COMPUTAÇÃO DE ALTO DESEMPENHO - TRABALHO 2

SÉRGIO CORDEIRO

1. Realize o cálculo do número PI, conforme expresso pela função $\int_0^1 \frac{4}{1+x^2}\ dx$. Utilize por exemplo integração trapezoidal para resolver o problema. Após esta etapa paralelize seu código utilizando OpenMP. Utilize uma referência como resultado para comparar sua solução.

O programa em C listado abaixo calcula o valor de PI corretamente com algarismos significativos. A precisão é limitada pela resolução do tipo **double** nativo. Para desativar a paralelização, basta remover a linha indicada. O programa aceita como argumento um valor inteiro, que é o número de divisões do eixo x a ser empregado: um valor de n muito baixo levará a erro de arredondamento muito grande na integração, conforme a tabela a seguir.

LISTING 1. pi.c

```
1  /*
2  Cálculo do valor de PI por integração trapezoidal de função de referência
3  Uso:
4    pi n
5  onde n é o número de divisões do eixo x. O valor default é 400, que dá o valor exato.
6  Pode usar OpenMP para aumentar o desempenho.
7  */
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <omp.h>
11
12
13 int main(int argc, char * argv[]) {
14 // Calcula o valor de PI por integração trapezoidal de função de referência
15   int divisions;
16   if (argc == 1) {
17     divisions = 400;
18     }
19   else {
20     divisions = atol(argv[1]);
21     }
22   long i;
23   double sum = 0, h = 1.0 / divisions;
24   // Para desativar a paralelização por OpenMp, remover a linha abaixo
25   #pragma omp parallel for private(i) reduction(+:sum)
26   for(i = 1; i <= divisions; ++i) {
27     sum = sum + 1 / (1 + i * h * i * h) + 1 / (1 + (i-1) * h * (i-1) * h );
28     }
```

```
29    sum *= 2.0/divisions;
30    printf("PI = %f \n",sum);
31    }
```

n	resultado	tempo de parede (µs)	
		sem OpenMP	com OpenMP
10	3.139926	0.15	81.5
20	3.141176	0.46	81.5
30	3.141407	0.62	81.5
40	3.141488	0.78	81.5
50	3.141526	0.93	81.5
100	3.141576	1.71	81.5
200	3.141588	3.28	81.6
300	3.141591	5.00	81.6
> 381	3.141592	6.40	81.6

A tabela mostra que o uso de OpenMP neste caso não foi efetivo, pois o programa é muito simples. O ganho obtido com o paralelismo não compensa o custo associado.

2. Implemente computacionalmente o Método dos Gradientes Conjugados (GC), resolva o sistema linear a ser passado pelo professor. Aplique um pré-condicionador do tipo Jacobi no método e verifique se houve redução do número de iterações. Analise os resultados mostrando o número de condição da matriz, o número de iterações e o critério de parada utilizado.
Paralelize o solver implementado utilizando OpenMP. Verifique se houve redução no tempo de processamento e faça uma análise dos resultados.

O programa em C listado abaixo resolve o sistema proposto. A paralelização e o precondicionamento são ativados ou não de acordo com os parâmetros passados na linha de comando. Um limite para o número de iterações e o erro tolerado, que são critérios de parada independentes, também são parâmetros. Como aproximação do número de condição foi tomada a diferença entre o maior e o menor coeficientes presentes na matriz. A matriz proposta está muito mal condicionada, e o precondicionador Jacobiano leva a expressiva redução no número de iterações necessárias, como mostra a tabela seguinte.

LISTING 2. gc.c

```
1  /*
2  Solução de sistema de equações lineares (Ax = b) pelo método dos gradientes conjugados.
3  Pode empregar OpenMP para aumentar o desempenho.
4  Uso:
5    gc Afile bfile maxiter erro prec omp
6  onde
7    Afile é o nome do arquivo contendo os valores da matriz A
8    bfile é o nome do arquivo contendo os valores do vetor b
9    maxiter é o número máximo de iterações permitido
10   erro é o erro tolerado
11   prec indica se deve ou não usar precondicionamento
12     0: não usar precondicionador
13     1: usar precondicionador Jacobiano
14   omp indica se vai usar o OpenMP
15     0: não usar
16     1: usar
17 Limitações:
18   a matriz precisa ser simétrica para que o algoritmo funcione,
19   para o número de condição foi empregada uma fórmula aproximada,
20 */
21 #include <stdio.h>
22 #include <stdlib.h>
23 #include <string.h>
24 #include <ctype.h>
25 #include <math.h>
26 #include <omp.h>
27
28 #include <sys/time.h>
29 #include <windows.h>
30 #include <time.h>
31
32 #define bufsize 5000       // para leitura dos dados em arquivo
```

```
33 #define size 37          // tamanho do sistema
34
35
36 int load(char * fname, double * dados, int nrow, int ncol);
37 double fgetval(char * buffer);
38
39
40 int main(int argc, char * argv[]) {
41 // Solução de sistema de equações lineares pelo método dos gradientes conjugados
42   // Carrega os dados a partir dos arquivos em disco
43   double wstart, wend;
44
45   double A [size][size], b[size], x[size];          // Ax = b
46   double r[size], d[size], q[size], c[size], s[size];    // auxiliares
47   int noread;
48   noread = load(argv[1], &A[0][0], size, size);
49   if (noread > 0) {
50     return 1;
51     }
52   noread = load(argv[2], b, size, 1);
53   if (noread > 0) {
54     return 1;
55     }
56   int j, maxiter = atoi(argv[3]) ;
57   if (maxiter <= 0) {
58     printf("Número máximo de iterações inválido: %d", maxiter);
59     return 1;
60     }
61   double erro = atof(argv[4]);
62   if (erro <= 0) {
63     printf("Erro tolerado inválido: %f", erro);
64     return 1;
65     }
66   int prec = atoi(argv[5]);
67   if (prec < 0 || prec > 1) {
68     printf("Precondicionador desconhecido: %d", prec);
69     return 1;
70     }
71   int omp = atoi(argv[6]);
72   if (omp == 0) {
73     omp_set_num_threads(1);
74     }
75   // Resolve o sistema
76     HANDLE hProcess = GetCurrentProcess();
77     FILETIME ftCreation, ftExit, ftKernel, ftUser1, ftUser2;
78     SYSTEMTIME stUser1,stUser2;
79     GetProcessTimes(hProcess, &ftCreation, &ftExit, &ftKernel, &ftUser1);
80   wstart = omp_get_wtime();
81   // ... inicialização
82   int i, k, jjj = 0;
83   double sum, p, qq;
84   for(int count = 0; count < 1; ++ count) {
85   #pragma omp parallel for private(i)
86   for (i = 0; i < size ; ++ i) {
87     // ... calcula o precondicionador
88     if (prec == 1) {
89       // precondicionador Jacobiano
90       c[i] = 1.0 / A[i][i] ;
91       }
```

```
92      // ... inicializa o vetor solução (x0)
93      // (teoricamente, pode ser qualquer coisa)
94      x[i] = 0;
95      }
96    p = 0;
97    #pragma omp parallel for private(i,sum,k) reduction(+:p)
98    for (i = 0; i < size ; ++ i) {
99      // ... calcula os vetores iniciais r(0) e d(0)
100     sum = 0;
101     // r(0) = b - A x(0)
102     for (k = 0; k < size; ++ k) {
103       sum = sum + A[i][k] * x[k];
104       }
105     r[i] = b[i] - sum;
106     if (prec == 0) {
107       // d(0) = r(0)
108       // p(0) = r(0)' r(0)
109       d[i] = r[i];
110       p += r[i] * r[i];
111       }
112     if (prec == 1) {
113       // d(0) = c' r(0)
114       // p(0) = r(0)' d(0)
115       d[i] = c[i] * r[i];
116       p += r[i] * d[i];
117       }
118     }
119   // ... executa a iteração
120   for (j = 1; j < maxiter && fabs(p) > erro; ++ j) {
121     // q(j) = A d(j)
122     // qq(j) = d(j)' A d(j)
123     qq = 0;
124     #pragma omp parallel for private(i,sum,k) reduction(+:qq)
125     for (i = 0; i < size; ++ i) {
126       sum = 0;
127       for (k = 0; k < size; ++ k) {
128         sum = sum + A[i][k] * d[k];
129     #pragma omp atomic
130     ++ jjj;
131         }
132       qq += d[i] * sum;
133       q[i] = sum;
134       }
135     // x(j+1) = x(j) + p/qq d(j)
136     // r(j+1) = r(i) - p/qq A d(j)
137     double alfa = p/qq;
138     #pragma omp parallel for private(i)
139     for (i = 0; i < size; ++ i) {
140       x[i] += alfa * d[i];
141       r[i] -= alfa * q[i];
142       }
143     double lastp = p;
144     p = 0;
145     #pragma omp parallel for private(i) reduction(+:p)
146     for (i = 0; i < size; ++ i) {
147       if (prec == 0) {
148         // p(j+1) = r(j+1)' r(j+1)
149         p += r[i] * r[i];
150         }
```

```
151       if (prec == 1) {
152         // s(j+1) = c' r(j+1)
153         // p(j+1) = r(j+1)' s(j+1)
154         s[i] = c[i] * r[i];
155         p += r[i] * s[i];
156         }
157       }
158     double beta = p/lastp;
159     #pragma omp parallel for private(i)
160     for (i = 0; i < size; ++ i) {
161       if (prec == 0) {
162         // d(j+1) = r(j+1) + p(j+1)/p(j) d(j)
163         d[i] = r[i] + beta * d[i];
164         }
165       if (prec == 1) {
166         // d(j+1) = s(j+1) + p(j+1)/p(j) d(j)
167         d[i] = s[i] + beta * d[i];
168         }
169       }
170     // if (omp_get_thread_num() == 0) printf("%Iteração %d %d: erro = %f \n", count, j,
           p);
171     }
172   }
173     GetProcessTimes(hProcess, &ftCreation, &ftExit, &ftKernel, &ftUser2);
174   wend = omp_get_wtime();
175     FileTimeToSystemTime(& ftUser1, & stUser1);
176     FileTimeToSystemTime(& ftUser2, & stUser2);
177   double twall = 1000.0 * (wend - wstart);
178     double tuser = 1000.0 * (stUser2.wSecond - stUser1.wSecond) + stUser2.wMilliseconds
          - stUser1.wMilliseconds;
179   printf("tempo gasto = %f ms, user time = %f ms %d\n", twall, tuser, jjj);
180   // Exibe a solução
181   double ka, val, max = 0, min = 1e9;
182   printf("Solução após %d iterações: [", j - 1);
183   for (i = 0; i < size ; ++ i) {
184     if (i > 0) {
185       printf(", ");
186       }
187     printf("%f", x[i]);
188     if (prec == 0) {
189       val = fabs(A[i][i]);
190       if (val > max) max = val;
191       if (val < min) min = val;
192       }
193     }
194   if (prec == 0) {
195     ka = max / min;
196     }
197   if (prec == 1) {
198     ka = 1;
199     }
200   printf("] \nErro < %f, ka = %f, ", fabs(p), ka);
201   return 0;
202   }
203
204 int load(char * fname, double * dados, int nrow, int ncol) {
205 // Carrega os dados do arquivo "fname" na matriz "dados", de dimensão "nrow" x "ncol"
206   char * pchar, buf [bufsize];
207   FILE * fp = fopen (fname, "r");
```

```
208   if (fp == NULL) {
209     printf("Não conseguiu ler o arquivo %s. \n", fname);
210     return 1;
211     }
212   fgets(buf,bufsize,fp);
213   fgets(buf,bufsize,fp);
214   for (int i = 0; i < nrow; ++ i) {
215     fgets(buf,bufsize,fp);
216     for (int k = 0; k < ncol; ++ k) {
217       * dados = fgetval(buf);
218       dados ++ ;
219       }
220     }
221   fclose(fp);
222   return 0;
223   }
224
225
226 double fgetval(char * buffer) {
227 // Extrai o primeiro valor existente no "buffer"
228   char * pchar = buffer, valor [bufsize], * pvalor = valor;
229   while (isblank(*pchar) ) {
230     pchar ++;
231     }
232   while (! isblank(*pchar) )  {
233     * pvalor ++ = * pchar ++;
234     }
235   * pvalor = '\0';
236   strcpy(buffer,pchar);
237   return atof(valor);
238   }
```

