Integration of PETSc Linear Solver Package into ISIS-CFD Flow Solver

by

Jarunan PANYASANTISUK

Master Thesis

Presented to the Equipe de Modélisation Numérique
Of the Ecole Central de Nantes
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science in Computation Mechanics

Ecole Centrale de Nantes
June 2009

Integration of PETSc Linear Solver Package into ISIS-CFD Flow Solver

by

Jarunan PANYASANTISUK
Ecole Centrale de Nantes, 2009

SUPERVISOR: Dr. Ganbo DENG

Abstract: In an incompressive flow solver ISIS-CFD, the most time consuming part is the linear solver for the pressure equation. Its preconditioning method and solver are neither scalable nor optimized for parallel computation. The Portable, Extensible Toolkit for Scientific Computation (PETSc) contains many of the mechanisms needed within parallel application codes as well as scalable parallel preconditioners. The performances of linear solver programmed with PETSc and one in ISIS-CFD are analyzed through this research.

CONTENTS

Contents

1. INTRODUCTION

1.1 ISIS-CFD

ISIS-CFD is an incompressible flow solver, uses the incompressible unsteady Raynold-averaged Navier Stokes equation (RANSE). The solver is based on the finite volume method to build the spatial discretization of the transport equations. The face-based method is generalized to two-dimensional, rotationally-symmetric, or three-dimensional unstructured meshes for which non-overlapping control volumes are bounded by an arbitrary number of constitutive faces.

The ISIS-CFD flow solver has developed by EMN (Equipe Modélisation Numérique) at Ecole Centrale de Nantes since more than 10 years. Its accuracy and robustness have been demonstrated in various international workshops, classical benchmarks, and EU research projects. From the end of 2006, it is commercialized by Numeca in a software package named Fine/Marine including Hexpress, a hexahedral unstructured mesh generator, ISIS-CFD flow solver, and a postprocessor CFView. Fine/Marine users grow quickly in spite of dominating position of a few CFD commercial softwares.

To remain competitive in the market, constant improvements are required. Reducing the computation time is one of the tasks with top priority. With the current version of ISIS-CFD, a typical computation using a grid with several million nodes requires 3-4 days of computation with about 20 processors. It is desirable to reduce the computation time within 1 day. The most time consuming part of the code is the linear solver for the pressure equation. It is solved with the BICGSTAB algorithm with an incomplete LU preconditioner, which is not scalable and is not optimized for parallel computation.

1.2 PETSc

The Portable, Extensible Toolkit for Scientific Computation (PETSc) is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the MPI standard for all message-passing communication for parallel computation.

PETSc consists of a variety of libraries (similar to classes in C++). Each library manipulates a particular family of objects (for instance, vectors) and the operations one would like to perform on the objects. A large suite of parallel linear and nonlinear equation solvers are easily used in application codes written in C, C++, Fortran and Python. PETSc provides many of the mechanisms needed within parallel application codes, such as simple parallel matrix and vector assembly routines that allow the overlap of communication and computation. Features include:

- Parallel vectors
- Parallel matrices
- Scalable parallel preconditioners
- Krylov subspace methods
- Parallel Newton-based nonlinear solvers
- Parallel timestepping (ODE) solvers
- Complete documentation
- Automatic profiling of floating point and memory usage
- Consistent user interface
- Intensive error checking
- Portable to UNIX and Windows
- Over one hundred examples
- PETSc is supported and will be actively enhanced for many years

In this research, on the features parallel vectors, parallel matrices, scalable parallel preconditioners and Krylov subspace methods are focused.

1.3 Installation PETSc and MPICH

1.3.1 Installation MPICH

MPICH is an essential part in installing PETSc as a compiler wrapper such as mpicc, mpicxx and mpif90. Although PETSc offers an option `--download-mpich=1` in the configuration to download MPICH2 but this latest released of MPI is not compatible to cluster computing. If no standard MPICH in the system is provided, the installation can be processed by the following steps.

- The current release mpich-1.2.7p1 can be downloaded at http://www-unix.mcs.anl.gov/mpi/mpich1/download.html

```
tar zxof mpich.tar.gz
(gunzip -c mpich.tar.gz | tar zxovf -)

cd mpich-1.2.7p1
```

- Define fortran compiler.

```
export FC=/usr/local/intel_91/fc/bin/ifort
export F77=/usr/local/intel_91/fc/bin/ifort
export F90=/usr/local/intel_91/fc/bin/ifort
```

- Set the configuration and install MPICH. The option `--with-common-prefix=dir` can be used to set the directory path for installing tools such as upshot and jumpshot that are independent of the MPICH device.

  ```
  ./configure --with-device=ch_p4 \
  --prefix=/work/common/petsc/mpich-1.2.7p1/ch_p4 \
  --with-commen-prefix=/work/common/petsc/mpich-1.2.7p1

  make
  make install
  ```

- Run the test to check if the MPICH is installed correctly.

  ```
  cd example/test/pt2pt/
  make testing
  ```

  Notes: `pi3f90.f` in `ch_p4` device is missing, therefore, an error occurs during `make install`. The problem can be fixed by commands

  ```
  cd /work/common/petsc/mpich-1.2.7p1/ch_p4/examples
  /work/common/petsc/mpich-1.2.7p1/ch_p4/bin/mpif90 -c pi3f90.f90
  /work/common/petsc/mpich-1.2.7p1/ch_p4/bin/mpif90 -o pi3f90 pi3f90.o
  cd /work/common/petsc/mpich-1.2.7p1/
  make install
  ```

1.3.2 Installation PETSc

- The latest PETSc release tarball (petsc-3.0.0-p2.tar.gz) can be downloaded from http://www.mcs.anl.gov/petsc/petsc-as/download/index.html

  ```
  cd $PATH
  gunzip -c petsc-3.0.0-p2.tar.gz | tar -xof –

  cd petsc-3.0.0-p2/
  ```

- Set the environment/make variable `PETSC_DIR` (bash shell) to define the directory where PETSc is installed.

  ```
  export PETSC_DIR= $PATH/petsc-3.0.0-p2
  ```

- Set the configuration. In this research, the external packages, HyPre and Trillinos/ML, are included.

  ```
  ./config/configure.py --with-mpi-dir=$PATH/mpich-1.2.7p1 \
  --download-f-blas-lapack=1 --download-hypre=1 \
  --download-ml=1 --with-shared=0
  ```

- Add the environment variables and paths in `~/.bashrc`. The environment variable `PETSC_ARCH` must be set to specify the architecture. For shared libraries which could not be found, the user can set their path with `LD_LIBRARY_PATH`.

```
PATH=/work/common/petsc/mpich-1.2.7p1/bin:
/work/common/isiscfd/interface/numeca_software/bin:${PATH}
export PETSC_DIR=/work/common/petsc/petsc-3.0.0-p2
export PETSC_ARCH=linux
export LD_LIBRARY_PATH=/usr/local/intel_91/fc/lib
```

2. WRITING PETSC PROGRAM IN FORTRAN

2.1 Include Files

The Fortran include files for PETSc are located in the directory `${PETSC_DIR}/include/finclude` and should be used via statements such as the following:

```
#include "finclude/includefile.h"
```

In Fortran one must explicitly list each of the include files and must be very careful to not include each file no more than once. The Fortran file suffix must be .F rather than .f. This convention enables use of the CPP preprocessor, which allows the use of the `#include` statements that define PETSc objects and variables.

2.2 Initialization and Finalization

Most PETSc programs in Fortran begin with a call to

```
call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
```

`PetscInitialize()` automatically calls `MPI_Init()` if MPI has not been not previously initialized. In certain circumstances in which MPI needs to be initialized directly (or is initialized by some other library), the user can first call `MPI_Init()`, or have the other library do it, and then call `PetscInitialize()`. By default, `PetscInitialize()` sets the PETSc "world" communicator, given by `PETSC_COMM_WORLD`, to `MPI_COMM_WORLD`.

MPI communicator is a way of indicating a collection of processes that will be involved together in a calculation or communication. Communicators have the variable type `MPI_COMM`. In most cases users can employ the communicator `PETSC_COMM_WORLD` to indicate all processes in a given run and `PETSC_COMM_SELF` to indicate a single process.

All PETSc programs should call `PetscFinalize()` as their final (or nearly final) statement.

```
call PetscFinalize(ierr)
```

This routine handles options to be called at the conclusion of the program, and calls `MPI_Finalize()` if `PetscInitialize()` began MPI. If MPI was initiated externally from PETSc but by either the user or another software package, the user is responsible for calling `MPI_Finalize()`.

2.3 Passing Null Pointers

In several PETSc functions, there are options of passing null argument, for example

```
       KSPMonitorSet(ksp,Monitor,PETSC_NULL_OBJECT,
     $       PETSC_NULL_FUNCTION,ierr)
```

For Fortran, users must pass `PETSC_NULL_XXX` to indicate a null argument, where `XXX` is `INTEGER`, `DOUBLE`, `CHARACTER`, or `SCALAR` depending on the type of argument required.

2.4 Error Checking

Each PETSc routine has as its final argument an integer error variable. For example,

```
call KSPSolve(ksp,rhs,u,ierr)
```

where `ierr` denotes the error variable. The error code is set to be nonzero if an error has been detected, otherwise, it is zero.

2.5 Matrix and Vector Indices

All matrices and vectors in PETSc use zero-based indexing, whereas Fortran is one-based indexing language. The interface routines, such as `MatSetValues()` and `VecSetValues()`, always use zero-based indexing.

3. PROGRAMMING WITH PETSC

3.1 Vector

Vector module is denoted by `Vec`. A vector for parallel computing can be created by command

```
      call VecCreateMPI(PETSC_COMM_WORLD,N,PETSC_DECIDE,x,ierr)
```

where integer `N` is local size and `x` is the vector. In this routine, the size of each processor's local portion (`N`) is specified, and let PETSc compute the global size by passing `PETSC_DECIDE` instead of giving global size. Alternatively, if one can pass the global size and use `PETSC_DECIDE` for the local size. PETSc will choose a reasonable partition trying to put nearly an equal number of elements on each processor. To create a new vector, here is vector `u`, of the same format as an existing vector, the command is

```
      call VecDuplicate(x,u,ierr)
```

Setting values to a vector by calling `VecSetValues()` always specify global locations of vector entries. The array `Local_to_Global_Mapping` contains global indices where to add values. Each processor can contribute any vector entries, regardless of which processor "owns" them. Any nonlocal contributions will be transferred to the appropriate processor during the assembly process. Here the flag `INSERT_VALUES` indicates that all contributions will be inserted and delete the old value.

```
      call VecSetValues(x,N,Local_to_Global_Mapping,
     $     Sol,INSERT_VALUES,ierr)
```

`Sol` is an array containing the referenced solutions. To Assemble vector, using the 2-step process `VecAssemblyBegin()` and `VecAssemblyEnd()`. Computations can be done while messages are in transition by placing code between these two statements.

```
      call VecAssemblyBegin(x,ierr)
      call VecAssemblyEnd(x,ierr)
```

A vector can be examine with the command `VecView()`, while the option `PETSC_VIEWER_STDOUT_WORLD` synchronize standard output where only the first processor opens the file. All other processors send their data to the first processor to print. By default, the option `PETSC_VIEWER_STDOUT_SELF` is set for the standard output. When a vector is no longer needed, it should be destroyed by calling `VecDestroy()`

```
      call VecView(x,PETSC_VIEWER_STDOUT_WORLD,ierr)
      call VecDestroy(x,ierr)
```

3.2 Matrix

Matrix module is denoted by `Mat`. The routine below shows an easy mechanism of creating a matrix and setting its configuration.

`MatCreate()` is the simplest routine for creating a matrix, as seen in line 1 it creates a matrix `D`. `MatSetSizes()` in line 2 is used for defining the matrix size, in which the local dimension `N x N` is specified and let PETSc computes the global size by passing `PETSC_DETERMINE` into the command. A matrix type for parallel computation is `MATMPIAIJ` set via `MatSetType()`, line 3.

```
1     call MatCreate(PETSC_COMM_WORLD,D,ierr)
2     call MatSetSizes(D,N,N,PETSC_DETERMINE,PETSC_DETERMINE,
    $     ierr)
3     call MatSetType(D,MATMPIAIJ,ierr)
4     call MatMPIAIJSetPreallocation(D,nz,PETSC_NULL_INTEGER,
    $     2,PETSC_NULL_INTEGER,ierr)
```

`nz` and `2` in the command `MatMPIAIJSetPreallocation()`, line 4, are numbers of diagonal nonzero and off-diagonal nonzero per row, respectively. Consequently, `PETSC_NULL_INTEGER` can be passed into the command instead of passing arrays containing number of nonzero in the various rows of the diagonal and off-diagonal portion of the local submatrix, which are possibly different for each row. The preallocation of memory for parallel AIJ sparse matrices is explained in Appendix C.

