Relatório de estudos individual - 2017

Pétterson Vinícius Pramiu

Resumo: Este relatório objetiva registrar de forma detalhada e sistemática as atividades de pesquisa e estudos relacionados à temática de proposta de tese do autor. Nele serão revistos alguns conceitos fundamentais de mecânica de fluidos, métodos numéricos em equações diferenciais, instalação e configuração de bibliotecas para implementação computacional como PETSC e MPI, entre outros. Exemplos da resolução de sistemas de equações lineares e da solução de algumas EDPs como a equação de Laplace, de Poisson, da equação do calor transiente e da equação de Stokes serão resolvidos detalhadamente em diferenças finitas ou volumes finitos, empregando malhas estruturadas e em programação serial ou paralela. A linguagem selecionada para implementação dos códgidos fonte é a linguagem C, e alguns exemplos também serão ilustrados e implementados em Scilab. Com a realização destes exemplos espera-se obter os requisitos necessários para compreensão de trabalhos e implementação de código computacional envolvendo simulações em escoamentos turbulentos empregando o método de Simulação de Grandes Escalas - *Large Eddy Simulation*.

1. Instalação e Configuração do PETSC

Esta seção é breve e tem como objetivo auxiliar na instalação e configuração do pacote de ferramentas PETSC, seguindo o manual do software. Informações adicionais podem ser obtidas diretamente no site do desenvolvedor da ferramenta: https://www.mcs.anl.gov/petsc/.

Nesta seção serão apresentados também alguns requisitos computacionais necessários à implementação dos métodos numéricos e resolução das equações discretizadas. Também será exibido uma introdução, instalação e configuração da biblioteca PETSC para o sistema Linux Ubuntu 16.04. PETSc (Portable, Extensible Toolkit for Scientific Computation), é um conjunto de estruturas de dados e rotinas para solução paralela de aplicações científicas modeladas por meio de equações diferenciais parciais.

Para instalação do PETSc é necessário a realização do download da distribuição mais atual do pacote, o que pode ser obtido no link https://www.mcs.anl.gov/petsc/download/index.html. Após a obtenção do arquivo `.tar.gz` é necessário extraí-lo em alguma pasta do computador local. Isto pode ser feito por meio do terminal e empregando o comando `tar -xf petsc-3.7.5.tar.gz`. Após a extração, deve-se acessar a pasta que foi criada e que neste tutorial chama-se `/petsc-3.7.5.tar.gz` e executar o comando para configuração do PETSC:

```
./configure -with-cc=gcc -with-cxx=g++ -with-X=1 -with-fc=gfortran -download-mpich
-download-fblaslapack
```

Esse comando, além de configurar o PETSC, também instala algumas ferramentas como compiladores e bibliotecas. Caso a configuração ocorra corretamente uma mensagem semelhante à da Figura 1 deve ser exibida no terminal.

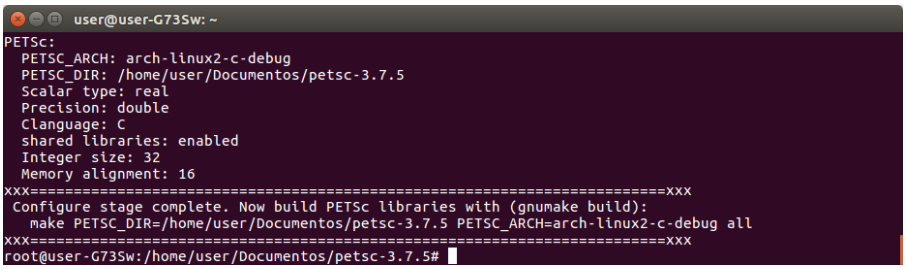

Figura 1: Terminal configuração PETSC.

Como indicado, o usuário deve em seguida executar o seguinte comando:

```
make PETSC_DIR=/home/user/Documentos/petsc-3.7.5 PETSC_ARCH=arch-linux2-c-debug all
```

o que resultará na seguinte Figura 2.

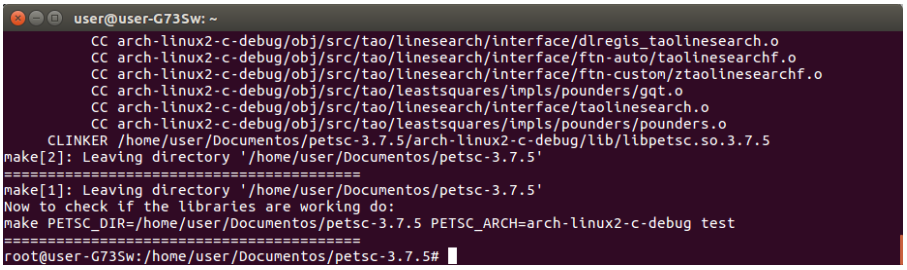

Figura 2: Terminal configuração PETSC.

Como indicado, o usuário deve verificar se as bibliotecas estão trabalhando corretamente, por meio do comando:

```
make PETSC_DIR=/home/user/Documentos/petsc-3.7.5 PETSC_ARCH=arch-linux2-c-debug test
```

o que resultará na seguinte Figura 3.