Iterações	Erro	Usou precond.	Tempo de parede (ms)		Tempo de usuário (ms)	
			sem OMP	com OMP	sem OMP	com OMP
123	10^{-6}	Não	11.4	26.6	4.16	11.7
58	10^{-6}	Sim	5.39	18.6	1.75	8.33
146	10^{-7}	Não	13.5	29.0	4.70	12.7
66	10^{-7}	Sim	6.11	19.5	2.06	10.2
158	10^{-8}	Não	14.5	34.1	4.77	19.7
83	10^{-8}	Sim	7.69	21.3	2.78	11.6
185	10^{-9}	Não	17.0	40.5	5.75	16.6
89	10^{-9}	Sim	8.25	28.6	2.67	12.1

A tabela também mostra que o uso de OpenMP neste caso não foi efetivo, pois a dimensão do problema é muito pequena. O ganho obtido com o paralelismo não compensa o custo associado.

3. Utilizando o sistema do item anterior, implemente um esquema que permita a partição do sistema de modo a ser processado em 4 máquinas interligadas em rede e operando em cluster com MPI. Apresente o algoritmo que realiza este procedimento.

O programa em C listado abaixo resolve o sistema proposto com paralelização OpenMPI em lugar de OpenMP. No esquema proposto, o mestre interage com o usuário e lê os dados em disco, que anuncia para os demais processadores. A partir daí, cada etapa é calculada em paralelo e, ao final, os resultados são publicados para emprego de todos os processadores. Não foi realizado nenhum teste devido à indisponibilidade de um ambiente computacional adequado.