In each iteration, ISIS-CFD computes new matrix values, array `a`, and new right hand side values, array `src`, therefore, updating the matrix must be redone before starting a new iteration. To define a matrix with arrays, ISIS-CFD gives

- The array a contains values of the matrix
- The array `IndCon_CC` contains local column indices of value in `a`
- The array `IpntCF_CC` contains indices pointing into `IndCon_CC`, where to begin a new row.

From these arrays, the values can be passed to a Matrix in PETSc with the command `MatSetValues()` which inserts or adds a block of values into a matrix. `ADD_VALUES` indicates to add a value into the specified location, if there previously was no value, just put the value into that location. The following routine shows adding values by row into the matrix `D`.

```
1     LoToGlo=Local_to_Global_Mapping(IndCon_CC)

2     do i=1,N
3          globalIndRow=Local_to_Global_Mapping(i)
4          pntBegin=IpntCF_CC(i)
5          pntEnd=IpntCF_CC(i+1)-1
6          j=pntEnd-pntBegin+1
7          call MatSetValues(D,1,globalIndRow,j,
    $           LoToGlo(pntBegin:pntEnd),
    $           a(pntBegin:pntEnd),ADD_VALUES,ierr)
8      end do
```

Setting values to the matrix `D` in line 7, `1` is number of row and the integer `globalIndRow` is its global index. The integer `j` and the integer array `LoToGlo` are number of columns and their global indices.

Table 3.1 Time of updating a matrix in second run with 2 processors

Preconditioner	Fine	Medium	Coarse	Very coarse	Vv coarse
ISIS-CFD	11.8470	5.6323	3.3168	1.3877	0.6346
HyPre/ILU(1)	2285.3920	659.3274	124.1123	7.2885	17.5339
HyPre/Multigrid	2181.5770	688.6473	144.2728	41.0015	24.4377

From Table 3.1, we can see that this routine takes much time to update a matrix which refers to high computation cost. Alternatively, we can create a matrix and set its values with the command `MatCreateMPIAIJWithSplitArrays()` by giving arrays of values and their indices. In this way, we could save computation cost in updating a matrix as seen in Table 3.2. However, values and their indices must be prepared by separating between diagonal portion and off-diagonal portion, and sorting the column indices.

```
      call MatCreateMPIAIJWithSplitArrays(PETSC_COMM_WORLD,N,
     $     N,PETSC_DETERMINE,PETSC_DETERMINE,pointer,column,v,
     $     opointer,ocolumn,ov,D,ierr)
```

The local dimension `N x N` is set for the matrix `D`. Passing `PETSC_DETERMINE` let PETSc calculates the global dimension. `pointer, column` and `v` are the row indices, column indices and values for diagonal portion of matrix, while `opointer, ocolumn` and `ov` are for off-diagonal portion.

Table 3.2 Time of updating a matrix in second

Preconditioner	Fine	Medium	Coarse	Very coarse	Vv coarse
ISIS-CFD	11.8470	5.6323	3.3168	1.3877	0.6346
Block Jacobi	14.7509	7.5555	1.0829	2.0193	0.3630

However, this routine can run neither with preconditioners in the external package HyPre nor with a single processor. Anyway, this routine will be used for the tests in chapter 5, when the preconditioner is set to Block Jacobi, Additive schwarz method, Multigrid method in PETSc, or Multigrid method in the external package Trillinos.

After the matrix completed setting the values, these values may be cached, so `MatAssemblyBegin()` and `MatAssemblyEnd()` must be called.

```
      call MatAssemblyBegin(D,MAT_FINAL_ASSEMBLY,ierr)
      call MatAssemblyEnd(D,MAT_FINAL_ASSEMBLY,ierr)
```

`MatView()` let the users examine the matrix, and `MatDestroy()` is called when a matrix is no longer needed and should be destroyed.

```
      call MatView(D,PETSC_VIEWER_STDOUT_WORLD,ierr)
      call MatDestroy(D,ierr)
```

3.3 KSP solver and Preconditioner

To solve a linear system with Krylov subspace methods, a solver context (KSP) must be created with `KSPCreate()`. The flag `DIFFERENT_NONZERO_PATTERN` in `KSPSetOperators()` is to indicate that the preconditioner matrix does not have the same nonzero structure. Alternatively, users can set the flag `SAME_PRECONDITIONER` to indicate that the preconditioner matrix is identical to that of the previous linear solver, and `SAME_NONZERO_PATTERN` to indicate that the preconditioning matrix has the same nonzero structure during successive linear solvers. In case the structure of a matrix is not known a priori, one should use the flag `DIFFERENT_NONZERO_PATTERN`. Here the matrix that defines the linear system, the matrix `D`, also serves as the preconditioning matrix.

```
      call KSPCreate(PETSC_COMM_WORLD,ksp,ierr)
      call KSPSetOperators(ksp,D,D,
     $      DIFFERENT_NONZERO_PATTERN,ierr)
```

The default solver within KSP is restarted GMRES, preconditioned for the uniprocess case with ILU(0), and for the multiprocess case with the block Jacobi method (with one block per process, each of which is solved with ILU(0)). A variety of other solvers and options are also available. To set any of the preconditioner or Krylov subspace options directly within the code, PETSc provide routines that extract the PC and KSP contexts

```
      call KSPGetPC(ksp,pc,ierr)
```

To employ a particular preconditioning method, the user can either select it from the options database using input of the form `-pc_type <methodname>` or set the method with the command

```
      call PCSetType(pc,PCType,ierr)
```

The list of preconditioning method supported in PETSc is shown in Table 3.3. For the external package HyPre, its type can be set with the command `PCHYPRESetType()`, where by default is '`boomeramg`', the Algebraic Multigrid method.

`KSPSetTolerances()` is to set the relative, absolute, divergence, and maximum iteration tolerances used by the default KSP convergence testers. `PETSC_DEFAULT_DOUBLE_PRECISION` is used for retaining the default values of any tolerances.

```
      call KSPSetTolerances(ksp,tol,
     $      PETSC_DEFAULT_DOUBLE_PRECISION,
     $      PETSC_DEFAULT_DOUBLE_PRECISION,maxits,ierr)
```

The option `-ksp_monitor_true_residual` prints the true residual norm as well as the preconditioned residual norm in each iteration of an iterative solver.

```
      call PetscOptionsSetValue('-ksp_monitor_true_residual',
     $     'monitor.dat',ierr)
```

To set the Krylov solver, `KSPSetType()` is provided, the KSPType is listed in Table 3.4. `KSPSetFromOptions()` indicates that KSP options are set from the options database. To solve a linear system, the right hand side vector, `rhs`, and solution vector, `u`, must be set, then execute the command `KSPSolve()`.

```
      call KSPSetType(ksp,KSPType,ierr)
      call KSPSetFromOptions(ksp,ierr)
      call KSPSolve(ksp,rhs,u,ierr)
```

`KSPView()` prints KSP data structure. Once the KSP context is no longer needed, it should be destroyed with the command

```
      call KSPView(ksp,PETSC_VIEWER_STDOUT_WORLD,ierr)
      call KSPDestroy(ksp,ierr)
```

Table 3.3 Preconditioning methods

Preconditioner					
Algorithm	PCType	Matrix types*	External Package	Parallel	Complex
Jacobi	PCJACOBI	aij,baij,sbaij,dense		X	X
Point Block Jacobi	PCPBJACOBI	baij,bs=2,3,4,5		X	X
SOR	PCSOR	seqdense,seqaij, seqsbaij,mpiaij			X
Point Block SOR		seqbaij,bs=2,3,4,5			X
Block Jacobi	PCBJACOBI	aij,baij,sbaij		X	X
Additive Schwarz	PCASM	aij,baij,sbaij		X	X
ILU(k)	PCILU/PCICC	seqaij,seqbaij			X
ICC(k)		seqaij,seqbaij			X
ILU dt		seqaij	Sparsekit		
ILU(0)/ICC(0)		aij	BlockSolve95	X	
ILU(k)	PCHYPRE	aij	Euclid/HyPre	X	
ILU dt		aij	Euclid/HyPre	X	
Matrix-free	PCSHELL			X	X
Multigrid/infrastructure	PCMG			X	X
Multigrid/geometric structured grid	DMMG			X	X
Multigrid algebraic	PCHYPRE	aij	BoomerAMG/HyPre	X	
	PCML	aij	ML/Trilinos	X	
	PC	baij	Prometheus	X	
Approximate inverses	PCHYPRE	aij	Parasails/HyPre	X	
	PCSPAI	aij	SPAI	X	
Balancing Neumann-Neumann	PCNN	is		X	X
Direct solver					
LU	PCLU	seqaij,seqbaij			X
		seqaij	MATLAB		X
		aij	Spooles	X	X
		aij	PastuiX	X	X
		aij	SuperLU, Sequential/Parallel	X	X
		aij	MUMPS	X	X
		seqaij	ESSL		
		seqaij	UMFPACK		
		dense	PLAPACK	X	X
Cholesky	PCCHOLESKY	seqaij,seqbaij			X
		sbaij	Spooles	X	X
		sbaij	PastuiX	X	X
		sbaij	MUMPS	X	X
		seqsbaij	DSCPACK	X	
		dense	PLAPACK	X	X
		matlab	MATLAB		
		aij		X	
QR		matlab	MATLAB		
XXt and XYt		aij		X	

* Matrix types

aij	- A matrix type to be used for sparse matrix
baij	- A matrix types to be used for block sparse matrix
sbaij	- A matrix type to be used for symmetric block sparse matrices
seqaij	- A matrix type to be used for sequential sparse matrices, based on compressed sparse row format.
mpiaij	- A matrix type to be used for parallel sparse matrices.
seqbaij	- A matrix type to be used for sequential block sparse matrices, based on block sparse compressed row format.
seqsbaij	- A matrix type to be used for sequential symmetric block sparse matrices, based on block compressed sparse row format.
dense	- A matrix type to be used for dense matrices.
seqdense	- A matrix type to be used for sequential dense matrices.
is	- A matrix type to be used for using the Neumann-Neumann type preconditioners.

Table 3.4 Krylov Sybspace Methods

Krylov Sybspace Method	KSPType
Richardson	KSPRICHARDSON
Chebychev	KSPCHEVBYCHEV
Conjugate Gradients	KSPCG
GMRES	KSPGMRES
Bi-CG-stab	KSPBCGS
Transpose-free Quasi Minimal-Residual	KSPTFQMR
Conjugate Residuals	KSPCR
Conjugate Gradient Squared	KSPCGS
Bi-Conjugate Gradient	KSPBICG
Minimum Residual Method	KSPMINRES
Flexible GMRES	KSPFGMRES
Least Squares Method	KSPLSQR
SYMMLQ	KSPSYMMLQ
LGMRES	KSPLGMRES
Conjugate gradient on the normal equations	KSPCGNE

4. COMPILING AND RUNNING PETSC PROGRAM

4.1. Makefile

All makefile commands and customizations to enable portability across different architectures can be found in the directory `${PETSC_DIR}/conf`, whereas most makefile commands for maintaining the PETSc system are defined in the file `${PETSC_DIR}/conf`.

Two makefiles `petscvariables` and `petscrules` are automatically generated in `${PETSC_DIR}/${PETSC_ARCH}/conf`, when `config/configure.py` is run. They contain rules specific to this machine and the definition of compilers and linkers, respectively. The architecture independent makefiles, are located in `${PETSC_DIR}/conf`, and the machine specific makefiles get included from here.

The most important line in the makefile is the line starting with include

```
include ${PETSC_DIR}/conf/base
```

This line includes other makefiles that provide the needed definitions and rules for the particular base PETSc installation specified by `${PETSC_DIR}` and architecture specified by `${PETSC_ARCH}`. The library required for the linker is the highest level library in that PETSc program. The makefile used for the PETSc program in this research reads

```
RM      = /bin/rm

MYSRCS  = $(wildcard *.F)
MYOBJS  = $(subst .F,.o,$(MYSRCS))

include ${PETSC_DIR}/conf/base

test_with_petsc:  $(MYOBJS) chkopts
   -${FLINKER} -o test_with_petsc $(MYOBJS) ${PETSC_KSP_LIB}
    ${RM} *.o

include ${PETSC_DIR}/conf/test
```

4.2. Running a PETSc program

To run the PETSc executable in multiprocessor, the command is

```
mpirun –np 2 –machinefile machines ./test_with_petsc
```

Options in PETSc can be added at the end of command. For example, to list the options available in the program `test_with_petsc`

```
mpirun –np 2 –machinefile machines ./test_with_petsc –help
```

5. TESTS

The linear solver of ISIS-CFD uses a preconditioner ILU(1) with Block Jacobi and a solver PCGSTAB, which equal to BiCGStab. Pressure equation in a double model computation with different grid density is used for the test. The geometry is the KVLCC2 tanker. Wall function is used for the computation.

