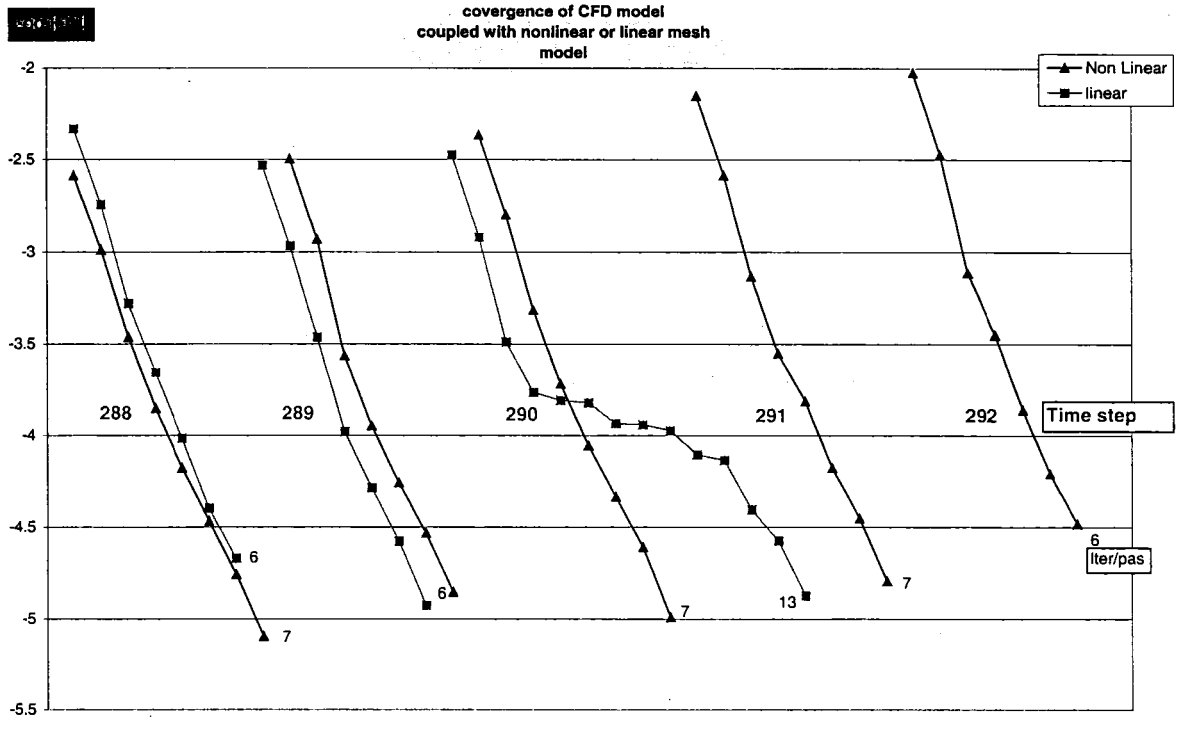


Figure 7. Convergence history for the CFD computations using respectively. A linear and a nonlinear mesh model