LISTING 3. gcmpi.c

```
1  /*
2  Solução de sistema de equações lineares (Ax = b) pelo método dos gradientes conjugados.
3  Emprega Open MPI para aumentar o desempenho.
4  Uso:
5    gcmpi Afile bfile maxiter erro prec
6  onde
7    Afile é o nome do arquivo contendo os valores da matriz A
8    bfile é o nome do arquivo contendo os valores do vetor b
9    maxiter é o número máximo de iterações permitido
10   erro é o erro tolerado
11   prec indica se deve ou não usar precondicionamento
12     0: não usar precondicionador
13     1: usar precondicionador Jacobiano
14 Limitações:
15   a matriz precisa ser simétrica para que o algoritmo funcione,
16   para o número de condição foi empregada uma fórmula aproximada,
17 */
18 #include <stdio.h>
19 #include <stdlib.h>
20 #include <string.h>
21 #include <ctype.h>
22 #include <math.h>
23 #include "mpi.h"
24
25
26 #define bufsize 5000       // para leitura dos dados em arquivo
27 #define size 37          // tamanho do sistema
28
29
30 int load(char * fname, double * dados, int nrow, int ncol);
31 double fgetval(char * buffer);
32
33
34 int main(int argc, char * argv[]) {
35 // Solução de sistema de equações lineares pelo método dos gradientes conjugados
36   // Carrega os dados a partir dos arquivos em disco
37   double A[size][size], b[size], x[size];          // Ax = b
38   double r[size], d[size], q[size], c[size], s[size];    // auxiliares
39   int me, nproc, j, maxiter, prec, mysize;
40   double erro;
41   MPI_Init(& argc , & argv);
```

```
42  MPI_Comm_rank(MPI_COMM_WORLD, & me);
43  MPI_Comm_size(MPI_COMM_WORLD, & nproc);
44  if (me == 0) {
45    // ... só o mestre executa a leitura dos dados e dos parâmetros
46    int noread;
47    noread = load(argv[1], &A[0][0], size, size);
48    if (noread > 0) {
49      return 1;
50      }
51    noread = load(argv[2], b, size, 1);
52    if (noread > 0) {
53      return 1;
54      }
55    maxiter = atoi(argv[3]) ;
56    if (maxiter <= 0) {
57      printf("Número máximo de iterações inválido: %d", maxiter);
58      return 1;
59      }
60    erro = atof(argv[4]);
61    if (erro <= 0) {
62      printf("Erro tolerado inválido: %f", erro);
63      return 1;
64      }
65    prec = atoi(argv[5]);
66    if (prec < 0 || prec > 1) {
67      printf("Precondicionador desconhecido: %d", prec);
68      return 1;
69      }
70    // ... faz o tamanho do problema ser divisível pelo número de processadores
71    my_size = size + size % (nproc - 1);
72    // ... comunica os valores aos demais
73    MPI_Bcast (& my_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
74    MPI_Bcast (& prec, 1, MPI_INT, 0, MPI_COMM_WORLD);
75    MPI_Bcast (& erro, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
76    MPI_Bcast (& maxiter, 1, MPI_INT, 0, MPI_COMM_WORLD);
77    MPI_Bcast (& A [0][0] , my_size * size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
78    MPI_Bcast (b , my_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
79    }
80  else {
81    // ... os demais processadores recebem os valores passados pelo mestre
82    MPI_Recv (& my_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
83    MPI_Recv (& prec, 1, MPI_INT, 0, MPI_COMM_WORLD);
84    MPI_Recv (& erro, 1, MPI_INT, 0, MPI_COMM_WORLD);
85    MPI_Recv (& maxiter, 1, MPI_INT, 0, MPI_COMM_WORLD);
86    MPI_Recv (& A [0][0] , my_size * size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
87    MPI_Recv (b , my_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
88    }
89  // Resolve o sistema
90  // ... inicialização
91  int i, k;
92  double sum, p, pt;
93  for (i = me * my_size; i < (me + 1) * my_size ; ++ i) {
94    // ... calcula o precondicionador
95    if (prec == 1) {
96      // precondicionador Jacobiano
97      c[i] = 1.0 / A[i][i] ;
98      }
99    // ... inicializa o vetor solução (x0)
100   // (teoricamente, pode ser qualquer coisa)
```

```
101     x[i] = 0;
102     }
103   // ... todos anunciam os vetores inicializados
104   MPI_Bcast (c + me * my_size, my_size, MPI_DOUBLE, me, MPI_COMM_WORLD);
105   MPI_Bcast (x + me * my_size, my_size, MPI_DOUBLE, me, MPI_COMM_WORLD);
106   // ... todos recebem os vetores inicializados
107   MPI_Reduce(c, 0, my_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD );
108   MPI_Reduce(x, 0, my_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD );
109   pt = p = 0;
110   for (i = me * my_size; i < (me + 1) * my_size ; ++ i) {
111     // ... calcula os vetores iniciais r(0) e d(0)
112     sum = 0;
113     // r(0) = b - A x(0)
114     for (k = 0; k < size; ++ k) {
115       sum = sum + A[i][k] * x[k];
116       }
117     r[i] = b[i] - sum;
118     if (prec == 0) {
119       // d(0) = r(0)
120       // p(0) = r(0)' r(0)
121       d[i] = r[i];
122       pt += r[i] * r[i];
123       }
124     if (prec == 1) {
125       // d(0) = c' r(0)
126       // p(0) = r(0)' d(0)
127       d[i] = c[i] * r[i];
128       pt += r[i] * d[i];
129       }
130     }
131   // ... todos anunciam os valores calculados
132   MPI_Bcast(r + me * my_size, my_size, MPI_DOUBLE, me, MPI_COMM_WORLD);
133   MPI_Bcast(d + me * my_size, my_size, MPI_DOUBLE, me, MPI_COMM_WORLD);
134   MPI_Bcast(& pt, 1, MPI_DOUBLE, me, MPI_COMM_WORLD);
135   // ... todos recebem os valores calculados
136   MPI_Reduce(r, 0, my_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD );
137   MPI_Reduce(d, 0, my_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD );
138   MPI_Reduce(& pt, & p, my_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD );
139   // ... executa a iteração
140   for (j = 1; j < maxiter && fabs(p) > erro; ++ j) {
141     // q(j) = A d(j)
142     // qq(j) = d(j)' A d(j)
143     double qq = qt = 0 ;
144     for (i = me * my_size; i < (me + 1) * my_size ; ++ i) {
145       sum = 0;
146       for (k = 0; k < size; ++ k) {
147         sum = sum + A[i][k] * d[k];
148         }
149       qq += d[i] * sum;
150       q[i] = sum;
151       }
152     // ... todos anunciam os valores calculados
153     MPI_Bcast(q + me * my_size, my_size, MPI_DOUBLE, me, MPI_COMM_WORLD);
154     MPI_Bcast(& qq, 1, MPI_DOUBLE, me, MPI_COMM_WORLD);
155     // ... todos recebem os valores calculados
156     MPI_Reduce(q, 0, my_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD );
157     MPI_Reduce(& qt, & qq, my_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD );
158     // x(j+1) = x(j) + p/qq d(j)
159     // r(j+1) = r(i) - p/qq A d(j)
```

```
160     double alfa = p/qt;
161     for (i = me * my_size; i < (me + 1) * my_size ; ++ i) {
162       x[i] += alfa * d[i];
163       r[i] -= alfa * q[i];
164       }
165     // ... todos anunciam os valores calculados
166     MPI_Bcast(x + me * my_size, my_size, MPI_DOUBLE, me, MPI_COMM_WORLD);
167     MPI_Bcast(r + me * my_size, my_size, MPI_DOUBLE, me, MPI_COMM_WORLD);
168     // ... todos recebem os valores calculados
169     MPI_Reduce(x, 0, my_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD );
170     MPI_Reduce(r, 0, my_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD );
171     double lastp = p;
172     pt = p = 0;
173     for (i = me * my_size; i < (me + 1) * my_size ; ++ i) {
174       if (prec == 0) {
175         // p(j+1) = r(j+1)' r(j+1)
176         pt += r[i] * r[i];
177         }
178       if (prec == 1) {
179         // s(j+1) = c' r(j+1)
180         // p(j+1) = r(j+1)' s(j+1)
181         s[i] = c[i] * r[i];
182         pt += r[i] * s[i];
183         }
184       }
185     // ... todos anunciam os valores calculados
186     if (prec == 1) {
187       MPI_Bcast(s + me * my_size, my_size, MPI_DOUBLE, me, MPI_COMM_WORLD);
188       }
189     MPI_Bcast(& pt, 1, MPI_DOUBLE, me, MPI_COMM_WORLD);
190     // ... todos recebem os valores calculados
191     if (prec == 1) {
192       MPI_Reduce(s, 0, my_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD );
193       }
194     MPI_Reduce(& pt, & p, my_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD );
195     double beta = p/lastp;
196     for (i = me * my_size; i < (me + 1) * my_size ; ++ i) {
197       if (prec == 0) {
198         // d(j+1) = r(j+1) + p(j+1)/p(j) d(j)
199         d[i] = r[i] + beta * d[i];
200         }
201       if (prec == 1) {
202         // d(j+1) = s(j+1) + p(j+1)/p(j) d(j)
203         d[i] = s[i] + beta * d[i];
204         }
205       }
206     // ... todos anunciam os valores calculados
207     MPI_Bcast(d + me * my_size, my_size, MPI_DOUBLE, me, MPI_COMM_WORLD);
208     // ... todos recebem os valores calculados
209     MPI_Reduce(d, 0, my_size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD );
210     }
211   if (me != 0) {
212     return 0;
213     }
214   // ... o mestre exibe a solução
215   double ka, val, max = 0, min = 1e9;
216   printf("Solução após %d iterações: [", j - 1);
217   for (i = 0; i < size ; ++ i) {
218     if (i > 0) {
```

```
219       printf(", ");
220       }
221     printf("%f", x[i]);
222     if (prec == 0) {
223       val = fabs(A[i][i]);
224       if (val > max) max = val;
225       if (val < min) min = val;
226       }
227     }
228   if (prec == 0) {
229     ka = max / min;
230     }
231   if (prec == 1) {
232     ka = 1;
233     }
234   printf("] \nErro < %f, ka = %f", fabs(p), ka);
235   return 0;
236   }
237
238 int load(char * fname, double * dados, int nrow, int ncol) {
239 // Carrega os dados do arquivo "fname" na matriz "dados", de dimensão "nrow" x "ncol"
240   char * pchar, buf [bufsize];
241   FILE * fp = fopen (fname, "r");
242   if (fp == NULL) {
243     printf("Não conseguiu ler o arquivo %s. \n", fname);
244     return 1;
245     }
246   fgets(buf,bufsize,fp);
247   fgets(buf,bufsize,fp);
248   for (int i = 0; i < nrow; ++ i) {
249     fgets(buf,bufsize,fp);
250     for (int k = 0; k < ncol; ++ k) {
251       * dados = fgetval(buf);
252       dados ++ ;
253       }
254     }
255   fclose(fp);
256   return 0;
257   }
258
259
260 double fgetval(char * buffer) {
261 // Extrai o primeiro valor existente no "buffer"
262   char * pchar = buffer, valor [bufsize], * pvalor = valor;
263   while (isblank(*pchar) ) {
264     pchar ++;
265     }
266   while (! isblank(*pchar) )  {
267     * pvalor ++ = * pchar ++;
268     }
269   * pvalor = '\0';
270   strcpy(buffer,pchar);
271   return atof(valor);
272   }
```