In this chapter, properties of preconditioning methods and Krylov solvers in PETSc are examined and compared with the ones in ISIS-CFD. The tests presenting are:-

- Convergence of PETSc preconditioning methods, section 5.1
- Convergence of PETSc Krylov methods, section 5.2
- Scalability, section 5.4
- Memory usage, section 5.5
- Speedup, section 5.6
- Convergence, section 5.7

Grids using for the tests are:-

- Fine grid, 1813351 cells
- Medium grid, 962717 cells
- Coarse grid, 397625 cells
- Very coarse grid, 242374 cells
- The coarsest grid named as Vv coarse grid, 115836 cells

Two methods of matrix creation in PETSc are used depending on the preconditioner using

a) Adding values by row by the command `MatSetValues()` is used when the preconditioner is set to ILU(k) and Multigrid method in the external Package HyPre , and

b) `MatCreateMPIAIJWithSplitArrays()` is used when the preconditioner is set to Block Jacobi, Additive Schwarz method, Multigrid method in PETSc or Multigrid method in the external package Trillinos.

Hereafter, only CPU time of one processor used to solve the linear system is compared. All the tests are run on PC local station.

5.1 Preconditioners

This test compares convergences of preconditioners:

- ILU(1) (with Block Jacobi) in ISIS-CFD
- ILU(k) (with Block Jacobi) in HyPre package
- Additive Schwarz method(ILU(k))
- Multigrid methods in PETSc, Trillinos package and HyPre package.

The Krylov methods of PETSc linear solver is set to BiCGStab, the same as the one in ISIS-CFD. This test is run with the coarse grid with 2 processors.

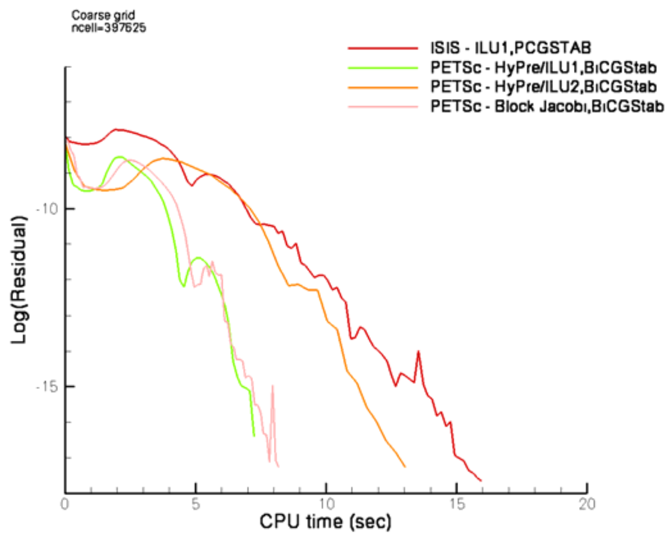

Figure 5.1 Convergence of ISIS-CFD, PETSc with preconditioners HyPre/ILU(k) and Block Jacobi

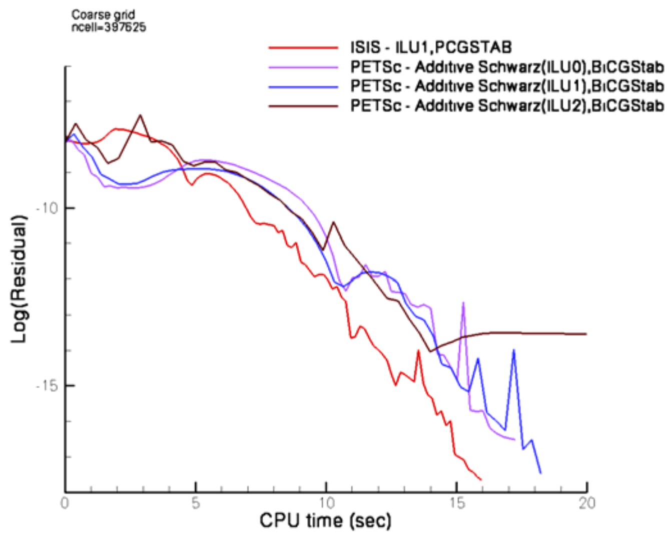

Figure 5.2 Convergence of ISIS-CFD, PETSc with preconditioner Additive Schwarz Method

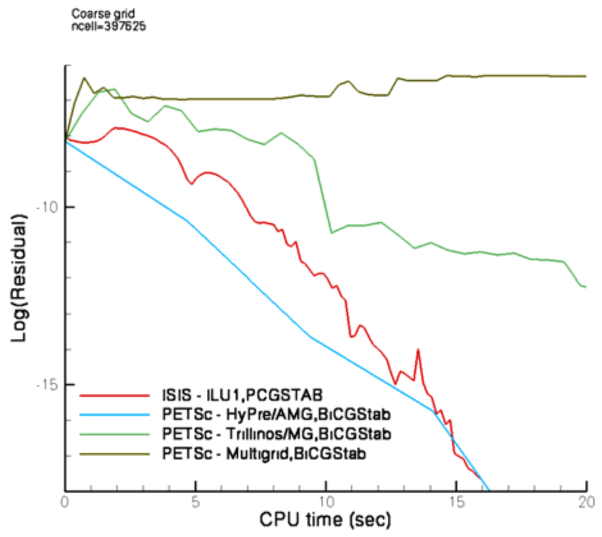

Figure 5.3 Convergence of ISIS-CFD, PETSc with preconditioner Multigrid Method

From Figure 5.1, 5.2 and 5.3, preconditioners which provide fast convergences are selected for testing properties of PETSc linear solvers in section 5.4-5.7. They are Block Jacobi, ILU(1) and Multigrid method in the external package HyPre.

5.2 Krylov methods

Convergences of different Krylov solvers are examined in this test. The PETSc preconditioner Block Jacobi is retained while comparing Krylov methods BiCGStab, Conjugate Gradient, GMRES, Chebychev and Richardson. The test is run with coarse grid with 2 processors.

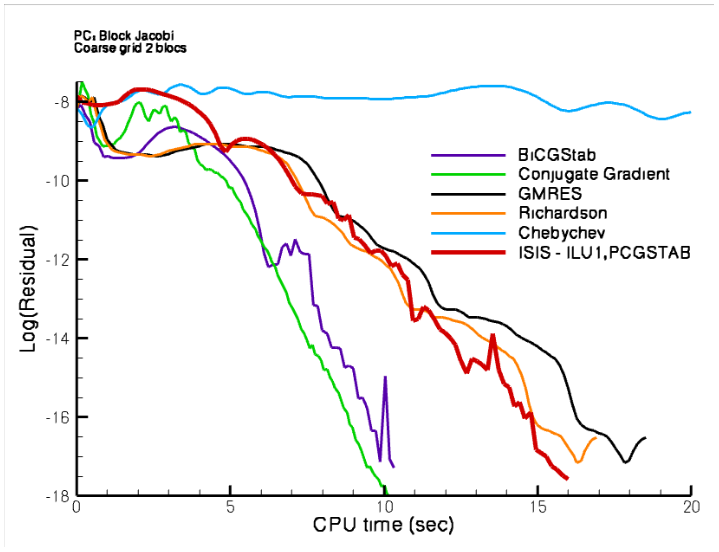

Figure 5.4 Convergence of ISIS-CFD and PETSc
with different Krylov methods

From Figure 5.4, obviously, Krylov methods BiCGStab and Conjugate Gradient have the fastest convergences. BiCGStab will be used in the scalability test, Memory usage, Speedup and the convergence test, while Conjugate Gradient will be applied in the convergence test.

5.3 Methods used in the tests

5.3.1 The Conjugate Gradient Algorithm

The Conjugate Gradient algorithm is one of the best known iterative techniques for solving sparse Symmetric Positive Definite linear systems. Described in one sentence, the method is a realization of an orthogonal projection technique onto the Krylov subspace $K_m(r_0, A)$ where r_0 is the initial residual. It is therefore mathematically equivalent to FOM.

Algorithm 5.3.1: Conjugate Gradient

```
1. Compute $r_0 := b - Az_0,\ p_0 = r_0$
2. For $j = 0,1,\ldots,$ until convergence Do:
3.     $\alpha_j := (r_j, r_j)/(Ap_j, p_j)$
4.     $x_{j+1} := x_j + \alpha_j p_j$
5.     $r_{j+1} := r_j - \alpha_j Ap_j$
6.     $\beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$
7.     $p_{j+1} := r_{j+1} + \beta_j p_j$
```

8. EndDo

5.3.2 BiCGStab

The Bi-Conjugate Gradient Stabilized (BiCGStab) algorithm is a variation of Conjugate Gradient Squared (CGS). As CGS is based on squaring the residual polynomial, and, in cases of irregular convergence, this may lead to substantial build-up of rounding errors, or possibly even overflow. BICGSTAB was developed to remedy this difficulty.

<u>Algorithm 5.3.2: BiCGStab</u>

```
1. Compute $r_0 := b - Ax_0; r_0^*$ arbitary;
2. $p_0 = r_0$
3. For $j = 0,1,...,$ until convergence Do:
4.      $\alpha_j := (r_j, r_j^*) / (Ap_j, r_j^*)$
5.      $s_j := r_j + \alpha_j Ap_j$
6.      $\omega_j := (As_j, s_j) / (As_j, r_j)$
7.      $x_{j+1} := x_j + \alpha_j p_j + \omega_j s_j$
8.      $r_{j+1} := s_j - \omega_j As_j$
9.      $\beta_j := \frac{(r_{j+1}, r_0^*)}{(r_j, r_0^*)} \times \frac{\alpha_j}{\omega_j}$
10.     $p_{j+1} := r_{j+1} + \beta_j (p_j - \omega_j Ap_j)$
11. EndDo
```

5.3.3 Incompleted LU

The Incomplete LU factorization technique with no fill-in, denoted by ILU(0), consists of taking the zero pattern *P* to be precisely the zero pattern of *A*. In the following, we denote by $b_{i,*}$ the *i*-th row of a given matrix *B*, and by *NZ(B)* , the set of pairs $(i, j), 1 \le i, j \le n$ such that $b_{i,j} \ne 0$

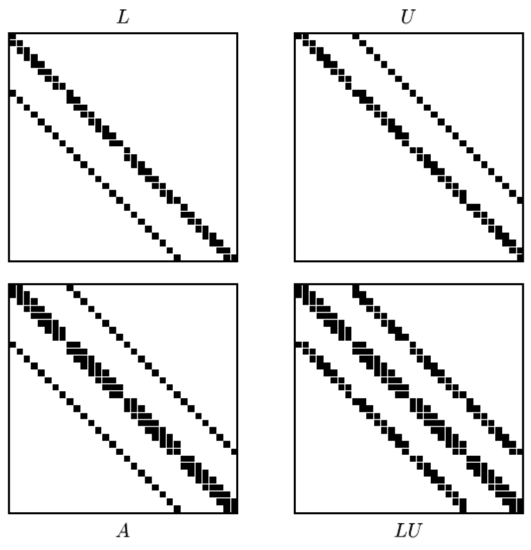

Figure 5.5 The ILU(0) factorization for a five-point matrix

The accuracy of the ILU(0) incomplete factorization may be insufficient to yield an adequate rate of convergence. More accurate Incomplete LU factorizations are often more efficient as well as more reliable. These more accurate factorizations will differ from ILU(0) by allowing some fill-in. Thus, ILU(1) keeps the “first order fill-ins” a term which will be explained shortly.

To illustrate ILU(p) with the same example as before, the ILU(1) factorization results from taking P to be the zero pattern of the product LU of the factors L,U obtained from ILU(0). This pattern is shown at the bottom right of Figure 5.5. Pretend that the original matrix has this “augmented” pattern $NZ_1(A)$. In other words, the fill-in positions created in this product belong to the augmented pattern $NZ_1(A)$, but their actual values are zero. The new pattern of the matrix A is shown at the bottom left part of Figure 5.6. The factors L_1 and U_1 of the ILU(1) factorization are obtained by performing an ILU(0) factorization on this “augmented pattern” matrix. The patterns of L_1 and U_1 are illustrated at the top of Figure 5.6. The new LU matrix shown at the bottom right of the figure has now two additional diagonals in the lower and upper parts.

