# PALM_PARASOL
# version 2.0

# Manuel utilisateur

---


Thierry Morel, Florent Duchaine

CERFACS / Global Change and Climate Modelling Team



Décembre 2016

TR-CMGC-16-307


---

# SOMMAIRE

SOMMAIRE ..... 2
1 Introduction ..... 3
2 Installation ..... 3
3 Principe de fonctionnement ..... 3
4 Utilisation ..... 5
5 Description des données de parasol ..... 9
6 Exemple d'utilisation : quantification de l’incertitude du nombre de Reynolds sur la simulation d’une cavité entrainée. ..... 11

# 1 Introduction

PALM_PARASOL est un utilitaire permettant de lancer automatiquement en parallèle un nombre quelconque d'instances d'un même code de calcul. Son but n'est pas de paralléliser automatiquement un code de calcul via une décomposition de domaine, son intérêt est plutôt d'exécuter n instances d'un même code avec des entrées (et par voie de conséquence des sorties) différentes. Il a été développé initialement pour répondre au besoin spécifique d'une application de couplage faisant intervenir le modèle agronomique STICS [1] (INRA Avignon) dans la plateforme intégrée EauDyssée, développée au centre de Géosciences de Mines ParisTech. Dans ce projet, l'idée était de "spatialiser" le code STICS en le lançant sur un ensemble de parcelles constituant un bassin versant.

PALM_PARASOL est fortement contraint par l'utilisation du coupleur OpenPALM [2] (CERFACS/ONERA) car il s'appuie sur lui pour créer automatiquement du code source constituant une unité PALM. Bien que développé pour paralléliser les sols (d'où son nom) PALM_PARASOL est applicable à toute application nécessitant de lancer en parallèle un nombre conséquent d'instances du même code, on peut donc y trouver un intérêt sur les problèmes basés sur des plans d'expériences ou certaines méthodes d'assimilation de données de type filtre de Kalman d'ensemble.

# 2 Installation

PALM_PARASOL étant écrit en langage interprété tcl [3], il ne nécessite aucune installation particulière si ce n'est la présence du langage tcl sur votre station de travail, la commande tclsh ou wish (interpreteur tcl ou tcl/tk) doit donc être accessible à l'utilisateur, ce qui est généralement le cas sur n'importe quel système Linux ou Mac. Le résultat de PALM_PARASOL est l'écriture automatique d'une unité OpenPALM, elle même basée sur l'utilisation de la librairie de message passing MPI, et de sous programmes écrits en Fortran90 et C. Il est donc indispensable d'installer le logiciel OpenPALM pour utiliser PALM_PARASOL. Pour l'installation d'OpenPALM on doit se référer à la documentation accessible en ligne sur le site Web du CERFACS ou dans la distribution d'OpenPALM. PALM_PARASOL est compatible avec la version OpenPALM 4.0.0 et les suivantes. Notons que PALM_PARASOL tourne en pré processeur de l'application OpenPALM proprement dite qui va elle se charger de faire tourner les modèles. Le déploiement du ou des codes de calcul se fait donc sur la machine de calcul visée là où OpenPALM et les codes de calcul sont eux-mêmes installés. Enfin il est nécessaire d'installer OpenPALM en mode MPI_2, on doit donc s'appuyer sur une version de MPI qui implémente les mécanismes client/serveur et lancement dynamique de processus de la norme MPI-2, par exemple mpich [4] ou openmpi [5] sous Linux.

# 3 Principe de fonctionnement

Lancer en parallèle plusieurs instances d'un programme n'est pas difficile en soi. Même si le code n'a jamais vu de près ou de loin le calcul parallèle, il suffit d'un travail minime pour l'adapter à la librairie MPI qui sait très bien faire cela. Le but de PARASOL consiste à faire travailler ces

différentes instances de code sur des données différentes et de collecter les résultats dans le bon ordre. Au passage on cherchera un cadre générique, c'est à dire facilement paramétrable quant au nombre de processeurs, pour équilibrer les calculs sur des machines différentes dont les ressources en processeurs sont très différentes. L'utilisation d'OpenPALM offre cette souplesse.

Dans un premier temps, notons **n** le nombre d'instances de code à faire tourner, on appellera n par la suite "taille du problème". On se propose de traiter ce problème en s'aidant grandement des paradigmes mis à disposition par le coupleur dynamique de codes parallèles OpenPALM. L'idée est d'externaliser "l'alimentation" des codes en données via des communications par mémoire "à la PALM" et de collecter les résultats par le même mécanisme. Pour l'application, données (IN) et résultats (OUT) seront stockés quelque part en mémoire dans des tableaux multidimensionnels dont l'une des dimensions est la taille du problème. Remarquons que le code de calcul à "PARASOLiser" ne voit jamais la taille n du problème, c'est un paramètre de l'application couplée et non du code, ce qui permet d'être un minimum intrusif dans ce code. Pour fixer les idées prenons un exemple simpliste où le code résout l'équation du 1er degré ax+b=0. Ce qu'on cherche, c'est, sans modifier ce code, résoudre en parallèle n équations du premier degré, sachant qu'on aura au préalable défini, dans d'autres unités PALM, deux tableaux d'entrée de taille n pour a et b et qu'on collectera les résultats (-b/a) dans un troisième tableau lui aussi de taille n. Pour plus de souplesse, et parce qu'on ne dispose pas toujours de n processeurs mais plutôt de **m** (m<n) on traitera non pas n opérations en parallèle mais seulement m, quitte à boucler n/m fois pour lancer tous les calculs.

