Relatório de estudos individual

Pétterson Vinícius Pramiu

27 de março de 2017

Sumário

Capítulo 1

Introdução

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*.

Capítulo 2

Instalação e Configuração do PETSc

Esta seção 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 2.1 deve ser exibida no terminal.

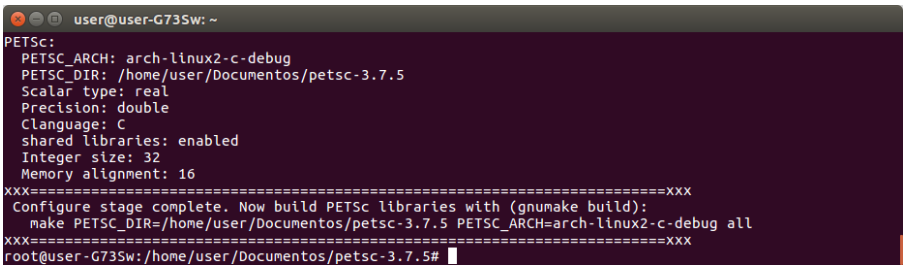


Figura 2.1: Terminal configuração PETSc.

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 2.2.

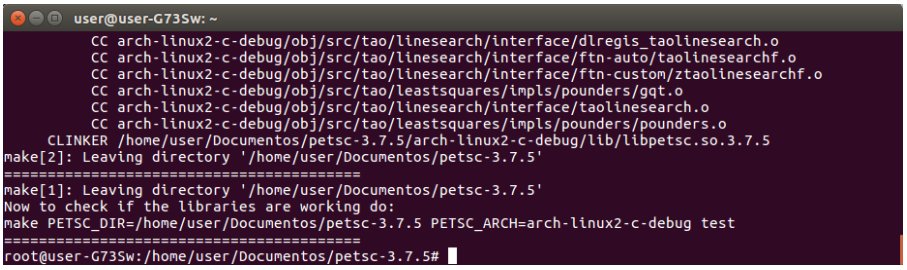


Figura 2.2: Terminal configuração PETSc.

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 2.3.

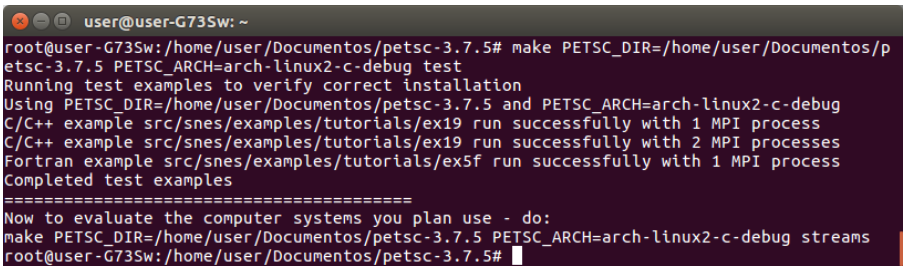


Figura 2.3: Terminal configuração PETSc.

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 2.4.

Figura 2.4: Terminal configuração PETSc.

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

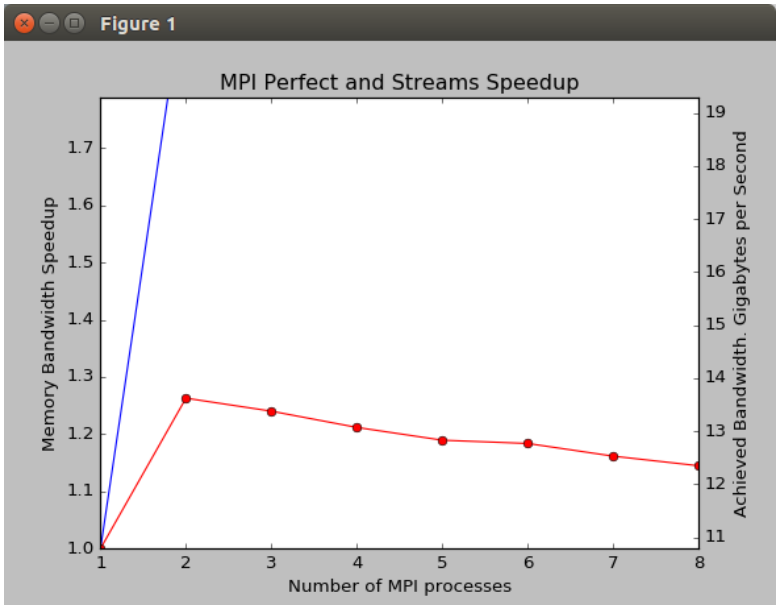


Figura 2.5: Terminal configuração PETSc.

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 2.6.

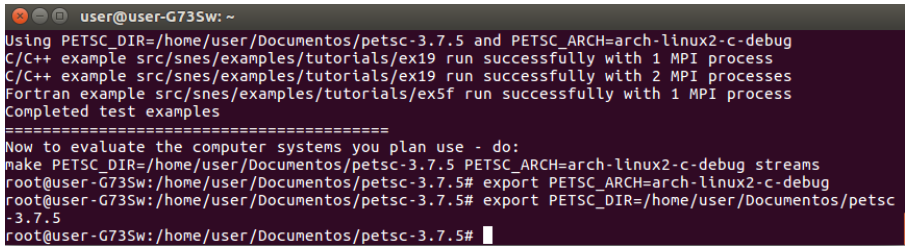


Figura 2.6: Terminal configuração PETSc.

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 2.7.

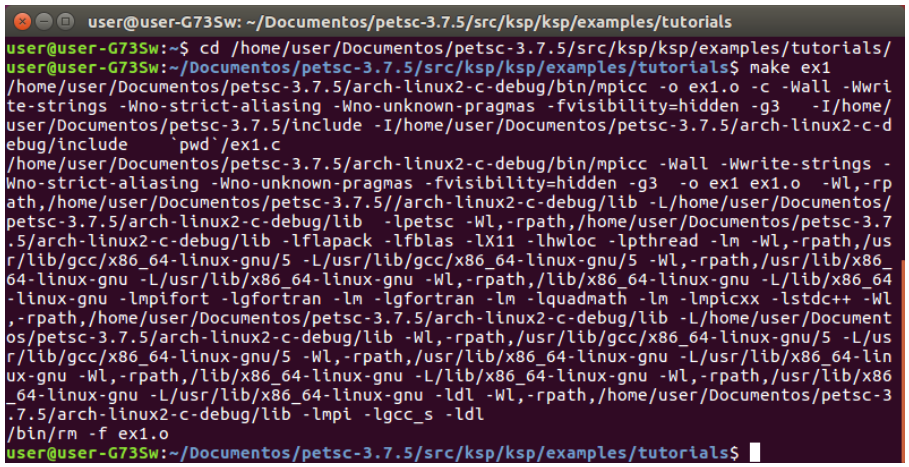


Figura 2.7: Terminal exemplo ex1.c.