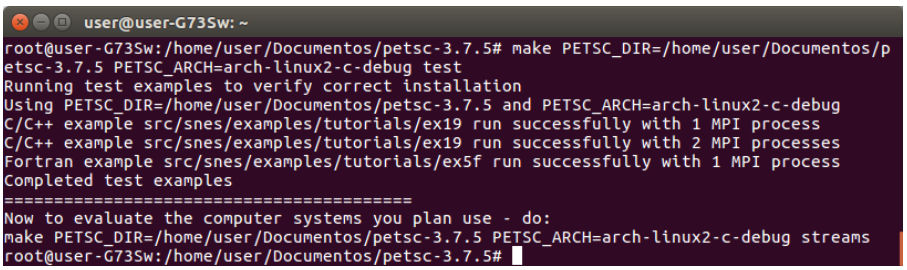

Figura 3: Terminal configuração PETSC.

Para finalizar a configuração, o usuário deve o sistema, por meio do comando:

`make PETSC_DIR=/home/user/Documentos/petsc-3.7.5 PETSC_ARCH=arch-linux2-c-debug streams`

o que resultará na seguinte Figura 4.

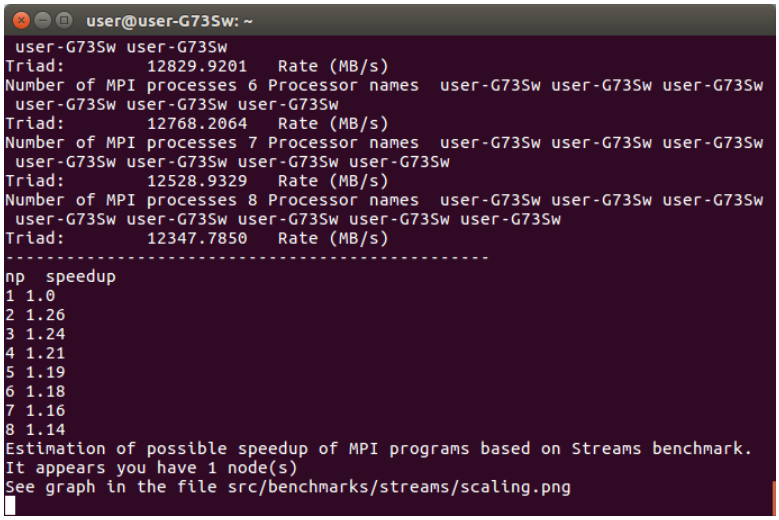

Figura 4: Terminal configuração PETSC.

Se a instalação foi realizada com êxito uma janela com o speedup em função do número de processadores será exibida.

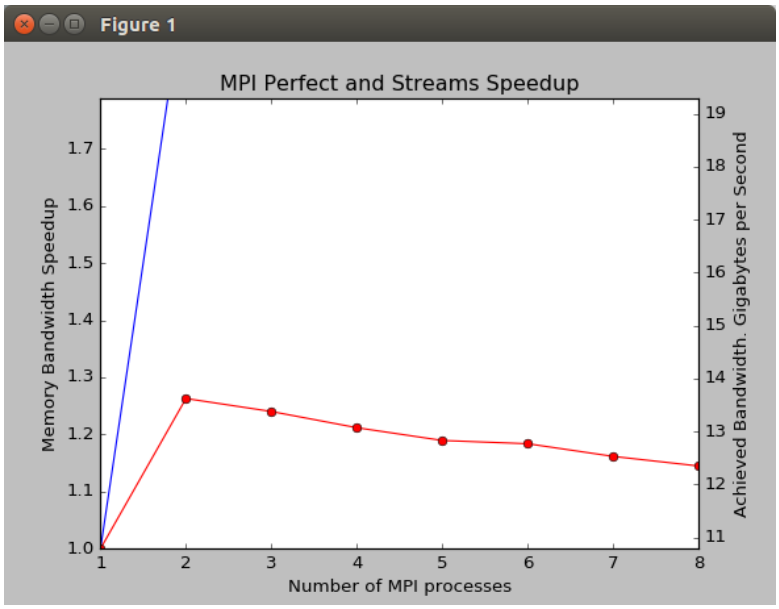

Figura 5: Terminal configuração PETSC.

Ao término da instalação deve-se também configurar, via terminal, as variáveis de ambiente `PETSC_DIR=/home/user/Documentos/petsc-3.7.5` e `PETSC_ARCH=arch-linux2-c-debug` com auxílio do comando `export`. Ver Figura 6.

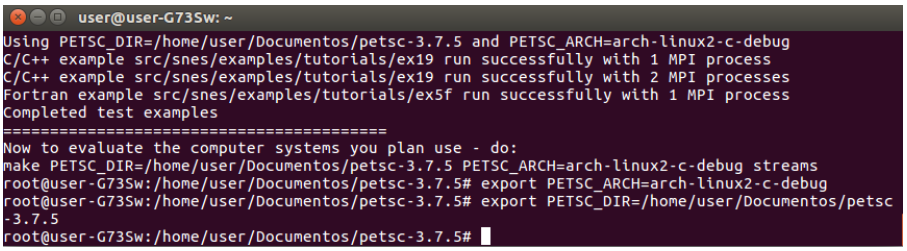

Figura 6: Terminal configuração PETSC.