4. Faça uma análise comparativa de desempenho de operações de níveis 1, 2 e 3 da BLAS entre implementações suas e equivalentes utilizando a MKL da Intel. Em seu programa utilize um número de repetições que permitam o programa executar aproximadamente 1, 2 e 3 Gflop para os níveis 1, 2 e 3 respectivamente. Mostre como foram feitos estes cálculos de operações de ponto flutuante. Utilize o aplicativo GPROF e o comando time para auxílio de sua análise.

O programa em C listado abaixo implementa as funções **ddot**, **dgemv** e **dgemm** da BLAS. Não se fez nenhuma tentativa séria de obter alto desempenho. O teste utilizou valores aleatórios gerados pelo próprio programa. O nível BLAS, o número de repetições e o tamanho dos vetores e matrizes é passado pela linha de comando, de forma a se atingirem as quantidades de operações desejadas em cada nível. Como não foi empregada paralelização, mediu-se apenas o tempo de usuário, como se vê na tabela seguinte.

LISTING 4. myblas.c

```
1  /*
2  Testa a implementação de funções de diversos níveis da BLAS.
3  Uso:
4    myblas rep n size1 size2 ... sizen
5  onde
6    rep é o número de repetições a executar
7    n é o nível BLAS, um inteiro de 1 a 3
8    size1, size2, sizen são o número de elementos nos vetores e matrizes de teste
9  O programa calcula o número de operações de ponto flutuante teoricamente executadas.
10 */
11 #include <stdlib.h>
12 #include <stdio.h>
13 #include <math.h>
14 #include <sys/time.h>
15 #include <windows.h>
16
17
18 // #define IMPRIMIR
19
20
21 void Init (int size, double * vals) {
22 // Inicializa um conjunto de valores aleatoriamente
23   while (size --- > 0)
24     * vals ++ = rand() % 10;
25   }
26
27 double ddot(double * x, double * y, int size) {
28 // Retorna o produto escalar de dois vetores
29   double sum = 0;
30   while (size --- > 0)
31     sum += (* x ++) * (* y ++);
32   return sum;
33   }
34
35 void dgemv (double * A, double * x, double * y, int nrows, int ncols) {
```

```
36 // Calcula o produto de uma matriz por um vetor
37   while (nrows --> 0) {
38     *y++ = ddot(A, x, ncols);
39     A += ncols;
40     }
41   return ;
42   }
43
44 void dgemm (double *A, double *B, double *C, int nrowsA, int nrowsB, int ncolsC) {
45 // Calcula o produto de duas matrizes
46   double *ptB, *tB = (double *) malloc(nrowsB * ncolsC * sizeof(double));
47   if (tB == NULL) {
48     printf("Falha ao alocar a memória de trabalho necessária");
49     exit(1);
50     }
51   ptB = tB;
52   for (int i = 0 ; i < nrowsB ; ++i) {
53     for (int j = 0 ; j < ncolsC ; ++j) {
54       *ptB++ = *(B + j * ncolsC + i);
55       }
56     }
57   while (nrowsA --> 0) {
58     ptB = tB;
59     for (int i = 0; i < nrowsB; ++i) {
60       *C++ = ddot(A, ptB, ncolsC);
61       ptB += ncolsC;
62       }
63     A += nrowsB;
64     }
65   }
66
67 int main(int argc, char *argv[]) {
68 // Testa a implementação de funções de diversos níveis da BLAS
69   int level = atoi(argv[1]);
70   if (level < 1 || level > 3) {
71     printf("Nível inválido: %d", level);
72     return 1;
73     }
74   int rep = atoi(argv[2]);
75   if (rep <= 0) {
76     printf("Número de repetições inválido: %d", rep);
77     return 1;
78     }
79     HANDLE hProcess = GetCurrentProcess();
80     FILETIME ftCreation, ftExit, ftKernel, ftUser1, ftUser2;
81     SYSTEMTIME stUser1,stUser2;
82   double gflop;
83   if (level == 1) {
84     if (argc != 4 ) {
85       printf("Número inválido de argumentos");
86       return 1;
87       }
88     int ncols = atoi(argv[3]);
89     gflop = rep * ncols/1e9;
90     int mcols = ncols/10;
91     double *x = (double *) malloc(ncols * sizeof(double));
92     double *y = (double *) malloc(ncols * sizeof(double));
93     if (x == NULL || y == NULL) {
94       printf("Falha ao alocar a memória de trabalho necessária");
```

```
95        return 1;
96        }
97      Init(ncols, x);
98      Init(ncols, y);
99      GetProcessTimes(hProcess, &ftCreation, &ftExit, &ftKernel, &ftUser1);
100     for (int i = 0; i < rep; ++ i)
101       double result = ddot(x, y, ncols);
102     GetProcessTimes(hProcess, &ftCreation, &ftExit, &ftKernel, &ftUser2);
103 #ifdef IMPRIMIR
104     printf("[");
105     for (int i = 0; i < ncols; ++ i) {
106       if (i > 0) printf(" ");
107       printf("%f", x[i]);
108       }
109     printf("] .\n[");
110     for (int i = 0; i < ncols; ++ i) {
111       if (i > 0) printf(" ");
112       printf("%f", y[i]);
113       }
114     printf("]\n\t= %f", result);
115 #endif
116     }
117   if (level == 2) {
118     if (argc != 5) {
119       printf("Número inválido de argumentos");
120       return 1;
121       }
122     int nrows = atoi(argv[3]);
123     int ncols = atoi(argv[4]);
124     gflop = rep * (2.0 * ncols - 1) * nrows / 1e9;
125     double * pA, * A = (double *) malloc(nrows * ncols * sizeof(double));
126     double * px, * x = (double *) malloc(ncols * sizeof(double));
127     double * py, * y = (double *) malloc(nrows * sizeof(double));
128     if (A == NULL || x == NULL || y == NULL) {
129       printf("Falha ao alocar a memória de trabalho necessária");
130       return 1;
131       }
132     Init(nrows * ncols, A);
133     Init(ncols, x);
134     GetProcessTimes(hProcess, &ftCreation, &ftExit, &ftKernel, &ftUser1);
135     for (int i = 0; i < rep; ++ i)
136       dgemv(A, x, y, nrows, ncols);
137     GetProcessTimes(hProcess, &ftCreation, &ftExit, &ftKernel, &ftUser2);
138 #ifdef IMPRIMIR
139     pA = A;
140     for (int i = 0; i < nrows; ++ i) {
141       printf("|");
142       for (int j = 0; j < ncols; ++ j) {
143         if (j > 0) printf(" ");
144         printf("%f", * pA ++) ;
145         }
146       printf("|\n");
147       }
148     printf("x [");
149     px = x;
150     for (int i = 0; i < ncols; ++ i) {
151       if (i > 0) printf(" ");
152       printf("%f", * px ++);
153       }
```

```
154     printf("]\n\t= [");
155     py = y;
156     for (int i = 0; i < nrows; ++ i) {
157       if (i > 0) printf(" ");
158       printf("%f" , * py ++);
159       }
160     printf("]\n");
161 #endif
162     }
163   if (level == 3) {
164     if (argc != 6) {
165       printf("Número inválido de argumentos");
166       return 1;
167       }
168     int nrowsA = atoi(argv[3]);
169     int nrowsB = atoi(argv[4]);
170     int ncolsC = atoi(argv[5]);
171     gflop = rep * (2.0 * nrowsB - 1) * nrowsA * ncolsC / 1e9;
172     double * pA, * A = (double *) malloc(nrowsA * nrowsB * sizeof(double));
173     double * pB, * B = (double *) malloc(nrowsB * ncolsC * sizeof(double));
174     double * pC, * C = (double *) malloc(nrowsA * ncolsC * sizeof(double));
175     if (A == NULL || B == NULL || C == NULL) {
176       printf("Falha ao alocar a memória de trabalho necessária");
177       return 1;
178       }
179     Init(nrowsA * nrowsB, A);
180     Init(nrowsB * ncolsC, B);
181     GetProcessTimes(hProcess, &ftCreation, &ftExit, &ftKernel, &ftUser1);
182     for (int i = 0; i < rep; ++ i)
183       dgemm(A, B, C, nrowsA, nrowsB, ncolsC);
184     GetProcessTimes(hProcess, &ftCreation, &ftExit, &ftKernel, &ftUser2);
185 #ifdef IMPRIMIR
186     pA = A;
187     for (int i = 0; i < nrowsA; ++ i) {
188       printf("|");
189       for (int j = 0; j < nrowsB; ++ j) {
190         if (j > 0) printf(" ");
191         printf("%f" , * pA ++);
192         }
193       printf("|\n");
194       }
195     printf("x\n");
196     pB = B;
197     for (int i = 0; i < nrowsB; ++ i) {
198       printf("|");
199       for (int j = 0; j < ncolsC; ++ j) {
200         if (j > 0) printf(" ");
201         printf("%f" , * pB ++);
202         }
203       printf("|\n");
204       }
205     printf("=\n");
206     pC = C;
207     for (int i = 0; i < nrowsA; ++ i) {
208       printf("|");
209       for (int j = 0; j < ncolsC; ++ j) {
210         if (j > 0) printf(" ");
211         printf("%f" , * pC ++);
212         }
```

```
213         printf("|\n");
214         }
215 #endif
216       }
217     FileTimeToSystemTime(& ftUser1, & stUser1);
218     FileTimeToSystemTime(& ftUser2, & stUser2);
219     printf("GFlops = %f, User time = %d ms \n", gflop,
220       stUser2.wSecond * 1000 - stUser1.wSecond * 1000 + stUser2.wMilliseconds - stUser1.
            wMilliseconds);
221     return 0;
222     }
```

