```
1  ! CGV/skel/cgv_02.F90
2
3  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
4  !                                                                     !
5  ! This file has been written as a sample solution to an             !
6  ! exercise in a course given at the High Performance                 !
7  ! Computing Centre Stuttgart (HLRS).                                 !
8  ! It is made freely available with the understanding that           !
9  ! every copy of this file must include this header and that         !
10 ! HLRS takes no responsibility for the use of the enclosed          !
11 ! teaching material.                                                !
12 !                                                                    !
13 ! Authors: Dr. Rolf Rabenseifner                                    !
14 !          Stephan Scheiderer                                       !
15 !                                                                    !
16 ! Contact: rabenseifner@hlrs.de                                     !
17 !                                                                    !
18 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
19                                                                !EOH
20
21 ! >>>>>>>>>>>>>>>>>>>>> begin of task 00 <<<<<<<<<<<<<<<<<<<<<
22
23 PROGRAM cgv_fortran
24
25 #ifndef serial
26   INCLUDE 'mpif.h'
27 #endif
28
29 ! ****************** declarations and global data ***************
30
31 ! --- options: ---
32 INTEGER :: m,n         ! size of the physical-heat matrix without borders
33                        ! m=vertical, n=horizontal
34 INTEGER :: iter_max    ! maximal number of iterations in CG
35 REAL :: epsilon        ! Limit for sqr( || residual vector || )
36 INTEGER :: print_level ! print_level = 1: Only result-error and partial result matrix
37                                 ! = 2: and residual norm after each iteration
38                                 ! = 3: and result physical-heat matrix
39                                 ! = 4: and all vector and halo data in first iteration
40                                 ! = 5: and all vector and halo data in _all_ iterations
41 INTEGER :: decomp_dims      ! dimensins of the domain decomposition
42 INTEGER :: m_procs, n_procs ! number of processes in both directions of the 2-dim domain decomposition
43 INTEGER :: mprt, nprt       ! number of rows and columns when printing the partial result matrix
44
45 ! global structures and data:
46
47 #ifndef serial
48   INTEGER :: ierror, comm_cart
49   DOUBLE PRECISION :: commtime_start=0, commtime_end=0, commtime_sum=0
50 #endif
51 INTEGER :: numprocs, my_rank
52 REAL :: flops
53
54 ! size of the physical area this process is about to calculate
55 ! start_i   : start row
56 ! end1_i    : end row + 1
57 ! start_j   : start column
58 ! end1_j    : end column + 1
59 ! rowsize   : number of data entries in a row of the local chunk
60 !             (end1_j - start_j)
61 ! colsize   : number of data entries in a column of the local chunk
62 !             (end1_i - start_i)
63 ! chunksize : number of data entries in the local chunk PLUS halo data
64 !             ((rowsize + 2) * (colsize + 2))
65 INTEGER :: start_i, end1_i, start_j, end1_j, rowsize, colsize, chunksize
66
67 ! SIDX (Serial InDeX): computes the serial index of a two-dimensional array
68 !   input: i = 0...(colsize + 1) and j = 0...(rowsize + 1)
69 !   output: SIDX in 0..((colsize + 2) * (rowsize + 2) - 1)
70 #define SIDX(i, j) ((rowsize + 2) * (i) + (j))
71
72 ! --- data structures ---
73
74 ! halo structure
75 TYPE halo_struct
76     REAL, DIMENSION(:), POINTER :: north
77     INTEGER                     :: north_rank
78     INTEGER                     :: north_size
79     REAL, DIMENSION(:), POINTER :: east
80     INTEGER                     :: east_rank
81     INTEGER                     :: east_size
82     REAL, DIMENSION(:), POINTER :: south
83     INTEGER                     :: south_rank
84     INTEGER                     :: south_size
85     REAL, DIMENSION(:), POINTER :: west
86     INTEGER                     :: west_rank
87     INTEGER                     :: west_size
88 END TYPE halo_struct
89 TYPE(halo_struct) :: halo
90
91 ! needed global vectors
92 REAL, DIMENSION(:), ALLOCATABLE :: x, b
93 ! needed send and receive scratch buffers
94 REAL, DIMENSION(:), ALLOCATABLE :: recv_scratch_buff_east, recv_scratch_buff_west, &
95                                    & recv_scratch_buff_north, recv_scratch_buff_south
96
97 ! >>>>>>>>>>>>>>>>>>>>> -end- of task 00 <<<<<<<<<<<<<<<<<<<<<
```

```
98  !>>>>>>>>>>>>>>>>>>>>>> begin of task 08 <<<<<<<<<<<<<<<<<<<<<<
99
100 !********************** main program ************************
101
102     INTEGER                           :: num_iter
103     REAL                              :: norm_r, norm_err, maxnorm_err
104     REAL, DIMENSION(:), ALLOCATABLE :: u, e
105     REAL                              :: time_start_cpu, time_end_cpu, time_elapsed_cpu
106 #ifndef serial
107     DOUBLE PRECISION :: time_start=0, time_end=0
108     INTEGER :: dims(0:1)
109     INTEGER :: percentage
110 #endif
111     REAL :: mflops = 0, global_mflops = 0, time_elapsed, global_mflops_cpu
112
113 #ifdef serial
114     numprocs = 1
115     my_rank = 0
116 #endif
117 #ifndef serial
118     CALL MPI_INIT(ierror)
119     CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierror)
120     CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierror)
121 #endif
122