Para verificar se a instalação e configuração foram realizadas com êxito, é possível navegar até a o diretório: `cd /home/user/Documentos/petsc-3.7.5/src/ksp/ksp/examples/tutorials/` e executar os comandos:

```
make ex1
```

o que resultará na Figura 7.

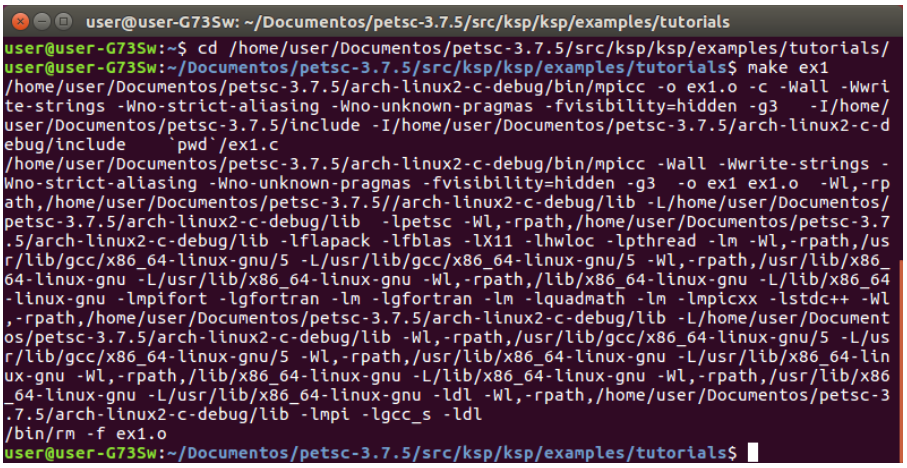

Figura 7: Terminal exemplo ex1.c.

Para executar o código acima compilado, digita-se `./ex1`, o que deve resultar em algo semelhante à Figura 8.

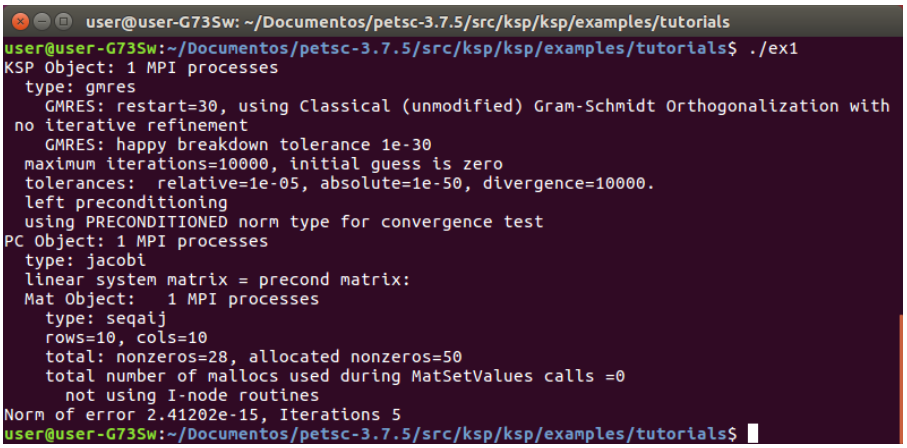

Figura 8: Execução do exemplo ./ex1

Para uma configuraçã completa do PETSC, talvez seja necessária a instalação de alguns pacotes adicionais como é o caso do X11 para criação de janelas gráficas. Sua instalação pode ser realizada via terminal com o comando: `apt install libxt-dev`.

2. Primeiros exemplos com PETSC

Como já mencionado o PETSC é uma suite de ferramentas que permite a solução de sistemas de equações em paralelo. Essa suite foi desenvolvida para resolução de Equações Diferenciais Parciais, em que sua resolução conduz à resolução de sistemas de equações de grandes dimensões, o que demanda algoritmos eficientes e programação paralela. Desta forma o propóstico da biblioteca PETSC é auxiliar na solução de problemas científicos e de engenharia em computadores multiprocessados. O primeiro exemplo aqui ilustrado é apresentado em **?** e aproxima a constante de Euler por meio da série de Maclaurin:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} \approx 2.718281828 \tag{1}$$