Nível BLAS	FLOPs	tempo (s)
1	1.000	3.78
2	1.999	3.69
3	3.032	5.47

5. Utilizando a biblioteca Sparsekit resolva o mesmo sistema do item 2, utilizando os solvers: CG, BiCG, GMRES, BCGSTAB disponíveis na mesma. Analise as respostas com e sem pré-condicionadores do tipo ILU, mostrando número de iterações, critério de parada utilizado os valores do lfill utilizado. Não se esqueça de fazer uma análise comparativa entre os solvers utilizados, apresentando as principais características de cada um.

O programa em C listado abaixo pode utilizar todos os solvers e pré-condicionadores disponíveis na biblioteca Sparskit. A tabela abaixo mostra o resultado apenas para os solvers solicitados no enunciado do problema e os pré-condicionadores de melhor desempenho. Foi usado um valor fixo de 5 para o lfil e os pre-condiconadores foram aplicados à esquerda e à direita em todos os casos.

LISTING 5. sparskit.c

```
1  /*
2  path=%path%;C:\Octave\Octave-4.0.0\bin
3  Solução de sistema de equações lineares (Ax = b) por meio dos solvers interativos da
       biblioteca Sparskit.
4  Uso:
5    solve Afile bfile maxiter erro prec solver
6  onde
7    Afile é o nome do arquivo contendo os valores da matriz A
8    bfile é o nome do arquivo contendo os valores do vetor b
9    maxiter é o número máximo de iterações permitido
10   erro é o erro absoluto tolerado
11   prec é o nome do precondicionador a usar
12   solver é o nome do solver a ser usado
13 Para mais informações, consultar o código fonte da biblioteca (arquivos iters.f, ilut.f
       , matvec.f e formats.f e blassm.f).
14 Listas dos solvers e precondicionadores permitidos encontram-se no código abaixo.
15 Os arquivos formats.f e blassm.f contêm muitas dependências, por isso apenas a parte
       que interessava para esta aplicação foi mantida.
16 Limitações:
17   para o número de condição foi empregada uma fórmula aproximada,
18   não foram utilizadas todas as funcionalidades dos precondicionadores
19 */
20
21
22 #include <stdio.h>
23 #include <stdlib.h>
24 #include <string.h>
25 #include <ctype.h>
26 #include <math.h>
27
28
29 #define bufsize 5000
30 #define size 37
31 #define numsolvers 10
32 #define numprecs 8
33 #define Krylov 15
34 // #define TESTA_PREC
35
```

```
36
37 // Assinatura redundante apenas para compatibilidade com a biblioteca Sparskit
38 extern"C" { double distdot_(int *, double *, int *, double *, int *); }
39 extern"C" { double ddot_(int *, double *, int *, double *, int *); }
40 extern"C" { void daxpy_(int *, double *, double *, int *, double *, int *); }
41 extern"C" { double dnrm2_(int *, double *, int *); }
42
43 // Funções da biblioteca Sparskit (em Fortran)
44 // ... solvers
45 extern"C" { void cg_(int *, double *, double *, int *, double *, double *); }
46 extern"C" { void cgnr_(int *, double *, double *, int *, double *, double *); }
47 extern"C" { void bcg_(int *, double *, double *, int *, double *, double *); }
48 extern"C" { void dbcg_(int *, double *, double *, int *, double *, double *); }
49 extern"C" { void bcgstab_(int *, double *, double *, int *, double *, double *); }
50 extern"C" { void tfqmr_(int *, double *, double *, int *, double *, double *); }
51 extern"C" { void fom_(int *, double *, double *, int *, double *, double *); }
52 extern"C" { void gmres_(int *, double *, double *, int *, double *, double *); }
53 extern"C" { void fgmres_(int *, double *, double *, int *, double *, double *); }
54 extern"C" { void dqgmres_(int *, double *, double *, int *, double *, double *); }
55 // ... BLAS
56 extern"C" { void amux_(int *, double *, double *, double *, int *, int *); }
57 extern"C" { void atmux_(int *, double *, double *, double *, int *, int *); }
58 extern"C" { void amub_(int *, int *, int *, double *, int *, int *, double *, int *,
      int *, double *, int *, int *, int *, int *); }
59 // ... conversões
60 extern"C" { void dnscsr_(int *, int *, int *, double *, int *, double *, int *, int *,
      int *); }
61 extern"C" { void csrdns_(int *, int *, double *, int *, int *, double *, int *, int *);
       }
62 extern"C" { void csrmsr_(int *, double *, int *, int *, double *, int *, double *, int
      *); }
63 extern"C" { void msrcsr_(int *, double *, int *, double *, int *, int *, double *, int
      *); }
64 // ... ILU
65 extern"C" { void ilut_(int *, double *, int *, int *, int *, double *, double *, int *,
       int *, int *, double *, int *, int *); }
66 extern"C" { void ilud_(int *, double *, int *, int *, double *, double *, double *, int
       *, int *, int *, double *, int *, int *); }
67 extern"C" { void ilutp_(int *, double *, int *, int *, int *, double *, double *, int
      *, double *, int *, int *, int *, double *, int *, int *, int *); }
68 extern"C" { void iludp_(int *, double *, int *, int *, double *, double *, double *,
      int *, double *, int *, int *, int *, double *, int *, int *, int *); }
69 extern"C" { void iluk_(int *, double *, int *, int *, int *, double *, int *, int *,
      int *, int *, double *, int *, int *); }
70 extern"C" { void ilu0_(int *, double *, int *, int *, double *, int *, int *, int *,
      int *); }
71 extern"C" { void milu0_(int *, double *, int *, int *, double *, int *, int *, int *,
      int *); }
72 extern"C" { void lusol_(int *, double *, double *, double *, int *, int *); }
73 extern"C" { void lutsol_(int *, double *, double *, double *, int *, int *); }
74 extern"C" { void pgmres_(int *, int *, double *, double *, double *, double *, int *,
      int *, double *, int *, int *, double *, int *, int *, int *); }
75
76 // Wrappers para chamada de precondicionadores da biblioteca
77 void myilut(int *, double *, int *, int *, double *, int *, int *, int *, double *, int
       *, int *);
78 void myilud(int *, double *, int *, int *, double *, int *, int *, int *, double *, int
       *, int *);
```

```
79  void myilutp(int *, double *, int *, int *, double *, int *, int *, int *, double *,
        int *, int *);
80  void myiludp(int *, double *, int *, int *, double *, int *, int *, int *, double *,
        int *, int *);
81  void myiluk(int *, double *, int *, int *, double *, int *, int *, int *, double *, int
         *, int *);
82  void myilu0(int *, double *, int *, int *, double *, int *, int *, int *, double *, int
         *, int *);
83  void mymilu0(int *, double *, int *, int *, double *, int *, int *, int *, double *,
        int *, int *);
84
85  // Cálculo do espaço de Krylov
86  int Ksize1();
87  int Ksize2();
88  int Ksize3();
89
90
91  typedef void SolverFn(int *, double *, double *, int *, double *, double *);
92  typedef int KrylovFn(void);
93  typedef void PrecFn(int *, double *, int *, int *, double *, int *, int *, int *,
        double *, int *, int *);
94  typedef struct {
95    const char * nome;
96    SolverFn * function;
97    int wsize;
98    KrylovFn * kfn ;
99    } SolverInfo;
100 typedef struct {
101   const char * nome;
102   PrecFn * function;
103   int mode;
104   } PrecInfo;
105
106
107 // Solvers implementados
108 static SolverInfo Solver[] = {
109   { "CG",       cg_,     5,  NULL, },
110   { "CGNR",     cgnr_,     5,  NULL, },
111   { "BCG",      bcg_,    7,  NULL, },
112   { "DBCG",     dbcg_,     11, NULL, },
113   { "BCGSTAB",  bcgstab_, 8,  NULL, },
114   { "TFQMR",      tfqmr_,    11, NULL, },
115   { "FOM",      fom_,    0,  Ksize1, },
116   { "GMRES",      gmres_,    0,  Ksize1, },
117   { "FGMRES",     fgmres_,   0,  Ksize2, },
118   { "DQGMRES",  dqgmres_, 0,  Ksize3, }
119   };
120
121 // Precondicionadores implementados
122 static PrecInfo Preconditioner[] = {
123   { "0",        NULL,    0, },
124   { "ILUT",     myilut,    2, },
125   { "ILUTP",      myilutp,   2, },
126   { "ILUD",     myilud,    2, },
127   { "ILUDP",      myiludp,   2, },
128   { "ILUK",     myiluk,    2, },
129   { "ILU0",     myilu0,    2, },
130   { "MILU0",      mymilu0,   2, }
131   };
```

```
132
133
134 // Funções de apoio (BLAS nível 1)
135 double distdot_(int * psize, double * x, int * ix, double * y, int * iy) {
136 // Calcula o produto escalar de dois vetores.
137 // Assinatura pre-definida pela biblioteca Sparskit.
138   double sum = 0;
139   int vsize = * psize;
140   if (* ix != 1 || * iy != 1) {
141     printf("Situação não tratada (inesperada) no cálculo do produto escalar");
142     exit(1);
143     }
144   while (vsize --> 0) {
145     sum += (* x ++) * (* y ++);
146     }
147   return sum;
148   }
149
150 double ddot_(int * psize, double * x, int * ix, double * y, int * iy) {
151 // Calcula o produto escalar de dois vetores.
152 // Assinatura pre-definida pela biblioteca Sparskit.
153   return distdot_(psize, x, ix, y, iy) ;
154   }
155
156 double dnrm2_(int * psize, double * x, int * ix) {
157 // Calcula a norma Euclideana de um vetor
158 // Assinatura pre-definida pela biblioteca Sparskit.
159     double sum = 0;
160   int vsize = * psize;
161   if (* ix != 1) {
162     printf("Situação não tratada (inesperada) no cálculo do produto escalar");
163     exit(1);
164     }
165   while (vsize --> 0) {
166     sum += (* x) * (* x);
167     ++ x;
168     }
169   return sqrt(sum);
170   }
171
172 void daxpy_(int * psize, double * pa, double * x, int * ix, double * y, int * iy) {
173 // Calcula y = ax + y, onde a é um escalar e x e y são vetores
174 // Assinatura pre-definida pela biblioteca Sparskit.
175     double sum = 0;
176   int vsize = * psize, a = * pa;
177   if (* ix != 1 || * iy != 1) {
178     printf("Situação não tratada (inesperada) no cálculo do produto escalar");
179     exit(1);
180     }
181   while (vsize --> 0) {
182     * y ++ += a * (* x ++);
183     }
184   }
185
186
187 // Funções para leitura de dados em disco
188 double fgetval(char * buffer) {
189 // Extrai o primeiro valor existente no "buffer"
190   char * pchar = buffer, valor [bufsize], * pvalor = valor;
```

```
191   while (isblank(*pchar) ) {
192     pchar ++;
193     }
194   while (! isblank(*pchar) )  {
195     * pvalor ++ = * pchar ++;
196     }
197   * pvalor = '\0';
198   strcpy(buffer,pchar);
199   return atof(valor);
200   }
201
202 int load(char * fname, double * dados, int nrow, int ncol) {
203 // Carrega os dados do arquivo "fname" na matriz "dados", de dimensão "nrow" x "ncol"
204   char * pchar, buf [bufsize];
205   FILE * fp = fopen (fname, "r");
206   if (fp == NULL) {
207     printf("Não conseguiu ler o arquivo %s. \n", fname);
208     return 1;
209     }
210   fgets(buf,bufsize,fp);
211   fgets(buf,bufsize,fp);
212   for (int i = 0; i < nrow; ++ i) {
213     fgets(buf,bufsize,fp);
214     for (int k = 0; k < ncol; ++ k) {
215       * dados = fgetval(buf);
216       dados ++ ;
217       }
218     }
219   fclose(fp);
220   return 0;
221   }
222
223
224 // Funções específicas
225 // Wrappers para chamada de precondicionadores da biblioteca
226 void myilut(int * n, double * a, int * ja, int * ia, double * alu, int * jlu, int * ju,
        int * iwk, double * w, int * jw, int * ierr) {
227   int lfil = 5;
228   double droptol = 1e-3;
229     ilut_(n, a, ja, ia, & lfil, & droptol, alu, jlu, ju, iwk, w, jw, ierr);
230   }
231