123     flops = 0
124     m = 4
125     n = 4
126     iter_max = 500
127     epsilon = 1e-6
128     print_level = 1
129     decomp_dims = 1
130     mprt = 11
131     nprt = 11
132     m_procs=1
133     n_procs=1
134 #ifndef serial
135     dims(0) = 0
136     dims(1) = 0
137     CALL MPI_DIMS_CREATE(numprocs, 2, dims, ierror)
138     m_procs = dims(0)
139     n_procs = dims(1)
140 #endif
141
142     ! Parse command line options
143     CALL read_options()
144
145     ! Init the domain decomposition
146     CALL Init_Decomp()
147     IF (print_level >= 1) THEN
148         CALL print_decomp()
149     END IF
150
151     ! reserve chunk of vector b and assign border data
152     ALLOCATE(b(0:chunksize-1))
153     b = 0
154     CALL Init_Boundary_Vec(b)
155
156     IF (print_level>=4) THEN
157         CALL print_vec('b',b)
158     END IF
159
160     ! x is initialized with zeros
161     ALLOCATE(x(0:chunksize-1))
162     x = 0
163
164 #ifndef serial
165     time_start = MPI_WTIME()
166 #endif
167     CALL CPU_TIME(time_start_cpu)
168
169     CALL CG_solve(num_iter, norm_r)
170
171 #ifndef serial
172     time_end = MPI_WTIME()
173     time_elapsed = time_end - time_start
174     mflops = flops / (1000000 * time_elapsed)
175     CALL MPI_REDUCE(mflops, global_mflops, 1, MPI_REAL, MPI_SUM, 0, comm_cart, ierror)
176 #endif
177     CALL CPU_TIME(time_end_cpu)
178     time_elapsed_cpu = time_end_cpu - time_start_cpu
179     global_mflops_cpu = flops / (1000000 * time_elapsed_cpu)
180 #ifndef serial
181     global_mflops_cpu = global_mflops
182 #endif
183
184     ALLOCATE(u(0:chunksize-1))
185     u = 0
186     ALLOCATE(e(0:chunksize-1))
187     e = 0
188     CALL Init_Exact_Solution_vec(u)
189     ! p = u - x
190     CALL add_vec_vec(u, -1.0, x, e)
191     norm_err = SQRT(sqr_norm_vec(e))
192     maxnorm_err = max_norm_vec(e)
193
194     IF (print_level>=3) THEN
```

```
195         CALL print_vec2d('x',x,m,n)
196     END IF
197
198     IF (my_rank==0) THEN
199         WRITE(*,'(A, I5, A, E8.2, A, E8.2)') 'main: ', num_iter, ' iterations, sqrt(eps) = ', SQRT(epsilon), &
200                 & ', normalized: ', SQRT(epsilon/(n*m))
201         WRITE(*,'(A, E8.2, A, E8.2)') 'main:          ||r||    = ', norm_r, ', normalized: ', norm_r/SQRT(REAL (n*m))
202         WRITE(*,'(A, E8.2, A, E8.2)') 'main:          ||error|| = ', norm_err, ', normalized: ', norm_err/SQRT(REAL (n*m))
203         WRITE(*,'(A, E8.2, A, E8.2)') 'main:          max(error)= ', maxnorm_err, ' = normalized: ', maxnorm_err
204         WRITE(*,'(A, I, A, I, A, I)') 'main:        n*m = ', n, '*', m, ' = ', n*m
205         WRITE(*,'(A, E8.2, A)') 'main:          ', time_elapsed_cpu, ' sec (exec. time CPU_TIME)'
206 #ifndef serial
207         WRITE(*,'(A, E8.2, A)') 'main:          ', time_elapsed, ' sec (exec. time MPI_WTIME)'
208         WRITE(*,'(A, E8.2, A)') 'main:          ', commtime_sum, ' sec (comm. time)'
209         percentage = commtime_sum * 100.0 / time_elapsed
210         WRITE(*,'(A, I2, A)') 'main: ratio (comm. time) / (exec. time): ', percentage, '%'
211 #endif
212         WRITE(*,'(A, F8.2)') 'Calculated MFLOPS (avg) CPU_TIME: ', global_mflops_cpu / numprocs
213         WRITE(*,'(A, F8.2)') 'Calculated MFLOPS (all) CPU_TIME: ', global_mflops_cpu
214 #ifndef serial
215         WRITE(*,'(A, F8.2)') 'Calculated MFLOPS (avg) MPI_WTIME: ', global_mflops / numprocs
216         WRITE(*,'(A, F8.2)') 'Calculated MFLOPS (all) MPI_WTIME: ', global_mflops
217 #endif
218     END IF
219
220     IF (print_level>=1) THEN
221         CALL print_vec2d('x',x,mprt,nprt)
222     END IF
223
224 #ifndef serial
225     CALL MPI_FINALIZE(ierror)
226 #endif
227
228 ! beginning of the subroutine part
229 CONTAINS
230
231 ! ********************** memory management **********************
232
233 SUBROUTINE abrt
234 #ifndef serial
235     CALL MPI_ABORT(MPI_COMM_WORLD, -1, ierror)
236 #endif
237     STOP
238 END SUBROUTINE abrt
239
240 ! ********************** commandline options *******************
241
242 SUBROUTINE read_options()
243     INTEGER :: optid = 1 ! start at first optional argument
244     INTEGER :: argc
245     LOGICAL :: err_detected = .FALSE.
246     CHARACTER(len=255) :: arg
247     argc = iargc()
248     IF ( my_rank == 0 ) THEN
249         DO WHILE( optid <= argc )
250             CALL getarg(optid, arg)
251             IF (arg == '-m') THEN
252                 optid = optid + 1
253                 CALL getarg(optid, arg)
254                 READ (arg, *) m
255                 IF (m<=0) THEN
256                     err_detected = .TRUE.
257                     EXIT
258                 END IF
259             ELSE IF (arg == '-n') THEN
260                 optid = optid + 1
261                 CALL getarg(optid, arg)
262                 READ (arg, *) n
263                 IF (n<=0) THEN
264                     err_detected = .TRUE.
265                     EXIT
266                 ENDIF
267             ELSE IF (arg == '-imax') THEN
268                 optid = optid + 1
269                 CALL getarg(optid, arg)
270                 READ (arg, *) iter_max
271                 IF (iter_max<=0) THEN
272                     err_detected = .TRUE.
273                     EXIT
274                 ENDIF
275             ELSE IF (arg == '-eps') THEN
276                 optid = optid + 1
277                 CALL getarg(optid, arg)
278                 READ (arg, *) epsilon
279                 IF (epsilon<=0) THEN
280                     err_detected = .TRUE.