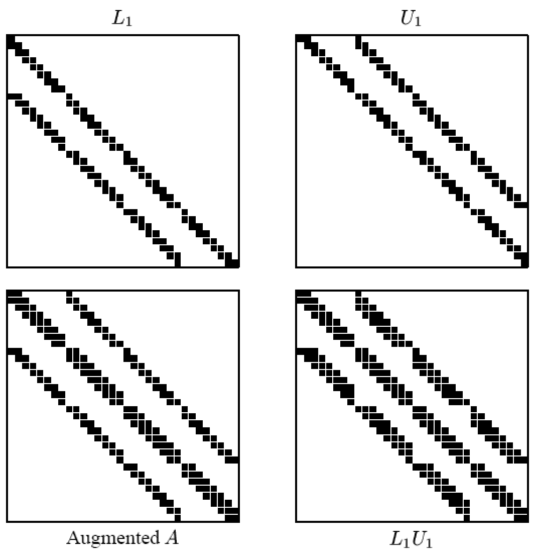

Figure 5.6 The ILU(1) factorization

Algorithm 5.3.3: ILU(p)

```
1. For all nonzero elements $a_{ij}$ define $lev(a_{ij}) = 0$
2. For $i = 2,...,n$ Do:
3. For each $k = 1,...,i-1$ and for $lev(a_{ij}) \le p$ Do:
4.     Compute $a_{ik} = a_{ik} / a_{kk}$
5.     Compute $a_{i*} := a_{i*} - a_{ik}a_{k*}$
6.     Update the level of fill of the nonzero $a_{i,j}$'s using (10.18)
7. EndDo
8. Replace any element in row i with $lev(a_{ij}) > p$ by zero
9. EndDo
```

In PETSc, ILU(k) can be called when the external package HyPre is installed. After setting preconditioning type to `PCHYPRE` (line1) we have to set HyPre type to `'euclid'` as seen in line 2. By default in HyPre is `'boomeramg'`, the algebraic Multigrid method. Many HyPre options can be set with the command `PetscOptionsSetValue()` as seen in line 3 and 4. The option to set its fill-ins is `-pc_hypre_euclid_levels` and `-pc_hypre_euclid_bj` is to set the ILU(k) method with block Jacobi. To list the HyPre options put `–help` after the run command line at runtime.

```
1     call PCSetType(pc,PCHYPRE,ierr)
2     call PCHYPRESetType(pc,'euclid',ierr)
3     call PetscOptionsSetValue('-pc_hypre_euclid_levels',
    $       '1',ierr)
4     call PetscOptionsSetValue('-pc_hypre_euclid_bj',
    $       'TRUE',ierr)
```

5.3.4 Algebraic Multigrid Method

Multigrid methods are a state-of-the art technique to solve large systems of linear equations *A x = b*, where $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$. This system can be represented as a graph of *n* nodes where an edge (*i*,*j*) represents a non-zero coefficient $A_{i,j}$. To simplify the following illustration, we assume that graph to be a regular two dimensional grid. The basic idea of multigrid is to define a hierarchy of grids as illustrated in Figure 5.7. Each node at the coarser grid level represents a set of nodes at the finer level. Coefficients at some grid level *i* are derived from coefficients at grid level *i*+1 (prolongation) or from coefficients at grid level *i*-1 (restriction). The grid hierarchy is traversed in V or W-cycles. On each level of the hierarchy an iterative solver is called.

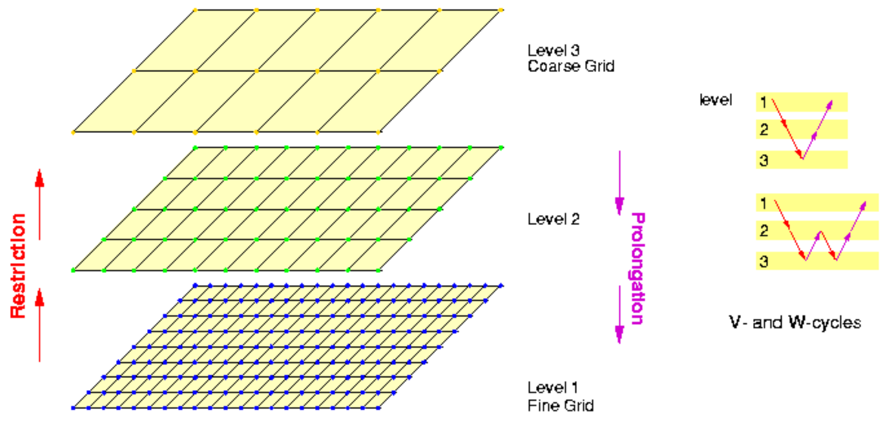

Figure 5.7 Multigrid Method

In geometric multigrid methods, coarse grids are determined based on geometry information (such as grid spacing) alone. In contrast, algebraic multigrid takes into account coefficient values, too. The algorithm below is a simple multigrid method.

Given a system of linear equations *Au*=*f* and an approximate solution *v*. The error *e* is defined as *e* = *v* - *u*. Thus, *Ae* = *A*(*v*-*u*) and from linearity of matrix-vector products we get *Ae* = *Av*-*Au*. We substitute *f* for *Au*, and get *Ae* = *Av* - *f*, that is, *Ae* is equal to the residue, as defined above.

Given *A* and *f*

1. Perform one step of an iterative method towards solving *Au* = *f* using initial guess of *v*=0
2. Calculate the residue *r*=*Av* - *f*

3. Reduce A and r to a coarser grid
4. Determine the error e by solving $Ae=r$ on the coarser grid
5. Prolong e to the original grid
6. Correct $v = v-e$
7. Perform another step of the iterative method towards solving $A\ u = f$, now using initial guess v

Observe that step 4 can be solved by applying this algorithm recursively, until the grid only contains a trivial number of points. Thus, one can descend through coarser and coarser grids, and then ascend back to the original grid.

There are many options database keys for HyPre/Multigrid (BoomerAMG). Its information can be seen in Annexe B as a result file. In the code it is set as the following.

```
      call PCSetType(pc,PCHYPRE,ierr)
      call PCHYPRESetType(pc,'boomeramg',ierr)
      call PetscOptionsSetValue(
     $     '-pc_hypre_boomeramg_max_levels','10',ierr)
```

5.3.5 Block Jacobi Method

Block versions of the Jacobi preconditioner can be derived by a partitioning of the variables. If the index set $S = \{1, \ldots, n\}$ is partitioned as $S = \bigcup_i S_i$ with the sets S_i mutually disjoint, then

$$m_{i,j} = \begin{cases} a_{i,j} & \text{if } i \text{ and } j \text{ are in the same index subset} \\ 0 & \text{otherwise} \end{cases}$$

The preconditioner is now a block-diagonal matrix.

Often, natural choices for the partitioning suggest themselves:

- In problems with multiple physical variables per node, blocks can be formed by grouping the equations per node.
- In structured matrices, such as those from partial differential equations on regular grids, a partitioning can be based on the physical domain. Examples are a partitioning along lines in the 2D case, or planes in the 3D case.
- On parallel computers it is natural to let the partitioning coincide with the division of variables over the processors.

5.4 Scalability test

Scalable precondition is desired in large-scale computation. Scalability indicates its ability to maintain the number of iterations when the number of processor used for computation increases.

The test is run with the coarse grid with a single processor, 2, 4, 8 and 16 processors, using BiCGStab as a solver. The number of iterations is observed as seen in Table 5.1.

Table 5.1 Number of iterations

Solver	Preconditioner	1 bloc	2 blocs	4 blocs	8 blocs	16 blocs
ISIS-CFD	ILU(1)	90	93	92	97	99
PETSc	HyPre/ILU(1)	79	77	82	84	87
	HyPre/Multigrid	8	10	9	8	10
	Block Jacobi	-	121	119	121	123

As expected, the Multigrid method is scalable. The iteration number remains fairly constant as well as Block Jacobi which shows its scalability. The number of iteration in single-processor of the preconditioner Block Jacobi case is missing because its matrix creation `MatCreateMPIAIJWithSplitArrays()` does not work with a single processor.

5.5 Memory usage

To monitor the maximum memory usage, the option `–memory_info` can be set at runtime. The memory usage will be printed at the end of the run. The medium grid is used for this test, run with 2, 4, 8 and 16 processors.

Table 5.2 Memory usage

Solver	Preconditioner	2 blocs	4 blocs	8 blocs	16 blocs
ISIS-CFD	ILU(1)	405.6 Mb	384.8 Mb	374.4 Mb	374.4 Mb
PETSc	HyPre/ILU(1)	716.8 Mb	738.4 Mb	761.8 Mb	819.1 Mb
	HyPre/Multigrid	1,432.1 Mb	1,421.2 Mb	1,520.0 Mb	1,696.8 Mb
	Block Jacobi	532.6 Mb	545.3 Mb	569.5 Mb	617.0 Mb

From Table 5.2, the total memory of PETSc increases according to number of processors and behaves as

$$Total\_Memory(Mb) = \frac{4 * N\_cells * (a * N\_procs + b)}{10^6}$$

where a is the number of integer array of global size per processor, and b is the number of integer array of local size per processor, shown in Table 5.3. The total memory of ISIS-CFD is fairly constant.

Table 5.3 a and b constant

Solver	Preconditioner	a	b
PETSc	HyPre/ILU(1)	1.948	182.245
	HyPre/Multigrid	3.804	364.281
	Block Jacobi	1.597	135.112

5.6 Speed up

This test is to see the convergence of the linear solvers run with the medium grid with 2, 4, 8 and 16 processors. For the PETSc linear solver, the Krylov method BiCGStab is used. Figure 5.8-5.11 shows the speed up performance of the linear solvers in ISIS-CFD and PETSc with the preconditioner HyPre/ILU(1), HyPre/Multigrid and Block Jacobi, respectively. Table 5.4 shows number of iterations in the computations.

Table 5.4 Number of iteration of Speed up test

Solver	Preconditioner	2 blocs	4 blocs	8 blocs	16 blocs
ISIS-CFD	ILU(1)	141	131	133	122
PETSc	HyPre/Multigrid	7	9	8	34
	HyPre/ILU(1)	57	62	59	61
	Block Jacobi	93	92	99	93

According to the tests are run on the PC local station, no speed up is observed in the beginning from 8 blocs due to the hardware limitation of local PC network.

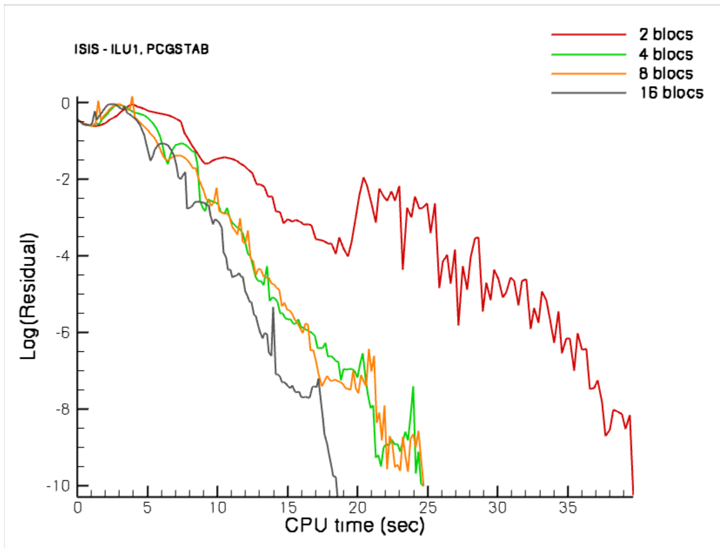

Figure 5.8 Convergence of ISIS-CFD run with 2, 4, 8 and 16 blocks

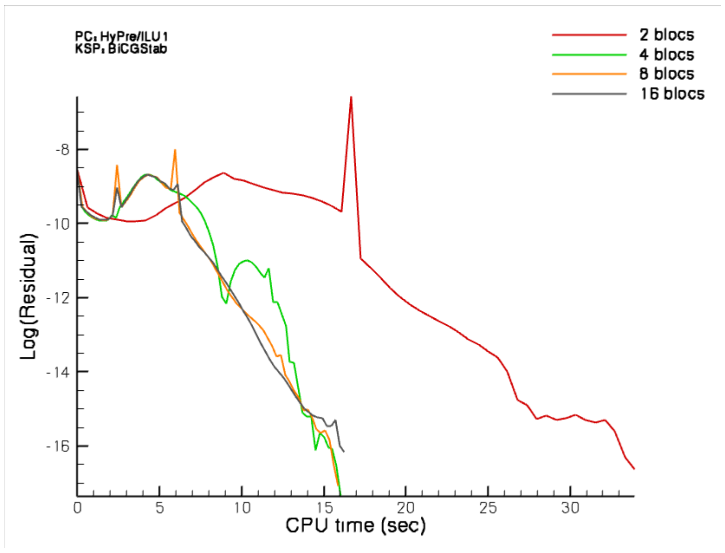

Figure 5.9 Convergence of PETSc with preconditioner HyPre/ILU(1) run with 2, 4, 8 and 16 blocks