PALM_PARASOL est basée sur un mode maitre/ouvrier disponible dans la norme MPI_2. Un programme "master" va recevoir sous forme de PALM_Get les tableaux de taille n, puis dans une boucle de taille n/m il va lancer dynamiquement (par la commande MPI_Comm_spawn) m instances du code auxquelles le master va envoyer les données dispatchées, le programme Master se charge également de collecter les résultats de tous les modèles lancés pour finalement les regrouper dans un tableau de taille n. Au total, on lancera donc n instances du code. Bien qu'il soit tout à fait possible d'optimiser les lancements en faisant une boucle interne dans le code, un peu comme on peut le faire dans PALM en ajoutant un block autour d'une boucle, cette solution n'a pas été retenue car elle suppose que le code soit adapté à ce mode de fonctionnement, il faudrait en effet être certain que tous les fichiers ouverts par le code soient fermés et que toutes les variables allouées dynamiquement soient désallouées, c'est en général loin d'être le cas pour la majorité des codes de calcul.

La génération du code Master, de la mécanique MPI pour envoyer/recevoir les tableaux, et l'encapsulation du code à lancer est faite automatiquement et une fois pour toute (si l'interface du code ne change pas) par PALM_PARASOL. La taille n du problème et le nombre de processeurs disponibles deviennent paramétrables dans l'interface graphique PrePALM car ces données apparaissent sous forme d'entrées de l'unité OpenPALM générée. L'utilisateur n'a besoin de remplir qu'un descriptif des entrées et sorties du codes, de remplacer le programme principal par un subroutine Fortran ou une fonction C. L’alimentation en donnée et la récupération des résultats se fait soit par des arguments donnés à cette subroutine (ou fonction C) soit par l’appel de primitives Get/put générées par PARASOL.

# 4 Utilisation

Prenons un exemple simple pour comprendre comment utiliser PARASOL. Cet exemple est fourni avec PARASOL dans le sous répertoire test/equation. Repartons du code résolvant l'équation du 1er degré, en Fortran90 il pourrait se présenter comme ceci :

```
program axpb
implicit none
double precision :: a, b, x
open(18, file='axpb.in')
read(18,*) a, b
close(18)
x = -b/a
print *, 'résultat : ',x
end program axpb
```

Ce programme lit ses données dans le fichier axpb.in et affiche le résultat sur la sortie standard. Dans un contexte de couplage, ou simplement d'automatisation pour lancer n fois ce programme, entrées et sorties vont maintenant être envoyées et reçues quelque part par des PALM_Put et des PALM_Get. Pour le code axpb on ne se préoccupe pas pour l'instant de qui va produire les entrées et recevoir les résultats, ceci sera l'affaire d'une ou d'autres unités PALM.

La seule adaptation de ce code pour PALM_PARASOL consiste à remplacer l'instruction program par subroutine et à faire remonter entrées et sorties :

- soit au niveau de la liste d'appel de ce subroutine (méthode 1),
- soit par l'appel à des primitives get/put générées par PALM_PARASOL (méthode 2).

L'adaptation du code axpb à PARASOL se présente comme ceci :

Méthode 1, par arguments :

```
subroutine axpb(a,b,x)
implicit none
double precision :: a, b, x
x = -b/a
end subroutine axpb
```

Méthode 2, par accesseurs :

```
subroutine axpb()
implicit none
integer :: il_err
double precision :: a, b, x
call PARASOL_Get_a(a, il_err)
call PARASOL_Get_b(b, il_err)
x = -b/a
call PARASOL_Put_x(x, il_err)
end subroutine axpb
```

Attention : dans la version 1.0 de PALM_PARASOL, le subroutine devait avoir comme premier argument un entier correspondant au rang que donnera PARASOL à l'exécution de chaque instance du code, indice variant de 1 à n. Cet argument ne doit plus être donné dans la liste d'appel avec la version 2.0, pour avoir accès à l'instance, il faut maintenant appeler la fonction PARASOL_Get_index(idx, id_err).

Notons qu'on a éliminé la partie qui lit les données d'entrée dans le fichier puisqu'elles seront maintenant fournies par des communications PALM reçues au niveau d'un nouveau programme principal qui lui sera généré par PARASOL et PrePALM. L'affichage du résultat d'une seule équation sur le stdout n'a pas non plus d'intérêt en soit, ce qui nous intéressera par la suite sera le vecteur résultat des n instances du code, la partie affichage a donc été commentée.

Maintenant que les entrées sorties du code sont définies, il convient de faire tourner le script parasol.tcl en lui décrivant ces entrées sorties ainsi que le nom du sous-programme à appeler. La description des variable demande de les typer, et éventuellement de donner la taille des tableaux pour pouvoir établir les communications MPI entre le programme master et les différentes instances du code. Ceci se fait dans la section réservée à l'utilisateur de parasol.tcl sous forme d'une liste tcl, ou directement dans un fichier d'entrée pour PARASOL :

Fichier parasol.in :

Méthode 1, par arguments :

```
set subroutine axpb
set varsub  {{in a {{double precision} MPI_DOUBLE_PRECISION 1} n {a} distr}  {in b {{double precision} MPI_DOUBLE_PRECISION 1} n {b} distr}  {out x {{double precision} MPI_DOUBLE_PRECISION 1} n {x}}}
```

Méthode 2, par accesseurs :

```
set subroutine axpb
set varsub  {{get a {{double precision} MPI_DOUBLE_PRECISION 1} n {a} distr}  {get b {{double precision} MPI_DOUBLE_PRECISION 1} n {b} distr}  {put x {{double precision} MPI_DOUBLE_PRECISION 1} n {x}}}
```

Une fois ce script lancé par la commande :

```
>tclsh parasol.tcl parasol.in
```

Deux fichiers sources Fortran90 et un fichier « makefile » sont générés, il est recommandé de ne pas modifier ces fichiers.