281                     EXIT
282                 END IF
283             ELSE IF (arg == '-prtlev') THEN
284                 optid = optid + 1
285                 CALL getarg(optid, arg)
286                 READ (arg, *) print_level
287                 IF (print_level<0) THEN
288                     err_detected = .TRUE.
289                     EXIT
290                 END IF
291             ELSE IF (arg == '-twodims') THEN
```

```
292                 decomp_dims = 2
293             ELSE IF (arg == '-mprocs') THEN
294                 optid = optid + 1
295                 CALL getarg(optid, arg)
296                 READ (arg, *) m_procs
297                 IF (m_procs<=0) THEN
298                     err_detected = .TRUE.
299                     EXIT
300                 END IF
301             ELSE IF (arg == '-nprocs') THEN
302                 optid = optid + 1
303                 CALL getarg(optid, arg)
304                 READ (arg, *) n_procs
305                 IF (n_procs<=0) THEN
306                     err_detected = .TRUE.
307                     EXIT
308                 END IF
309             ELSE
310                 err_detected = .TRUE.
311                 EXIT
312             END IF
313             optid = optid + 1
314         END DO
315         IF (err_detected) THEN
316             WRITE(*,*) 'unknown command line option: ', arg
317             WRITE(*,*) 'Synopsis: mpirun -np <number of processes> ./cg_all <options>'
318             WRITE(*,*) 'with options [defaults]:'
319             WRITE(*,*) ' -m <number of rows in physical area of the Laplace equation> [4]'
320             WRITE(*,*) ' -n <number of columns in physical area of the Laplace equation [4]'
321             WRITE(*,*) ' -imax <maximal number of iterations> [500]'
322             WRITE(*,*) ' -eps  <epsion> [1e-6]'
323             WRITE(*,*) ' -prtlev <print and debug level 0..5> [1]'
324             WRITE(*,*) ' -twodims [only one-dimensional domain decomposition (DD)]'
325             WRITE(*,*) ' -mprocs <horizontal number of processes in 2-dim DD'
326             WRITE(*,*) ' -nprocs <vertical   number of processes in 2-dim DD'
327             WRITE(*,*) ''
328             CALL abrt
329         END IF
330     END IF
331 #ifndef serial
332     CALL MPI_BCAST(m,          1, MPI_INTEGER,    0, MPI_COMM_WORLD, ierror);
333     CALL MPI_BCAST(n,          1, MPI_INTEGER,    0, MPI_COMM_WORLD, ierror);
334     CALL MPI_BCAST(iter_max,   1, MPI_INTEGER,    0, MPI_COMM_WORLD, ierror);
335     CALL MPI_BCAST(epsilon,    1, MPI_REAL,       0, MPI_COMM_WORLD, ierror);
336     CALL MPI_BCAST(print_level,1, MPI_INTEGER,    0, MPI_COMM_WORLD, ierror);
337     CALL MPI_BCAST(decomp_dims,1, MPI_INTEGER,    0, MPI_COMM_WORLD, ierror);
338     CALL MPI_BCAST(m_procs,    1, MPI_INTEGER,    0, MPI_COMM_WORLD, ierror);
339     CALL MPI_BCAST(n_procs,    1, MPI_INTEGER,    0, MPI_COMM_WORLD, ierror);
340 #endif
341 END SUBROUTINE read_options
342
343 ! >>>>>>>>>>>>>>>>>>>>>> -end- of task 08 <<<<<<<<<<<<<<<<<<<<<<
```

```
344 ! >>>>>>>>>>>>>>>>>>>>>> begin of task 01 <<<<<<<<<<<<<<<<<<<<<<
345
346 ! ********************* vector routines ***********************
347
348 !-------------------------------------------
349 ! add_vec_vec
350 !  IN  REAL(0:) v1      : vector 1
351 !  IN  REAL     lambda  : factor lambda
352 !  IN  REAL(0:) v2      : vector 2
353 !  OUT REAL(0:) v3      : vector 3
354 !
355 ! RETURNS:
356 !   --
357 !
358 ! TASK:
359 !   calculate:   v3 := v1 + lambda * v2
360 !
361 ! GLOBAL DATA:
362 !   IN     INTEGER rowsize : number of data entries in a row of the local chunk
363 !   IN     INTEGER colsize : number of data entries in a column of the local chunk
364 !   IN/OUT REAL    flops   : amount of FLOPS
365 !
366 ! USED FUNCTIONS:
367 !   --
368 SUBROUTINE add_vec_vec(v1, lambda, v2, v3)
369     REAL, DIMENSION(0:), INTENT(IN)  ::  v1
370     REAL, INTENT(IN)                 ::  lambda
371     REAL, DIMENSION(0:), INTENT(IN)  ::  v2
372     REAL, DIMENSION(0:), INTENT(OUT) ::  v3
373     INTEGER                          ::  i
374
375     ! addition of two vectors, second vector is multiplied by a scalar
376     DO i = (rowsize + 2), ((rowsize + 2) * (colsize + 1)) - 1
377         v3(i) = v1(i) + lambda * v2(i);
378     END DO
379
380     ! for the MFLOPS calculation
381     flops = flops + (rowsize + 2) * colsize * 2
382 END SUBROUTINE add_vec_vec
383
384 !------------------------------
385 ! dot_product_vec_vec
386 !  IN  REAL(0:) v1  : vector 1
387 !  IN  REAL(0:) v2  : vector 2
388 !
389 !  RETURNS:
390 !    REAL: dot product
391 !
392 ! TASK:
393 !   calculate dot product for vector 1 and vector 2
394 !
395 ! GLOBAL DATA:
396 !   IN     INTEGER rowsize : number of data entries in a row of the local chunk
397 !   IN     INTEGER colsize : number of data entries in a column of the local chunk
398 !   IN/OUT REAL    flops   : amount of FLOPS
399 !