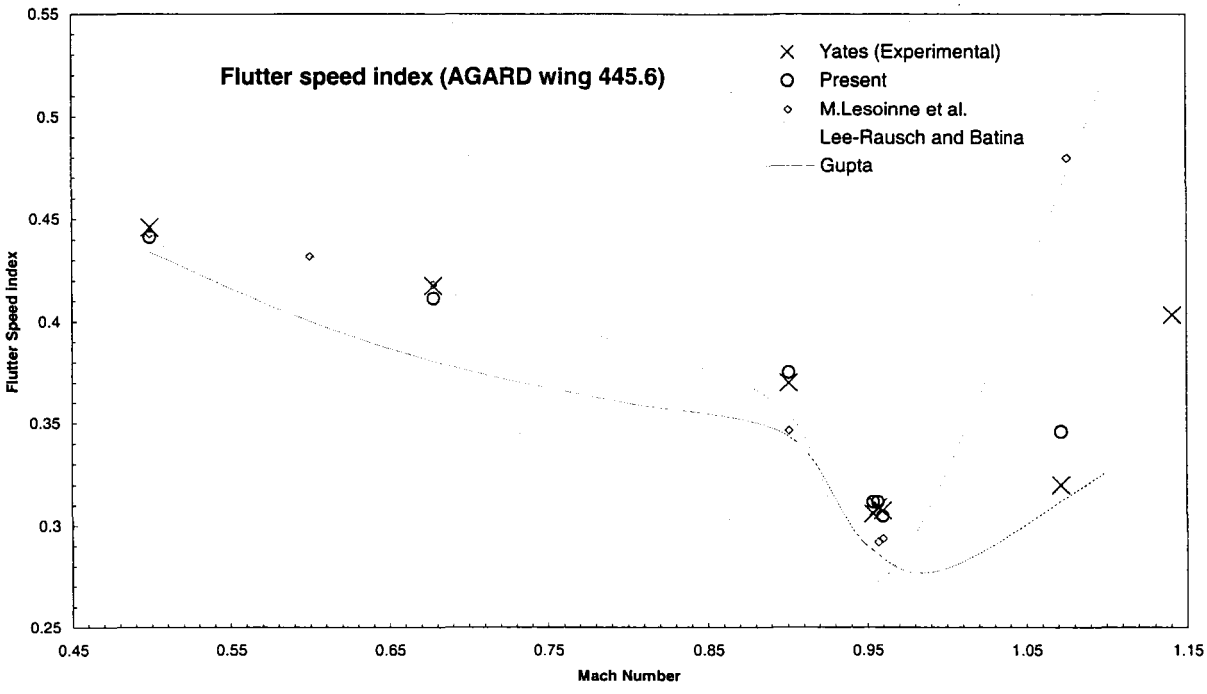


Figure 8. Flutter speed index (AGARD wing 445.6)

ANNEXES 4

Temps de calcul

Temps de calcul sur la machine **Thunderbird**

N de processeurs			Problème d'aéroélasticité				Problème non-couplé		
			Temps CPU (min)		Mémoire (Mo)	Temps Réel (min)	Temps CPU (min)	Mémoire (Mo)	Temps Réel (min)
Total	Fluide	Mesh	Fluide	Mesh	Fluide		Fluide	Fluide	Fluide
5	3	2	278	56	500	362	242	498	301
6	4	2	215	56	395	228	186	394	189
	3	3	278	43	500	356	242	498	301
7	5	2	181	56	314	192	157	311	161
	4	3	212	43	396	220	186	394	189
8	6	2	133	56	266	165	115	263	137
	5	3	181	43	314	192	157	311	161
10	8	2	109	56	217	125	93	215	101
	7	3	124	43	240	144	105	238	117
	6	4	133	31	266	165	186	394	189
11	9	2	90	56	190	114	75	188	90
	8	3	109	43	217	125	93	215	101
	7	4	124	26	240	143	105	238	117
	6	5	134	21	266	165	186	394	189
12	10	2	78	56	174	104	64	171	82
	9	3	90	43	189	114	75	188	90
	8	4	109	26	217	124	93	215	101
	7	5	124	21	240	143	105	238	117
13	11	2	76	56	165	93	63	163	71
	10	3	78	44	173	104	64	171	82
	9	4	90	26	189	114	75	188	90
	8	5	109	21	217	124	93	215	101
14	12	2	68	56	152	87	57	151	68
	11	3	76	43	165	93	63	163	71
	10	4	78	31	173	104	64	171	82

	9	5	90	27	189	114	75	188	90
	8	6	109	19	217	125	93	215	101
15	13	2	62	56	142	82	51	140	63
	12	3	68	43	152	87	57	151	68
	11	4	76	26	165	93	63	163	71
	10	5	78	21	173	104	64	171	82
	9	6	90	19	189	114	75	188	90
	15						48	123	53
16	14	2	55	56	129	79	50	125	60
	13	3	62	42	142	83	51	140	63
	12	4	68	26	152	87	57	151	68
	3	13	275	14	500	335	242	498	301
	16						46	116	50

ANNEXES 5

Liste des routines

Routine d'initialisation du communicateur **Neighbors :**

```
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~!
      SUBROUTINE INIT_voisinage(nproc,proc)
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~!
      use global_data,only:myid,Family,Neighbors,NUMBROTHER
      USE INTERFACE_DATA,only:IALLOCINIT
      implicit none
      include 'mpif.h'
      INTEGER :: i,FamilyGroup,voisins(nproc+1)
      INTEGER :: nproc,proc(*),ierr,proc_voisins(NUMBROTHER)
      INTEGER :: NPSIZE(NUMBROTHER),pointeur(NUMBROTHER+1),a,b(4)
      INTEGER,DIMENSION(:),POINTER ::voisinage
      INTEGER :: voisinagedim,test,j

  CALL IALLOCINIT(Neighbors,NUMBROTHER,'Neighbors . Groups')
!---- Connaitre le groupe associe au communicateur FAMILY
      call MPI_COMM_GROUP(FAMILY,FamilyGroup,ierr)

      voisins(1)=myid-1
      do i=2,nproc+1
      voisins(i)=proc(i-1)-1
      enddo
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      call MPI_ALLgather(nproc+1,1,MPI_INTEGER,NPSIZE,1,MPI_INTEGER,FAMILY,ierr)
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      pointeur=0
      pointeur(1)=1
      do i=2,numbrother+1
      pointeur(i)=pointeur(i-1)+npsize(i-1)
      enddo
      voisinagedim=sum(NPSIZE)
      CALL IALLOCINIT(voisinage,voisinagedim,'voisinage . Groups')
      voisinage=0
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      call MPI_ALLgatherv(voisins,nproc+1,MPI_INTEGER,voisinage,npsize,
     *               pointeur-1,MPI_INTEGER,FAMILY,ierr)
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
      do i=1,numbrother
!---- Creer les groupes de processeurs voisins

      call MPI_GROUP_INCL(FamilyGroup,npsize(i),voisinage(pointeur(i):pointeur(i+1)-1),
     *                              proc_voisins(i),ierr)

!---- Creer le communicateur entre les voisins
      Call MPI_COMM_CREATE(FAMILY,proc_voisins(i),Neighbors(i),ierr)
      enddo

      RETURN

      END SUBROUTINE INIT_VOISINAGE
```