**master**_axpb.f90 : Unité PALM à lancer dans PrePALM, le rôle de cette unité est de collecter/dispatcher les données, recevoir n et m, et de lancer dynamiquement par groupe de m le code slaves.

**slaves**_axpb.f90 : Programme principal d'encapsulation du code (dont ont a fait une subroutine) sous PARASOL.

**makefile_slaves_**axpb : Fichier pour compiler le programme slaves_axpb.f90

L'unité PALM master_axpb contient à la fois le code et la carte d'identité pour PrePALM. C'est cette unité qu'on doit charger dans l'interface graphique PrePALM. La taille du problème n et le nombre de processeurs m sont demandés sous forme de PALM_Get dans cette unité. Un troisième plot permet de préciser s'il on veut ou non faire tourner le code dans l'exécutable du programme master_axpb, en effet, entre le moment où elle a terminée le lancement dynamique des esclaves et le moment où elle doit récupérer les résultats, cette unité PALM monopolise un processus MPI juste pour effectuer des attentes, une optimisation a donc consisté à faire tourner également le subroutine dans cette unité PALM. Attention cependant si ce mode est choisi, car l'exécutable master reste en mémoire du début à la fin du processus de calcul des n instances, si le subroutine est appelé plusieurs fois par le master (dans le cas où n>m), on se retrouve dans le même cas de figure qu'une unité PALM lancée en boucle contenue dans un bloc PALM, avec des effets désastreux si le code n'est pas prévu pour être appelé plusieurs fois par le même exécutable. Il est donc prudent de mettre au point l'application sans faire tourner le subroutine dans l'unité master (choisir l'option 0 : only on slaves). Une quatrième entrée permet de préciser le niveau de verbosité des messages écrits dans le fichier PL_OUT.

Pour alimenter les différentes instances du code, il faut créer des vecteurs contenant les différentes valeurs de a et b, et il faut prévoir également un vecteur pour récupérer les résultats x. Pour notre application les vecteur a, b et x sont tous envoyés/reçus dans la même unité qui s'appelle "donnees". Notons que dans l'unité master, les objets sont déclarés avec un espace NULL pour pouvoir hériter des caractéristiques des objets envoyés. La mise en place du schémas PrePALM et des communication est assez triviale pour cette application, elle consiste à lancer l'unité "donnees" et l'unité "master_axpb" en parallèle et à décrire les communications. Le schéma PrePALM est présenté ci après. Pour compiler l'application, il est nécessaire de compiler l'application PALM avec le fichier Makefile écrit par PrePALM ainsi que les esclaves en lançant les commandes :

```
> make
> make -f makefile_slaves_axpb
```