400 ! USED FUNCTIONS:
401 !   APPL:
402 !     --
403 !   MPI :
404 !     MPI_Allreduce
405 REAL FUNCTION dot_product_vec_vec(v1, v2)
406     REAL, DIMENSION(0:), INTENT(IN) :: v1
407     REAL, DIMENSION(0:), INTENT(IN) :: v2
408
409     INTEGER :: i
410     REAL    :: sum, global_sum
411
412     ! sum = dot product of vectors
413     sum = 0
414     DO i = (rowsize + 2),((rowsize + 2) * (colsize + 1)) - 1
415         sum = sum + v1(i) * v2(i)
416     END DO
417
418 ! global sum
419 #ifdef serial
420     global_sum = sum
421 #endif
422 #ifndef serial
423     commtime_start = MPI_WTIME()
424 !01!CALL MPI_ALLREDUCE(sum, global_sum, 1, MPI_REAL, MPI_SUM, comm_cart, ierror)
425     commtime_end = MPI_WTIME()
426     commtime_sum = commtime_sum + commtime_end - commtime_start
427     flops = flops + 1
428 #endif
429
430     ! for the MFLOPS calculation
431     flops = flops + (rowsize + 2) * colsize * 2
432
433     dot_product_vec_vec = global_sum
434 END FUNCTION dot_product_vec_vec
435
```

```
436 !------------------------------
437 ! max_norm_vec
438 !  IN  REAL(0:) v1  : vector 1
439 !
440 !  RETURNS:
441 !    REAL: maximum norm
442 !
443 ! TASK:
444 !   calculate (vector maximum norm) for vector 1
445 !
446 ! GLOBAL DATA:
447 !   IN INTEGER rowsize : number of data entries in a row of the local chunk
448 !   IN INTEGER colsize : number of data entries in a column of the local chunk
449 !
450 ! USED FUNCTIONS:
451 !   APPL:
452 !     --
453 !   MPI :
454 !     MPI_ALLREDUCE
455 REAL FUNCTION max_norm_vec(v1)
456     REAL, DIMENSION(0:), INTENT(IN) :: v1
457
458     INTEGER :: i
459     REAL    :: global_max = 0
460     ! maxabs = maximum norm of the local vector
461     REAL    :: maxabs = 0
462 !01!DO i = (rowsize + 2), ((rowsize + 2) * (colsize + 1)) - 1
463 !01!   IF ( v1(i) > maxabs) THEN
464 !01!        maxabs =  v1(i)
465 !01!    END IF
466 !01!    IF (-v1(i) > maxabs) THEN
467 !01!        maxabs = -v1(i)
468 !01!    END IF
469 !01!END DO
470
471     ! global maximum
472 #ifdef serial
473 !01!global_max = maxabs
474 #endif
475 #ifndef serial
476     commtime_start = MPI_WTIME()
477 !01!CALL MPI_ALLREDUCE(maxabs, global_max, 1, MPI_REAL, MPI_MAX, comm_cart, ierror)
478     commtime_end = MPI_WTIME()
479     commtime_sum = commtime_sum + commtime_end - commtime_start
480 #endif
481     ! no flops added, because we have only assignments and no operations
482 !01!max_norm_vec = global_max
483 END FUNCTION max_norm_vec
484
485 !------------------------------
486 ! sqr_norm_vec
487 !  IN  REAL(0:) v1  : vector 1
488 !
489 !  RETURNS:
490 !    REAL: square of norm
491 !
492 ! TASK:
493 !   calculate global (vector norm Euklid)^2 for vector 1
494 !
495 ! GLOBAL DATA:
496 !   IN     INTEGER rowsize : number of data entries in a row of the local chunk
497 !   IN     INTEGER colsize : number of data entries in a column of the local chunk
498 !   IN/OUT REAL    flops   : amount of FLOPS
499 !
500 ! USED FUNCTIONS:
501 !   APPL:
502 !     --
503 !   MPI :
504 !     MPI_ALLREDUCE
505 REAL FUNCTION sqr_norm_vec(v1)
506     REAL, DIMENSION(0:), INTENT(IN) :: v1
507
508     INTEGER :: i
509     REAL    :: sum, global_sum
510
511     ! sum = dot product of vector
512 !01!sum = 0
513 !01!DO i = (rowsize + 2), ((rowsize + 2) * (colsize + 1)) - 1
514 !01!    sum = sum + v1(i) * v1(i)
515 !01!END DO
516
517     ! global sum
518 #ifdef serial
519 !01!global_sum = sum;
520 #endif
521 #ifndef serial
522     commtime_start = MPI_WTIME()
523 !01!CALL MPI_Allreduce(sum, global_sum, 1, MPI_REAL, MPI_SUM, comm_cart, ierror)
524     commtime_end = MPI_WTIME()
525     commtime_sum = commtime_sum + commtime_end - commtime_start
526     flops = flops + 1
527 #endif
528     ! for the MFLOPS calculation
529     flops = flops + (rowsize + 2) * colsize * 2
530 !01!sqr_norm_vec = global_sum
531 END FUNCTION sqr_norm_vec
532 !>>>>>>>>>>>>>>>>>>>>> -end- of task 01 <<<<<<<<<<<<<<<<<<<<<
```

```
533 !>>>>>>>>>>>>>>>>>>>>>> begin of task 02 <<<<<<<<<<<<<<<<<<<<<<
534
535 !********************** halo routines ***********************
536
537 #ifndef serial
538 !------------------------------
539 !Init_Halo_Struct
540 !
541 ! TASK:
542 !   Initialize the halo structure.
543 !