O programa `code_euler.c` , ilustrado no código **??**, realiza a computação de cada termo da série infinita em cada processo, resultando numa melhor estimativa de e quando executado em vários processos MPI. Embora seja um exemplo ingênuo do emprego da biblioteca PETSC, ele auxilia na compreensão de algumas ideias envolvidas em computação paralela.

```
#include <petsc.h>

int main(int argc, char **argv) {
  PetscErrorCode ierr;
  int            rank, i;
  double         localval, globalsum;

  PetscInitialize(&argc,&argv,NULL,"Calcula 'e' em paralelo com PETSc.\n\n");

  ierr = MPI_Comm_rank(PETSC_COMM_WORLD,&rank); CHKERRQ(ierr);
```

```
  // calcula  1 / n!  onde n = (rank do processo) + 1
  localval = 1.0;
  for (i = 2; i < rank+1; i++)
      localval /= i;

  // soma as contribuiÃğÃţes de cada processo
  ierr = MPI_Allreduce(&localval, &globalsum, 1, MPI_DOUBLE, MPI_SUM,
                       PETSC_COMM_WORLD); CHKERRQ(ierr);

  // imprime a estimativa de e
  ierr = PetscPrintf(PETSC_COMM_WORLD,
      "O valor da constante 'e' Ãľ aproximadamente: %17.15f\n",globalsum); CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_SELF,
      "rank %d did %d flops\n",rank,(rank > 0) ? rank-1 : 0); CHKERRQ(ierr);

  PetscFinalize();

  return 0;
}
```

Como qualquer programa escrito em linguagem C, o código é iniciado com uma função chamada `main()` a qual tem os argumentos `argc` e `argv` passados via linha de comando. No exemplo ilustrado esses argumentos serão passados à biblioteca através da função `PetscInitialize()`, e a biblioteca obtém as informações passadas em linha de comando. A função `main()` também tem como retorno um valor inteiro, que é igual a 0 se o programa foi executado corretamente. Além disso, é importante utilizar a função PETSC para vericação de erros associados à sua utilização, `CHKERRQ(ierr)`, a qual retorna um valor inteiro diferente de 0 caso alguma anomalia ocorra na execução de alguma função pertencente à biblioteca.

Como indicado no manual **?** para compilar um arquivo que utiliza PETSC, deve-se ter no mesmo diretório do arquivo fonte, um arquivo `makefile`, cujo conteúdo é exibido no código **??**.

```
include ${PETSC_DIR}/lib/petsc/conf/variables
include ${PETSC_DIR}/lib/petsc/conf/rules

code_euler: code_euler.o  chkopts
        -${CLINKER} -o code_euler code_euler.o  ${PETSC_LIB}
        ${RM} code_euler.o

build_vector: build_vector.o  chkopts
        -${CLINKER} -o build_vector build_vector.o  ${PETSC_LIB}
        ${RM} build_vector.o

build_matrix: build_matrix.o  chkopts
        -${CLINKER} -o build_matrix build_matrix.o  ${PETSC_LIB}
        ${RM} build_matrix.o
```

```
solve_linear_ksp: solve_linear_ksp.o  chkopts
	-${CLINKER} -o solve_linear_ksp solve_linear_ksp.o  ${PETSC_LIB}
	${RM} solve_linear_ksp.o

solve_linear_arbitrary: solve_linear_arbitrary.o  chkopts
	-${CLINKER} -o solve_linear_arbitrary solve_linear_arbitrary.o  ${PETSC_LIB}
	${RM} solve_linear_arbitrary.o
```

Após ter criado o arquivo `makefile` é possível compilar o código programa `code_euler.c` com o seguinte comando:

```
user@user-G73Sw:~$ make code_euler
```

Para executar o código compilado basta digitar

```
user@user-G73Sw:~$ ./code_euler
O valor da constante 'e' é aproximadamente: 1.000000000000000
rank 0 did 0 flops
```

O valor obtido para $e = 1.0$ é uma estimativa muito ruim, e isso pode ser melhorado com a execução de mais processos MPI, da seguinte forma:

```
user@user-G73Sw:~$ mpiexec -n 5 ./code_euler
O valor da constante 'e' é aproximadamente: 2.708333333333333
rank 0 did 0 flops
rank 1 did 0 flops
rank 2 did 1 flops
rank 3 did 2 flops
rank 4 did 3 flops
```

Executando o mesmo programa em 10 processos, obtemos uma boa aproximação constante

```
user@user-G73Sw:~$ mpiexec -n 10 ./code_euler
O valor da constante 'e' é aproximadamente: 2.718281525573192
rank 0 did 0 flops
.....
```

Com base na execução dos 10 processos acima, pode-se imaginar que o código tenha sido escrito usando um cluster com no mínimo 10 processadores físicos. Na verdade, esses 5 e 10 processos funcionam muito bem em um laptop com 2 núcleos. Os processos MPI são criados conforme necessário, usando um recurso

antigo de sistemas operacionais: multitarefa. Obviamente a aceleração real do paralelismo (speedup) é outra questão.

No exemplo do programa `code_euler.c`, cada processo MPI calcula o termo $1/n!$, onde n é o retorno de `MPI_Comm_rank()`. É importante notar que `PETSC_COMM_WORLD` é um comunicador MPI contendo todos os processos gerados usando `mpiexec -n N` na linha de comando. Uma chamada para `MPI_Allreduce()` calcula a soma parcial de expressão (1) e envia o resultado de volta para cada processo. Esses usos diretos da API MPI são uma parte (relativamente pequena) do uso do PETSc, mas ocorrem porque o PETSc geralmente evita a duplicação da funcionalidade MPI.