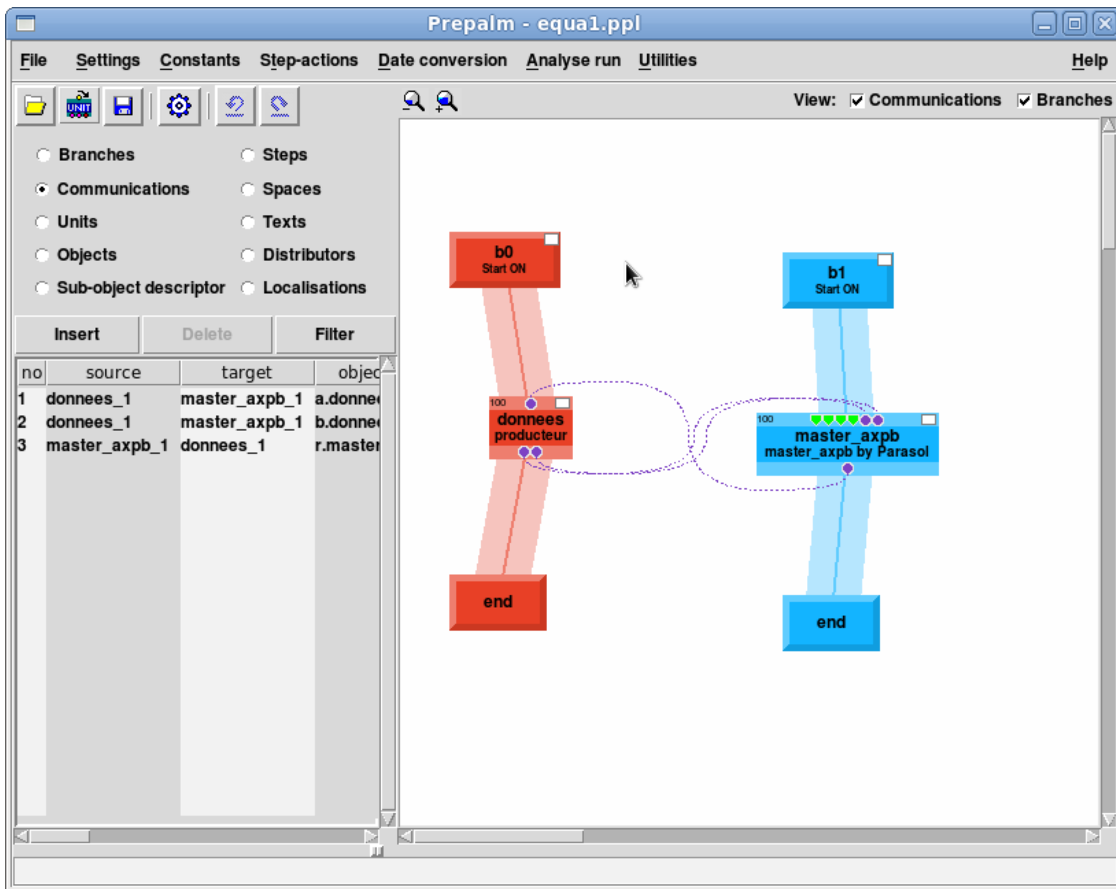


**Figure 1 : Canevas PrePALM de l'application axpb.**

Dans le nombre de processus à donner pour PALM (Menu Settings->Palm execution settings->Number of proc) il convient de compter les processus nécessaires pour faire tourner les esclaves. Avant de lancer un groupe de m esclaves, l'unité master demande à PALM le nombre de processus disponibles (qui peut éventuellement être inferieur à m), et il lance les esclaves sur ce nombre de processus disponibles.

Dans notre application le résultat est écrit dans le ficher PL_OUT d'OpenPALM, fichier b0_000.log.

Dans la version 1.0 de PALM_PARASOL il n'était possible de lancer que des codes écrits en langage Fortran, depuis la version 2.0 il est possible de lancer des codes en langages C/C++ et Python

Un exemple avec un code C est donné dans le répertoire equation_getput_C.

Un exemple avec un code Python est donné dans le répertoire sum_python.

# 5 Description des données de parasol

La plupart des données d'entrées du script parasol.tcl sont décrites sous forme de listes, pour écrire ces listes il faut utiliser des accolades et des espaces comme séparateur, par exemple :

{element1 {element2 est une sous liste} element3} : liste de trois éléments dont l'un est lui même une liste

{} : liste vide

Les données sont décrites avec le mot clé "set" suivi d'un nom de variable et de sa valeur.
Le fichier d'entrée peut contenir des commentaires en ajoutant un caractère # en début de ligne.

1) `set subroutine sub_name` : nom du point d'entrée du programme, subroutine fortran ou fonction C à appeler pour dérouler le programme principal du code de calcul. Comme pour les unités PALM, lorsque le code de calcul est un programme principal, il convient de le remplacer par un subroutine fortran.

2) `set language lang` : optionnel, défaut fortran, type de langage (fortran, c ou python).

3) `set module liste_modules` : optionnel, défaut {}, liste contenant les modules fortran90 qui peuvent être nécessaires aux sous programmes crées par parasol, ces noms de modules se retrouveront sous forme de "use" dans l'unité PALM master_sub_name et dans le programme principal slaves_sub_name appelant le code.

4) `set object_files liste_objets` : optionnel, défaut {}, liste contenant les noms des fichiers objets (.o .a) nécessaires à l'édition de lien des programmes master et slaves.

5) `set extraflags` : optionnel, défaut {}, liste contenant des options de compilation additionnelles nécessaires au code.

6) `set dependencies liste_dependences` : optionnel, défaut {}, description des dépendances éventuelles pour la création du fichier makefile_slaves_sub_name.

7) `set dependencies_master liste_dependences`: optionnel, défaut {} description des dépendances éventuelles pour la création de la liste object_files de la carte d'identité de l'unité master_sub_name.

8) `set makefile_extralines` : optionnel, défaut {}, lignes additionnelle à ajouter au fichier make_slaves_sub_name

9) `set parall_mode` : optionnel, défaut "no", indique si le code de calcul est lui même un code parallèle MPI.

10) `set varsub liste_variables` : liste ordonnée, contenant la description des variables du subroutine utilisateur, pour chaque variable il faut décrire une liste comprenant :

10.1) `intention` : in, out, inout, get ou put

in : variable dans la liste d’appel en entrée du subroutine

out : variable dans le liste d'appel en sortie du subroutine
inout : entrée et sortie
get : variable en entrée à récupérer par une primitive PARASOL_Get_nom
put : variable en sortie, à mettre à jour par une primitive PARASOL_Put_nom

Les variables en entrée (in et get) correspondront à des PALM_Get dans l'unité master_sub_name, en sortie (out et put) à des PALM_Put des résultats.

10.2) `nom` : nom donné à la variable dans le subroutine

10.3) `{type_FORTRAN type_message_MPI taille}` : liste descriptive du type informatique à manipuler avec :

`type_FORTRAN` : integer, real ou {double precision}

`type_message_MPI` : MPI_INTEGER, MPI_REAL ou MPI_DOUBLE_PRECISION

`taille` : 1 pour les types simples ou taille du message MPI pour de type dérivés, dans ce cas il faut utiliser le type MPI_INTEGER et donner la taille du type dérivé en nombre d'entiers.