Para executar o código acima compilado, digita-se `./ex1`, o que deve resultar em algo semelhante à Figura 2.8.

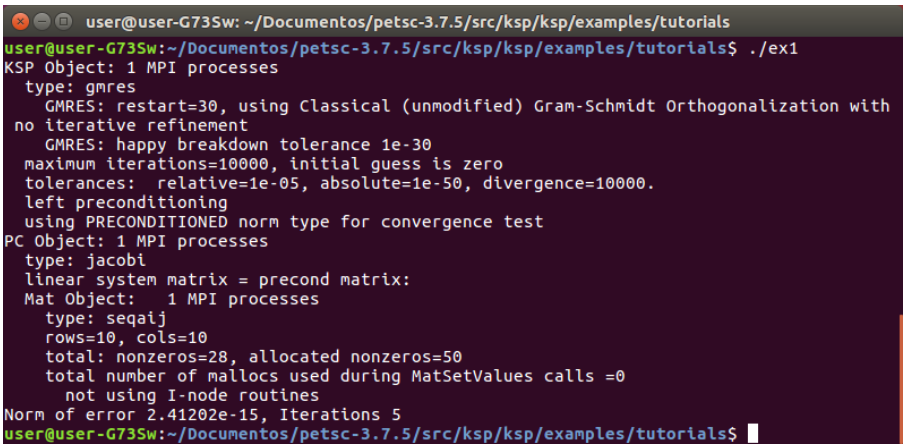


Figura 2.8: Execução do exemplo ./ex1

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`.

2.1 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 biblioteca 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 [4] e aproxima a constante de Euler por meio da série de Maclaurin:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} \approx 2.718281828 \tag{2.1}$$

O programa `code_euler.c` , ilustrado no código 2.1, 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.

```
1 #include <petsc.h>
2
3 int main(int argc, char **argv) {
4   PetscErrorCode ierr;
5   int            rank, i;
6   double         localval, globalsum;
```

```
7
8   PetscInitialize(&argc,&argv,NULL,"Calcula 'e' em paralelo com PETSc.\n\n");
9
10  ierr = MPI_Comm_rank(PETSC_COMM_WORLD,&rank); CHKERRQ(ierr);
11
12  // calcula  1 / n!  onde n = (rank do processo) + 1
13  localval = 1.0;
14  for (i = 2; i < rank+1; i++)
15      localval /= i;
16
17  // soma as contribuiÃğÃţes de cada processo
18  ierr = MPI_Allreduce(&localval, &globalsum, 1, MPI_DOUBLE, MPI_SUM,
19                       PETSC_COMM_WORLD); CHKERRQ(ierr);
20
21  // imprime a estimativa de e
22  ierr = PetscPrintf(PETSC_COMM_WORLD,
23      "O valor da constante 'e' Ãl aproximadamente: %17.15f\n",globalsum); CHKERRQ(ier
24  ierr = PetscPrintf(PETSC_COMM_SELF,
25      "rank %d did %d flops\n",rank,(rank > 0) ? rank-1 : 0); CHKERRQ(ierr);
26
27  PetscFinalize();
28
29  return 0;
30 }
```

Listing 2.1: Código constante de Euler.

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 [1] 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 2.2.

```
1 include ${PETSC_DIR}/lib/petsc/conf/variables
2 include ${PETSC_DIR}/lib/petsc/conf/rules
3
4 code_euler: code_euler.o  chkopts
5   -${CLINKER} -o code_euler code_euler.o  ${PETSC_LIB}
6   ${RM} code_euler.o
7
8 build_vector_01: build_vector_01.o  chkopts
```

```
 9   -${CLINKER} -o build_vector_01 build_vector_01.o  ${PETSC_LIB}
10   ${RM} build_vector_01.o
11
12
13 build_vector: build_vector.o  chkopts
14   -${CLINKER} -o build_vector build_vector.o  ${PETSC_LIB}
15   ${RM} build_vector.o
16
17 build_matrix: build_matrix.o  chkopts
18   -${CLINKER} -o build_matrix build_matrix.o  ${PETSC_LIB}
19   ${RM} build_matrix.o
20
21 solve_linear_ksp: solve_linear_ksp.o  chkopts
22   -${CLINKER} -o solve_linear_ksp solve_linear_ksp.o  ${PETSC_LIB}
23   ${RM} solve_linear_ksp.o
24
25 solve_linear_arbitrary: solve_linear_arbitrary.o  chkopts
26   -${CLINKER} -o solve_linear_arbitrary solve_linear_arbitrary.o  ${PETSC_LIB}
27   ${RM} solve_linear_arbitrary.o
28
29 poisson_fd_2d: poisson_fd_2d.o  chkopts
30   -${CLINKER} -o poisson_fd_2d poisson_fd_2d.o  ${PETSC_LIB}
31   ${RM} poisson_fd_2d.o
```

Listing 2.2: Exemplo de `makefile`.

Após criar 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 deve-se executar

```
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 computador pessoal com 2 núcleos (processodor i3 2120). 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 (2.1) 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 programa 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
.....
```