Routine d'initialisation du communicateur **Neighbors :**

```
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~!
      SUBROUTINE INIT_voisinage(nproc,proc)
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~!
      use global_data,only:myid,Family,Neighbors,NUMBROTHER
      USE INTERFACE_DATA,only:IALLOCINIT
      implicit none
      include 'mpif.h'
      INTEGER :: i,FamilyGroup,voisins(nproc+1)
      INTEGER :: nproc,proc(*),ierr,proc_voisins(NUMBROTHER)
      INTEGER :: NPSIZE(NUMBROTHER),pointeur(NUMBROTHER+1),a,b(4)
      INTEGER,DIMENSION(:),POINTER ::voisinage
      INTEGER :: voisinagedim,test,j

  CALL IALLOCINIT(Neighbors,NUMBROTHER,'Neighbors . Groups')
!---- Connaitre le groupe associe au communicateur FAMILY
      call MPI_COMM_GROUP(FAMILY,FamilyGroup,ierr)

      voisins(1)=myid-1
      do i=2,nproc+1
      voisins(i)=proc(i-1)-1
      enddo
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      call MPI_ALLgather(nproc+1,1,MPI_INTEGER,NPSIZE,1,MPI_INTEGER,FAMILY,ierr)
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      pointeur=0
      pointeur(1)=1
      do i=2,numbrother+1
      pointeur(i)=pointeur(i-1)+npsize(i-1)
      enddo
      voisinagedim=sum(NPSIZE)
      CALL IALLOCINIT(voisinage,voisinagedim,'voisinage . Groups')
      voisinage=0
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      call MPI_ALLgatherv(voisins,nproc+1,MPI_INTEGER,voisinage,npsize,
     *              pointeur-1,MPI_INTEGER,FAMILY,ierr)
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
c          ix(1:i1) contains the nodes adjacent to processor proc(1)
c          ix(i1+1:i2)  the nodes adjacent to proc(2), etc..
c    ipr   = pointer to beginning of boundary nodes for each processor,
c          in the above example ipr(1) = 1, ipr(2) = i1+1, ....
c
c return code:
c-------------
c
c      ierr =     0 --- boundary information has been sent
c                    and received
c                less than 0 --- error
c
c-----------------------------------------------------------------------
c    local variables
     include 'mpif.h'
     integer  len, kin, kout, ii, j, iproc,lof,i

     if( nproc.gt.99) then
       ierr = -3
       return
     endif

c    gather boundary data in y
c
     kin=nloc+1
     do j=ipr(1), ipr(nproc+1)-1
       y(j+nloc) = x(ix(j))
     enddo

c
c
       call MPI_BARRIER(FAMILY,ierr)

      ipr2=1
      iwk=0
```

```
      msg=0
      do ii = 1,nproc
      ipr2(ii+1)=ipr(ii)+nloc
      msg(ii+1)  = ipr(ii+1)-ipr(ii)
      iwk(proc(ii))=proc(nproc+ii)
      enddo

      do ii=1,numbrother
      if(Neighbors(ii)==0) cyCle
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      call MPI_Scatterv(y,msg,ipr2-1,MPI_double_precision,
     *                  x(kin),iwk(ii),MPI_double_precision,
     *                   0,Neighbors(ii),ierr)
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      kin=kin+iwk(ii)
      enddo
     return
     end
```