10.4) `shape` : description de la variable, scalaire ou tableau à plusieurs dimensions, sous forme d'une expression donnant la dimension du tableau et la position de l'élément sur lequel doit se faire la distribution ou la collecte des variables entre les vecteurs de taille n (dans le master) et les tableaux dont la taille est réduite sur la position où se trouve n. Pour une variable scalaire (0d) la description est n, pour un tableau 3d on peut avoir :,:,n ou n,:,: ou :,n,:

10.5) `commentaire` : zone de texte pour le commentaire de la carte d'identité de l'unité master_sub_name.

10.6) `mode` : optionnel, défaut distr, indicateur de duplication (dupl) ou de distribution (distr) uniquement pour les variables in, permet de savoir si le tableau en entrée doit être distribué ou dupliqué. Le mode de fonctionnement par défaut est distr. Si le mode est dupl, l'objet demandé en entrée de l'unité master comporte une dimension de moins (sur la position n) et ce même objet est envoyé à toutes les instances du code, voir l'exemple dans PARASOL/test/equation_a_dupl/

**Remarques sur les variables get et put**

Pour chaque variable déclarée en get ou put dans la liste d'entrée varsub de PARASOL, des fonctions permettant de les manipuler dans le code de calcul sont créées, ces fonctions sont accessibles depuis les langages Fortran, C/C++ ou python. En Fortran les fonctions prennent un argument supplémentaire qui est le code d'erreur de retour, en C ce code d'erreur est retourné par la fonction, par exemple :

Forme d'appel Fortran : call PARASOL_Get_a(a, il_err)
Forme d'appel C : il_err = PARASOL_Get_a(a);
Forme d'appel Python : PARASOL.get_a(a)

Pour chaque variable « var », deux fonctions supplémentaires sont construites, la première PARASOL_Get_shape_var(ila_shape , il_err) permet de récupérer le shape du tableau, en effet cette taille peut être dynamique, la taille des tableaux multidimensionnels n'est connue qu'au moment où le master fait son PALM_Get. Par exemple pour un tableau 3d déclaré sous la forme :, :,n dans les entrées de PARASOL, le tableau ila_shape sera de dimension 2 et retournera ce qui a été définit aux positions des « : ».

La seconde fonction PARASOL_Get_var(var, il_err) ou PARASOL_Put_var(var, il_err) permettent d'accéder à la valeur (Get) ou de mettre à jour la variable (Put).

Ces fonctions sont accessibles dans n'importe quel ordre et depuis n'importe quel endroit du code de calcul. Les fonctions peuvent être appelées plusieurs fois si nécessaire, si le code est écrit par un mixte de langages (par exemple du Fortran appelant du C) il est possible d'appeler la fonction depuis les parties écrites Fortran comme depuis les parties écrites en C.

Dans tous les cas une fonction PARASOL_Get_index(index, il_err) permet retourner le numéro du run, donc un entier variant de 1 à n.

**Remarques pour le langage Python**

Si le code est écrit en langage Python un exemple est donné dans le répertoire test/sum_python. Comme pour OpenPALM, les variables à manipuler par le primitives PARASOL doivent être des tableaux numpy. Une fonction d'initialisation PARASOL.init() doit être appelée au début du code et une fonction de finalisation PARASOL.finalize() doit être appelée à fin du code python.

# 6 Exemple d'utilisation : quantification de l'incertitude du nombre de Reynolds sur la simulation d'une cavité entrainée.

La quantification des incertitudes d'un modèle numérique permet de caractériser de manière quantitative l'impact des paramètres non exactement connus sur la solution recherchée. L'exemple que l'on se propose d'étudier ici vient de la mécanique des fluides. Il s'agit d'un cas bien connu pour tester les codes incompressibles : la cavité entrainée. Le système est constitué d'une boite initialement au repos. L'écoulement est généré par la mise en mouvement d'une des parois à une vitesse donnée, toutes les autres restant fixes (Figure 2). Le cisaillement imposé au fluide par le couvercle d'entraînement produit un écoulement dont la dynamique est complexe et fortement non linéaire, avec formation et interaction d'instabilités. Dans le cadre d'un écoulement de fluide visqueux newtonien incompressible, les instabilités qui se développent sont directement liées au nombre de Reynolds :

$$Re = \frac{\rho U L}{\mu}$$

Avec $\rho$ la densité et $\mu$ la viscosité du fluide, L et U la longueur et la vitesse de la paroi. Lorsque la vitesse d'entraînement du fluide est faible (nombre de Reynolds petit), on observe un gros tourbillon central ainsi qu'une séquence infinie de tourbillons dans les coins de la cavité. À des vitesses plus élevées (donc à nombre de Reynolds plus élevé), le fluide entre dans un régime turbulent, dans lequel la structure de l'écoulement devient chaotique en espace et en temps. Dans le cadre de cet exemple, nous nous limiterons aux faibles nombres de Reynolds ce qui nous permet de

partir sur une discrétisation bidimensionnelle du problème et non sur un de couteuses simulations tridimensionnelles.