2.1.1 Objetos do tipo vetores e matrizes em PETSc

A maioria dos métodos de resolução numérica de equações diferenciais culmina na solução de sistemas lineares de dimensão finita. Como esses sistemas lineares se tornam representações mais precisas da EDP à medida que seu tamanho vai para o infinito, busca-se resolver os maiores sistemas lineares que a tecnologia disponível possa ser capaz de computar. 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 em memória distribuída - representa a biblioteca 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.

```c
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();
```

```
23   return 0;
24 }
```

codigo/build_vector.c

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

```
1  include ${PETSC_DIR}/lib/petsc/conf/variables
2  include ${PETSC_DIR}/lib/petsc/conf/rules
3
4  code_euler: code_euler.o  chkopts
5    -${CLINKER} -o code_euler code_euler.o  ${PETSC_LIB}
6    ${RM} code_euler.o
7
8  build_vector_01: build_vector_01.o  chkopts
9    -${CLINKER} -o build_vector_01 build_vector_01.o  ${PETSC_LIB}
10   ${RM} build_vector_01.o
11
12
13 build_vector: build_vector.o  chkopts
14   -${CLINKER} -o build_vector build_vector.o  ${PETSC_LIB}
15   ${RM} build_vector.o
16
17 build_matrix: build_matrix.o  chkopts
18   -${CLINKER} -o build_matrix build_matrix.o  ${PETSC_LIB}
19   ${RM} build_matrix.o
20
21 solve_linear_ksp: solve_linear_ksp.o  chkopts
22   -${CLINKER} -o solve_linear_ksp solve_linear_ksp.o  ${PETSC_LIB}
23   ${RM} solve_linear_ksp.o
24
25 solve_linear_arbitrary: solve_linear_arbitrary.o  chkopts
26   -${CLINKER} -o solve_linear_arbitrary solve_linear_arbitrary.o  ${PETSC_LIB}
27   ${RM} solve_linear_arbitrary.o
28
29 poisson_fd_2d: poisson_fd_2d.o  chkopts
30   -${CLINKER} -o poisson_fd_2d poisson_fd_2d.o  ${PETSC_LIB}
31   ${RM} poisson_fd_2d.o
```

codigo/makefile

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`:

```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;
}
```

codigo/build_matrix.c

Neste exemplo a matriz é montada por linhas, e em cada iteração do laço 'for', são inseridas quatro colunas em cada linha, a saber as colunas indicadas pelo vetor j. 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.
```

2.1.2 Solução de sistema linear

Como descrito em [4], 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:

$$\begin{bmatrix} 1 & 2 & 3 & 0 \\ 2 & 1 & -2 & -3 \\ -1 & 1 & 1 & 0 \\ 0 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 1 \\ 1 \\ 3 \end{bmatrix} \tag{2.2}$$

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 (2.8) é apresentado a seguir:

```c
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;
```

```
48 }
```

codigo/solve_linear_ksp.c

O resultado 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 2.3.

```
1  //STARTSETUP
2  static char help[] =
3    "Solve a tridiagonal system of arbitrary size.  Option prefix = tri_.\n";
4
5  #include <petsc.h>
6
7  int main(int argc,char **args) {
8    PetscErrorCode ierr;
9    Vec     x, b, xexact;
10   Mat     A;
11   KSP     ksp;
12   int     m = 4, i, Istart, Iend, j[3];
13   double v[3], xval, errnorm;
14
15   PetscInitialize(&argc,&args,NULL,help);
16
17   ierr = PetscOptionsBegin(PETSC_COMM_WORLD,"tri_","options for tri",""); CHKERRQ(ierr
18   ierr = PetscOptionsInt("-m","dimension of linear system","tri.c",m,&m,NULL); CHKERRQ
19   ierr = PetscOptionsEnd(); CHKERRQ(ierr);
20
21   ierr = VecCreate(PETSC_COMM_WORLD,&x); CHKERRQ(ierr);
22   ierr = VecSetSizes(x,PETSC_DECIDE,m); CHKERRQ(ierr);
23   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;
}
```

```
72 //ENDSOLVE
```

Listing 2.3: Resolução de sistema com tamanho arbitrário.

É importante destacar que embora o tamanho do sistema 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 realizado 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 a função `MatGetOwnershipRange(A, &Istart, &Iend)` 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 $(A\,x_{ex} = b)$. 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 foi reduzido, neste caso caiu de 17 para 5 segundos.

2.2 A equação de Poisson

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 construída 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` [2].

Neste exemplo a equação de Poisson será resolvida numa região quadrada, $S, (0,1) \times (0,1)$ cujo contorno é representado por ∂S. O domínio do problema é ilustrado na Figura 2.9 e formulado em (2.3).

$$\begin{aligned} -\nabla^2 u &= f \quad \text{em } S \\ u &= 0 \quad \text{em } \partial S \end{aligned} \tag{2.3}$$

O Laplaciano de $u(x, y)$ é especificado por:

$$\begin{aligned} \nabla^2 u &= \nabla \cdot (\nabla u) \\ &= \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \end{aligned} \tag{2.4}$$

e aparece com frequência em modelos matemáticos que expressam a conservação de alguma quantidade u, juntamente com a suposição de que o fluxo de u é proporcional ao seu gradiente. O problema aqui estudado está sujeito à condições de contorno homogêneas de Dirichlet.

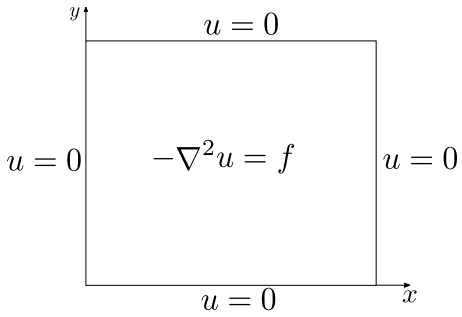


Figura 2.9: Domínio do problema.

O problema de Poisson pode modelar o potencial eletrostático, a distribuição de equilíbrio de certas caminhos aleatórios ou vários outros fenômenos físicos. Para um exemplo, a condução de calor em sólidos segue a lei de Fourier, que diz que o fluxo de calor é $q = -k\nabla u$, onde k é a condutividade térmica. A conservação da energia diz que $c\rho \partial u/\partial t = -\nabla \cdot q + f$ se f representa uma fonte de calor no interior do domínio. O coeficiente $c\rho$ parametriza a capacidade do material para manter o calor por um ganho de temperatura. Se k é constante, então em estado estacionário essas condições resultam na equação de Poisson $0 = k\nabla^2 u + f$. Mantendo a temperatura nula ao longo do limite da região (contorno), tem-se o problema (2.3), que será resolvido numericamente através do método de diferenças finitas.

2.2.1 Geração da malha

O método de diferenças finitias é desenvolvido sobre uma malha composta por $m_x m_y$ pontos igualmente espaçados, como ilustrado na Figura 2.10, cujo espaçamento na direção x é especificado por $h_x = 1/(m_x - 1)$ e por $h_y = 1/(m_y - 1)$ na direção y.

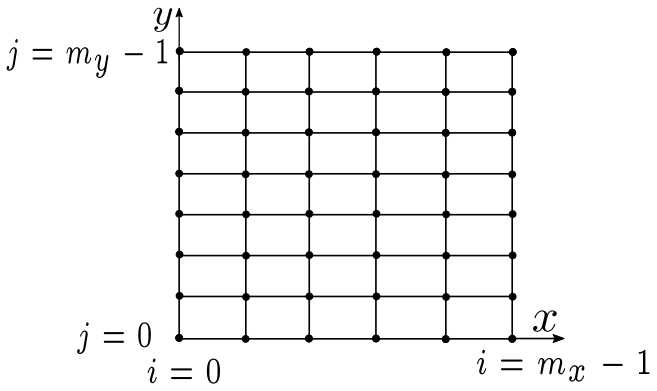


Figura 2.10: Malha computacional.

Considerando a malha da Figura 2.11 em $m_x = 5$ e $m_y = 7$, as coordenadas da malha são $(x_i = ih_x, y_j = jh_y)$, para $i = 0, ..., m_x - 1$ e $j = 0, ..., m_y - 1$.

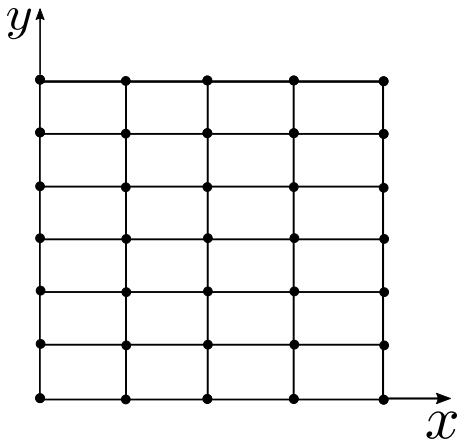


Figura 2.11: Malha computacional 5×7.

Para construção dessa malha 2D de forma distribuida em processos MPI, utiliza-se um novo objeto PETSc que cria uma instância do tipo PETSc `DM` para descrever a topologia (conexão) da malha, a forma como ela é distribuída através de processos MPI e a forma como cada processo pode acessar dados de seus processos vizinhos. Este caso específico aqui é um `DMDA`, que é uma sub-classe de `DM`. A designação `DM` pode significar "distributed mesh" ou "domain management", e `DA` significa "distributed array". Ao executar os comandos abaixo:

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