A estimativa calculada de e é impressa de uma só vez. Além disso, cada processo também imprime seu `rank` e o trabalho que ele fez. O comando de impressão formatado `PetscPrintf()`, semelhante ao `fprintf()` da biblioteca padrão C, é chamado duas vezes no código. Na primeira vez MPI usa o comunicador `PETSC_COMM_WORLD` e a segunda vez `PETSC_COMM_SELF`. O primeiro desses trabalhos de impressão é, portanto, coletivo em todos os processos, e apenas uma linha de saída é produzida, enquanto a segunda é individual para cada processo e obtemos n linhas impressas. As linhas de saída `PETSC_COMM_SELF` podem aparecer em ordem aparentemente aleatória uma vez que a impressão ocorre na ordem que essa classificação encontra o comando `PetscPrintf()` no código.

Todo programa ou parte de comando que utiliza a biblioteca PETSC, deve iniciar e terminar com as funções `PetscInitialize()` e `PetscFinalize()`, respectivamente. Observa-se que o último argumento da função `PetscInitialize(&argc, &argv, NULL, help)` fornece uma string que informa ao usuário uma breve descrição do programa, e pode ser visualizada através do comando:

```
user@user-G73Sw:~$ ./code_euler --help
O valor da constante 'e' é aproximadamente: 2.718281525573192
rank 0 did 0 flops
.....
```

2.1. Objetos do tipo vetores e matrizes em PETSC

A maioria dos métodos de resolução numérica de equações diferenciais cumlmina na solução de sistemas lineares de dimensão finita. Como esses sistemas lineares se tornam representações mais precisas do PDE à medida que seu tamanho vai para o infinito, busca-se resolver os maiores sistemas lineares que a tecnologia de computação disponível possa ser capaz de suportar. Resolver tais sistemas lineares, usando algoritmos que têm o potencial de escalar para tamanhos muito grandes - tão grandes, por exemplo, que a solução vetorial do sistema deve ser distribuída através de muitos processadores até mesmo se encaixar na memória - representa a tecnologia de núcleo em PETSc.

Uma observação a ser feita, é que muitas das PDEs discretizadas geram sistemas lineares com estrutura explorável, especialmente a esparcidade, o que significa que há poucas entradas diferentes de zero por linha na matriz. Para que os métodos convirjam, também precisa haver outra estrutura no sistema linear, tal como a regularidade das entradas de matrizes que surgem da suavidade dos coeficientes na PDE. A

aplicação ingênua de métodos diretos é na maioria das vezes, muito lenta.

O código exibido a seguir ilustra um exemplo da criação de um vetor 10×1 em PETSc.

```
static char help[] = "Monta um vetor 10x1 usando Vec.\n";

#include <petsc.h>

int main(int argc, char **args) {
        Vec      x;
        int      i[10] = {0, 1, 2, 3,4, 5, 6, 7, 8, 9};
        double v[10] = {11.0, 7.0, 5.0, 3.0, 6.0,
                        11.0, 7.0, 5.0, 3.0, 6.0};

        PetscInitialize(&argc, &args, NULL, help);

        VecCreate(PETSC_COMM_WORLD,&x);
        VecSetSizes(x,PETSC_DECIDE,10);
        VecSetFromOptions(x);
        VecSetValues(x,10,i,v,INSERT_VALUES);
        VecAssemblyBegin(x);
        VecAssemblyEnd(x);

        VecDestroy(&x);

        PetscFinalize();
        return 0;
}
```

Para compilar o código acima, deve-se alterar o arquivo `makefile` para deixá-lo semelhante ao apresentado abaixo:

```
include ${PETSC_DIR}/lib/petsc/conf/variables
include ${PETSC_DIR}/lib/petsc/conf/rules

code_euler: code_euler.o  chkopts
        -${CLINKER} -o code_euler code_euler.o  ${PETSC_LIB}
        ${RM} code_euler.o

build_vector: build_vector.o  chkopts
        -${CLINKER} -o build_vector build_vector.o  ${PETSC_LIB}
        ${RM} build_vector.o

build_matrix: build_matrix.o  chkopts
        -${CLINKER} -o build_matrix build_matrix.o  ${PETSC_LIB}
        ${RM} build_matrix.o

solve_linear_ksp: solve_linear_ksp.o  chkopts
```

```
	-${CLINKER} -o solve_linear_ksp solve_linear_ksp.o  ${PETSC_LIB}
	${RM} solve_linear_ksp.o

solve_linear_arbitrary: solve_linear_arbitrary.o  chkopts
	-${CLINKER} -o solve_linear_arbitrary solve_linear_arbitrary.o  ${PETSC_LIB}
	${RM} solve_linear_arbitrary.o
```

Em seguida o código é compilado e executado através dos comandos:

```
user@user-G73Sw:~$ make build_vector
user@user-G73Sw:~$ ./build_vector -vec_view
Mat Object: 1 MPI processes
type: seqaij
11.
7.
5.
3.
6.
11.
7.
5.
3.
6.
```