232 void myilud(int * n, double * a, int * ja, int * ia, double * alu, int * jlu, int * ju,
        int * iwk, double * w, int * jw, int * ierr) {
233   double alph = 0.5, tol = 1e-3;
234     ilud_(n, a, ja, ia, & alph, & tol, alu, jlu, ju, iwk, w, jw, ierr);
235   }
236
237 void myilutp(int * n, double * a, int * ja, int * ia, double * alu, int * jlu, int * ju
      , int * iwk, double * w, int * jw, int * ierr) {
238   int mbloc = * n, lfil = 5;
239   int * iperm = (int *) malloc(2 * (*n) * sizeof(int));
240   if (iperm == NULL) {
241     printf("Não conseguiu alocar a memória de trabalho necessária para o
          precondicionador");
242     exit(1);
243     }
244   double permtol = 0, droptol = 1e-3;
```

```
245     ilutp_(n, a, ja, ia, &lfil, &droptol, &permtol, &mbloc, alu, jlu, ju, iwk, w,
            jw, iperm, ierr);
246   }
247
248 void myiludp(int * n, double * a, int * ja, int * ia, double * alu, int * jlu, int * ju
      , int * iwk, double * w, int * jw, int * ierr) {
249   int mbloc = * n;
250   int * iperm = (int *) malloc(2 * (*n) * sizeof(int));
251   if (iperm == NULL) {
252     printf("Não conseguiu alocar a memória de trabalho necessária para o
            precondicionador");
253     exit(1);
254     }
255   double alph = 0.5, tol = 1e-3, permtol = 0;
256     iludp_(n, a, ja, ia, &alph, &tol, &permtol, &mbloc, alu, jlu, ju, iwk, w, jw,
            iperm, ierr);
257   }
258
259 void myiluk(int * n, double * a, int * ja, int * ia, double * alu, int * jlu, int * ju,
       int * iwk, double * w, int * jw, int * ierr) {
260   int lfil = 5, * levs = (int *) malloc((*iwk) * sizeof(int));
261   if (levs == NULL) {
262     printf("Não conseguiu alocar a memória de trabalho necessária para o
            precondicionador");
263     exit(1);
264     }
265     iluk_(n, a, ja, ia, &lfil, alu, jlu, ju, levs, iwk, w, jw, ierr);
266   }
267
268 void myilu0(int * n, double * a, int * ja, int * ia, double * alu, int * jlu, int * ju,
       int * iwk, double * w, int * jw, int * ierr) {
269     ilu0_(n, a, ja, ia, alu, jlu, ju, jw, ierr);
270   }
271
272 void mymilu0(int * n, double * a, int * ja, int * ia, double * alu, int * jlu, int * ju
      , int * iwk, double * w, int * jw, int * ierr) {
273     milu0_(n, a, ja, ia, alu, jlu, ju, jw, ierr);
274   }
275
276
277 void teste_ilu(double * b, double * x, double maxerr, int maxiter, double * a, int * ja
      , int * ia, double * alu, int * jlu, int * ju) {
278 // Testa o resultado do precondicionamento.
279 // Apenas para fins de desenvolvimento.
280   int zero = 1, n = size, retcode, wksize = Krylov;
281   double * pvv = (double *) malloc((wksize + 1) * size * sizeof(double));
282   if (pvv == NULL) {
283     printf("Não conseguiu alocar a memória de trabalho necessária para o solver de
            teste");
284     exit(1);
285     }
286   pgmres_(& n, & wksize, b, x, pvv, & maxerr, & maxiter, & zero, a, ja, ia, alu, jlu,
          ju, & retcode);
287   if (retcode != 0) {
288     printf("Erro no cálculo do solver de teste PGMRES: %d", retcode);
289     exit(1);
290     }
291   printf("Solução do solver de teste PGMRES: [");
292   for (int i = 0; i < size ; ++ i) {
```

```
293       if (i > 0) {
294         printf(", ");
295         }
296       printf("%f", x[i]);
297       }
298     }
299
300
301   void getprec(double * a, int * ja, int * ia, PrecInfo * pprec, double * alu, int * jlu,
          int * ju) {
302   // Obtém o precondicionador em formato MSR (modified sparse row)
303     double * wp = (double *) malloc((size + 1) * sizeof(double));
304     int * jwp = (int *) malloc(2 * size * sizeof(int));
305     int n = size, n2 = n * n + 100, retcode;
306     (pprec -> function) (& n, a, ja, ia, alu, jlu, ju, & n2, wp, jwp, & retcode);
307     if (retcode != 0) {
308       printf("Erro no cálculo do precondicionador: %d", retcode);
309       exit(1);
310       }
311     free(wp); free(jwp);
312     }
313
314
315   // Cálculo do espaço de Krylov
316   int Ksize1() {
317     return (size + 3) * (Krylov + 2) + Krylov * ( Krylov + 1) / 2.0 + 1;
318     }
319
320   int Ksize2() {
321     return 2 * size * (Krylov + 1) + Krylov * ( Krylov + 1) / 2.0 + 3 * Krylov + 3;
322     }
323
324   int Ksize3() {
325     return size + (Krylov + 1) * (2 * size + 4);
326     }
327
328   int solve(double * a, int * ja, int * ia, double * b, double * x, int maxiter, double
        maxerr, int idxprec, int idxsolver) {
329   // Resolve o sistema linear ax = b
330   // a está na forma CSR
331   // Utiliza diversas combinações de solvers e precondicionadores
332     // Inicializa o precondicionador, se houver
333     PrecInfo * pprec = & Preconditioner[idxprec];
334     double * alu;
335     int * jlu, * ju;
336     if (idxprec > 0) {
337       alu = (double *) malloc(size * size * sizeof(double));
338       jlu = (int *) malloc(size * size * sizeof(int));
339       ju = (int *) malloc(size * sizeof(int));
340       if (alu == NULL || jlu == NULL || ju == NULL) {
341         printf("Não conseguiu alocar a memória de trabalho necessária para o
              precondicionador");
342         exit(1);
343         }
344       getprec(a, ja, ia, pprec, alu, jlu, ju);
345   #ifdef TESTA_PREC
346       teste_ilu(b, x, maxerr, maxiter, a, ja, ia, alu, jlu, ju);
347       exit(0);
348   #endif
```

```
349     }
350   // Inicializa o solver
351   int retcode, n = size, one = 1;
352   int ipar[16];
353   double fpar[16], flop;
354   SolverInfo * psolver = & Solver[idxsolver];
355   int wksize = size * psolver -> wsize;
356   if (wksize <= 0) {
357     wksize = (psolver -> kfn)();
358     }
359   double * ws = (double *) malloc(wksize * sizeof(double));
360   if (ws ==  NULL) {
361     printf("Não conseguiu alocar a memória de trabalho necessária para o solver");
362     return 1;
363     }
364   fpar[0] = maxerr;
365   fpar[1] = maxerr * 1e-4;
366   fpar[10] = flop = 0;
367   ipar[0] = 0;
368   ipar[1] = pprec -> mode;
369   ipar[2] = 1;
370   ipar[3] = wksize;
371   ipar[4] = Krylov;
372   ipar[5] = maxiter;
373   ipar[6] = 0;
374   // Executa o solver interativamente
375   do {
376     (psolver -> function) (& n, b, x, ipar, fpar, ws);
377     retcode = ipar[0];
378     switch (retcode) {
379       case 1:
380         amux_(& n, ws + ipar[7] - 1, ws + ipar[8] - 1, a, ja, ia ) ;
381         flop += (2 * size - 1) * size;
382         printf(".");
383         break;
384       case 2:
385         atmux_(& n, ws + ipar[7] - 1, ws + ipar[8] - 1, a, ja, ia ) ;
386         flop += (2 * size - 1) * size;
387         printf("|");
388         break;
389       case 3:
390       case 5:
391         if (idxprec == 0) {
392           printf("Situação não tratada (inesperada): condicionador chamado");
393           exit(1);
394           }
395         lusol_(& n, ws + ipar[7] - 1, ws + ipar[8] - 1, alu, jlu, ju);
396         flop += (2 * size - 1) * size;
397         printf("+");
398         break;
399       case 4:
400       case 6:
401         if (idxprec == 0) {
402           printf("Situação não tratada (inesperada): condicionador chamado");
403           exit(1);
404           }
405         lutsol_(& n, ws + ipar[7] - 1, ws + ipar[8] - 1, alu, jlu, ju);
406         flop += (2 * size - 1) * size;
407         printf("-");
```

```
408         break;
409       default:
410         if (retcode > 0) {
411           printf("Situação não tratada (inesperada): retcode = %d\n", retcode);
412           exit(1);
413           }
414         }
415     } while (retcode > 0);
416   if (retcode < 0) {
417     printf("Não obteve a solução após %d iterações. Erro = %d\n", ipar[6], retcode);
418     return 1;
419     }
420   // Exibe a solução
421   double ka, val, max = 0, min = 1e9;
422   double gflop = (flop + fpar[10]) / 1e9;
423   printf("Solução após %d iterações (%f GFlops): [", ipar[6], gflop);
424   for (int i = 0; i < size ; ++ i) {
425     if (i > 0) {
426       printf(", ");
427       }
428     printf("%f", x[i]);
429     }
430   if (idxprec == 0) {
431     ka = max / min;
432     }
433   if (idxprec == 1) {
434     ka = 1;
435     }
436   printf("] \nErro < %f, ka = %f", fpar[5], ka);
437   return 0;
438   }
439
440
441 int findsolver(const char * nome) {
442   for (int i = 0; i < numsolvers; ++ i) {
443     if (! strcmp (nome, Solver[i].nome))
444       return i;
445     }
446   return -1;
447   }
448
449
450 int findprec(const char * nome) {
451   for (int i = 0; i < numprecs; ++ i) {
452     if (! strcmp (nome, Preconditioner[i].nome))
453       return i;
454     }
455   return -1;
456   }
457
458
459 int convert(double * A, double ** pA, int ** pja, int ** pia) {
460 // Converte uma matriz para o formato CSR (compressed sparse row)
461   int retcode, n = size, nmax = 0;
462   * pA = A + size * size - 1;
463   while (* pA >= A) {
464     nmax += (** pA != 0) ;
465     -- (* pA);
466     }
```

```
467   * pA = (double *) malloc(nmax * sizeof(double));
468   * pja = (int *) malloc(nmax * sizeof(int));
469   * pia = (int *) malloc((size + 1) * sizeof(int));
470   if (* pA == NULL || * pja == NULL || * pia == NULL) {
471     printf("Não conseguiu alocar memória para a matriz convertida");
472     exit(1);
473     }
474   dnscsr_(& n, & n, & nmax, A, & n, * pA, * pja, * pia, & retcode);
475   if (retcode != 0) {
476     printf("Erro %d ao converter matriz para o formato CSR", retcode);
477     exit(1);
478     }
479   return nmax;
480   }
481
482
483 int main(int argc, char * argv[]) {
484 // Solução de sistema de equações lineares (Ax = b) por meio dos solvers da biblioteca
      Sparskit.
485   // Carrega os dados a partir dos arquivos em disco
486   double A[size][size], b[size], x[size];    // Ax = b
487   int noread;
488   noread = load(argv[1], &A[0][0], size, size);
489   if (noread > 0) {
490     return 1;
491     }
492   noread = load(argv[2], b, size, 1);
493   if (noread > 0) {
494     return 1;
495     }
496   // Lê os demais parâmetros
497   int maxiter = atoi(argv[3]) ;
498   if (maxiter <= 0) {
499     printf("Número máximo de iterações inválido: %d", maxiter);
500     return 1;
501     }
502   double erro = atof(argv[4]);
503   if (erro <= 0) {
504     printf("Erro tolerado inválido: %f", erro);
505     return 1;
506     }
507   int idxprec = findprec(argv[5]);
508   if (idxprec < 0) {
509     printf("Precondicionador desconhecido: %s", argv[5]);
510     return 1;
511     }
512   int idxsolver = findsolver(argv[6]);
513   if (idxsolver < 0 ) {
514     printf("Solver desconhecido: %s", argv[6]);
515     return 1;
516     }
517   // Inicializa o vetor solução (x0)
518   for (int i = 0; i < size ; ++ i) {
519     // (teoricamente, pode ser qualquer coisa)
520     x[i] = 0;
521     }
522   // Converte a matriz de coeficientes para o formato CSR
523   double * pA;
524   int * pja, * pia;
```

```
525   int nmax = convert(& A[0][0], & pA, & pja, & pia);
526   printf("nmax = %d (%s esparsa)\n", nmax, (nmax * 1.0 / (size * size)) > 0.5 ? "pouco"
          : "muito");
527   // Resolve o sistema
528   solve(pA, pja, pia, b, x, maxiter, erro, idxprec, idxsolver);
529   }
```

