Introduction au solveur PaStiX

Introduction

Le solveur PaStiX est disponible sur `http://pastix.gforge.inria.fr`. Vous pouvez également accéder aux forums de discussions (`http://gforge.inria.fr/forum/?group_id=186`) et à la mailing liste (`pastix-users@lists.gforge.inria.fr`).
PaStiX a besoin de différentes bibliothèques pour pouvoir être installé :
– une bibliothèque de communication MPI (Mpich2 , OpenMPI , ...)
– une bibliothèque de BLAS (GotoBLAS , ACML , MKL , ...)
– la bibliothèque Scotch pour la renumérotation des inconnues (la bibliothèque METIS est facultative).

Compilation et installation

La compilation et l'installation du solveur PaStiX se font en 5 étapes :

1. Paramétrer le fichier `config.in` en fonction de la machine et des bibliothèques (communications et BLAS) disponibles. Des fichiers d'exemple de configuration pour différentes architectures sont disponibles dans le répertoire `src/config` de l'archive.
2. Choisir les options de compilation spécifique à l'utilisateur : type d'entiers, type arithmétique (réel ou complexe), support ou non des threads, version parallèle du solveur, ...
3. Compiler : `make`
4. Installer : `make install`
5. Compiler les programmes de test : `make examples` (exemples en C et en Fortran, avec ou sans utilisation de l'interface Murge)

Pour faciliter les tests dans le cadre de cette formation, les bibliothèques OpenMPI **(avec le support complet des threads MPI_THREAD_MULTIPLE),** MKL **,** Scotch **et** METIS **ont été installées sur le calculateur** Plafrim **sous la forme de modules :**

```
module load mds-intel-openmpi mds-scotch/5.1.11 mds-metis/4.0.1 mds-pastix/5.1.4
```

ou encore plus simplement :

```
module load mds-meta-pastix
```

Pour récupérer une collection d'exemples (disponible également dans l'archive des distributions récentes de PaStiX) :

```
mkdir -p ~/examples-pastix && cp -R $PASTIX_EXAMPLES/src ~/examples-pastix/
cd ~/examples-pastix/src
make
cd ../bin
```

Pour exécuter un exemple simple :

```
mpirun -np 2 ./simple -lap 1000
```

Un ensemble de matrices sont disponibles dans différents formats dans le répertoire `$MATRIX\_HOME` (après chargement du module `mds-matrix`). [1]

1. les exemples peuvent lire des matrices au format Harwell Boeing (`.rsa` ou `.rua`) ou Matrix Market (`.mtx`).

Pour exécuter un second exemple, en précisant une matrice réelle symétrique au format Harwell Boeing :

```
mpirun -np 2 ./dsimple -rsa $MATRIX\_HOME/small.rsa
```

Pour exécuter troisième exemple, en utilisant un prétraitement parallèle sur une matrice complexe au format Matrix Market :

```
mpirun -np 2 ./zsimple_dist -mm $MATRIX\_HOME/young4c.mm
```

Utiliser les options de PaStiX

PaStiX utilise deux vecteurs de paramètres pour déterminer tout un ensemble d'options. Ce sont les vecteurs `iparm` et `dparm` , voir la fiche suivante pour plus de détails :

```
evince $PASTIX_HOME/doc/refcard.pdf
```

En particulier :
- `IPARM_SYM` : matrice symétrique ou non
- `IPARM_FACTORIZATION_TYPE` : type de factorisation
- `IPARM_VERBOSE` : niveau des informations affichées
- `IPARM_ITERMAX` : nombre maximum d'itérations de l'étape de raffinement
- `DPARM_EPSILON_REFINEMENT` : précision demandée

Testez ces différents paramètres sur les matrices fournies.
Remarque : l'exemple `simple` associe automatiquement le type de factorisation avec le type de la matrice détectée lors de la lecture. L'option `-h` permet d'afficher la liste des paramètres disponibles.

Choix du partitionneur de graphes

Il est possible d'imposer à PaStiX sa propre renumération des inconnues en fournissant le vecteur `perm` lors de l'appel au solveur. Il est cependant conseillé de faire appel à l'étape de renumérotation afin de minimiser le remplissage et maximiser le parallélisme. Le choix du partitionneur est réglé lors de la compilation, et il est possible de disposer à la fois de METIS et Scotch .