Routine d'allocation Dynamique :

```
SUBROUTINE IALLOCINIT (ITAB, DIM, name, r,restart)
!=============================================================================!
!Version Parallele
! ============================================================================!
!  * Si [ITAB] n est pas associee :
!+ Allocation de memoire pour un tableau dynamique [ITAB] d etendue DIM !
!+ Gestion des erreurs d allocation memoire
!+ Initialisation du tableau a la valeur entière 0
!+ name : nom de la table, optionel
!
!  * Si [ITAB] est deja associee :
!+ creation d une table temporaire [temp]
!+ copier [ITAB] dans [temp]
!- creer la table [ITAB] de nouveau avec la nouvelle dimension :
!+ Cette dimension est DIM si ce dernier est fournit
!+ si non la dimension sera SIZE(ITAB)*r
!+ copier [temp] dans [ITAB]
!+ liberer [temp]
! ============================================================================!
use global_data,only: hist,NIMP
implicit none
INTEGER, dimension(:),pointer :: ITAB,temp
INTEGER, optional :: DIM,restart
INTEGER :: ALLOC_OK,dim2,dim3,mp=33
CHARACTER (LEN = *),OPTIONAL :: name
real,optional :: r
real::cor2
cor2=1.2
! ----------------------------------------------------------------------------
if(.not.associated(ITAB) ) then
! ----------------------------------------------------------------------------
!-----  ALLOCATION

IF (.not.PRESENT(DIM)) then
```

```
write(mp,'("Dimension introuvable")')
IF (PRESENT(name)) then
write(mp,'("lors de l allocation de ",a18)')name
      endif
      call flush(mp)
    call fin_de_programme(2,'Probleme de programmation lors de l allocation de '//name)
endif


! ----- ALLOCATION DE MEMOIRE
ALLOCATE (ITAB(DIM), STAT= ALLOC_OK)

! ----- GESTION D ERREURS
      IF (ALLOC_OK > 0) THEN
      write(mp,'("MEMOIRE INSUFFISANTE")')
      write(mp,'("IMPOSSIBLE D ALLOUER LA MEMOIRE POUR LA TABLE [", a18,"]")') name
      write(mp,'("PROGRAMME ARRETE.")')
      call flush(mp)
      call fin_de_programme(2,'Probleme de memoire')

 ENDIF

! ----- INITIALISATION
ITAB = 0
     if(NIMP>0)
   * write(hist,'(1x,"Alloc",4x,a18,2x,"DIM :",i10)') name,dim
! ---------------------------------------------------------------------------
else
! ---------------------------------------------------------------------------
!----- REALLOCATION

!----- Calcule de la dimension de [temp]

dim2=size(ITAB)
IF (PRESENT(r)) cor2 = r
IF (.not.PRESENT(DIM)) then
dim3=ceiling(dim2*cor2)
else
```

```
dim3=DIM
endif
!----- Allocation de la memoire pour [temp]

allocate(temp(min(dim3,dim2) ), STAT= ALLOC_OK)

! ---- GESTION D ERREURS

IF (ALLOC_OK > 0) THEN

if (dim2 >= dim3) then
      if(NIMP>0) then
      write(hist,'("MEMOIRE INSUFFISANTE")')
      write(hist,'("IMPOSSIBLE D ALLOUER LA MEMOIRE POUR LA TABLE [",a18,"]")') name
write(hist,*) 'Continue..'
call flush(hist)
endif
go to 100
else

IF ((PRESENT(restart)).and.(restart==0))then
restart=1
deallocate(ITAB)
      if(NIMP>0)write(hist,*) 'Restart...',name
goto 100
else
      write(mp,'("MEMOIRE INSUFFISANTE")')
      write(mp,'("IMPOSSIBLE D ALLOUER LA MEMOIRE POUR LA TABLE [",a18,"]")') name
write(mp,'("PROGRAMME ARRETE.")')
call flush(mp)

    call fin_de_programme(2,'Probleme d allocation')
    endif
 endif

 ENDIF
```

```
!----- copier [ITAB] dans [temp]
 temp=ITAB(1:size(temp))

!---- liberer la memoire

 deallocate(ITAB)

!----- REAllocation de la memoire pour [ITAB]
ALLOCATE (ITAB(dim3), STAT= ALLOC_OK)
! ---- GESTION D ERREURS

IF (ALLOC_OK > 0) THEN
       write(mp,'("MEMOIRE INSUFFISANTE")')
       write(mp,'("IMPOSSIBLE D ALLOUER LA MEMOIRE POUR LA TABLE [",a18,"]")') name
       write(mp,'("PROGRAMME ARRETE.")')
       call flush(mp)
     call fin_de_programme(2,'Probleme de memoire')
 ENDIF

!----- Initialisation
 ITAB=0

!----- copier [ITAB] dans [temp]
ITAB(1:size(temp))=temp

!---- liberer la memoire
deallocate(temp)
      if(NIMP>0) then
      write(hist,'("* Realloc ",a18,2x,"DIM :",i10,1x,"-->",1x,i10)') name,dim2,dim3
      endif
! ----------------------------------------------------------------------
end if
! ----------------------------------------------------------------------
       if(NIMP>0) call flush(hist)
100  END SUBROUTINE IALLOCINIT
```