A seguir é apresentado um exemplo simples de como preencher uma matriz 4×4, usando um loop 'for' sobre o índice de linha i. O programa é denominado de `build_matrix.c`:

```
static char help[] = "Monta uma matriz 4x4 usando Mat.\n";

#include <petsc.h>

int main(int argc, char **args) {
	Mat      A;
	int      i, j[4] = {0, 1, 2, 3};
	double   aA[4][4] ={{1.0, 2.0, 3.0, 0.0},
	                    { 2.0, 1.0, -2.0, -3.0},
	                    {-1.0, 1.0, 1.0, 0.0},
	                    { 0.0, 1.0, 1.0, -1.0}};

	PetscInitialize(&argc, &args, NULL, help);

	MatCreate(PETSC_COMM_WORLD, &A);
	MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,4,4);
	MatSetFromOptions(A);
	MatSetUp(A);
```

```
        for (i=0; i<4; i++) {
                MatSetValues(A,1,&i,4,j,aA[i],INSERT_VALUES);
        }
        MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
        MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);

        MatDestroy(&A);

        PetscFinalize();
        return 0;
}
```

O resultado ou a matriz criada pode ser visualizada de várias formas, e aqui faremos duas visualizações uma no formato esparso e outra mostrando todos os seus elementos.

```
user@user-G73Sw:~$ ./build_matrix -mat_view
Mat Object: 1 MPI processes
type: seqaij
row 0: (0, 1.)  (1, 2.)  (2, 3.)  (3, 0.)
row 1: (0, 2.)  (1, 1.)  (2, -2.)  (3, -3.)
row 2: (0, -1.)  (1, 1.)  (2, 1.)  (3, 0.)
row 3: (0, 0.)  (1, 1.)  (2, 1.)  (3, -1.)

user@user-G73Sw:~$ ./build_matrix -mat_view ::ascii_dense
Mat Object: 1 MPI processes
type: seqaij
 1.00000e+00  2.00000e+00  3.00000e+00  0.00000e+00
 2.00000e+00  1.00000e+00 -2.00000e+00 -3.00000e+00
-1.00000e+00  1.00000e+00  1.00000e+00  0.00000e+00
 0.00000e+00  1.00000e+00  1.00000e+00 -1.00000e+00
```

É possível também salvar a matriz impressa no terminal através do comando:

```
./build_matrix -mat_view ascii:build_matrix.txt:ascii_dense.
```

Como descrito em **?**, embora PETSC seja escrito em C, e não em C ++, ela é uma biblioteca orientada a objetos. Para construir o nosso primeiro código PETSc para resolver um sistema linear, vamos usar os tipos de dados `Vec` e `Mat`, que são essencialmente objetos, que possuem vetores e matrizes. O exemplo a seguir descreve a solução do sistema linear:

$$\begin{bmatrix} 1 & 2 & 3 & 0 \\ 2 & 1 & -2 & -3 \\ -1 & 1 & 1 & 0 \\ 0 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 1 \\ 1 \\ 3 \end{bmatrix} \tag{2}$$

Um objeto KSP resolve o sistema linear, com o algoritmo de solução específico escolhido apenas em tempo de execução. O código fonte que contém o programa para resolver o sistema (2) é apresentado a seguir:

```
static char help[] = "Resolve um sistema linear 4x4 usando Vec, Mat, and KSP.\n";

#include <petsc.h>

int main(int argc, char **args) {
        Vec     x, b;
        Mat     A;
        KSP     ksp;
        int     i, j[4] = {0, 1, 2, 3};
        double  ab[4] = {7.0, 1.0, 1.0, 3.0};
        double  aA[4][4] ={{1.0, 2.0, 3.0, 0.0},
                           { 2.0, 1.0, -2.0, -3.0},
                           {-1.0, 1.0, 1.0, 0.0},
                           { 0.0, 1.0, 1.0, -1.0}};

        PetscInitialize(&argc, &args, NULL, help);

        VecCreate(PETSC_COMM_WORLD, &b);
        VecSetSizes(b,PETSC_DECIDE,4);
        VecSetFromOptions(b);
        VecSetValues(b,4,j,ab, INSERT_VALUES);
        VecAssemblyBegin(b);
        VecAssemblyEnd(b);

        MatCreate(PETSC_COMM_WORLD, &A);
        MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,4,4);
        MatSetFromOptions(A);
        MatSetUp(A);
        for (i=0; i<4; i++) {
                MatSetValues(A,1,&i,4,j,aA[i],INSERT_VALUES);
        }
        MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);
        MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);

        KSPCreate(PETSC_COMM_WORLD,&ksp);
        KSPSetOperators(ksp, A, A);
        KSPSetFromOptions(ksp);
        VecDuplicate(b,&x);
        KSPSolve(ksp,b,x);
        VecView(x,PETSC_VIEWER_STDOUT_WORLD);

        KSPDestroy(&ksp);
        MatDestroy(&A);
        VecDestroy(&x);
```

```
        VecDestroy(&b);
        PetscFinalize();
        return 0;
}
```