```
user@user-G73Sw:~$./poisson -da_grid_x 5 -da_grid_y 7
```

uma malha correspondente à da Figura 2.11 será criada, em que todos os nodos são construídos e pertencentes a um único processo MPI. No entanto, ao digitar o comando:

```
user@user-G73Sw:~$mpiexec -n 4 ./poisson -da_grid_x 5 -da_grid_y 7
```

então a biblioteca PETSc faz o equilíbrio de carga para os pontos da malha entre os processos, com a restrição de que cada processo MPI possui uma sub-malha retangular. Como observado na Figura 2.12, PETSc distribui os quatro processos através dos 35 pontos da malha de forma relativamente uniforme (O processo rank 0 possui 12 pontos e o rank 3 possui 6, enquanto os outros estão entre eles).

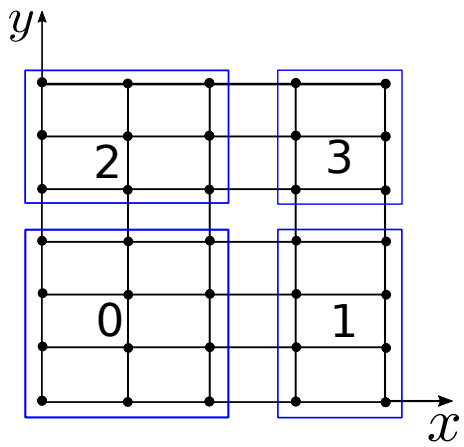


Figura 2.12: Malha computacional 5×7 e 4 processos.

O bloco de código referente ao emprego do objeto `DMDACreate2d` é descrito a seguir:

```
1
2   // default size (9 x 9) can be changed using -da_grid_x M -da_grid_y N
3   ierr = DMDACreate2d(PETSC_COMM_WORLD,
4                 DM_BOUNDARY_NONE, DM_BOUNDARY_NONE, DMDA_STENCIL_STAR,
```

Listing 2.4: DMDA()

No Código 2.4, o primeiro argumento é referente ao comunicador. O segundo e terceiro argumentos são do tipo `DM_BOUNDARY_NONE` pois as condições de Dirichlet não precisam de comunicação para o próximo processo. No quarto argumento, escolhe-se o método `DMDA_STENCIL_STAR` porque somente os vizinhos cardeais de um ponto da malha são usados na discretização. A Figura 2.13 mostra seu stencil computacional.

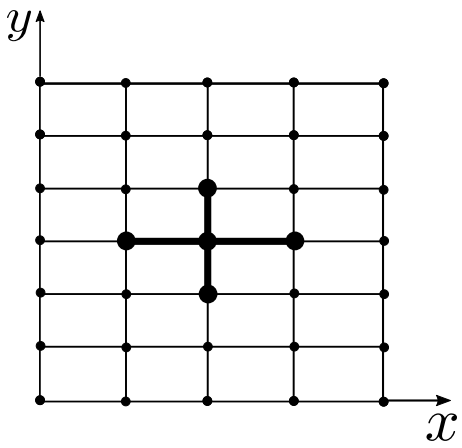


Figura 2.13: Stencil computacional epregado na discretização.

Os dois argumentos `PETSC_DECIDE` depois disso dão ao `PETSc` autonomia para distribuir a malha sobre os processos de acordo com a quantidade de processos no comunicador MPI. Os dois argumentos seguintes, na nona e décima posições, dizem que nossa PDE é escalar, com grau de liberdade igual a 1 (dof $= 1$) e que o método de diferenças finitas só precisará de um vizinho em cada direção (s $= 1$). Os próximos dois argumentos depois disso são NULL porque não estamos dizendo PETSc quaisquer detalhes sobre como distribuir processos sobre a malha. Finalmente, o objeto `DMDA` é retornado através de um argumento ponteiro.

2.2.2 Discretização em diferenças finitas

Considerando uma discretização da equação (2.3) em diferenças finitas, ela pode ser escrita na forma discreta como:

$$-\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} - \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} = f(x_i, y_j) \tag{2.5}$$

que se aplica a todos os pontos interiores do domínio, ou seja, quando $0 < i < m_x - 1$ e $0 < j < m_j - 1$. As condições de contorno são impostas como:

$$u_{0,j} = 0, \quad u_{m_x-1,j} = 0, \quad u_{i,0} = 0, \quad u_{i,m_y-1} = 0, \tag{2.6}$$

para todo i, j.

Todos os valores $u_{i,j}$ serão tratados como incógnitas, sejam do interior ou da fronteira, resultando num total de $L = m_x m_y$ incógnitas. Isso permite a construção de um sistema de equações linear:

$$A\boldsymbol{u} = \boldsymbol{b} \tag{2.7}$$

em que a matriz A tem dimensão $L \times L$, e os vetores $\boldsymbol{u}$ e $\boldsymbol{b}$, dimensão $L \times 1$. No entanto, para montar as entradas de A e $\boldsymbol{b}$ no sistema linear (2.7), deve-se ordenar as incógnitas. Essa ordenação é

implementada dentro do objeto DMDA, e o código só precisa usar as coordenadas (i, j). A ordenação usada em um processo único (serial) executado por um DMDA 2D é mostrada na Figura 2.14. Em uma malha m_x por m_y, pode-se escrever o novo índice global como $k = jm_x + i$ de modo que $u_{i,j}$ seja a k-ésima incógnita do sistema. Esse mapeamento de índice feito pelo objeto DMDA fica transparente ao usuário.

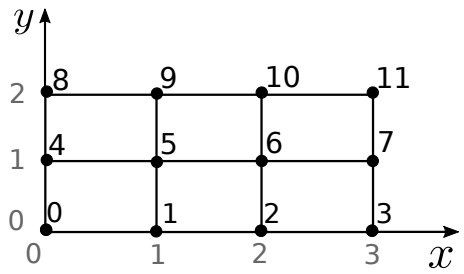


Figura 2.14: Malha computacional 4×3.

Como exemplo, a seguir será construído o sistema de equações para o caso da malha representada na Figura 2.14, em que $m_x = 4$ e $m_y = 3$, o que resulta em $h_x = 1/3$ e $h_y = 1/2$. Somente os pontos com índice global $k = 5$ e $k = 6$ não são condições de contorno e o sistema linear é dado por:

$$\begin{bmatrix} 1 & & & & & & & & & & & \\ & 1 & & & & & & & & & & \\ & & 1 & & & & & & & & & \\ & & & 1 & & & & & & & & \\ & & & & 1 & & & & & & & \\ c & & & b & a & b & & & c & & & \\ & c & & & b & a & b & & & c & & \\ & & & & & & 1 & & & & & \\ & & & & & & & 1 & & & & \\ & & & & & & & & 1 & & & \\ & & & & & & & & & 1 & & \\ & & & & & & & & & & 1 & \end{bmatrix} \begin{bmatrix} u_{0,0} \\ u_{1,0} \\ u_{2,0} \\ u_{3,0} \\ u_{0,1} \\ u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{0,2} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ f_{1,1} \\ f_{2,1} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.8}$$

em que $a = 2/h_x^2 + 2/h_y^2 = 26$, $b = -1/h_x^2 = -9$ e $c = -1/h_y^2 = -4$. A matriz A não é simétrica e seu número de condição na norma 2 é $k(A) = ||A||_2||A^{-1}||_2 = 43.16$. Dois detalhes importantes devem ser considerados nessa primeira discretização.

O primeiro é que a equação (2.5) tem diferentes escalas ou ordem de grandeza. Por exemplo, ao tomar $m_x = m_y = 1001$, tem-se $h_x = h_y = 0,001$ o que resulta em coeficientes para os nodos internos do domínio da ordem de $4/0.001^2 = x \times 10^6$, enquanto que os coeficientes para os nodos do contorno são iguais a 1. Para mitigar esse problema, pode-se multiplicar (2.5) pela área do elemento de malha, $h_x h_y$, resultando em:

$$2(a + b)u_{i,j} - a(u_{i+1,j} + u_{i-1,j}) - a(u_{i,j+1} + u_{i,j-1}) = h_x h_y f(x_i, y_j) \tag{2.9}$$

em que $a = h_y/h_x$ e $b = h_x/h_y$.

Em segundo, as equações de diferenças finitas podem ser reescritas para formar uma matriz A simétrica. Por exemplo, em um ponto da malha adjacente ao limite esquerdo do domínio, o caso $i = 1$ de (2.9), o valor de localização da incógnita $u_{0,j}$ aparece na equação. A matriz do sistema linear será simétrica ao mover tais valores para o vetor do lado direito, $\boldsymbol{b}$. Isso é possível pois o valor $u_{0,j}$ é explicitado pelas condições de contorno. Isso converte as entradas das sub-diagonais de A para zero nas colunas correspondentes aos valores de contorno conhecidos. Desta forma, pode-se resolver o sistema de equações por métodos mais eficientes, como gradientes conjugados, KSP e pré-condicionadores de Cholesky. Ao realizar estas duas alterações no sistema linear (2.8), tem-se:

$$\begin{bmatrix} 1 & & & & & & & & & & & \\ & 1 & & & & & & & & & & \\ & & 1 & & & & & & & & & \\ & & & 1 & & & & & & & & \\ & & & & 1 & & & & & & & \\ & & & & & \alpha & \beta & & & & & \\ & & & & & \beta & \alpha & & & & & \\ & & & & & & & 1 & & & & \\ & & & & & & & & 1 & & & \\ & & & & & & & & & 1 & & \\ & & & & & & & & & & 1 & \\ & & & & & & & & & & & 1 \end{bmatrix} \begin{bmatrix} u_{0,0} \\ u_{1,0} \\ u_{2,0} \\ u_{3,0} \\ u_{0,1} \\ u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{0,2} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ (1/6)f_{1,1} \\ (1/6)f_{2,1} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.10}$$

em que $\alpha = 2(h_x/h_y + h_y/h_x) = 13/3$, $\beta = -h_y/h_x = -3/2$. A nova matriz A é simétrica e positiva definida, está melhor escalada que antes e tem um número de condição $k(A) = 5.83$.

2.2.3 Montagem da matriz A em PETSc

A função denominada `formMatrix()` no Código 2.5 é responsável pela montagem da matriz A do sistema linear (2.10).