La Routine **PGMRES** :

```
!=======================================================================
      subroutine pgmres1(myproc,n, im, rhs, sol, vv, eps, maxits,
     * iout,  aa, ja, ia, alu, jlu, ju, ier,
     * ipas,iter,xnorm, nbnd,nproc,proc,ix,iprr,neq,
     * riord,jb,tmp,tmpd,icount,iter_count)
!=======================================================================

c-----------------------------------------------------------------------
c
c              *** ILUT - Preconditioned GMRES ***
c
c-----------------------------------------------------------------------
c This is a simple version of the ILUT preconditioned GMRES algorithm.
c The ILUT preconditioner uses a dual strategy for dropping elements
c instead  of the usual level of-fill-in approach. See details in ILUT
c subroutine documentation. PGMRES uses the L and U matrices generated
c from the subroutine ILUT to precondition the GMRES algorithm.
c The preconditioning is applied to the right. The stopping criterion
c utilized is based simply on reducing the residual norm by epsilon.
c This preconditioning is more reliable than ilu0 but requires more
c storage. It seems to be much less prone to difficulties related to
c strong nonsymmetries in the matrix. We recommend using a nonzero tol
c (tol=.005 or .001 usually give good results) in ILUT. Use a large
c lfil whenever possible (e.g. lfil = 5 to 10). The higher lfil the
c more reliable the code is. Efficiency may also be much improved.
c Note that lfil=n and tol=0.0 in ILUT  will yield the same factors as
c Gaussian elimination without pivoting.
c
c ILU(0) and MILU(0) are also provided for comparison purposes
c USAGE: first call ILUT or ILU0 or MILU0 to set up preconditioner and
c then call pgmres.
c-----------------------------------------------------------------------
c Coded by Y. Saad - This version dated May, 7, 1990.
c-----------------------------------------------------------------------
```

```
c parameters
c----------
c on entry:
c==========
c
c n    == integer. The dimension of the matrix.
c im   == size of krylov subspace:  should not exceed 50 in this
c         version (can be reset by changing parameter command for
c         kmax below)
c rhs  == real vector of length n containing the right hand side.
c         Destroyed on return.     ontaining an initial guess to the
c sol  == real vector of length n containing an initial guess to the

c         solution on input. approximate solution on output
c eps  == tolerance for stopping criterion. process is stopped
c         as soon as ( ||.|| is the euclidean norm):
c         || current residual||/||initial residual|| <= eps
c maxits== maximum number of iterations allowed
c iout  == output unit number number for printing intermediate results
c         if (iout .le. 0) nothing is printed out.
c
c aa, ja,
c ia   == the input matrix in compressed sparse row format:
c         aa(1:nnz)  = nonzero elements of A stored row-wise in order
c         ja(1:nnz) = corresponding column indices.
c         ia(1:n+1) = pointer to beginning of each row in aa and ja.
c         here nnz = number of nonzero elements in A = ia(n+1)-ia(1)
c
c alu,jlu== A matrix stored in Modified Sparse Row format containing
c          the L and U factors, as computed by subroutine ilut.
c
c ju    == integer array of length n containing the pointers to
c          the beginning of each row of U in alu, jlu as computed
c          by subroutine ILUT.
c
c on return:
c==========
c sol  == contains an approximate solution (upon successful return).
c ier  == integer. Error message with the following meaning.
```

```
c        ier = 0 --> successful return.
c        ier = 1 --> convergence not achieved in itmax iterations.
c        ier =-1 --> the initial guess seems to be the exact
c                    solution (initial residual computed was zero)
c
c-----------------------------------------------------------------------
c
c work arrays:
c=============
c vv    == work array of length  n x (im+3) (Used to store the Arnoli
c         basis)
c-----------------------------------------------------------------------
c subroutines called :
c amux   : SPARSKIT routine to do the matrix by vector multiplication
c         delivers y=Ax, given x  -- see SPARSKIT/BLASSM/amux
c lusol0 : combined forward and backward solves (Preconditioning ope.)
c BLAS1  routines.
c-----------------------------------------------------------------------
c Modified by Azzeddine SOULAIMANI : la matrice a des lignes et des
c colonnes ordonnes comme dans PSPARSLIB
c-----------------------------------------------------------------------
      use global_data,only:FAMILY, myfamily, famtype
      IMPLICIT NONE
      include 'mpif.h'
 integer n, im, maxits, iout, ier,nbnd, ja(*), ia(*),jlu(*), ju(*), nproc, neq, icount(*),im3,im2,i1,itype
 DOUBLE PRECISION vv(n,*), rhs(n), sol(n),eps1,eps,eps2,ddot,gam,tloc,tmp(*),tmpd(*),
     *                          vnorm_rhs,vnorm_sol,dnrm2
integer iter_count,riord(*), jb(*),iprr(*),ix(*),proc(*)
DOUBLE PRECISION summ,zero
integer ::iptrn,n1,its,ierr,j,i,k, k1,ii
DOUBLE PRECISION    aa(*)
DOUBLE PRECISION    alu(*)
integer ,parameter:: kmax=50
DOUBLE PRECISION hh(kmax+1,kmax), c(kmax) ,epsmac
DOUBLE PRECISION s(kmax), rs(kmax+1),t, wnormt,ro
DOUBLE PRECISION hhloc(kmax+1,kmax),dsqrt, wnorm
DOUBLE PRECISION xnorm,eps_pert,dnorm,dnormt,rao
integer ipas,iter,kkj
integer myproc
```

```
c-------------------------------------------------------------
c arnoldi size should not exceed kmax=50 in this version..
c to reset modify paramter kmax accordingly.
c-------------------------------------------------------------
      data epsmac/1.d-16/
      IF(trim (famtype(myfamily))=='mesher')THEN
      eps2=1.d-07
      ELSE !-----
      eps2=1.d-02
      ENDIF !----
c
      zero=0.0d0
      itype=12
      iptrn=12
c
      n1 = n + 1
      im2=im+2
      im3=im+3
      its = 0

c-------------------------------------------------------------