544 ! GLOBAL DATA:
545 !   IN INTEGER rowsize : number of data entries in a row of the local chunk
546 !   IN INTEGER colsize : number of data entries in a column of the local chunk
547 !   IN/OUT halo_structure halo : the halo
548 !   OUT REAL(0:) recv_scratch_buff_north : receive buffer for north halo
549 !   OUT REAL(0:) recv_scratch_buff_east  : receive buffer for east halo
550 !   OUT REAL(0:) recv_scratch_buff_south : receive buffer for south halo
551 !   OUT REAL(0:) recv_scratch_buff_west  : receive buffer for west halo
552 ! USED FUNCTIONS:
553 !   APPL:
554 !     --
555 !   MPI:
556 !     MPI_CART_SHIFT
557 SUBROUTINE Init_Halo_Struct(v1)
558     REAL, DIMENSION(0:), TARGET :: v1
559     INTEGER                     :: maxScratchBuffSize
560
561     ! allocate sufficient memory for the send scratch buffer
562     IF (rowsize > colsize) THEN
563         maxScratchBuffSize = rowsize
564     ELSE
565         maxScratchBuffSize = colsize
566     END IF
567     ! allocate sufficient memory for the receive scratch buffer for the halo
568     ALLOCATE(recv_scratch_buff_north(0:maxScratchBuffSize-1))
569     ALLOCATE(recv_scratch_buff_east(0:maxScratchBuffSize-1))
570     ALLOCATE(recv_scratch_buff_south(0:maxScratchBuffSize-1))
571     ALLOCATE(recv_scratch_buff_west(0:maxScratchBuffSize-1))
572
573     ! determine neighbors (1-dim and 2-dims)
574     halo%north => v1(1:rowsize)
575     halo%north_size = rowsize
576     halo%south => v1((rowsize + 2) * (colsize + 1) + 1:(rowsize + 2) * (colsize + 1) + rowsize)
577     halo%south_size = rowsize
578     ! compute north_rank and south_rank
579     CALL MPI_CART_SHIFT(comm_cart, 0, 1, halo%north_rank, halo%south_rank, ierror)
580
581     halo%west => v1((rowsize + 2):(colsize * (rowsize + 2)):(rowsize + 2))
582     halo%west_size = colsize
583     halo%east => v1((rowsize + 2) * 2 - 1:((colsize + 1) * (rowsize + 2) - 1):(rowsize + 2))
584     halo%east_size = colsize
585     ! compute west_rank and east_rank
586 !02!_____________________________________________________________________
587 END SUBROUTINE Init_Halo_Struct
588
589 ! halo communication: e.g. FROM_WEST_TO_EAST
590 !
591 ! halo is included in the local vector chunk
592 !
593 !   *************************       *************************
594 !   *  -----NORTH HALO-----  *       *  -----NORTH HALO-----  *
595 !   *W ******************** E*       *W ******************** E*
596 !   *E *                  *|* A*      *E *                    * A*
597 !   *S * WEST PROCESS    *------------> *   EAST PROCESS     * S*
598 !   *T *                  *|* T*      *T *                    * T*
599 !   *  *                  *|*  *      *  *                    *  *
600 !   *H ******************** H*       *H ******************** H*
601 !   *  -----SOUTH HALO-----  *       *  -----SOUTH HALO-----  *
602 !   *************************       *************************
603
```

```
!------------------------------
! comm_vec_halo
!
! IN  REAL(0:) v1: the vector with the data to send to other processes
!
! TASK:
!   Send neccessary data to the neighbors
!   and receive the needed data from the neighbors into
!   the halo area.
!
! USED GLOBAL DATA:
!   IN/OUT halo_structure halo : the halo
!   IN     INTEGER rowsize : number of data entries in a row of the local chunk
!   IN     INTEGER colsize : number of data entries in a column of the local chunk
!   OUT REAL(0:) recv_scratch_buff_north : receive buffer for north halo
!   OUT REAL(0:) recv_scratch_buff_east  : receive buffer for east halo
!   OUT REAL(0:) recv_scratch_buff_south : receive buffer for south halo
!   OUT REAL(0:) recv_scratch_buff_west  : receive buffer for west halo
!
! USED FUNCTIONS:
!   MPI:
!     MPI_IRECV, MPI_SEND, MPI_WAITALL
!
! message tags for the different communication directions
SUBROUTINE comm_vec_halo(v1)
    REAL, DIMENSION(0:), INTENT(IN), TARGET :: v1
    INTEGER :: i

    INTEGER :: req_array(4)
    INTEGER :: status_array(MPI_STATUS_SIZE, 4)
    INTEGER :: TAG_FROM_NORTH_TO_SOUTH = 101
    INTEGER :: TAG_FROM_EAST_TO_WEST   = 102
    INTEGER :: TAG_FROM_SOUTH_TO_NORTH = 103
    INTEGER :: TAG_FROM_WEST_TO_EAST   = 104

    REAL, DIMENSION(:), POINTER :: send_to_east_values
    REAL, DIMENSION(:), POINTER :: send_to_west_values
    REAL, DIMENSION(:), POINTER :: send_to_north_values
    REAL, DIMENSION(:), POINTER :: send_to_south_values

    ! make a non-blocking receive

    ! Receive halo (non blocking). If current rank has no neighbors,
    ! the neighbor rank is MPI_PROC_NULL and nothing is sent.

    ! ATTENTION:
    ! It is NOT allowed to use pointers or strided arrays in NON-blocking send/receive routines,
    ! because the memory MAY NOT be contiguous. One must use a contiguous scratch buffer
    ! for sending and receiving, if the memory is not contiguous or a pointer to an array.

    ! receive north halo
    CALL MPI_IRECV(recv_scratch_buff_north, halo%north_size, MPI_REAL, halo%north_rank, &
                & TAG_FROM_NORTH_TO_SOUTH, comm_cart, req_array(1), ierror)
    ! receive east halo
!02!_____________________________________________________________________________
!02!                 _____________________________________________________
    ! receive south halo
!02!_____________________________________________________________________________
!02!                 _____________________________________________________
    ! receive west halo
!02!_____________________________________________________________________________
!02!                 _____________________________________________________

    ! send to north neighbor (blocking)
    send_to_north_values => v1(SIDX(1, 1):SIDX(colsize, 1):1)
    CALL MPI_SEND(send_to_north_values, halo%north_size, MPI_REAL, halo%north_rank, &
                & TAG_FROM_SOUTH_TO_NORTH, comm_cart, ierror)

    ! send to east neighbor (blocking)
!02!_____________________________________________________________________________
!02!_____________________________________________________________________________
!02!                 _____________________________________________

    ! send to south neighbor (blocking)
!02!_____________________________________________________________________________
!02!_____________________________________________________________________________
!02!                 _____________________________________________

    ! send to west neighbor (blocking)
!02!_____________________________________________________________________________
!02!_____________________________________________________________________________
!02!                 _____________________________________________

    ! wait for all halo info
!02!_____________________________________________

    ! copy the received halo from the scratch buff into the real halo
    ! ATTENTION: pointers always start with the index 1!!!