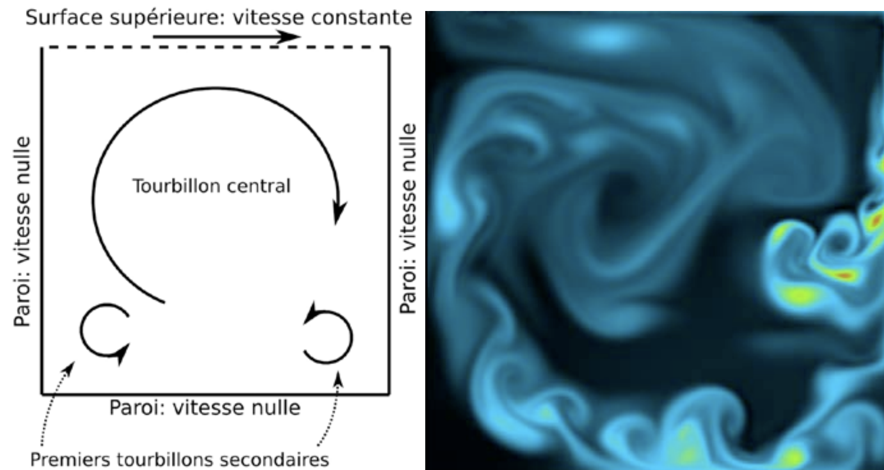


**Figure 2 : Schéma de principe de l'écoulement en cavité entrainée (à gauche) et résultat instantané obtenu avec le code SGE AVBP (à droite).**

Le code utilisé pour résoudre cet écoulement est le code 2d *nsproj* qui résout les équations de Navier Stokes incompressibles par une méthode de projection. Le code est adimensionné de telle sorte que les paramètres d'entée soient :

- le nombre de Reynolds,
- le nombre de point de discrétisation dans chaque direction,
- le temps total à simuler,
- les instants de sauvegarde des solutions,
- le type de schéma numérique spatial (centré ou décentré amont),
- le nombre d'itérations maximum du solveur de Poisson pour la pression.

Etant donnée la très forte dépendance de cet écoulement au nombre de Reynolds, il est légitime de quantifier l'impact d'une incertitude de cette quantité sur la solution. Pour cela on se propose de se limiter à un nombre de Reynolds de 400. On se donne alors une incertitude $\sigma$ sur ce nombre de Reynolds et le but de l'analyse est de quantifier l'impact de cette incertitude sur les champs de vitesse. Pour cela, nous proposons d'utiliser une méthode simple dite non intrusive de quantification d'incertitude. La méthodologie est la suivante :

- on considère que le nombre de Reynolds est une variable aléatoire qui suit une loi Gaussienne de moyenne *Re* et d'écart type $\sigma$,
- on génère ainsi une population de *nbp* individus de cette distribution. La Figure 3 présente un exemple de distribution représentée sur 20 intervalles. A noter que plus *nbp* est grand et plus on s'assure de la convergence statistique du processus, mais plus le coût de la quantification est élevé en temps de calcul,
- on exécute ensuite les *nbp* simulations correspondant aux *n* nombres de Reynolds ainsi générés,
- on post traite l'ensemble des *nbp* champs de vitesses obtenus afin de tirer des quantifications d'incertitudes.

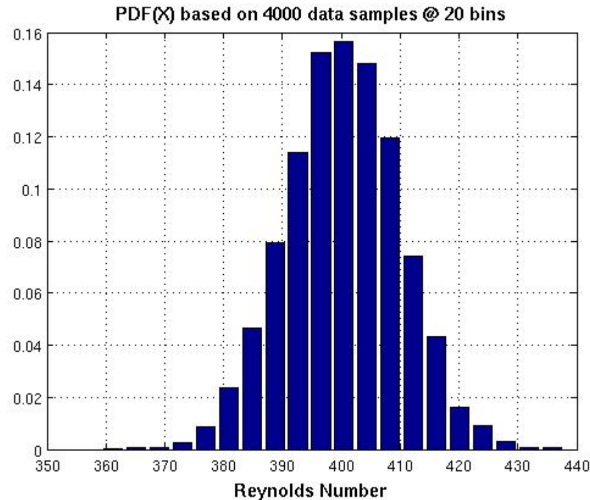


**Figure 3 Distribution gaussienne de nombre de Reynolds représentée sur 20 bins avec un écart type σ = 10 et une population de 4000 individus.**

L'unité *uq* (voir carte d'identité ci-dessous) permet de générer la distribution gaussienne, d'envoyer le tableau des nombres de Reynolds correspondant (tableau *controlp* dont la taille va être dynamiquement initialisée à *nbp*).

PARASOL va nous aider à lancer ce grand nombre de calculs en parallèle, le code n'est pas très gourmand en temps de calcul ni en mémoire, l'application tournera sur un PC linux à 16 processeurs. Il faut noter que faire tourner cette application sur un plus grand nombre de processeurs se fait en changeant uniquement un paramètre dans le schémas PrePALM, le travail préliminaire pour utiliser PARASOL est donc payant par rapport à la modularité qu'il apporte sur le déploiement de l'application. On commence donc à écrire la carte d'identité de l'unité qui nourrit le code en données :

```
!PALM_UNIT -name uq\
!          -functions {F90 uq}\
!          -object_files {uq.o}\
!          -comment {Pour faire du UQ}\
!
!PALM_SPACE -name tab1d\
!           -shape (:)\
!           -element_size PL_DOUBLE_PRECISION\
!           -comment {Tableau 1D double precision}\
!
!PALM_SPACE -name tab3d\
!           -shape (:,:,:)\
!           -element_size PL_DOUBLE_PRECISION\
!           -comment {Tableau 3D double precision}\
!
!PALM_OBJECT -name pbsize\
!            -space one_integer\
```

```
!            -intent OUT\
!            -comment {Taille du probleme}
!
!PALM_OBJECT -name controlp\
!            -space tab1d\
!            -intent OUT\
!            -comment {Vecteur de controle}
!
!PALM_OBJECT -name velu\
!            -space tab3d\
!            -intent IN\
!            -comment {Tableau de Velu}
!
!PALM_OBJECT -name velv\
!            -space tab3d\
!            -intent IN\
!            -comment {Tableau de Velv}
```

L'unité *uq* lit dans un fichier d'entrée la valeur moyenne et l'écart type du nombre de Reynolds, le nombre *nbp*, les nombres de point de discrétisation dans chaque direction pour *nsproj*, ainsi que deux paramètres pour la génération de la distribution gaussienne.