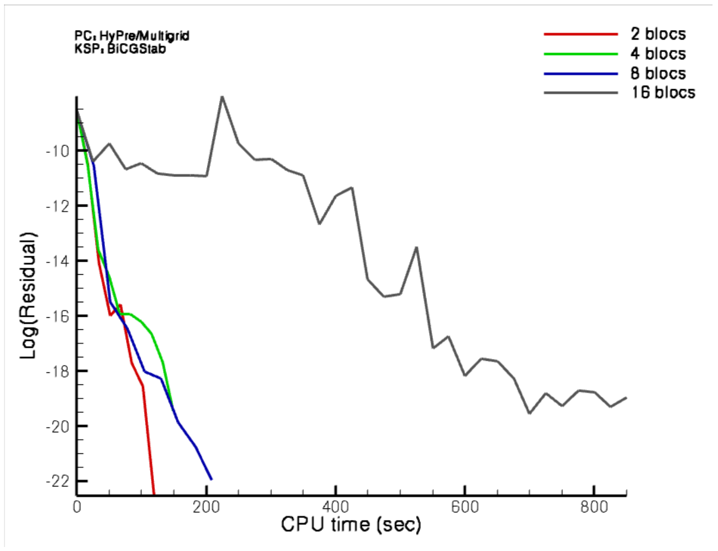

Figure 5.10 Convergence of PETSc with preconditioner HyPre/Multigrid run with 2, 4, 8 and 16 blocks

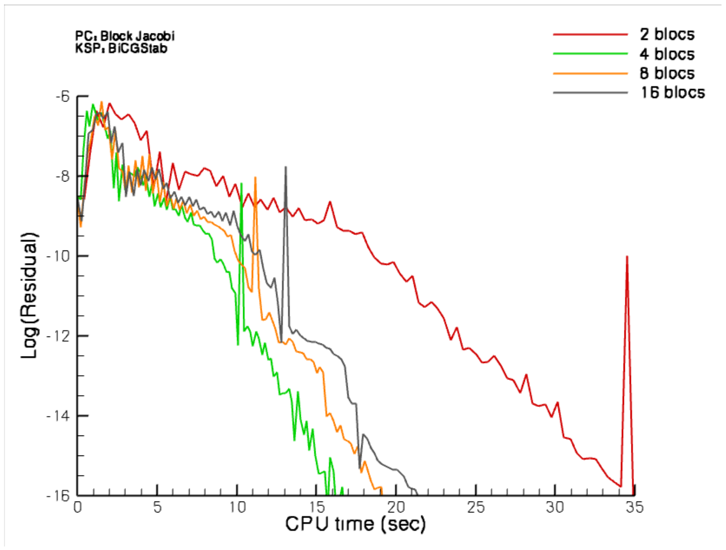

Figure 5.11 Convergence of PETSc with Preconditioner Block Jacobi run with 2, 4, 8 and 16 blocks

5.7 Convergence test

The objective of this test is to study the variation of number of iteration with respect to number of cells. The convergence test is run with 2 blocks with different grids. BiCGStab and Conjugate Gradient are used as the solver methods, and preconditioners in PETSc such as ILU(1) and Multigrid method from the external package HyPre and Block Jacobi are applied. Figure 5.12 – 5.18 illustrate convergences of linear solver in ISIS-CFD and PETSc.

- ISIS-CFD linear solver

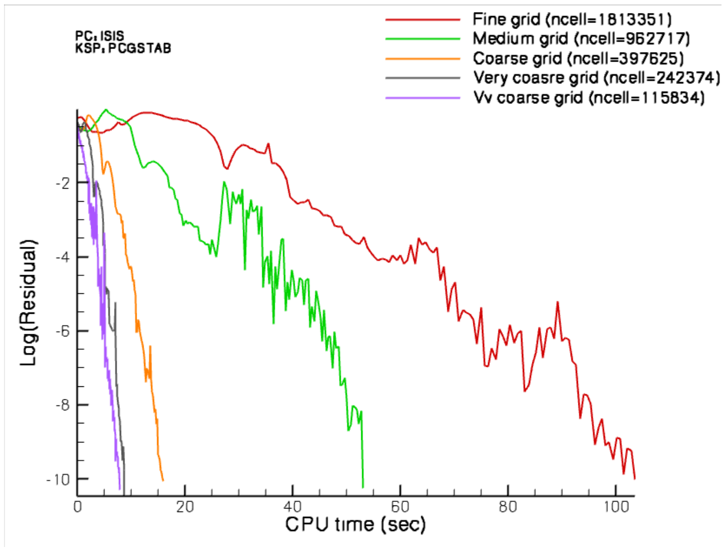

Figure 5.12 Convergence of ISIS-CFD run with different grids

- PETSc linear solver with preconditioner HyPre/ILU(1)

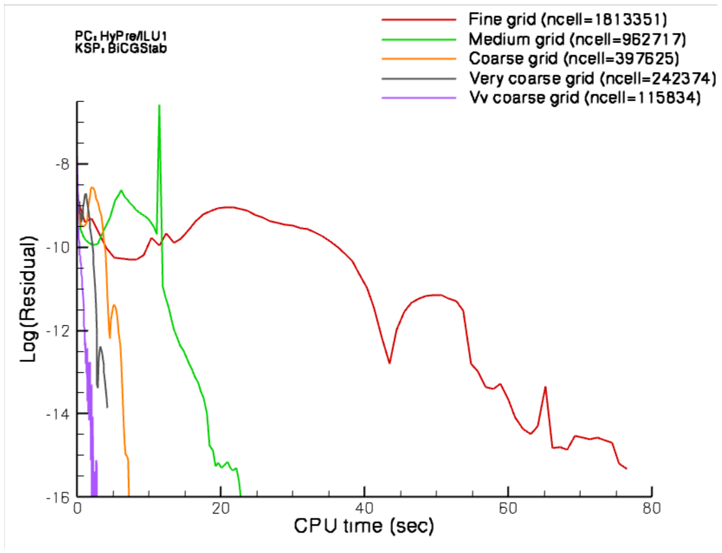

Figure 5.13 Convergence of PETSc- HyPre/ILU(1), BiCGStab run with different grids

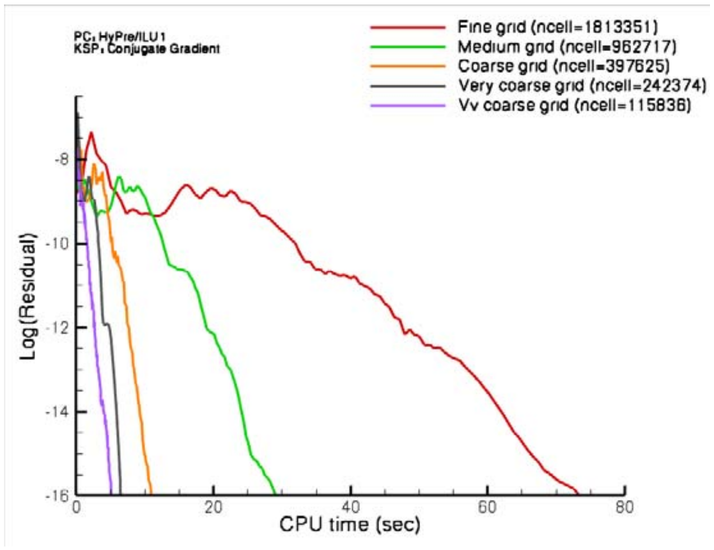

Figure 5.14 Convergence of PETSc- HyPre/ILU(1), Conjugate Gradient run with different grids

- PETSc linear solver with preconditioner HyPre/Multigrid method

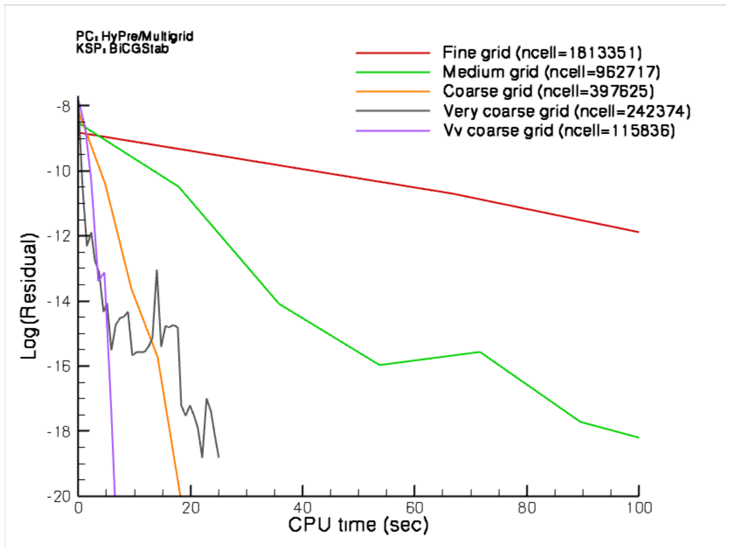

Figure 5.15 Convergence of PETSc- HyPre/Multigrid, BiCGStab run with different grids

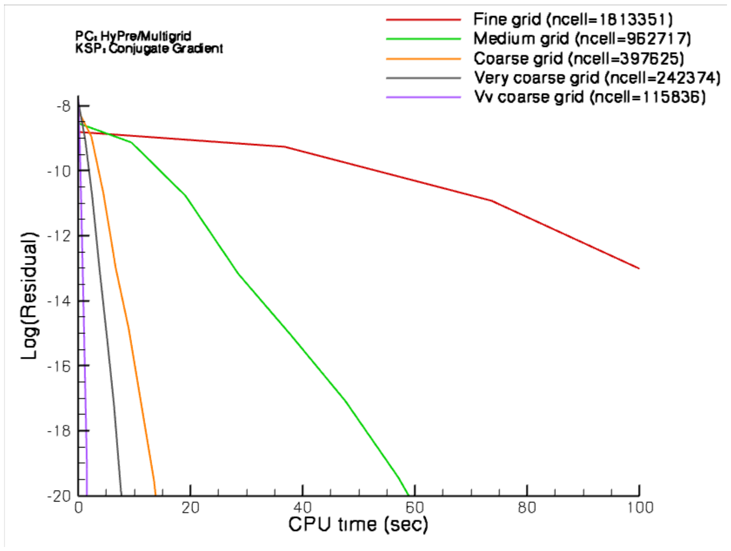

Figure 5.16 Convergence of PETSc- HyPre/Multigrid, Conjugate Gradient run with different grids

- PETSc linear solver with the preconditioner Block Jacobi

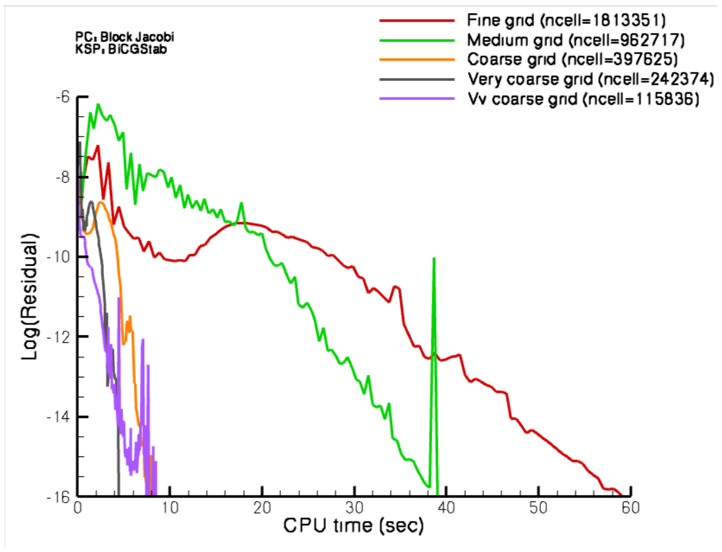

Figure 5.17 Convergence of PETSc- Block Jacobi, BiCGStab run with different grids