    IF (halo%north_rank /= MPI_PROC_NULL) THEN
        DO i = 0, halo%north_size - 1
            halo%north(i+1) = recv_scratch_buff_north(i)
        END DO
    END IF
!02!_____________________________________________
!02!    _____________________________________
!02!        _____________________________________________
!02!    ______
```

```
!02!______
!02!_____________________________________________
!02!    ___________________________________
!02!        _____________________________________________
!02!    _______
!02!_______
!02!_____________________________________________
!02!    ___________________________________
!02!        _____________________________________________
!02!    _______
!02!______
END SUBROUTINE comm_vec_halo
#endif

!>>>>>>>>>>>>>>>>>>>>>> -end- of task 02 <<<<<<<<<<<<<<<<<<<<<<
!>>>>>>>>>>>>>>>>>>>>>> begin of task 03 <<<<<<<<<<<<<<<<<<<<<<

!********************* matrix-vector-multiply ***************************

!------------------------------
! mult_A_vec
!      IN  REAL(0:) v1 : vector to multiply A with
!      OUT REAL(0:) v2 : output vector
!
! TASK: Calculates matrix vector product v2 = A * v1
!
! USED GLOBAL DATA:
!   IN     INTEGER rowsize: number of data entries in a row of the local chunk
!   IN     INTEGER colsize: number of data entries in a column of the local chunk
!   IN/OUT REAL    flops  : amount of FLOPS
!
! USED FUNCTIONS:
!   --
SUBROUTINE mult_A_vec(v1, v2)
    REAL, DIMENSION(0:), INTENT(IN)  :: v1
    REAL, DIMENSION(0:), INTENT(OUT) :: v2
    INTEGER :: i, j

#ifdef FASTMEMXS
    INTEGER :: k, calcend
#endif

    !*************************************************************************
    !* Matrix-Vector mult. without distinction of border/halo neighbored data *
    !*************************************************************************

#ifdef FASTMEMXS
    ! multiplication only for the inner values
    calcend = (colsize + 1) - 2 - 1
    IF (MODULO(calcend, 2) == 0) THEN
        calcend = calcend - 1
    END IF
    DO i = 1, calcend, 2
        DO j = 1, (rowsize + 1) - 1
            v2(SIDX(i, j)) = -1 * v1(SIDX((i - 1), j)) &
                           & - 1 * v1(SIDX(i, (j - 1))) &
                           & + 4 * v1(SIDX(i, j)) &
                           & - 1 * v1(SIDX(i, (j + 1))) &
                           & - 1 * v1(SIDX((i + 1), j))
            v2(SIDX(i + 1, j)) = -1 * v1(SIDX((i - 1 + 1), j)) &
                           & - 1 * v1(SIDX((i + 1), (j - 1))) &
                           & + 4 * v1(SIDX((i + 1), j)) &
                           & - 1 * v1(SIDX((i + 1), (j + 1))) &
                           & - 1 * v1(SIDX((i + 1 + 1), j))
        END DO
    END DO

    ! calculate the remaining rows
    DO k = calcend + 2, (colsize + 1) - 1
        DO j = 1, (rowsize + 1) - 1
            v2(SIDX(k, j)) = -1 * v1(SIDX((k - 1), j)) &
                           & - 1 * v1(SIDX(k, (j - 1))) &
                           & + 4 * v1(SIDX(k, j)) &
                           & - 1 * v1(SIDX(k, (j + 1))) &
                           & - 1 * v1(SIDX((k + 1), j))
        END DO
    END DO
#endif
#ifndef FASTMEMXS
    ! multiplication only for the inner values
    DO i = 1, (colsize + 1) - 1
        DO j = 1, (rowsize + 1) - 1
            v2(SIDX(i, j)) = -1 * v1(SIDX((i - 1), j)) &
                           & - 1 * v1(SIDX(i, (j - 1))) &
                           & + 4 * v1(SIDX(i, j)) &
                           & - 1 * v1(SIDX(i, (j + 1))) &
                           & - 1 * v1(SIDX((i + 1), j))
        END DO
    END DO
#endif

    ! for the MFLOPS calculation
    flops = flops + rowsize * colsize * 9
END SUBROUTINE mult_A_vec

!>>>>>>>>>>>>>>>>>>>>>> -end- of task 03 <<<<<<<<<<<<<<<<<<<<<<
```

```
!>>>>>>>>>>>>>>>>>>>>>> begin of task 04 <<<<<<<<<<<<<<<<<<<<<<

!****************** domain decomposition *********************

!----------------------- 1-dim decomposition -----------------
!
!   Example: m=6, n=7, numprocs=2
!
!       |       j=0     1     2     3     4     5     6 |
!   ----+-------------------------------------------------+-----------
!   i=0 |         0     1     2     3     4     5     6 |
!     1 |         7     8     9    10    11    12    13 | process 1
!     2 |        14    15    16    17    18    19    20 |
!   ----+-------------------------------------------------+-----------
!     3 |        21    22    23    24    25    26    27 |
!     4 |        28    29    30    31    32    33    34 | process 2
!     5 |        35    36    37    38    39    40    41 |
!   ----+-------------------------------------------------+-----------

!------------------------------
! Init_1dim
!
! TASK:
!   Do the one dimensional domain decomposition as depicted above.
!
! USED GLOBAL DATA:
!   OUT INTEGER start_i    : start row of the physical area this process is to calculate
!   OUT INTEGER end1_i     : end row + 1 of the physical area this process is to calculate
!   OUT INTEGER start_j    : start column of the physical area this process is to calculate
!   OUT INTEGER end1_j     : end column + 1 of the physical area this process is to calculate
!   IN/OUT INTEGER my_rank : the rank of the current process
!   OUT INTEGER comm_cart  : cartesian communicator
!   IN  INTEGER numprocs   : number of used processors
!   IN  INTEGER m          : vertical size of the physical problem
!   IN  INTEGER n          : horizontal size of the physical problem
!