Regarder l'impact du partitionneur sur le remplissage et le temps de factorisation.
Remarque : avec les exemples de type `*simple*` il est possible de choisir son partionneur avec l'option `-ord metis` ou `scotch`.

Enchaînements séparés

Le solveur PaStiX est décomposé en 8 étapes :
1. Initialisation de la structure du solveur
2. Renumérotation
3. Factorisation symbolique
4. Distribution et calcul de l'ordonnancement
5. Factorisation
6. Résolution
7. Raffinement
8. Libération des structures

Chacune de ces étapes peut être appelée séparément en utilisant les paramètres `IPARM_START_TASK` et `IPARM_END_TASK`.

Regardez le code de l'exemple `step-by-step` pour tester différentes combinaisons.

Analyse parallèle avec une matrice distribuée

Il est possible d'appeler le solveur PaStiX avec une matrice déjà distribuée (dans le code utilisateur). On utilise, dans ce cas, un appel à un partitionneur de graphe parallèle (PT-Scotch ou ParMETIS).

Regarder la scalabilité et l'impact du partitionneur parallèle sur le remplissage.
Remarque : la matrice (stockée sous une forme compressée CSC) doit alors être distribuée par colonne sur les processeurs. Afin de faciliter la distribution de la matrice, le driver de lecture d'une matrice (`read_matrix`) est étendu pour retourner une matrice déjà distribuée sur les processus MPI (`dread_matrix`) en faisant appel à la fonction `csc_dispatch` (voir refcard) disponible dans la bibliothèque PaStiX *. C'est ce que réalise l'exemple `simple_dist`.*

```
mpirun -np 4 ./simple_dist -rsa $MATRIX\_HOME/shipsec5.rsa
```

Utilisation du parallélisme de thread

PaStiX est un solveur qui utilise un parallélisme hybride MPI+thread. L'intérêt d'utiliser des threads plutôt que des processus MPI distincts et de pouvoir économiser la mémoire nécessaire aux buffers de communication et d'améliorer l'efficacité parallèle sur les architectures multi-cœurs.

Il faut cependant noter que ce modèle peut poser des problèmes lors de l'intégration d'une bibliothèque de ce type avec un code utilisateur n'utilisant pas ce modèle de programmation. D'une manière idéale, l'utilisateur doit réserver et placer un processus MPI par nœud multicœur. C'est le solveur qui s'occupe de créer, d'exécuter puis de terminer des threads dans chaque processus MPI lors de la factorisation et de la résolution numérique.

Testez différentes combinaisons entre nombre de threads (`IPARM_THREAD_NBR`) et processus MPI et regardez l'impact sur la mémoire utilisée [2].
Remarque : avec les exemples de type `*simple*` il est possible de spécifier le nombre de threads par processus MPI avec l'option `-t` :

```
mpirun -np 4 ./simple -rsa $MATRIX\_HOME/audi.rsa -t 4
```

Interface commune aux solveurs PaStiX et HIPS : Murge

Une nouvelle interface a été définie pour uniformiser et normaliser les appels à un solveur générique. Le développement de cette interface pour les solveur PaStiX et HIPS est documenté sur le site web du projet Murge : `http://murge.gforge.inria.fr`.

Un couplage fin entre le code de simulation et le solveur peut-être mis en place permettant de minimiser les étapes de redistribution ; en particulier, la distribution des éléments pour la phase d'assemblage (parallélisée en OpenMP par exemple) devrait s'appuyer sur la distribution des inconnues imposée par le solveur. Des routines sont disponibles pour masquer le parallélisme lors de la phase d'assemblage de la matrice. L'utilisateur peut se contenter de parcourir ses éléments locaux, fournir des matrices élémentaires, éventuellement avec plusieurs degrés de liberté, imposer ses conditions aux limites et spécifier le type d'opération à effectuer sur l'interface.

Regarder les appels à l'interface Murge dans l'exemple `Murge-Fortran`
Remarque : cet exemple détecte le nombre de cores de l'architecture et active le nombre de threads correspondant lors de l'appel du solveur. La boucle d'assemblage de la matrice est parallélisée à l'aide de directive OpenMP.

```
mpirun -np 2 ./Murge-Fortran 1000 3
```

2. le solveur peut être compilé avec l'option -DMEMORY_USAGE

Factorisation incomplète de type ILU(k)

Le solveur PaStiX a été initialement développé pour implémenter un solveur parallèle basé sur une méthode directe supernodale. Mais il a été dérivé pour pouvoir être utilisé comme un préconditionneur, basé sur une factorisation incomplète de type ILU(k), couplé avec des méthodes itératives.