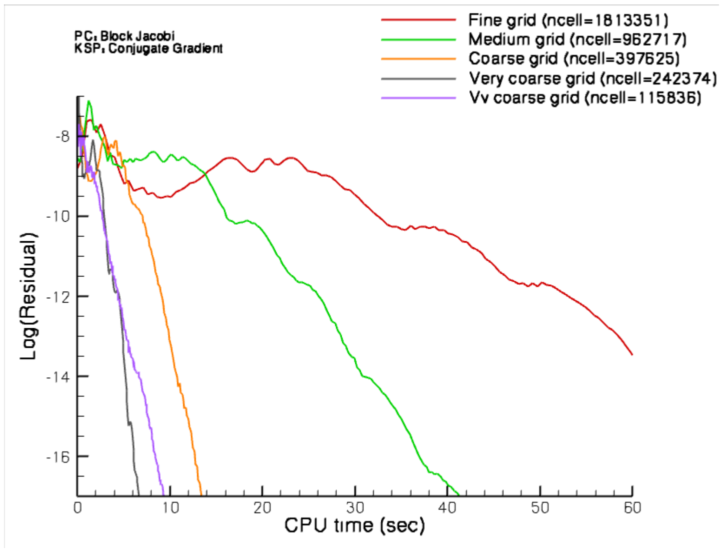

Figure 5.18 Convergence of PETSc- Block Jacobi, Conjugate Gradient run with different grids

Table 5.5 Number of iterations

	Preconditioner	Solver	Fine	Medium	Coarse	Very coarse	Vv coarse
ISIS-CFD	ILU(1)	PCGSTAB	153	141	93	82	149
PETSc	HyPre/ILU(1)	BiCGStab	74	57	46	43	88
	HyPre/ILU(1)	CG	140	102	83	72	199
	HyPre/MG	BiCGStab	9	7	10	34	7
	HyPre/MG	CG	11	12	10	9	252
	Block Jacobi	BiCGStab	115	93	71	61	140
	Block Jacobi	CG	206	170	121	104	212

Table 5.5 shows the number of iterations with respect to number of cells. Suppose that the Multigrid method has a constant number of iterations and the vv coarse grid case is not taken into account, comparison of ratios of number of iterations per cell are shown in Table 5.6. From the table, we can see that the higher of ratio of number of cells, the smaller change of number of iterations per cell. Every linear solver has fairly the same ratio for the same number of cells; 0.65 for Fine/Medium, 0.5 for Medium/Coarse and 0.7 for Coarse/Very coarse.

Table 5.6 Ratios of mumber of iterations per cells

	Preconditioner	Solver	Fine/Medium	Medium/Coarse	Coarse/Very c
ISIS-CFD	ILU(1)	PCGSTAB	5.76088E-01	6.26197E-01	6.91324E-01
PETSc	HyPre/ILU(1)	BiCGStab	6.89245E-01	5.11790E-01	6.52081E-01
	HyPre/ILU(1)	CG	7.28693E-01	5.07571E-01	7.02681E-01
	Block Jacobi	BiCGStab	6.56495E-01	5.41003E-01	7.09481E-01
	Block Jacobi	CG	6.43332E-01	5.80281E-01	7.09193E-01
Ratio of number of cells			1.883576378	2.421168186	1.640543128

CPU times used for reducing 5-order residual divided by number of cells are shown in Table 5.7, in which the fastest and the second fastest convergence are comment in red and blue, respectively. We can also see that, in general, the linear solvers in PETSc can reduce the residual faster than the linear solver in ISIS-CFD, except the case with the preconditioner HyPre/Multigrid for the fine grid and the case with the preconditioner Block Jacobi and the solver Conjugate Gradient for the vv coarse grid. There is an effect of the hardware limitation of PC local station in these tests. Multigrid Method in the external package HyPre with the solver BiCGStab performs best for the coarse grid and the very coarse grid cases with a very good performance for the medium grid case. A linear solver with the fastest convergence for the fine grid in this test is the case with the preconditioner Block Jacobi and the solver BiCGStab.

Table 5.7 CPU time for reducing 5-order residual divided by number of cells

	Preconditioner	Solver	Fine	Medium	Coarse	Very coarse	Vv coarse
ISIS-CFD	ILU(1)	PCGSTAB	3.83E-05	4.49E-05	2.72E-05	2.43E-05	3.80E-05
PETSc	HyPre/ILU(1)	BiCGStab	3.36E-05	1.97E-05	1.61E-05	1.73E-05	1.55E-05
	HyPre/ILU(1)	CG	3.31E-05	2.43E-05	2.11E-05	2.22E-05	2.76E-05
	HyPre/MG	BiCGStab	4.59E-05	2.17E-05	1.28E-05	2.89E-06	2.07E-05
	HyPre/MG	CG	6.07E-05	3.12E-05	1.73E-05	1.53E-05	6.47E-06
	Block Jacobi	BiCGStab	2.58E-05	3.32E-05	1.53E-05	1.61E-05	3.11E-05
	Block Jacobi	CG	3.42E-05	3.00E-05	2.49E-05	2.02E-05	4.58E-05

6. CONCLUSION AND FUTURE WORK

In this research, PETSc linear solver has in general better convergence than the linear solver in ISIS-CFD, especially, the Multigrid method in the external package HyPre solved with BiCGStab perform very well in most of grid sizes. Its number of iterations is fairly constant, when the number of cells increases as well as when the number of processors increases.

PETSc linear solver package contains scalable parallel preconditioners such as the Multigrid method in the external package HyPre and Block Jacobi. However, the PETSc linear solvers increase consuming memory when the number of processors increases, while the linear solver in ISIS-CFD remains fairly the constant memory usage.

The hardware limitation of PC local network affects the tests and their results such as in the convergence test, the performance of Multigrid method for the fine grid. As well as in the speed up test, no speed up beginning from 8 blocs is observed.

In the future work, I would like to recommend studying:

- The Multigrid method in the external package HyPre as it performs very well in different grid sizes and scalable. Its various options should be studied to develop its performance.
- `MatCreateMPIAIJWithSplitArrays()` can update the matrix very fast and should be enabled to work with the linear solvers in the external package HyPre.
- Smoothness is also a desirable property for the computation and should be focused