O resulta final da execução do programa `solve_linear_ksp.c` é:

```
user@user-G73Sw:~$ ./solve_linear_ksp
Vec Object: 1 MPI processes
type: seq
1.
0.
2.
-1.
```

O código `solve_linear_ksp.c` apresentou a solução de um sistema com dimensão fixa, no entanto pode ser necessário, alterar essa dimensão em tempo de execução, como é o caso do próximo exemplo. Ele resovle um sistema de equação com tamanho arbitrário e definido no momento da execução, através de um valor inteiro passado na função `PetscOptionsXXX()`. Além disso, nesse exemplo serão vistos algumas formas de manipulação de vetores como soma e determinação da sua norma euclidiana, utilizando `VecAXPY` e `VecNorm`, respectivamente. O programa é ilustrado no código.....

```
//STARTSETUP
static char help[] =
  "Solve a tridiagonal system of arbitrary size.  Option prefix = tri_.\n";

#include <petsc.h>

int main(int argc,char **args) {
  PetscErrorCode ierr;
  Vec    x, b, xexact;
  Mat    A;
  KSP    ksp;
  int    m = 4, i, lstart, lend, j[3];
  double v[3], xval, errnorm;

  PetscInitialize(&argc,&args,NULL,help);

  ierr = PetscOptionsBegin(PETSC_COMM_WORLD,"tri_","options for tri","");  CHKERRQ(ierr);
  ierr = PetscOptionsInt("-m","dimension of linear system","tri.c",m,&m,NULL);  CHKERRQ(ier
  ierr = PetscOptionsEnd();  CHKERRQ(ierr);

  ierr = VecCreate(PETSC_COMM_WORLD,&x);  CHKERRQ(ierr);
  ierr = VecSetSizes(x,PETSC_DECIDE,m);  CHKERRQ(ierr);
  ierr = VecSetFromOptions(x);  CHKERRQ(ierr);
```

```
  ierr = VecDuplicate(x,&b); CHKERRQ(ierr);
  ierr = VecDuplicate(x,&xexact); CHKERRQ(ierr);

  ierr = MatCreate(PETSC_COMM_WORLD,&A); CHKERRQ(ierr);
  ierr = MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,m,m); CHKERRQ(ierr);
  ierr = MatSetOptionsPrefix(A,"a_"); CHKERRQ(ierr);
  ierr = MatSetFromOptions(A); CHKERRQ(ierr);
  ierr = MatSetUp(A); CHKERRQ(ierr);
//ENDSETUP
  ierr = MatGetOwnershipRange(A,&Istart,&Iend); CHKERRQ(ierr);
  for (i=Istart; i<Iend; i++) {
    if (i == 0) {
      v[0] = 3.0;  v[1] = -1.0;
      j[0] = 0;    j[1] = 1;
      ierr = MatSetValues(A,1,&i,2,j,v,INSERT_VALUES); CHKERRQ(ierr);
    } else {
      v[0] = -1.0;  v[1] = 3.0;  v[2] = -1.0;
      j[0] = i-1;   j[1] = i;    j[2] = i+1;
      if (i == m-1) {
        ierr = MatSetValues(A,1,&i,2,j,v,INSERT_VALUES); CHKERRQ(ierr);
      } else {
        ierr = MatSetValues(A,1,&i,3,j,v,INSERT_VALUES); CHKERRQ(ierr);
      }
    }
    xval = exp(cos(i));
    ierr = VecSetValues(xexact,1,&i,&xval,INSERT_VALUES); CHKERRQ(ierr);
  }
  ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
  ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
  ierr = VecAssemblyBegin(xexact); CHKERRQ(ierr);
  ierr = VecAssemblyEnd(xexact); CHKERRQ(ierr);
  ierr = MatMult(A,xexact,b); CHKERRQ(ierr);

  ierr = KSPCreate(PETSC_COMM_WORLD,&ksp); CHKERRQ(ierr);
  ierr = KSPSetOperators(ksp,A,A); CHKERRQ(ierr);
  ierr = KSPSetFromOptions(ksp); CHKERRQ(ierr);
  ierr = KSPSolve(ksp,b,x); CHKERRQ(ierr);

  ierr = VecAXPY(x,-1.0,xexact); CHKERRQ(ierr);
  ierr = VecNorm(x,NORM_2,&errnorm); CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_WORLD,
         "error for m = %d system is |x-xexact|_2 = %.1e\n",m,errnorm); CHKERRQ(ierr);

  KSPDestroy(&ksp);  MatDestroy(&A);
  VecDestroy(&x);  VecDestroy(&b);  VecDestroy(&xexact);
  PetscFinalize();
  return 0;
}
```

//ENDSOLVE

É importante destacar que embora o tamanho do sistem seja arbitrário, ele sempre terá a forma tridiagonal, com valores 3 na diagonal principal e valor -1 na diagonal superior e inferior.