c outer loop starts here..
c
      itype=12
      iptrn=12

c--- calcul de la norme de sol



      dnorm=ddot(n,sol,1,sol,1)
c
      write(33,*)' Dnorm',dnorm
      call flush(33)
      call MPI_BARRIER(FAMILY,ierr)
      dnormt=zero
      call MPI_allreduce(dnorm,dnormt,1,MPI_double_precision,MPI_sum,FAMILY,ierr)
c
```

```
      dnormt=dsqrt(dnormt)

      eps_pert=eps2*(dnormt+eps2)
      vnorm_sol=dnormt

      write(iout, 198) ipas,eps_pert
      call flush(iout)

c store initial residual vector
      do 21 j=1,n
      vv(j,1) = rhs(j)
      vv(j,im3)=rhs(j)
21     continue

c-----------------------------------------------------------------
20     rao = ddot(n, vv, 1,vv,1)
      call MPI_BARRIER(FAMILY,ierr)
      ro=zero
      call MPI_allreduce(rao,ro,1,MPI_double_precision,MPI_sum,FAMILY,ierr)

      ro=dsqrt(ro)
      if(myproc.eq.1)then
      if (iout .gt. 0 .and. its .eq. 0)then
      write(iout, 199) its, ro
      call flush(iout)
      endif
      endif
c
199    format(' pgmres  its =', i4, ' res. norm =', d20.6)
198    format(' pgmres  ipas =', i4, ' eps_pert =', d20.6)

 if (ro .eq. 1.0d-16) goto 999
      t = 1.0d0/ ro
      do 210 j=1, n
      vv(j,1) = vv(j,1)*t
 210  continue

      if (its .eq. 0) eps1=eps*ro
```

```
c     ** initialize 1-st term  of rhs of hessenberg system..
      rs(1) = ro
      i = 0
      its = its + 1
 4    i=i+1
      i1 = i + 1
c
c    call preconditionner
c--- le resultat est dans tmp
      call lusol0 (n, vv(1,i), tmp, alu, jlu, ju)

c------> on fait  le moyennage  des coefficients partages
      call consis_av(n,nbnd,tmp,tmpd,nproc,proc,ix,iprr,itype,neq,iptrn,riord,jb,icount)

      do kkj=1,n
      rhs(kkj)=tmp(kkj)
      enddo

      wnorm= dnrm2(n,rhs,1)
      wnormt=zero
      call MPI_allreduce(wnorm,wnormt,1,MPI_double_precision,MPI_sum,FAMILY,ierr)
      eps_pert= eps2/wnormt

c compute the residual for the perturbed solution
      do k=1,n
      vv(k,im2)=sol(k)+eps_pert*rhs(k)
      enddo
c
      call flux(vv(1,im2), vv(1,i1),0)

c-- le vecteur residu de la solution perturbee est    vv(*,i1)
c
      itype=12
      iptrn=12
c
c----calcul du produit matrice vecteur
      do kkj=1,n
      vv(kkj,i1)=-(vv(kkj,i1)-vv(kkj,im3))/eps_pert
```

```
      enddo
c--------------------------------------
c     modified  gram - schmidt...
c--------------------------------------

      do 55 j=1, i

      tloc = ddot(n, vv(1,j),1,vv(1,i1),1)
      call MPI_BARRIER(FAMILY,ierr)
      t=zero
      call MPI_allreduce(tloc,t,1,MPI_double_precision,
     *                   MPI_sum,FAMILY,ierr)
c
      hh(j,i) = t
      call daxpy(n, -t, vv(1,j), 1, vv(1,i1), 1)
 55   continue
c
      dnorm=ddot(n,vv(1,i1),1,vv(1,i1),1)
      call MPI_BARRIER(FAMILY,ierr)
      t=zero
      call MPI_allreduce(dnorm,t,1,MPI_double_precision,MPI_sum,FAMILY,ierr)
c
      t=dsqrt(t)
      hh(i1,i) = t
      if ( t .eq. 0.0d0) goto 58
      t = 1.0d0/t
      do 57  k=1,n
      vv(k,i1) = vv(k,i1)*t
 57   continue

c
c     done with modified gram schimd and arnoldi step..
c     now  update factorization of hh
c
 58   if (i .eq. 1) goto 121
c--------perfrom previous transformations  on i-th column of h
      do 66 k=2,i
        k1 = k-1
        t = hh(k1,i)
```

```
      hh(k1,i) = c(k1)*t + s(k1)*hh(k,i)
      hh(k,i) = -s(k1)*t + c(k1)*hh(k,i)
 66   continue
 121  gam = sqrt(hh(i,i)**2 + hh(i1,i)**2)
c
c    if gamma is zero then any small value will do...
c    will affect only residual estimate
c
     if (gam .eq. 0.0d0) gam = epsmac
c
c    get  next plane rotation
c
     c(i) = hh(i,i)/gam
     s(i) = hh(i1,i)/gam
     rs(i1) = -s(i)*rs(i)
     rs(i) =  c(i)*rs(i)
c
c    detrermine residual norm and test for convergence-
c
     hh(i,i) = c(i)*hh(i,i) + s(i)*hh(i1,i)
     ro = abs(rs(i1))

     write(iout, 199)i, ro
     call flush(iout)

     if ((i .lt. im) .and. (ro .gt. eps1) ) goto 4

c
c    now compute solution. first solve upper triangular system.
c
     rs(i) = rs(i)/hh(i,i)
     do 30 ii=2,i
       k=i-ii+1
       k1 = k+1
       t=rs(k)
       do 40 j=k1,i
         t = t-hh(k,j)*rs(j)
 40    continue
       rs(k) = t/hh(k,k)
```

```
 30   continue
c
c    form linear combination of v(*,i)'s to get solution
c
      t = rs(1)
      do 15 k=1, n
      rhs(k) = vv(k,1)*t
 15   continue
      do 16 j=2, i
        t = rs(j)
        do 161 k=1, n
        rhs(k) = rhs(k)+t*vv(k,j)
 161     continue
 16   continue
c
c    call preconditioner.
c



c---  appliquer le preconditionneur a rhs qui est ordonne
c

c--- resultat du precond dans tmp
      call lusol0 (n, rhs, tmp, alu, jlu, ju)

c--- moyenner les coefficients partages
      call consis_av(n,nbnd,tmp,tmpd,nproc,proc,ix,iprr,itype,neq,iptrn,riord,jb,icount)

c--- copier tmp dans rhs
      do kkj=1,n
      rhs(kkj)=tmp(kkj)
      enddo
c


      dnorm=ddot(n,rhs,1,rhs,1)
      call MPI_BARRIER(FAMILY,ierr)
      vnorm_rhs=zero
```

```
      call MPI_allreduce(dnorm,vnorm_rhs,1,MPI_double_precision,
     *                   MPI_sum,FAMILY,ierr)

      vnorm_rhs=dsqrt(vnorm_rhs)
      do 17 k=1, n
      sol(k) = sol(k) + rhs(k)
 17   continue
c------ increase iterartion count
      iter_count= iter_count+i
      if(vnorm_sol.ne.zero) then
      xnorm=vnorm_rhs/vnorm_sol
      else
      xnorm=1.0d0
      endif
c
c     restart outer loop  when necessary
c
      if (ro .le. eps1) goto 990
      if (its .ge. maxits) goto 991
c
      call flux(sol,vv(1,im3),0)
c---- on a un nouveau residu dans vv(*,im3) qui est  ordonne.
      itype=12
      iptrn=12
c-- replacer ce nouveau residu dans vv(*,im3) et vv(*,1) --> ordonne
       do j=1,n
       vv(j,1)  =  vv(j,im3)
       enddo
      goto 20
990   ier = 0
      return
 991   ier = 1
      return
 999   continue
      ier = -1
      return
c-----------------end of pgmres ---------------------------------------
c---------------------------------------------------------------------
      end
```