APPENDIX A   The PETSc program

```
* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*     The program solves the linear system by using PETSc linear solver
*     Toolkids combining with MPI
* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      program test_with_petsc
      Include "precision.h"

* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*                    Include files
* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
!  petsc.h  - base PETSc routines  petscvec.h - vectors
!  petscmat.h - matrices           petscksp.h - Krylov subspace methods
!  petscpc.h  - preconditioners

#include "finclude/petsc.h"
#include "finclude/petscvec.h"
#include "finclude/petscmat.h"
#include "finclude/petscpc.h"
#include "finclude/petscsys.h"
#include "finclude/petscksp.h"

* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*                   Variable declarations
* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
!  Variables:
!     pc     - preconditioner context
!     ksp    - Krylov subspace method context, linear solver context
!     D      - matrix that defines linear system
!     x,u,b,rhs    - exact and approx sol, computed and given RHS vectors
!     errRHSmax    - maximum error of the right-hand-side
!     me,nproc     - The processor I am and the total number of processor
!     IpntCF_CC,IndCon_CC – local indices of 'a' where to begin a new row

      PC       pc
      KSP      ksp
      Mat      D
      Vec      u,rhs,x,b
      PetscInt N,NN,i,j,globalIndRow,globalIndCol,its,maxits,dummy,
     $        Istart,Iend,ii,jj,kk,ll,ojj,okk,kkA,kkB,kkN,pntBegin,
     $        pntEnd,yy,nrow,ncolumn,lg,matopt,pcopt,nz
      PetscErrorCode ierr
      PetscMPIInt    rank
      PetscScalar    neg_one
      PetscReal      errRHSmax,mem
      PetscTruth     flg0,flg1,oflg1,oflg2
      KSPType kspt
      PCType pct

      common /pvmmb/me,nproc
      Common/parallele/mybloc
      common /com/mytid,itids(1000)
      COMMON/STMPI /bloc
      character*4 bloc

      Common /umesg/ imesg
      CHARACTER*150 fname
      character*5   iluk

        integer,dimension (:), allocatable:: IpntCF_CC,IndCon_CC,
     $     nfcom,nblcom,ncell_local,Local_to_Global_Mapping,LoToGlo,
     $     column,pointer,ocolumn,opointer
        integer,dimension (:,:), allocatable:: Ind_send,Ind_Receive
        double precision,dimension (:), allocatable::
     $       a,Src,Sol,p,v,ov

!  Note: Any user-defined Fortran routines (such as MyKSPMonitor)
!  MUST be declared as external.

      external MyKSPMonitor,MyKSPConverged

      ! Timing variables
      Integer, Parameter :: iprec_single=selected_real_kind(4)
      Integer, Parameter :: iprec_double=selected_real_kind(8)

      Real(iprec_single) :: time
      Integer :: itime_start, itime_end, itime_rate, time_max
*-----------------------------------------------------------------------*
* Pré-Initialisations par défaut
*     o Langue, etc ...
      me=0
      nproc=1
      imesg=6

* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*     Choose the defining matrix, preconditioner and solver type
* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
***   Select Matrix creation case
!     [1] MatCreate() and MatSetValues() by row
!     [2] MatCreateMPIAIJWithSplitArrays()
      matopt=2

***   Preconditioner options
!     [0] PETSc Preconditioners; bjacobi,mg,asm
!     [1] Additive Schwarz Method
!     [2] HYPRE/ILU(K)
!     [3] HYPRE/Multigrid
!     [4] TRILLINOS/Multigrid
      pcopt=3
!     if 0 please enter type
      pct=PCBJACOBI
!     If 1,2 please enter the ilu level
      iluk='1'

***   KSP type
      kspt=KSPCG


* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*                     Begin the program
* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
      ! >>> Start timing 1
      Call SYSTEM_CLOCK(COUNT=itime_start, COUNT_RATE=itime_rate,
     $                  COUNT_MAX=time_max)

      call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
      call initmb1
      mybloc=me

* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*     Extract the values of pressure equation and indices, then define
*     the number of local nodes and global nodes
* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

*     Open files and get the values or send indices and receive indices
      if (me.gt.1) imesg=100+me
      if (nproc.eq.1) then
         open(10,file='pressure_equation.bin',status='unknown',
     $        form='unformatted')
      else
         write(bloc,'(A,I3.3)') 'b',me
         open(10,file=bloc//'/pressure_equation.bin',status='unknown',
     $        form='unformatted')
         open(11,file=bloc//'/index_send_receive.dat',status='unknown')
         read(11,*) nbloc,NFCOMMAX
         allocate(nfcom(nbloc),nblcom(nbloc),Ind_Send(NFCOMMAX,nbloc))
         allocate(Ind_Receive(NFCOMMAX,nbloc))
         do ibloc=1,nbloc       ! boucle sur les blocs de communication
            read(11,*) nfcom(ibloc),nblcom(ibloc)
            nbloc_nb=nblcom(ibloc) ! numero de bloc a communiquer
            if (nbloc_nb.ne.mybloc) then
               do iface=1,nfcom(ibloc)
                  read(11,*) Ind_Send(iface,ibloc),
     $                 Ind_Receive(iface,ibloc)
               end do
            else
               write(0,*) 'Unexpected communication'
               call killallmpi
            end if
         end do
         close(11)
      end if

        read(10) ncellule,nvariable
        write(imesg,*) 'Number of cells    : ',ncellule
        write(imesg,*) 'Number of variables: ',nvariable

*       Import the connectivity
        allocate(IpntCF_CC(ncellule+1))
        call read_bin_int(ncellule+1,IpntCF_CC)
        ndim_mat=IpntCF_CC(ncellule+1)-1
        allocate(IndCon_CC(ndim_mat),a(ndim_mat))
        call read_bin_int(ndim_mat,IndCon_CC)
*       Import the matrix
        call read_bin_float(ndim_mat,a)
        allocate(Src(nvariable),Sol(nvariable))
*       Import the right hand side
        call read_bin_float(nvariable,Src)
*       Import the solution
```

```
          call read_bin_float(nvariable,Sol)
        close(10)

*       Compute the total number of cells
        ncell_total=ncellule
        call glbsum_int(ncell_total)
        write(imesg,*) 'Total number of cells: ',ncell_total

        N=ncellule
        NN=ncell_total

*       Gather the local number of each bloc
        allocate(ncell_local(nproc))
        ncell_local(1)=ncellule
        call  gather_int(nproc,1,ncell_local,nleng)
        write(imesg,*) 'Local number of cell for each bloc:'
        do i=1,nproc
           write(imesg,*) i,ncell_local(i)
        end do

* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*               Establish local to global index mapping
* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

        allocate(Local_to_Global_Mapping(Nvariable))
        index0=0
        if (me.ne.1) then
           do i=1,me-1
              index0=index0+ncell_local(i)
           end do
        end if
        Local_to_Global_Mapping=0
        do i=1,ncellule
           Local_to_Global_Mapping(i)=index0+i-1
        end do
        call communicationint1(Local_to_Global_Mapping,
       $     mybloc,nbloc,nfcom,nblcom,
       &     Ind_Send,Ind_Receive,NFCOMMAX)


* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*                     Create vectors and matrix
* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

*       Vector: Creatte a parallel vector and duplicate it.
!       Create a parallel vector.
!        - In this case, we specify the size of each processor's local
!          portion, and PETSc computes the global size.  Alternatively,
!          if we pass the global size and use PETSC_DECIDE for the
!          local size PETSc will choose a reasonable partition trying
!          to put nearly an equal number of elements on each processor.
!       rhs - the given right hand side

        call VecCreateMPI(PETSC_COMM_WORLD,N,PETSC_DECIDE,rhs,ierr)
        call VecDuplicate(rhs,u,ierr) ! u - the approximated solution
        call VecDuplicate(rhs,b,ierr) ! b - the computed RHS
        call VecDuplicate(rhs,x,ierr)! x - the exact solution
*
```

```
*     LoToGlo - the global column indices of the array a
*
      allocate(LoToGlo(ndim_mat))
      LoToGlo=Local_to_Global_Mapping(IndCon_CC)

* The matrix creation case must be select at the top of the program
      select case (matopt)

*     [1] MatCreate() and MatSetValues() by row
      case (1)
      nz=0
      do i=IpntCF_CC(1),IpntCF_CC(2)-1
         if (a(i).ne.0.0) nz=nz+1
      enddo
      write(imesg,*) 'nz:',nz

      call MatCreate(PETSC_COMM_WORLD,D,ierr)
      call MatSetSizes(D,N,N,PETSC_DETERMINE,PETSC_DETERMINE,
     $     ierr)
      call MatSetType(D,MATMPIAIJ,ierr) ! to set type a parallel matrix
      call MatMPIAIJSetPreallocation(D,nz,PETSC_NULL_INTEGER,
     $     2,PETSC_NULL_INTEGER,ierr)
      call MatZeroEntries(D,ierr)

      write(imesg,*) '[1] MatCreate() and MatSetValues() by row'
      do i=1,N
          globalIndRow=Local_to_Global_Mapping(i)
          pntBegin=IpntCF_CC(i)
          pntEnd=IpntCF_CC(i+1)-1
          j=pntEnd-pntBegin+1
      call MatSetValues(D,1,globalIndRow,j,LoToGlo(pntBegin:pntEnd),
     $                 a(pntBegin:pntEnd),ADD_VALUES,ierr)
      end do


*     [2] MatCreateMPIAIJWithSplitArrays()
      case (2)

      Istart=index0 ! Istart - the start global column index of that bloc
      Iend=index0+N ! Iend - the end global column index of that bloc
      write(imesg,*) 'Istart:',Istart,'Iend:',Iend

      allocate(v(ndim_mat))      ! v - diagonal values
      allocate(column(ndim_mat)) ! column - diagonal column indices
      allocate(pointer(N+1))     ! pointer - diag row indices into column
      allocate(ov(ndim_mat))     ! ov - off-diagonal values
      allocate(ocolumn(ndim_mat))! ocolumn - off-diagonal column indices
      allocate(opointer(N+1))    ! opointer - off-diag ind into ocolumn

      v=0
      column=0
      pointer=0
      ov=0
      ocolumn=Iend
      if (Iend.eq.NN) ocolumn=Istart-1
      opointer=NN

      jj=1  ! index of v and column
```

```
kk=1  ! index of pointer
ojj=1 ! index of ov
okk=1 ! index of opinter

oflg2=PETSC_TRUE

 do ii=1,(ndim_mat+1) !index of array a and LoToGlo

 flg0=(abs(a(ii)-0).gt.(1.e-20))
 flg1=((LoToGlo(ii).ge.(Istart)).and.(LoToGlo(ii).lt.(Iend)))
 oflg1=((LoToGlo(ii).ge.0).and.(LoToGlo(ii).lt.NN))

 if (flg0) then! if nonzero

 if (flg1) then
     v(jj)=a(ii) ! put matrix value to diagonal array
     column(jj)=LoToGlo(ii)
    if (ii.ge.(IpntCF_CC(kk))) then ! if the index starts new row
       pointer(kk)=jj-1 ! PETSc is zero-based; FORTRAN is one-based
       kk=kk+1         ! index of new row pointer array
    end if
    jj=jj+1  ! move column index address and value to the next one
    yy=IpntCF_CC(kk)-1
  elseif (oflg1) then
     ov(ojj)=a(ii) !  put matrix value to off-diagonal array
     ocolumn(ojj)=LoToGlo(ii)
     if (ii.ge.(IpntCF_CC(okk))) then ! if the index starts new row
        opointer(okk)=ojj-1 ! PETSc:zero-based; FORTRAN:one-based
        okk=okk+1
     endif
     ojj=ojj+1
     oflg2=PETSC_FALSE
endif

endif

if (ii.eq.yy) then
  if (oflg2) then
     opointer(okk)=ojj-1
     ojj=ojj+1
     okk=okk+1
  else
      oflg2=PETSC_TRUE
   endif
endif
end do

pointer(kk)=jj-1 ! Last row pointer may missing
opointer(okk)=ojj-1 ! Last off-diag row pointer may missing

write(imesg,*) '[2] MatCreateMPIAIJWithSplitArrays()'

column=column-Istart ! set global column indices to local indices
do kk=1,N
   kkA=pointer(kk)+1
   kkB=pointer(kk+1)
   kkN=kkB-kkA+1
   call isort(column(kkA:kkB),v(kkA:kkB),kkN,2)
```

```
      end do

      ! Create the matrix
      call MatCreateMPIAIJWithSplitArrays(PETSC_COMM_WORLD,N,N,
     $     PETSC_DETERMINE,PETSC_DETERMINE,pointer,column,v,
     $     opointer,ocolumn,ov,D,ierr)

      end select
*
*     Assemble the matrix
*
      call MatAssemblyBegin(D,MAT_FINAL_ASSEMBLY,ierr) ! Assemble it
      call MatAssemblyEnd(D,MAT_FINAL_ASSEMBLY,ierr)

* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*                   Set values to vectors
* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

*     Set Value to the exact solution vector and RHS
!     Set the vector elements.
!      - Always specify global locations of vector entries.
!      - Each processor can contribute any vector entries,
!        regardless of which processor "owns" them; any nonlocal
!        contributions will be transferred to the appropriate processor
!        during the assembly process.
!      - In this example, the flag INSERT_VALUES indicates that all
!        contributions will be inserted and delete the old value.

      call VecSetValues(x,N,Local_to_Global_Mapping,
     $     Sol,INSERT_VALUES,ierr) ! Set the exact solution vector

!     Assemble vector, using the 2-step process:
!       VecAssemblyBegin(), VecAssemblyEnd()
!     Computations can be done while messages are in transition
!     by placing code between these two statements.
      call VecAssemblyBegin(x,ierr)
      call VecAssemblyEnd(x,ierr)

!     Set values for the right hand side vector
      call VecSetValues(rhs,N,Local_to_Global_Mapping,
     $     Src,INSERT_VALUES,ierr)
      call VecAssemblyBegin(rhs,ierr)
      call VecAssemblyEnd(rhs,ierr)
      write(imesg,*) 'The vector value is set and assembled.'


      ! <<< Stop timing 2
      Call SYSTEM_CLOCK(itime_end)

      ! The elapsed time in seconds 2
      time=REAL(itime_end - itime_start)/REAL(itime_rate)
      Print *, 'Elapsed time in seconds, Vec & Mat, proc',me,':',time

*     Check if the matrix has been defined correctly
      neg_one=-1.0
      call MatMult(D,x,b,ierr)
      call VecAXPY(b,neg_one,rhs,ierr)
      call VecAbs(b,ierr)
      call VecMax(b,i,errRHSmax,ierr)
      write(imesg,*) 'rhs-b =',errRHSmax

* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*          Create the linear solver and set various options
* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      ! >>> Start timing 3
      Call SYSTEM_CLOCK(COUNT=itime_start, COUNT_RATE=itime_rate,
     $                  COUNT_MAX=time_max)

*     Create linear solver context
      call KSPCreate(PETSC_COMM_WORLD,ksp,ierr)

*     Set operators. Here the matrix that defines the linear system
!     also serves as the preconditioning matrix.  Here are matrix D.
      call KSPSetOperators(ksp,D,D,DIFFERENT_NONZERO_PATTERN,ierr)

*     Returns a pointer to the preconditioner context
      call KSPGetPC(ksp,pc,ierr)

*     Preconditioner options can be selected at the top of the program
      select case (pcopt)

      case (0)
* 0   Preconditioner of PETSc which can be used for parallel computing
*     without external package: Block Jacobi, Additive Schwarz
      call PCSetType(pc,pct,ierr)

      case (1)
* 1  Preconditioner: Additive Schwarz Method
!      By default: subdomain=1, overlab=1, type=restrict, level=0
!      ilu - if want to use icc, set the matrix is symmetric.
!      Use in place is to destroy the matrix after use to save memory
      call PCSetType(pc,PCASM,ierr)
      call PCASMSetUseInPlace(pc,ierr)
      call PetscOptionsSetValue('-sub_pc_factor_levels',iluk,ierr)
      call PetscOptionsSetValue('-sub_pc_factor_shift_positive_definite'
     $     ,PETSC_NULL_CHARACTER,ierr) ! to avoid zero pivot

      case (2,3)
* 2,3 Preconditioner: HYPRE
      call PCSetType(pc,PCHYPRE,ierr)

      if (pcopt.eq.2) then
* 2   Euclid for ILU(k)
      call PCHYPRESetType(pc,'euclid',ierr)
      call PetscOptionsSetValue('-pc_hypre_euclid_levels',iluk,ierr)
      call PetscOptionsSetValue('-pc_hypre_euclid_bj','TRUE',ierr)

      else
* 3   BoomerAMG for Multigrid
      call PCHYPRESetType(pc,'boomeramg',ierr)
      call PetscOptionsSetValue('-pc_hypre_boomeramg_max_levels',
     $     '10',ierr)

      endif

      case (4)
```

```
* 2   Preconditioner: ML
      call PCSetType(pc,PCML,ierr)
      call PetscOptionsSetValue('-pc_ml_maxNlevels','5',ierr)
      call PetscOptionsSetValue(
     $    '-mg_coarse_redundant_pc_factor_zeropivot',
     $    '1e-25',ierr)

      end select
* End Preconditioner options-------------------------------------------

*     Set the relative,absolute,divergence,and maximum iteration
*       tolerances
      tol = 1.e-7
      maxits = 1000
      call KSPSetTolerances(ksp,tol,PETSC_DEFAULT_DOUBLE_PRECISION,     &
     &     PETSC_DEFAULT_DOUBLE_PRECISION,maxits,ierr)

*     Set user-defined monitoring routine if desired
      call PetscOptionsHasName(PETSC_NULL_CHARACTER,'-my_ksp_monitor',  &
     &                  flg,ierr)
      if (flg .eq. 1) then
         call KSPMonitorSet(ksp,MyKSPMonitor,PETSC_NULL_OBJECT,
&
     &                    PETSC_NULL_FUNCTION,ierr)
      Endif

*     To enable the ksp monitoring and write in a file
      call PetscOptionsSetValue('-ksp_monitor_true_residual',
     $     'monitor.dat',ierr)


*--------------Set KSP solver type-------------------
      call KSPSetType(ksp,kspt,ierr)
      call KSPSetFromOptions(ksp,ierr)

*----------------------------------------------------

*     Set convergence test routine if desired
      call PetscOptionsHasName(PETSC_NULL_CHARACTER,                    &
     &     '-my_ksp_convergence',flg,ierr)
      if (flg .eq. 1) then
         call KSPSetConvergenceTest(ksp,MyKSPConverged,
&
     &          PETSC_NULL_OBJECT,ierr)
      endif

* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*    Solve the linear system and see the computing time and view KSP
* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -


*-----------Solve the linear system-----------------

      call KSPSolve(ksp,rhs,u,ierr)

*---------------------------------------------------

      ! <<< Stop timing 3
      Call SYSTEM_CLOCK(itime_end)

      ! The elapsed time in seconds 3
      time=REAL(itime_end - itime_start)/REAL(itime_rate)
      Print *, 'Elapsed time in seconds, PC & KSP, proc',me,':',time

*     View the information of solver, preconditioner and matrix
      call KSPView(ksp,PETSC_VIEWER_STDOUT_WORLD,ierr)

* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*                         check the error
* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

*     Transfer the values from vector u, N elements, to array p.
!     Local_to_Global_Mapping is the global location to get the values.
      allocate(p(N))
      call VecGetValues(u,N,Local_to_Global_Mapping,p,ierr)

*     Check errors of the approximated solution
      err0=-1.0e+8
      err1=p(1)-sol(1)
      do i=1,ncellule
         err0=max(err0,abs(p(i)-sol(i)-err1))
      end do
      write(imesg,*) 'Maximum error = ',err0
      write(imesg,*) 'Difference at node 1 = ',err1

*     To get the iterations number used for computing
      call KSPGetIterationNumber(ksp,its,ierr)
      write(imesg,*) 'Iterations =', its

* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*              Clean up and exit the programFree work space.
* All PETSc objects should be destroyed when they are no longer needed.
* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      call VecDestroy(x,ierr)
      call VecDestroy(b,ierr)
      call VecDestroy(u,ierr)
      call VecDestroy(rhs,ierr)
      call MatDestroy(D,ierr)
      call KSPDestroy(ksp,ierr)

* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
*                       End the program.
*        Always call PetscFinalize() before exiting a program
* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      call PetscFinalize(ierr)
      write(imesg,*) 'Normal end'
      end

* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
!  MyKSPMonitor - This is a user-defined routine for monitoring
!  the KSP iterative solvers.
!
!  Input Parameters:
```

```
!     ksp   - iterative context
!     n     - iteration number
!     rnorm - 2-norm (preconditioned) residual value (may be estimated)
!     dummy - optional user-defined monitor context (unused here)
!
      subroutine MyKSPMonitor(ksp,n,rnorm,dummy,ierr)
      implicit none

#include 'finclude/petsc.h'
#include 'finclude/petscvec.h'
#include 'finclude/petscksp.h'

       KSP              ksp
       Vec              x
       PetscErrorCode ierr
       PetscInt n,dummy
       PetscMPIInt rank
       double precision rnorm

*  Build the solution vector

      call KSPBuildSolution(ksp,PETSC_NULL_OBJECT,x,ierr)

*  Write the solution vector and residual norm to stdout
!   - Note that the parallel viewer PETSC_VIEWER_STDOUT_WORLD
!     handles data from multiple processors so that the
!     output is not jumbled.

      call MPI_COMM_RANK(PETSC_COMM_WORLD,rank,ierr)
      if (rank .eq. 0) write(6,100) n
      call VecView(x,PETSC_VIEWER_STDOUT_WORLD,ierr)
      if (rank .eq. 0) write(6,200) n,rnorm

 100  format('iteration ',i5,' solution vector:')
 200  format('iteration ',i5,' residual norm ',e10.4)
      ierr = 0
      end

* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
!  MyKSPConverged - This is a user-defined routine for testing
!  convergence of the KSP iterative solvers.
!
!  Input Parameters:
!    ksp   - iterative context
!    n     - iteration number
!    rnorm - 2-norm (preconditioned) residual value (may be estimated)
!    dummy - optional user-defined monitor context (unused here)
!
      subroutine MyKSPConverged(ksp,n,rnorm,flag,dummy,ierr)

      implicit none

#include "finclude/petsc.h"
#include "finclude/petscvec.h"
#include "finclude/petscksp.h"

      KSP              ksp
```

```
PetscErrorCode ierr
PetscInt n,dummy
KSPConvergedReason flag
double precision rnorm

if (rnorm .le. .05) then
  flag = 1
else
  flag = 0
endif
ierr = 0

end
```