L'objectif est de trouver un compromis entre une diminution importante de la taille mémoire pour stocker la matrice factorisée par une méthode directe, et une conservation d'une certaine dose de remplissage (les blocs conservés seront toujours considérés comme pleins) pour exploiter suffisamment les effets super-scalaires dans les calculs BLAS3 du préconditionneur et atteindre globalement de bonnes performances en temps. Bien sûr, cette méthode conserve les techniques de renumérotation et de distribution/ordonnancement pour avoir une implémentation parallèle efficace du calcul par blocs du préconditionneur et des itérés.

Il est nécessaire d'activer cette option à l'aide du paramètre `IPARM_INCOMPLETE`, puis de régler le niveau de remplissage k avec le paramètre `IPARM_LEVEL_OF_FILL`, et enfin augmenter la valeur par défaut du niveau d'amalgamation α ou choisir une amalgamation automatique avec le paramètre `IPARM_AMALGAMATION_LEVEL`.
Plusieurs types de raffinement itératifs sont implémentés en interne (`IPARM_REFINEMENT`) :
– Gradient Conjugué (`API_RAF_GRAD`), pour des systèmes symétriques.
– GMRES (`API_RAF_GMRES`), pour des systèmes non symétriques.
Il est possible de définir la taille maximale de l'espace de Krylov (`IPARM_GMRES_IM`) ainsi que le critère d'arrêt (`DPARM_EPSILON_REFINEMENT`) et le nombre d'itérations maximum (`IPARM_ITERMAX`).

Faire varier k et α et regarder l'impact sur la convergence et le temps de factorisation.
Remarque : avec les exemples de type `*simple*` il est possible de spécifier les paramètres de la factorisation incomplète avec l'option `-incomp` k α :

```
./simple -mm $MATRIX\_HOME/nasa1824.mtx -incomp 3 40
```

Calcul d'un complément de Schur

Il est possible d'indiquer au solveur PaStiX les inconnues que l'on souhaite placer dans le calcul d'un complément de Schur. Dans sa version actuelle, le complément de Schur est dense et stocké de manière centralisée. Le solveur peut allouer lui même le complément de Schur et retourner un pointeur sur ce bloc, ou peut prendre en entrée un pointeur alloué par l'utilisateur et y placer le complément de Schur. Cette dernière solution permet d'éviter une double allocation dans le cadre d'un couplage avec un solveur hybrid comme HIPS ou MaPHyS .

Il est nécessaire d'activer l'option `IPARM_SCHUR` et de lister les numéros des inconnues à placer dans le complément de Schur à l'aide de la fonction `pastix_setSchurUnknownList`. Après l'appel à l'étape de factorisation numérique de PaStiX , le bloc du complément Schur est récupéré à l'aide de la fonction `pastix_getSchur`.

Essayer de placer la seconde moitié des inconnues (dans la numérotation initiale) dans le complément de Schur.
Remarque : la version 2 de l'exemple `schur` utilise une zone mémoire fournie par l'utilisateur pour éviter une copie.

```
./schur2 -mm $MATRIX\_HOME/wang1.mtx
```

Implémentation Out-of-Core

Une version "Out-of-Core" du solveur PaStiX a été implémentée (EXPERIMENTAL). Les supernœuds et les blocs de contributions sont stockés sur disque en anticipant de manière optimale les

opérations d'entrée/sortie pour recouvrir les temps d'accès aux disques par des calculs. Les mécanismes de pagination ont été implémentés en utilisant des primitives standards d'entrée/sortie mais, il est envisagé, dans les évolutions futures, d'utiliser des appels bas niveau plus performant. Pour utiliser cette fonctionnalité, il faut compiler le solveur avec l'option `-DOOC`. Ensuite, il est nécessaire de fixer la taille mémoire maximale disponible (exprimée en Mo) à l'aide de l'option `IPARM_OOC_LIMIT`.

Dans la version actuelle, le fonctionnement "Out-of-Core" est instable sur certaines architectures avec un parallélisme de type MPI. C'est pourquoi il est recommandé de n'utiliser cette fonctionnalité qu'avec la version séquentielle ou multi-threads du solveur.