! USED FUNCTIONS:
!   MPI:
!     MPI_CART_CREATE, MPI_COMM_RANK
SUBROUTINE Init_1dim()
    INTEGER :: slice_size
#ifndef serial
    INTEGER :: dims(0:1)
    LOGICAL :: periods(0:1)
    IF ((print_level >= 1) .AND. (my_rank == 0)) THEN
        WRITE(*,'(A,I3)') 'Using 1-dim domain decomposition, numprocs=', numprocs
    END IF
    ! Init the virtual topology (1-dim)
    dims(0) = numprocs
    dims(1) = 1
    periods(0) = .FALSE.
    periods(1) = .FALSE.
!04!CALL MPI_CART_CREATE(MPI_COMM_WORLD, 2, dims, periods, .FALSE., comm_cart, ierror)
!04!CALL MPI_COMM_RANK(comm_cart, my_rank, ierror)
#endif

    ! the size of each slice
    ! If m is not a multiple of numprocs, we round up the slice size
    slice_size = ((m - 1) / numprocs) + 1

    ! the rows of the physical area are distributed in slices of equal size
    ! (maybe except for the last slice) to each process
    start_i = (my_rank * slice_size)
    IF (start_i > m) THEN
        start_i = m
    END IF

    end1_i = start_i + slice_size
    IF (end1_i > m) THEN
        end1_i = m
    END IF

    ! we have the full vertical extent (all n columns) for each row
    start_j = 0
    end1_j = n
END SUBROUTINE Init_1dim

!----------------------- 2-dim decomposition -----------------
!
!   Example: m=5, n=7, m_procs=2, n_procs=2
!
!       |           j=0     1     2     3  |  4     5     6            |
!   ----+--------------------------------+-------------------------+
!   i=0 |             0     1     2     3  |  4     5     6            |
!     1 | process 1   7     8     9    10  | 11    12    13  process 2 |
!     2 |            14    15    16    17  | 18    19    20            |
!   ----+--------------------------------+-------------------------+
!     3 | process 3  21    22    23    24  | 25    26    27  process 4 |
!     4 |            28    29    30    31  | 32    33    34            |
!   ----+--------------------------------+-------------------------+

```

```
888 !------------------------------
889 ! Init_2dim
890 !
891 ! TASK:
892 !   Do the two dimensional domain decomposition as depicted above.
893 !
894 ! USED GLOBAL DATA:
895 !   OUT INTEGER start_i   : start row of the physical area this process is to calculate
896 !   OUT INTEGER end1_i    : end row + 1 of the physical area this process is to calculate
897 !   OUT INTEGER start_j   : start column of the physical area this process is to calculate
898 !   OUT INTEGER end1_j    : end column + 1 of the physical area this process is to calculate
899 !   IN  INTEGER my_rank   : the rank of the current process
900 !   IN  INTEGER m_procs   : vertical number of processors
901 !   IN  INTEGER n_procs   : horizontal number of processors
902 !   OUT INTEGER comm_cart : cartesian communicator
903 !   IN  INTEGER m         : vertical size of the physical problem
904 !   IN  INTEGER n         : horizontal size of the physical problem
905 !
906 ! USED FUNCTIONS:
907 !   APPL:
908 !     abrt
909 !   MPI:
910 !     MPI_CART_CREATE, MPI_COMM_RANK
911 SUBROUTINE Init_2dim()
912     INTEGER :: blocksize_i, blocksize_j
913     INTEGER :: startblock_i, startblock_j
914 #ifndef serial
915     INTEGER :: dims(0:1)
916     LOGICAL :: periods(0:1)
917     IF ((print_level >= 1) .AND. (my_rank == 0)) THEN
918         WRITE(*,'(A, I3, A, I3, A, I3)') 'Using 2-dim domain decomposition, m_procs=', m_procs, ', &
919        & n_procs=', n_procs, ', numprocs=', numprocs
920     END IF
921     ! Init the virtual topology (2-dims)
922     dims(0) = m_procs
923     dims(1) = n_procs
924     periods(0) = .FALSE.
925     periods(1) = .FALSE.
926 !04!CALL MPI_CART_CREATE(MPI_COMM_WORLD, 2, dims, periods, .FALSE., comm_cart, ierror)
927 !04!CALL MPI_COMM_RANK(comm_cart, my_rank, ierror)
928 #endif
929
930     IF ((my_rank == 0) .AND. ((m_procs * n_procs) /= numprocs)) THEN
931         WRITE(*,*) 'Number of processors does not fit! Use ', (m_procs * n_procs), ' processors instead. &
932       & Aborting...'
933         CALL abrt()
934     END IF
935
936     ! the number of rows for each block (the last block might have
937     ! less than the first blocks)
938 !04!blocksize_i = ((m - 1) / m_procs) + 1
939
940     ! the number of columns for each block (the last block might have
941     ! less than the first blocks)
942 !04!blocksize_j = ((n - 1) / n_procs) + 1
943
944     ! calculate the start and end row and column
945 !04!startblock_i = my_rank / n_procs
946 !04!startblock_j = MODULO (my_rank, n_procs)
947 !04!start_i = blocksize_i * startblock_i
948 !04!end1_i = start_i + blocksize_i
949 !04!IF (end1_i > m) THEN
950 !04!    end1_i = m
951 !04!END IF
952 !04!start_j = blocksize_j * startblock_j
953 !04!end1_j = start_j + blocksize_j
954 !04!IF (end1_j > n) THEN
955 !04!    end1_j = n
956 !04!END IF
957 END SUBROUTINE Init_2dim
958
959 !------------------------------
960 ! Init_Decomp
961 !
962 ! TASK:
963 !   Do the domain decomposition.
964 !