ANNEXES 6

Unités de lecture/écriture

Liste des unités réservée :

Lecture		Écriture	
Unité	Nom	Unité	Nom
53	process.ini	57	*.log
45	FMETIS	33	*.out
11	CORD	19	PFES Solution
10	COND	72	Tecplot Solution
14	LIMD	71	pas.dat
4	INPD	18	Z.txt
16	INID		
30	NORD		
99	RESD		
21	VITD		
13	NEQD		
15	NELD		
46	PARAM		
55	NFLUI		
58	EFLUI		
56	NSTRU		
67	ESTRU		
61	FREQD		
62	MODES		
73	FSKIN		

BIBLIOGRAPHIE

[1] A. Quarteroni and A. valli (1999). *Domain Decomposition Methods for Partial Differential Equations*. Oxford Science Publications.

[2] A.Soulaïmani, N.Ben Salah and Y.Saad. *ENHANCED GMRES ACCELERATION TECHNIQUES FOR SOME CFD PROBLEMS.* UMSI 2000/165 September 2000.

[3] A. Soulaïmani, Y. Saad and A. Rebaine. *Parallelization of the edge based stabilized finit element method using PSPARSLIB, in parallel computational fluid dynamics, towards teraflops, optimization and Novel Formulations.* D.Keyes, A. Ecer, N Satofuka, P. Fox and J. Periaux editors, pp.397-406, North-Holland, 2000.

[4] A. Soulaïmani, T.wong, Y. Azami, A. BenElHajAli. *An Object-Oriented Approach for PC Clusters.*

[5] A. Soulaïmani, Y. Saad & A. Rebain (1999). *An edge based stabilized finite element method for solving compressible flows: Formulation and parallel implementation.* Canadian Aeronautics and Space Institute, 46th Annual Conference, pp. 123-132, Montréal.