Optimisations performances

Choisir le facteur de blocage

Cette option correspond aux paramètres `IPARM_MIN_BLOCKSIZE` et `IPARM_MAX_ BLOCKSIZE`. Ils permettent de choisir la taille minimale et maximale des blocs denses. Le facteur de blocage dépend essentiellement de la bibliothèque BLAS utilisée. La valeur 60 est couramment utilisée.

Allouer les données au plus près du processeur

Cette option de compilation (`-DNUMA_ALLOC`) est activée par défaut. Elle a été principalement développée dans le cadre des travaux sur l'adaptation aux architectures NUMA (Non Uniform Memory Acces). Elle apporte un gain de l'ordre de 5% sur une machine cible avec des accès homogènes à la mémoire (nœuds SMP) et de plus de 15% pour des machines NUMA.

Associer une thread à un processeur

Suivant les architectures, le système d'exploitation aura tendance à "visser" les threads POSIX sur des processeurs ou au contraire les faire migrer d'un processeur à un autre suivant sa charge. Dans un cadre normal de fonctionnement du solveur, il devrait y avoir une thread de calcul par processeur ou cœur physique ; il est donc souhaitable de ne pas autoriser le système d'exploitation à migrer les threads en affectant une thread à un processeur ou à un cœur.
Cette option correspond au paramètre `IPARM_THREAD_COMM_MODE`. Dans certains cas particulier (comme l'utilisation du solveur pour résoudre des systèmes distincts en parallèle avec des communicateurs MPI différents), il est nécessaire de spécifier un vecteur de placement (à l'aide de la fonction `pastix_setBind`) indiquant quelle thread doit être associée à quel processeur ou cœur.

Utiliser une ou plusieurs threads dédiées aux communications

Dans le cadre de travaux sur le ré-ordonnancement dynamique interne du solveur, un nouveau mode de communication a été implémenté. Par défaut, chaque tâche de calcul est responsable de l'envoi et de la réception de ses données. Ce nouveau schéma utilise une (ou plusieurs) thread de communication supplémentaire qui prend en charge, au niveau du processus MPI, la réception (et l'ajout des contributions associées) et s'occupe également de faire progresser les communications.
Pour activer cette option, il est nécessaire de compiler le solveur avec l'option `-DTHREAD_COMM`. Dans ce cas, les échanges de données utilisent des communications persistantes. Autrement, il est possible d'activer l'utilisation des réceptions non bloquantes avec l'option de compilation `-DTEST_IRECV`.

Adapter les modèles

Il est possible d'ajuster les modèles de calcul BLAS ainsi que des modèles des échanges sur le réseau et en mémoire partagée. Ces modèles sont utilisés par l'ordonnancement statique du solveur. Cette procédure d'adaptation n'est pas automatique, mais des scripts sont disponibles pour effectuer des mesures sur l'architecture cible et générer les modèles correspondants.

Optimisations mémoires

Régler le niveau d'amalgamation

Cette option correspond au paramètre `IPARM_AMALGAMATION_LEVEL`. Le réglage de ce paramètre est particulièrement important si le solveur est utilisé comme préconditionneur avec une factorisation incomplète. Généralement, il est conseillé d'utiliser une valeur autour de 5% dans le cas d'une factorisation directe et autour de 40% pour des factorisations incomplètes (ou une valeur négative pour utiliser un critère automatique).

Gestion des données internes

Il est possible d'indiquer au solveur qu'il peut dés-allouer :
- la matrice CSC de l'utilisateur (option `IPARM_FREE_CSCUSER`), à la fin de l'initialisation des coefficients dans les structures internes du solveur.
- la matrice CSC interne du solveur (option `IPARM_FREE_CSCPASTIX`), mais dans ce cas, il ne sera pas possible d'appliquer un raffinement itératif sur la solution.

Optimisations numériques

Appliquer un prétraitement numérique

Le solveur PaStiX ne propose pas, en interne, de prétraitement numérique.

Régler le pivotage statique

Cette option correspond au paramètre `DPARM_EPSILON_MAGN_CTRL`.

Activer et choisir un raffinement itératif

Le solveur PaStiX implémente, en interne, plusieurs types de raffinement itératifs :
- Gradient Conjugué, pour des systèmes symétriques.
- GMRES, pour des systèmes non-symétriques.

Il est possible de définir la taille de l'espace de Krylov ainsi que le critère d'arrêt et le nombre d'itérations maximum.