Solver	Iterações	Erro	Precond.
CG	5	10^{-6}	-
BiCG	8	10^{-6}	-
GMRES	5	10^{-6}	-
BCGSTAB	9	10^{-6}	-
CG	5	10^{-7}	-
BiCG	8	10^{-7}	-
GMRES	5	10^{-7}	-
BCGSTAB	9	10^{-7}	-
CG	5	10^{-8}	-
BiCG	8	10^{-8}	-
GMRES	5	10^{-8}	-
BCGSTAB	9	10^{-8}	-
CG	29	10^{-9}	-
BiCG	56	10^{-9}	-
GMRES	14	10^{-9}	-
BCGSTAB	-	10^{-9}	-
CG	43	10^{-10}	-
BiCG	120	10^{-10}	-
GMRES	29	10^{-10}	-
BCGSTAB	-	10^{-10}	-
CG	-	10^{-6}	ILUT
BiCG	520	10^{-6}	ILUT
GMRES	-	10^{-6}	ILUT
BCGSTAB	-	10^{-6}	ILUT
CG	-	10^{-7}	ILUT
BiCG	-	10^{-7}	ILUT
GMRES	-	10^{-7}	ILUT
BCGSTAB	-	10^{-7}	ILUT
CG	3	10^{-6}	ILUD
BiCG	96	10^{-6}	ILUD
GMRES	3	10^{-6}	ILUD
BCGSTAB	3	10^{-6}	ILUD
CG	4	10^{-7}	ILUD
BiCG	128	10^{-7}	ILUD
GMRES	4	10^{-7}	ILUD
BCGSTAB	5	10^{-7}	ILUD
CG	-	10^{-8}	ILUD
BiCG	142	10^{-8}	ILUD
GMRES	8	10^{-8}	ILUD
BCGSTAB	179	10^{-8}	ILUD
CG	-	10^{-9}	ILUD
BiCG	152	10^{-9}	ILUD
GMRES	-	10^{-9}	ILUD
BCGSTAB	-	10^{-9}	ILUD
CG	-	10^{-10}	ILUD
BiCG	174	10^{-10}	ILUD
GMRES	-	10^{-10}	ILUD
BCGSTAB	-	10^{-10}	ILUD