Pour s'adapter à PARASOL, le programme *nsproj* est transformé en subroutine (avec comme données d'entrées :

- le nombre de Reynolds,
- les nombres de discrétisation *nx* et *ny* en x et en y,

et comme données de sortie

- les tableaux contenant les champs de vitesses en x et y.

Notons que les nombres de discrétisations sont passées deux fois à la subroutine du fait de la construction générique de PARASOL : en effet *nx* et *ny* dimensionnent les vecteurs de vitesse *vx* et *vy*.

La section réservée à l'utilisateur de l'outil parasol.tcl est présentée ci dessous :

```
set subroutine nsproj
set varsub  {{in Re {{double precision} MPI_DOUBLE_PRECISION 1} n {input Reynolds Number} distr}  {out velu  {{double precision} MPI_DOUBLE_PRECISION 1} :,:,n {Output Velocity u}} {out velv {{double precision} MPI_DOUBLE_PRECISION 1} :,:,n {Output Velocity v}}}
```

Remarquons ici l'utilisation de tableaux 3D pour les vitesses velu et velv, dans le code nsproj ces tableaux sont 2d, mais dans uq on veut récupérer toutes les solutions des simulations, d'où l'ajout de la troisième dimension. Notons que la position de n dans l'expression `:,:,n` correspond à la manière dont le tableau a été déclaré dans l'unité uq.

L'exécution du script PARASOL (`>tclsh parasol.tcl`) permet de générer les fichiers :

- master_nsproj.f90 : unité PALM à charger dans PrePALM,
- slaves_nsproj.f90 : programme principal qui lancera la subroutine *nsproj*,
- makefile_slave_nsproj : fichier pour compiler le programme slave_nsprog.f90.

La Figure 4 présente le canevas PrePALM correspondant à l'application UQ. Les communications à établir sont :

- l'envoi de la taille de la population à traiter *pbsize* de l'unité *uq* à l'unité *master_nsproj*,
- l'envoi du tableau des nombres de Reynolds à traiter *controlp* de l'unité *uq* à l'unité *master_nsproj*,
- l'envoi des résultats de champs de vitesse *velu* et *velv* de l'unité *master_nsproj* à l'unité *uq*.

Le nombre de processeurs *nb_slaves* dédiés au calculs du programme slaves_nsproj peut être directement initialisé via une constante PrePALM.