```
1  PetscErrorCode formMatrix(DM da, Mat A) {
2    PetscErrorCode ierr;
3    DMDALocalInfo info;
4    MatStencil     row, col[5];
5    double         hx, hy, v[5];
6    int            i, j, ncols;
7
8    ierr = DMDAGetLocalInfo(da,&info); CHKERRQ(ierr);
9    hx = 1.0/(info.mx-1);  hy = 1.0/(info.my-1);
10   for (j = info.ys; j < info.ys+info.ym; j++) {
```

```
11      for (i = info.xs; i < info.xs+info.xm; i++) {
12        row.j = j;            // row of A corresponding to (x_i,y_j)
13        row.i = i;
14        col[0].j = j;         // in this diagonal entry
15        col[0].i = i;
16        ncols = 1;
17        if (i==0 || i==info.mx-1 || j==0 || j==info.my-1) {
18          v[0] = 1.0;  // if on boundary, just insert diagonal entry
19        } else {
20          v[0] = 2*(hy/hx + hx/hy); // ... everywhere else we build a row
21          // if neighbor is NOT a known boundary value then we put an entry
22          if (i-1 > 0) {
23            col[ncols].j = j;    col[ncols].i = i-1;  v[ncols++] = -hy/hx;
24    //printf("linha j: %d linha i: %d Ncols: %d \n",row.j,row.i,ncols);
25    }
26          if (i+1 < info.mx-1) {
27            col[ncols].j = j;    col[ncols].i = i+1;  v[ncols++] = -hy/hx;  }
28          if (j-1 > 0) {
29            col[ncols].j = j-1;  col[ncols].i = i;    v[ncols++] = -hx/hy;  }
30          if (j+1 < info.my-1) {
31            col[ncols].j = j+1;  col[ncols].i = i;    v[ncols++] = -hx/hy;
32    //printf("linha j: %d linha i: %d Ncols: %d \n",row.j,row.i,ncols);
33    }
34        }
35    ierr = MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES); CHKERRQ(ierr);
36      }
37    }
38    ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
39    ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
40    //ierr = MatView(A,PETSC_VIEWER_STDOUT_SELF);
```

Listing 2.5: formMatrix()

A variável `DMDALocalInfo info` declarada no Código 2.5 precisa de uma descrição especial. Ela é uma estrutura de inteiros em C, definida pela biblioteca PETSc, e serve para descrever tanto o tamanho da malha global como a extensão das submalhas locais. A extensão da malha global está nos membros `info.mx` e `info.my`. Os processos locais possem sub-malhas retangulares com dimensão `info.xm` e `info.ym`, com um intervalo de índices locais em duas dimensões da forma:

$$\texttt{info.xs} \leq i \leq \texttt{info.xs} + \texttt{info.xm} - 1$$
$$\texttt{info.ys} \leq j \leq \texttt{info.ys} + \texttt{info.ym} - 1$$

Para melhor compreensão, considere a malha da Figura 2.11, em que `info.mx`=5 e `info.my`=7. Ao considerar o caso em que são empregados 4 processos MPI, tem-se que os processos com rank 0 e 2 têm `info.xs`=0 e `info.xm`=3, por outro lado, os processos com rank 1 e 3 têm

`info.xs`=3 e `info.xm`=2, respectivamente. Para a coordenada j, os processos com rank 0 e 1 têm `info.ys`=0 e `info.ym`=4, por outro lado, os processos com rank 2 e 3 têm `info.ys`=4 e `info.xm`=3, respectivamente. Com esses índices, é possível montar a matriz bidimensional em paralelo.

Em particular, os índices locais (i, j) podem ser usados para inserir entradas na `Mat A`. No Código 2.5, vemos um uso de `MatSetValuesStencil()` para cada ponto da malha local. Para um ponto interior genérico, este comando insere cinco coeficientes na matriz, cuja estrutura de dados é do tipo `MatStencil`, que é uma estrutura da biblioteca PETSc que apresenta quatro valores inteiros, k, j, i, c. No caso 2D, com um único grau de liberdade em cada nodo, usamos apenas os membros i e j da estrutura `MatStencil`. A partir da discretização (2.9), as entradas da matriz são $a_{i,i} = 2(h_y/h_x + h_x/h_y)$ na diagonal e $a_{i,j} = -h_y/h_x$ ou $a_{i,j} = -h_x/h_y$ para as diagonais secundárias.

2.2.4 Um problema particular

Neste seção, será especificada uma função $u(x, y)$ para que o problema de Poisson tenha uma solução exata que também satisfaça as condições de contorno de Dirichlet homogêneas ($u = 0$ ao longo de ∂S):

$$u(x, y) = (x^2 - x^4)(y^4 - y^2) \tag{2.11}$$

Cujo Laplaciano é igual a $f = -\nabla^2 u$:

$$f(x, y) = 2(1 - 6x^2)y^2(1 - y^2) + 2(1 - 6y^2)x^2(1 - x^2) \tag{2.12}$$

A expressão (2.11) será referenciada como solução exata, u_{ex} a partir de agora. Este mesmo problema é ilustrado no capítulo 4 [**?**] e em [4],

O Código 2.6 mostra como a solução exata é implementado na função `formExact()`, e mostra como (2.12) é implementada na função `formRHS()`. Os cálculos nestes códigos usam apenas coordenadas da malha local (i, j). Ou seja, a aritmética de ponteiros PETSc permite indexar as arrays que obtemos de `DMDAVecGetArray()` usando as coordenadas da malha i e j, sem conhecer os índices das incógnitas no `Vec u`, considerado como vetor de coluna de comprimento $L = m_x m_y$. Ao terminar de construir os vetores, restaura-se as arrays chamando a função `DMDAVecRestoreArray()`.