Solver	Iterações	Erro	Precond.
CG	-	10^{-6}	ILUDP
BiCG	6	10^{-6}	ILUDP
GMRES	3	10^{-6}	ILUDP
BCGSTAB	7	10^{-6}	ILUDP
CG	-	10^{-7}	ILUDP
BiCG	44	10^{-7}	ILUDP
GMRES	4	10^{-7}	ILUDP
BCGSTAB	55	10^{-7}	ILUDP
CG	-	10^{-8}	ILUDP
BiCG	48	10^{-8}	ILUDP
GMRES	7	10^{-8}	ILUDP
BCGSTAB	81	10^{-8}	ILUDP
CG	-	10^{-9}	ILUDP
BiCG	48	10^{-9}	ILUDP
GMRES	14	10^{-9}	ILUDP
BCGSTAB	87	10^{-9}	ILUDP
CG	-	10^{-10}	ILUDP
BiCG	70	10^{-10}	ILUDP
GMRES	93	10^{-10}	ILUDP
BCGSTAB	137	10^{-10}	ILUDP
CG	-	10^{-6}	ILUTP
BiCG	520	10^{-6}	ILUTP
GMRES	-	10^{-6}	ILUTP
BCGSTAB	-	10^{-6}	ILUTP
CG	-	10^{-7}	ILUTP
BiCG	-	10^{-7}	ILUTP
GMRES	-	10^{-7}	ILUTP
BCGSTAB	-	10^{-7}	ILUTP
CG	2	10^{-6}	ILUK
BiCG	2	10^{-6}	ILUK
GMRES	2	10^{-6}	ILUK
BCGSTAB	3	10^{-6}	ILUK
CG	2	10^{-7}	ILUK
BiCG	2	10^{-7}	ILUK
GMRES	2	10^{-7}	ILUK
BCGSTAB	3	10^{-7}	ILUK
CG	2	10^{-8}	ILUK
BiCG	2	10^{-8}	ILUK
GMRES	2	10^{-8}	ILUK
BCGSTAB	3	10^{-8}	ILUK
CG	2	10^{-9}	ILUK
BiCG	2	10^{-9}	ILUK
GMRES	2	10^{-9}	ILUK
BCGSTAB	3	10^{-9}	ILUK
CG	2	10^{-10}	ILUK
BiCG	2	10^{-10}	ILUK
GMRES	2	10^{-10}	ILUK
BCGSTAB	3	10^{-10}	ILUK

A tabela mostra que:

1) Todos os solvers são capazes de resolver o problema proposto se a precisão exigida não for muito grande.
2) À medida que a precisão requerida aumenta, cresce o número de iterações necessárias para a solução e, eventualmente, alguns solvers não conseguem atingir o objetivo.
3) O emprego de precondicionadores, em geral, faz com que a solução seja atingida com menos iterações, mas algumas combinações solver/precondicionador não funcionam bem.
4) O melhor solver parece ser o GMRES, que funciona bem com vários precondicionadores e consegue atingir boa precisão com poucas iterações.
5) O melhor precondicionador parece ser o ILUK, que funciona bem com todos os solvers e atinge alta precisão com muito poucas iterações.