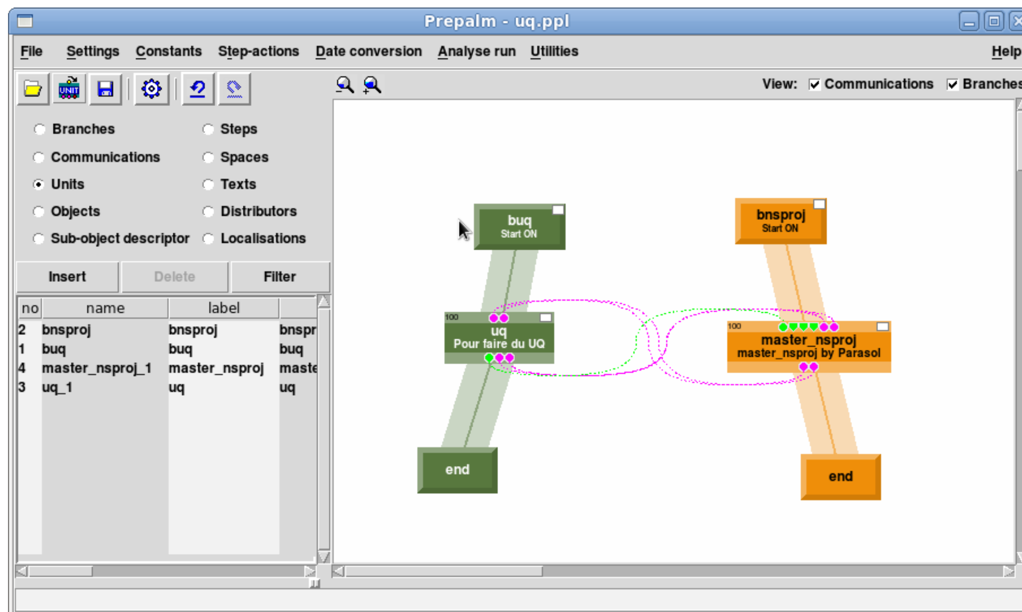


**Figure 4 Canevas PrePALM de l'application UQ.**

A la fin du calcul, l'unité *uq* récupère ainsi tous les champs de vitesses correspondant aux *nbp* nombres de Reynolds de la population gaussienne. Elle produit ensuite une analyse statistique basique. La Figure 5 présente la moyenne et l'écart type de la composante en X de la vitesse correspondant à cette population illustrant ainsi les zones de forte sensibilité du modèle au nombre de Reynolds. Les zones de fort écart type sont donc les plus sensibles à des incertitudes du nombre de Reynolds. Il s'agit des zones de l'écoulement présentant de forts cisaillements. Notons que les niveaux d'incertitude sont dans le cas présent très faibles devant les valeurs moyennes. Ce qu'il est ensuite intéressant de générer avec ce type de méthodologie est par exemple des barres d'erreurs sur les profils de vitesse. La Figure 6 montre qu'une imprécision correspondant à un écart type de $\sigma = 10$ sur un nombre de Reynolds de 400 mène à de très faibles incertitudes sur le profil de vitesse axiale pris au centre de la cavité.

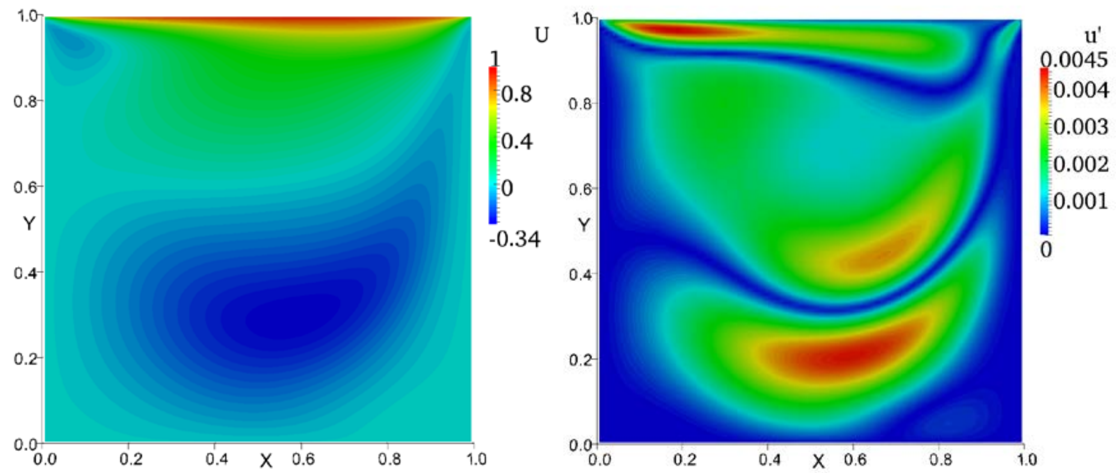


**Figure 5 Résultats produits par l'unité uq : moyenne de la vitesse en X (gauche) et écart type de la vitesse en X (droite). La grille utilisée comporte 101 points dans chaque direction.**

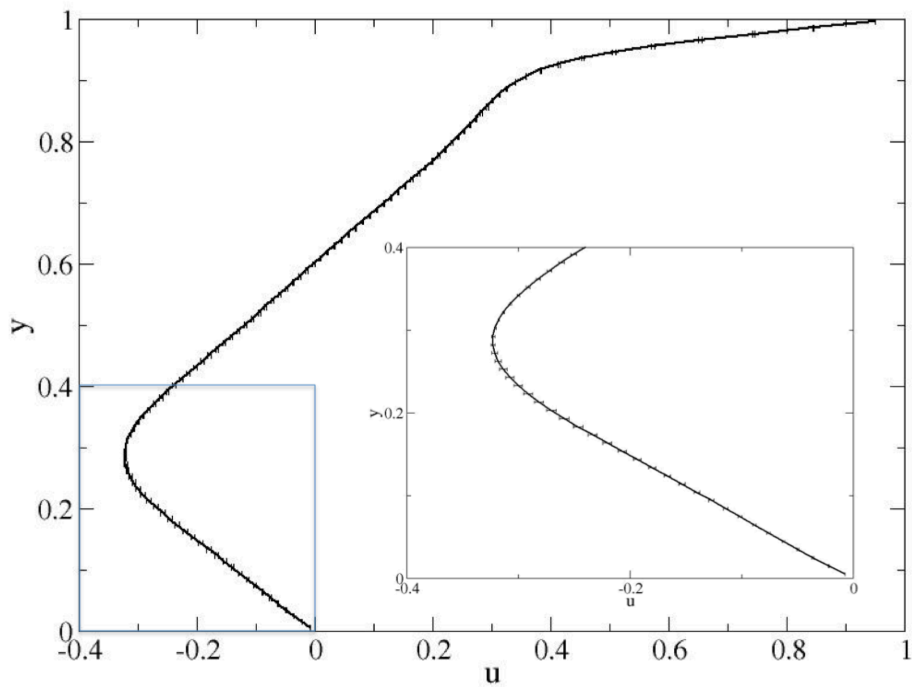


**Figure 6. Profil transverse au centre de la cavité de vitesse en X avec les barres d'erreurs provenant de l'analyse UQ.**

## Références


[1] STICS : www.avignon.inra.fr/agroclim_stics/modele_stics
[2] OpenPALM : www.cerfacs.fr/globc/PALM_WEB/
[3] tcl/tk : www.tcl.tk/software/tcltk/
[4] mpich2 : www.mcs.anl.gov/research/projects/mpich2/
[5] openmpi : www.open-mpi.org/
[6] lammpi : www.lam-mpi.org/