APPENDIX B A result file

```
 Number of cells    :       198812
 Number of variables:       220903
 Total number of cells:       397625
 Local number of cell for each bloc:
           1      198812
           2      198813
 nz:           7
 [2] MatCreate() and MatSetValues() by row
 The vector value is set and assembled.
 Elapsed time in seconds, Vec & Mat, proc       1 :   144.3628
 Elapsed time in seconds, Vec & Mat, proc       2 :   144.2728
 rhs-b =  3.774324602856538E-015
 Elapsed time in seconds, PC & KSP, proc        1 :   18.80410
KSP Object:
  type: bcgs
  maximum iterations=1000, initial guess is zero
  tolerances:  relative=1e-07, absolute=1e-50, divergence=10000
  left preconditioning
PC Object:
  type: hypre
    HYPRE BoomerAMG preconditioning
    HYPRE BoomerAMG: Cycle type V
    HYPRE BoomerAMG: Maximum number of levels 10
    HYPRE BoomerAMG: Maximum number of iterations PER hypre call 1
    HYPRE BoomerAMG: Convergence tolerance PER hypre call 0
    HYPRE BoomerAMG: Threshold for strong coupling 0.25
    HYPRE BoomerAMG: Interpolation truncation factor 0
    HYPRE BoomerAMG: Interpolation: max elements per row 0
    HYPRE BoomerAMG: Number of levels of aggressive coarsening 0
    HYPRE BoomerAMG: Number of paths for aggressive coarsening 1
    HYPRE BoomerAMG: Maximum row sums 0.9
    HYPRE BoomerAMG: Sweeps down         1
    HYPRE BoomerAMG: Sweeps up           1
    HYPRE BoomerAMG: Sweeps on coarse    1
    HYPRE BoomerAMG: Relax down          symmetric-SOR/Jacobi
    HYPRE BoomerAMG: Relax up            symmetric-SOR/Jacobi
    HYPRE BoomerAMG: Relax on coarse     Gaussian-elimination
    HYPRE BoomerAMG: Relax weight  (all)      1
    HYPRE BoomerAMG: Outer relax weight (all) 1
    HYPRE BoomerAMG: Using CF-relaxation
    HYPRE BoomerAMG: Measure type        local
    HYPRE BoomerAMG: Coarsen type        Falgout
    HYPRE BoomerAMG: Interpolation type  classical
  linear system matrix = precond matrix:
  Matrix Object:
    type=mpiaij, rows=397625, cols=397625
    total: nonzeros=2811329, allocated nonzeros=4261494
      not using I-node (on process 0) routines
 Elapsed time in seconds, PC & KSP, proc        2 :   18.80070
 Maximum error =   0.159961173834830
 Difference at node 1 =   -26.9312163083231
 Iterations =           4
Summary of Memory Usage in PETSc
[0]Current space PetscMalloc()ed 26708, max space PetscMalloced()
5.19728e+07
```

```
[0]Current process memory 3.92315e+07 max process memory 2.5618e+08
[1]Current space PetscMalloc()ed 26708, max space PetscMalloced()
6.46507e+07
[1]Current process memory 3.90021e+07 max process memory 2.82411e+08
 Normal end
```

APPENDIX C Preallocation of Memory for Parallel AIJ Sparse Matrices

```
      call MatMPIAIJSetPreallocation(Mat A,PetscInt d_nz,
     $       const PetscInt d_nnz[],PetscInt o_nz,
     $       const PetscInt o_nnz[],ierr)
```

Input parameter:-

`A`	- the matrix.
`d_nz`	- number of nonzeros per row in DIAGONAL portion of local submatrix (same value is used for all local rows)
`d_nnz`	- array containing the number of nonzeros in the various rows of the DIAGONAL portion of the local submatrix (possibly different for each row) or `PETSC_NULL_INTEGER`, if `d_nz` is used to specify the nonzero structure. The size of this array is equal to the number of local rows. One must leave room for the diagonal entry even if it is zero.
`o_nz`	- number of nonzeros per row in the OFF-DIAGONAL portion of local submatrix (same value is used for all local rows).
`o_nnz`	- array containing the number of nonzeros in the various rows of the OFF-DIAGONAL portion of the local submatrix (possibly different for each row) or `PETSC_NULL_INTEGER`, if `o_nz` is used to specify the nonzero structure. The size of this array is equal to the number of local rows.

Preallocation of memory is critical for achieving good performance during matrix assembly, as this reduces the number of allocations and copies required. We present an example for three processes to indicate how this may be done for the `MATMPIAIJ` matrix format. Consider the 8 by 8 matrix, which is partitioned by default with three rows on the first process, three on the second and two on the third.

$$\left(\begin{array}{ccc|ccc|cc}
1 & 2 & 0 & 0 & 3 & 0 & 0 & 4 \\
0 & 5 & 6 & 7 & 0 & 0 & 8 & 0 \\
9 & 0 & 10 & 11 & 0 & 0 & 12 & 0 \\
\hline
13 & 0 & 14 & 15 & 16 & 17 & 0 & 0 \\
0 & 18 & 0 & 19 & 20 & 21 & 0 & 0 \\
0 & 0 & 0 & 22 & 23 & 0 & 24 & 0 \\
\hline
25 & 26 & 27 & 0 & 0 & 28 & 29 & 0 \\
30 & 0 & 0 & 31 & 32 & 33 & 0 & 34
\end{array}\right)$$

The “diagonal” submatrix, `d`, on the first process is given by

$$\begin{pmatrix} 1 & 2 & 0 \\ 0 & 5 & 6 \\ 9 & 0 & 10 \end{pmatrix}$$

while the “off-diagonal” submatrix, `o`, matrix is given by

$$\begin{pmatrix} 0 & 3 & 0 & 0 & 4 \\ 7 & 0 & 0 & 8 & 0 \\ 11 & 0 & 0 & 12 & 0 \end{pmatrix}$$

For the first process one could set `d_nz` to 2 (since each row has 2 nonzeros) or, alternatively, set `d_nnz` to {2,2,2}. The `o_nz` could be set to 2 since each row of the `o` matrix has 2 nonzeros, or `o_nnz` could be set to {2,2,2}.

For the second process the `d` submatrix is given by

$$\begin{pmatrix} 15 & 16 & 17 \\ 19 & 20 & 21 \\ 22 & 23 & 0 \end{pmatrix}$$

Thus, one could set `d_nz` to 3, since the maximum number of nonzeros in each row is 3, or alternatively, one could set `d_nnz` to {3,3,2}, thereby indicating that the first two rows will have 3 nonzeros while the third has 2. The corresponding `o` submatrix for the second process is

$$\begin{pmatrix} 13 & 0 & 14 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 \\ 0 & 0 & 0 & 24 & 0 \end{pmatrix}$$

so that one could set `o_nz` to 2 or `o_nnz` to {2,1,1}.

Note that the user never directly works with the `d` and `o` submatrices, except when preallocating storage space as indicated above. Also, the user need not preallocate exactly the correct amount of space; as long as a sufficiently close estimate is given, the high efficiency for matrix assembly will remain.

The option `-info` will print information about the success of preallocation during matrix assembly. For the `MATMPIAIJ` and `MATMPIBAIJ` formats, PETSc will also list the number of elements owned by on each process that were generated on a different process. For example, the statements

```
MatAssemblyBegin MPIAIJ:Stash has 10 entries, uses 0 mallocs
MatAssemblyBegin MPIAIJ:Stash has  3 entries, uses 0 mallocs
```

indicate that very few values have been generated on different processes. On the other hand, the statements

```
MatAssemblyBegin MPIAIJ:Stash has 100000 entries,
uses 100 mallocs
MatAssemblyBegin MPIAIJ:Stash has  77777 entries
```

indicate that many values have been generated on the "wrong" processes. This situation can be very inefficient, since the transfer of values to the "correct" process is generally expensive. By using the command `MatGetOwnershipRange()` in application codes, the user should be able to generate most entries on the owning process.

Note: It is fine to generate some entries on the "wrong" process. Often this can lead to cleaner, simpler, less buggy codes. One should never make code overly complicated in order to generate all values locally. Rather, one should organize the code in such a way that most values are generated locally.

REFERENCES

Yousef Saad. *Iterative Methods for Sparse Linear Systems*. 2nd edition. N.p.: January 2000.

S. Balay, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. Curfman McInnes, B. Smith, and H. Zhang. *PETSc Users Manual Revision 3.0.0.* December 2008.