O primeiro novo recurso usado neste código é `PetscOptionsBegin()` e `PetscOptionsEnd()`, ou seja, a chamada para `PetscOptionsInt()`. O início do método define um prefixo `-tri_` para que a nova opção criada seja distinguida das muitas opções integradas do PETSc que começam por exemplo com `-ksp_` ou `-vec_` ou algo do tipo. Aqui `PetscOptionsInt()` cria a opção `-tri_m` para que o usuário possa definir a variável m e deixa como padrão $m = 4$ inalterado se a opção não for definida na execução.

Após configurar a nova opção em `solve_linear_arbitrary.c` a solução numérica `Vec x` é criada exatamente como fizemos no último exemplo. Mas agora também é necessário criar `Vec s`, `Vec b` e `Vec xexact`. O primeiro vetor é o lado direito do sistema linear e este último mantém a solução exata para o sistema linear para que seja possível avaliar o erro associado à solução numérica.

Em seguida, deve-se montar a matriz A, que como mencionado é uma matriz tridiagonal. É importante montá-la de forma eficiente em paralelo, algo que será relevante na resolução de equações diferenciais 2D e 3D posteriormente. No entanto, somente quando o `solve_linear_arbitrary.c` é executado, sabemos quantos processos estão em uso. O método `MatGetOwnershipRange()` informa o programa, executando em um determinado processo (rank), quais linhas ele possui localmente.

Como observado no início do código `solve_linear_arbitrary.c`, chamamos função `MatGetOwnershipRange(A,` para obter os índices de linha inicial e final para o processo local. Estes índices são usados como limites no loop 'for' que preenche as linhas da matriz localmente. utiliza-se `MatSetValues()` para realmente definir as entradas da matriz A e `MatAssemblyBegin/End()` para completar a montagem de A.

Adicionalmente, é necessário montar o lado direito do sistema linear e também a solução exacta para o sistema linear $(A\,xexato = b)$. A maneira mais simples de fazer isso, é escolher uma solução exata, e em seguida, calcular b, multiplicando A pela solução exata. Assim, definimos valores para `xexact`. Em seguida calculamos b com auxílio da função `MatMult(A, xexact, b)`.

Como em vecmatksp.c criamos o objeto KSP e então chamamos o solver `KSPSolve()` para resolver aproximadamente $Ax = b$. A opção `-ksp_monitor` imprime a norma residual $||b - Ax||_2$ em tempo de execução. Neste caso, também queremos ver que o erro real $||x - x_{ex}||_2$ é pequeno quando o solver KSP é concluido. Assim, depois de obter x do KSPSolve (), calculamos o erro com os códigos:

`VecAXPY(x,-1.0,xexact)` : $x \leftarrow -1.0x_{ex} + x$
`Vecnorm(x,NORM_2,&errnorm)` : `errnorm` $\leftarrow ||x||_2$

Obviamente o sistema linear resolvido neste exemplo é fácil de resolver por ser tridiagonal, simétrico, diagonal-dominante e positivo definido. Pode-se verificar o tempo necessário para resolução do sistema com auxílio do comando `time`, como segue:

```
user@user-G73Sw:~$ time ./solve_linear_arbitrary -tri_m 1000000
error for m = 1000000 system is (x-xexact)_2 = 4.8e-11
real 0m1.814s
user 0m1.772s
sys 0m0.040s
```

Somente o tempo 'real' deve ser considerado. Note a diferença ao executar o mesmo código com $m = 10000000$.

```
user@user-G73Sw:~$ time ./solve_linear_arbitrary -tri_m 10000000
error for m = 10000000 system is (x-xexact)_2 = 3.5e-10
real 0m17.154s
user 0m17.971s
sys 0m0.140s
```

A princípio um sistema de equações contendo 10^7 variáveis parece ser grande. Mas pensando numa simulação tridimensional da equação de Navier-Stokes em um domínio cúbico de 1 metro de lado e com espaçamento de malha igual a 0.01 m (1cm), isso geraria um sistema de equações dessa ordem de grandeza. Agora, o programa será executado empregando 8 processos.

```
user@user-G73Sw:~$ time mpiexec -n 8./solve_linear_arbitrary -tri_m 10000000
error for m = 10000000 system is (x-xexact)_2 = 9.9e-11
real 0m5.484s
user 0m38.976s
sys 0m2.260s
```

Como esperado, o tempo de execução caiu de 17 para 5 segundos, indicando um speedup de:

$$546546 \tag{3}$$

3. A equaçã de Poisson em uma malha estruturada

Esta seção é dedicada a resolução numérica do problema de Poisson em um quadrado. Este é um problema que permite compreender partes essenciais da implementação de códigos empregando a biblioteca PETSc. A discretização da EDP gera um sistema linear que é mais interessante do que o sistema tridiagonal que foi resolvido anteriormente. Será construiída uma malha estruturada usando um objeto `DMDA`, que será introduzido nesta seção, e depois será montada uma matriz em paralelo com base nessa malha. Por último o sistema linear resultante será resovido em paralelo usando um objeto `KSP`.