[6] A. Soulaïmani, A. BenElHajAli and Z. Feng (2002). *Nonlinear Computational Aeroelasticity : Formulations and Solution Algorithms.*

[7] A. Soulaïmani, A. Forest, Z. Feng, A. BenElhaj & Y. Azami (2001). *A distributed computing-based methodology for computational nonlinear aeroelasticity.* CASI 48th annual Conference, Toronto, Canada, pp. 123-135.

[8] A. Soulaïmani and Y. Saad (1998). *An arbitrary lagrangian Eulerian finite element formulation for solving three-dimensional free surface flows.*

[9] B.H.V Topping And A. I. Khan (1996). *Parallel Finite Element Computations.* Saxe-Coburg Publications.

[10] B. Lucquin, O. Pironneau (1996). *Introduction au calcul Scientifique.* Masson

[11] C. Farhat. *High performance simulation of coupled nonlinear transient aeroelastic problems.*

[12] C. Farhat and M. Lesoinne (1996). *On the accuracy, and performance of the solution of three-dimensional nonlinear transient aeroelastic problems by partitioned procedures.* AIAA-96-1388, 1996

[13] C. Farhat and M. Lesoinne (2000). *Two efficient staggered algorithms for the serial and parallel solution of three-dimensional nonlinear transient aeroelastic problems.* Comput. Math. Appl. Mech. Engrg, 182, 499-515, 2000.

[14] C. Frahat, M. Lesoinne and P. LeTallec (1998). *Load and motion transfer algorithms for fluid/structure interaction problems with non-matching discrete interfaces: Momentum and energy conservation, optimal discretization and application to aeroelasticity.* Comput. Maths. Appl. Mech. Engrg, vol. 157, pp. 95-114, 1998.

[15] C. Leopold. (2001). *Parallel and distributed computing, A survey of models, paradigms, and approaches.*

[16] D. Keyes, A. Ecer, N. Satofuka, P. Fox, J. Periaux. *Parallel Computational Fluid Dynamics, Towards Teraflops, Optimization And Novel Formulations.* Proceedings of the parallel CFD'99 Conference.

[17] E. C. Yates. *AGARD Standard Aeroelastic Configuration for Dynamics Response, Candidat Configuration I.-Wing 445.6.* NASA TM 100492, 1987.

[18] G. Karypis, V. Kumar (1998). *A fast and high-quality multi-level scheme for partitioning irregular graphs*, SIAM J. Sci. Comput. 20 (1998) 359-392.

[19] I. Ryhming (1991). *Dynamique des fluides.* ISBN 2-88074-224-2

[20] I. Foster (1995). *Designing and Building Parallel Programs, Concepts and tools for parallel software engineering.*

[21] J. E. Akin. *Object Oriented Programming via Fortran 90.* Engineering Computations, v. 16, n. 1, pp. 26-48, 1999

[22] M. Snir, S. Otto, S. Huss-lederman. D. Walker, J. Dongarra. MPI : *The Complete Reference.* The MPI Press, Cambridge, Massachusetts, London, England.

[23] S. Kusnetov, G. C. Lo, Y. Saad. *Parallel solution of general sparse linear systems using PSPARSLIB*, in choi-Hong Lai et al., editor, domain decomposition XI, pages 455-465, Domain decomposition Press, Bergen, Norway, 1999.

[24] T. Sato, S. Obayashi and K. Nakahashi. *Aerodynamic and Aeroelastic Simulations of unsteady flows over wings.*

[25] V. K. Decyk, C.D. Norton and B. K. Szymanski. *Object-Oriented Programming with Fortran90.*

[26] V. K. Decyk, C.D. Norton and B. K. Szymanski. *Introduction to Object-oriented Concepts using Fortran90.*

[27] Y. Saad & M. H. Schultz (1996). *GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems.* SIAM Journal on scientific and statistical computing, 7, pp. 856-869.

[28] Y. Saad, A. Malevsky. *PSPARSLIB : A portable library of distributed memory sparse iterative solvers.* In V. E. Malyshkin et al., editor, proceedings of Parallel Computing Technologies (PaCT-95), 3-rd International Conference, St. Petersburg, Russia, Spt. 1995.

[29] Y. Saad (1996). *Iterative methods for sparse linear systems.* PWS.

[30] Y. Saad. *SPARSKIT: A basic tool-kit for sparse matrix computations.* http://www.cs.umn.edu/research/arpa/SPARSKIT/sparskit.html

[31] Y. Saad. *A Portable Library of Parallel Sparse Iterative Solvers.* http://www.cs.umn.edu/Research/arpa/p_sparslib/psp-abs.html

[32] Y. Azami. *Realisation d'un environnement de traitement parallèle à partir d'un regroupement d'ordinateurs personnels.* Projet de maîtrise MGL, ÉTS, 2001

[33] W. Gropp, E. Lusk, A. Skjellum. *Using MPI, Portable Parallel Programming with the Message-Passing Interface.* Scientific and Engineering Computation Series 1996