```
1
2 //STARTEXACT
3 PetscErrorCode formExact(DM da, Vec uexact) {
4   PetscErrorCode ierr;
5   DMDALocalInfo info;
6   int            i, j;
7   double         hx, hy, x, y, **auexact;
```

```
 8
 9   ierr = DMDAGetLocalInfo(da,&info); CHKERRQ(ierr);
10   hx = 1.0/(info.mx-1);  hy = 1.0/(info.my-1);
11   ierr = DMDAVecGetArray(da, uexact, &auexact);CHKERRQ(ierr);
12   for (j = info.ys; j < info.ys+info.ym; j++) {
13     y = j * hy;
14     for (i = info.xs; i < info.xs+info.xm; i++) {
15       x = i * hx;
16       auexact[j][i] = x*x * (1.0 - x*x) * y*y * (y*y - 1.0);
17     }
18   }
19   ierr = DMDAVecRestoreArray(da, uexact, &auexact);CHKERRQ(ierr);
20   return 0;
21 }
22
23 PetscErrorCode formRHS(DM da, Vec b) {
24   PetscErrorCode ierr;
25   int            i, j;
26   double         hx, hy, x, y, f, **ab;
27   DMDALocalInfo info;
28
29   ierr = DMDAGetLocalInfo(da,&info); CHKERRQ(ierr);
30   hx = 1.0/(info.mx-1);  hy = 1.0/(info.my-1);
31   ierr = DMDAVecGetArray(da, b, &ab);CHKERRQ(ierr);
32   for (j=info.ys; j<info.ys+info.ym; j++) {
33     y = j * hy;
34     for (i=info.xs; i<info.xs+info.xm; i++) {
35       x = i * hx;
36       if (i==0 || i==info.mx-1 || j==0 || j==info.my-1) {
37         ab[j][i] = 0.0;                    // on bdry the eqn is 1*u = 0
38       } else {  // if not bdry; note  f = - (u_xx + u_yy)  where u is exact
39         f = 2.0 * ( (1.0 - 6.0*x*x) * y*y * (1.0 - y*y)
40                     + (1.0 - 6.0*y*y) * x*x * (1.0 - x*x) );
41         ab[j][i] = hx * hy * f;
42       }
43     }
44   }
45   ierr = DMDAVecRestoreArray(da, b, &ab); CHKERRQ(ierr);
```

Listing 2.6: `formExact()` e `formRHS`

2.2.5 Resolução do sistema de equações

O Código 2.7 mostra a função principal e os objetos necessários para resolver o problema de Poisson: um `DM`, um `Mat` e três `Vec`. Um objeto `DM` determina as dimensões da matriz e dos

vetores a partir das dimensões da malha, o que é feito quando chamamos `DMCreateMatrix()` e `DMCreateGlobalVector()` para criar objetos `Mat` e `Vec`, respectivamente. Então invoca-se as funções mostradas nos Códigos 2.5 e 2.6 para preencher a matriz e os vetores.

```
1
2 //STARTCREATE
3 int main(int argc,char **args) {
4   PetscErrorCode ierr;
5   DM             da;
6   Mat            A;
7   Vec            b,u,uexact;
8   KSP            ksp;
9   double         errnorm;
10  DMDALocalInfo  info;
11
12  PetscInitialize(&argc,&args,(char*)0,help);
13
14  // default size (9 x 9) can be changed using -da_grid_x M -da_grid_y N
15  ierr = DMDACreate2d(PETSC_COMM_WORLD,
16                DM_BOUNDARY_NONE, DM_BOUNDARY_NONE, DMDA_STENCIL_STAR,
17                -9,-9,PETSC_DECIDE,PETSC_DECIDE,1,1,NULL,NULL,
18                &da); CHKERRQ(ierr);
19
20  // create linear system matrix A
21  ierr = DMCreateMatrix(da,&A);CHKERRQ(ierr);
22  ierr = MatSetOptionsPrefix(A,"a_"); CHKERRQ(ierr);
23  ierr = MatSetFromOptions(A); CHKERRQ(ierr);
24
25  // create right-hand-side (RHS) b, approx solution u, exact solution uexact
26  ierr = DMCreateGlobalVector(da,&b);CHKERRQ(ierr);
27  ierr = VecDuplicate(b,&u); CHKERRQ(ierr);
28  ierr = VecDuplicate(b,&uexact); CHKERRQ(ierr);
29
30  // fill vectors and assemble linear system
31  ierr = formExact(da,uexact); CHKERRQ(ierr);
32  ierr = formRHS(da,b); CHKERRQ(ierr);
33  ierr = formMatrix(da,A); CHKERRQ(ierr);
34 //ENDCREATE
35 //STARTSOLVE
36  // create and solve the linear system
37  ierr = KSPCreate(PETSC_COMM_WORLD,&ksp); CHKERRQ(ierr);
38  ierr = KSPSetOperators(ksp,A,A); CHKERRQ(ierr);
39  ierr = KSPSetFromOptions(ksp); CHKERRQ(ierr);
40  ierr = KSPSolve(ksp,b,u); CHKERRQ(ierr);
41
42  // report on grid and numerical error
43  ierr = VecAXPY(u,-1.0,uexact); CHKERRQ(ierr);     // u <- u + (-1.0) uxact
44  ierr = VecNorm(u,NORM_INFINITY,&errnorm); CHKERRQ(ierr);
```

```
45   ierr = DMDAGetLocalInfo(da,&info);CHKERRQ(ierr);
46   ierr = PetscPrintf(PETSC_COMM_WORLD,
47               "on %d x %d grid:  error |u-uexact|_inf = %g\n",
48               info.mx,info.my,errnorm); CHKERRQ(ierr);
49
50   VecDestroy(&u);  VecDestroy(&uexact);  VecDestroy(&b);
51   MatDestroy(&A);  KSPDestroy(&ksp);  DMDestroy(&da);
52   PetscFinalize();
```

Listing 2.7: formExact()

Como já realizado, o sistema linear é resolvido por um objeto de solução linear KSP. O sistema é resolvido chamando `KSPSolve()`. Então calcula-se e imprimi-se na tela o erro numérico $||u - u_{ex}||$. Finalmente, destruímos objetos com o método `XXXDestroy()` e chamamos `PetscFinalize()` para encerrar. Uma primeira execução da solução pode ser feita através do comando:

```
user@user-G73Sw:~$ make poisson_fd_2d
user@user-G73Sw:~$ ./poisson_fd_2d -ksp_monitor
0 KSP Residual norm 1.020952970432e-01
1 KSP Residual norm 2.656923348626e-02
2 KSP Residual norm 8.679141000397e-03
3 KSP Residual norm 1.557150861763e-03
4 KSP Residual norm 2.239919982542e-04
5 KSP Residual norm 2.519822315367e-05
6 KSP Residual norm 2.152764600588e-06
7 KSP Residual norm 2.650467236964e-07
on 9 x 9 grid:  error |u-uexact|_inf = 0.000763959|
```

Pode-se examinar graficamente a malha para verificar a indexação dos objetos, em tempo de execução Assim, se o sistema X-window estiver corretamente configurado na instalação PETSc, então o comando:

```
user@user-G73Sw:~$ ./poisson_fd_2d -da_grid_x 5 -da_grid_y 7 -dm_view draw -draw_pause 5
```

irá plotar a Figura 2.15.

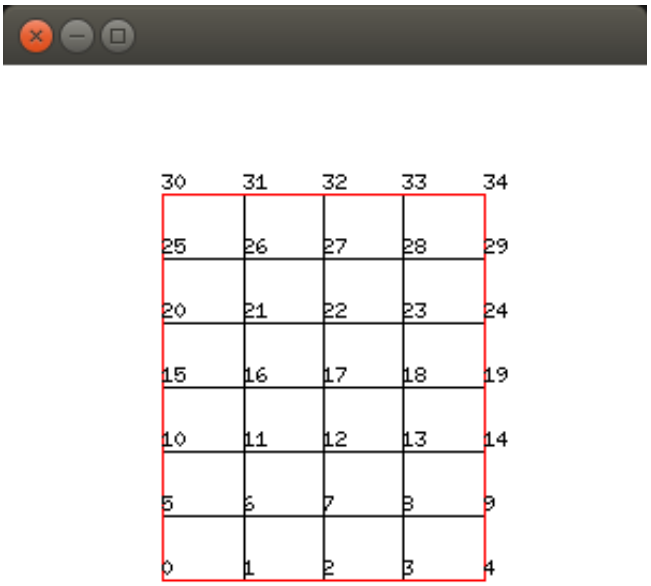


Figura 2.15: Malha computacional.

Outra visualização interessante é obtida com o comando:

```
user@user-G73Sw:~$ ./poisson_fd_2d -da_grid_x 5 -da_grid_y 7 -a_mat_view draw -draw_paus
```

que apresenta a estrutura da matriz A, conforme Figura 2.16.

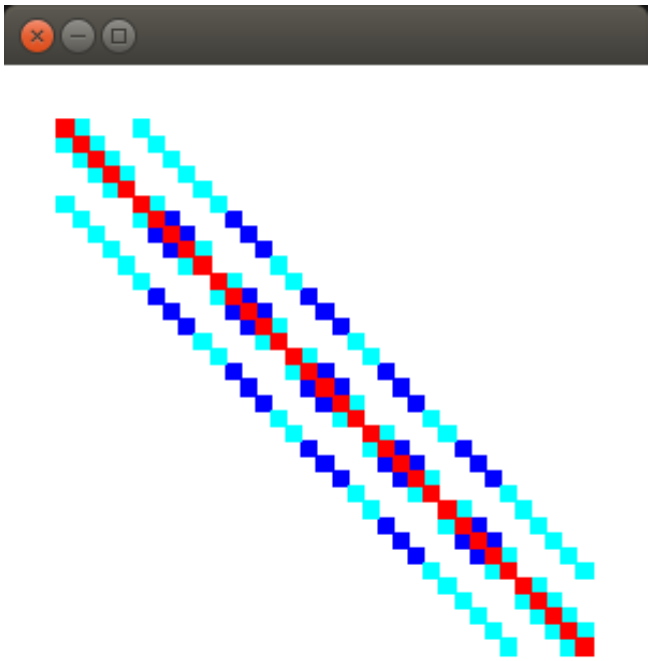


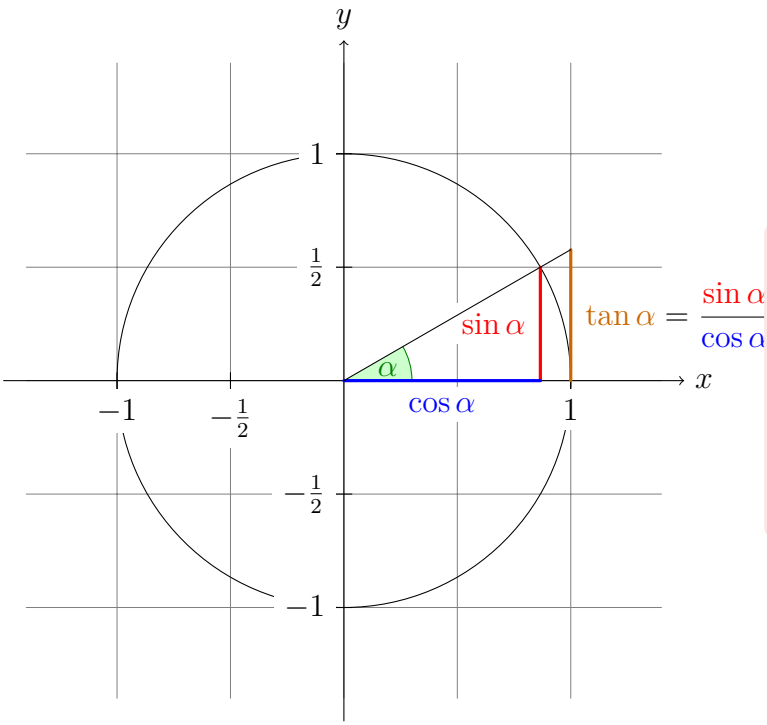
Figura 2.16: Estrutura da matriz A.

É possível também observar a variação do erro conforme o refinamento da malha com o comando:

```
user@user-G73Sw:~$ for K in 0 1 2 3 4 5; do ./poisson_fd_2d -da_refine $K; done
on 9 x 9 grid:  error u-uexact_inf = 0.000763959
on 17 x 17 grid:  error u-uexact_inf = 0.000196764
on 33 x 33 grid:  error u-uexact_inf = 4.91557e-05
on 65 x 65 grid:  error u-uexact_inf = 1.29719e-05
on 129 x 129 grid:  error u-uexact_inf = 3.76924e-06
on 257 x 257 grid:  error u-uexact_inf = 1.73086e-06
```

Note agora a diferença entre tempos de processamento para o código serial e utilizando 2 processadores.

```
user@user-G73Sw:~$ time ./poisson_fd_2d -da_refine 6
real 0m56.467s
user@user-G73Sw:~$ time mpiexec -n 2 ./poisson_fd_2d -da_refine 6
real 1m0.327s
```



The angle α is 30° in the example ($\pi/6$ in radians). The sine of α, which is the height of the red line, is

$$\sin\alpha = 1/2.$$

By the Theorem of Pythagoras ...

Referências Bibliográficas

[1] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc Web page. http://www.mcs.anl.gov/petsc, 2016.

[2] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016.

[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[4] Ed Bueler. *PETSc for Partial Differential Equations*. SIAM, 2018.