965 ! USED GLOBAL DATA:
966 !   IN/OUT INTEGER rowsize     : number of data entries in a row of the local chunk
967 !   IN/OUT INTEGER colsize     : number of data entries in a column of the local chunk
968 !   IN/OUT INTEGER chunksize   : number of data entries in the local chunk PLUS halo data
969 !   IN     INTEGER decomp_dims : dimension of the domain decomposition
970 !   IN     INTEGER my_rank     : the rank of the current process
971 !
972 ! USED FUNCTIONS:
973 !   APPL:
974 !     abrt, Init_1dim, Init_2dim
975 SUBROUTINE Init_Decomp()
976     SELECT CASE (decomp_dims)
977         CASE (1)
978             CALL Init_1dim()
979         CASE (2)
980             CALL Init_2dim()
981         CASE DEFAULT
982             WRITE(*,*) 'Wrong decomp_dims=', decomp_dims
983             CALL abrt()
984     END SELECT
```

```
985       ! calculate the size of the rows, cols and the whole chunk
986       rowsize   = end1_j - start_j
987       colsize   = end1_i - start_i
988       chunksize = (rowsize + 2) * (colsize + 2)
989
990       IF ((rowsize < 2) .OR. (colsize < 2)) THEN
991           WRITE(*,*) 'Each process must have at least 2 entries in each dimension.'
992           WRITE(*,*) 'Decrease amount of processes or increase physical area size.'
993           WRITE(*, '(A, I3, A, I, A, I)') 'Rank[', my_rank, ']: rowsize = ', rowsize, ' -- colsize = ', colsize
994           WRITE(*,*) 'Aborting.'
995           CALL abrt()
996       END IF
997   END SUBROUTINE Init_Decomp
998
999   !>>>>>>>>>>>>>>>>>>>>>> -end- of task 04 <<<<<<<<<<<<<<<<<<<<<<
1000  !>>>>>>>>>>>>>>>>>>>>>> begin of task 05 <<<<<<<<<<<<<<<<<<<<<<
1001  !************* boundary and exact solution init ***************
1002
1003  !------------------------------
1004  ! Init_Boundary_Vec
1005  !  OUT REAL(0:) b  : the boundary vector
1006  !
1007  ! TASK: Store all boundary values in the vector memory
1008  !       (that is already allocated).
1009  !       In this example, the boundaries are given by:
1010  !         u(i,j) = h * (i+1)  with  h = 1.0/(m+1) for i=-1,m and j=-1,n
1011  !
1012  ! GLOBAL DATA:
1013  !   IN INTEGER start_i : start row of the physical area this process is to calculate
1014  !   IN INTEGER end1_i  : end row + 1 of the physical area this process is to calculate
1015  !   IN INTEGER start_j : start column of the physical area this process is to calculate
1016  !   IN INTEGER end1_j  : end column + 1 of the physical area this process is to calculate
1017  !   IN INTEGER m       : vertical size of the physical problem
1018  !   IN INTEGER n       : horizontal size of the physical problem
1019  !
1020  ! USED FUNCTIONS:
1021  !     --
1022  SUBROUTINE Init_Boundary_Vec(b)
1023      REAL, DIMENSION(0:), INTENT(OUT) :: b
1024      INTEGER                          :: i, j
1025      REAL                             :: v, h
1026
1027      h = 1.0/(m+1)
1028      ! boundary vector data is all zero at this point
1029      DO i = start_i, end1_i - 1
1030          DO j = start_j, end1_j - 1
1031              v = 0.0
1032              ! next line not neccessary, because boundary value is zero
1033              ! if (i==0)   v = v + u(-1,j) = 0
1034              ! next line simplified to add 1.0 to vector, because of the boundary value of 1.0
1035              ! if (i==m-1) v = v + u(m, j): h * (m+1) = 1
1036              IF (i==m-1) THEN
1037                  v = 1.0
1038              END IF
1039              ! u(i,-1):  h * (i+1)
1040              IF (j==0) THEN
1041                  v = v + h * (i + 1)
1042              END IF
1043              ! u(i, n):  h * (i+1)
1044              IF (j==n-1) THEN
1045                  v = v + h * (i + 1)
1046              END IF
1047              b(SIDX((i - start_i + 1), (j - start_j + 1))) = v
1048          END DO
1049      END DO
1050  END SUBROUTINE Init_Boundary_Vec
1051
1052  !------------------------------
1053  ! Init_Exact_Solution_Vec
1054  !   OUT REAL(0:) u  : the local chunk of the exact solution vector
1055  !
1056  ! TASK: Store the exact solution for the given boundary in the vector u.
1057  !       In this example, the inner values are given by:
1058  !       u(i,j) = h * (i+1)  with  h = 1.0/(m+1) for i=0..m-1 and j=0..n-1
1059  !
1060  ! GLOBAL DATA:
1061  !   IN INTEGER start_i : start row of the physical area this process is to calculate
1062  !   IN INTEGER end1_i  : end row + 1 of the physical area this process is to calculate
1063  !   IN INTEGER start_j : start column of the physical area this process is to calculate
1064  !   IN INTEGER end1_j  : end column + 1 of the physical area this process is to calculate
1065  !   IN INTEGER m       : vertical size of the physical problem
1066  !
1067  ! USED FUNCTIONS:
1068  !   --
1069  SUBROUTINE Init_Exact_Solution_vec(u)
1070      REAL, DIMENSION(0:), INTENT(OUT) :: u
1071      INTEGER                          :: i, j
1072      REAL                             :: h
1073
1074      h = 1.0 / (m + 1)
1075      DO i = start_i, end1_i - 1
1076          DO j = start_j, end1_j - 1
1077              u(SIDX((i - start_i + 1), (j - start_j + 1))) = h * (i + 1)
1078          END DO
1079      END DO
1080  END SUBROUTINE Init_Exact_Solution_vec
1081  !>>>>>>>>>>>>>>>>>>>>>> -end- of task 05 <<<<<<<<<<<<<<<<<<<<<<
```

```
1082 !>>>>>>>>>>>>>>>>>>>>>> begin of task 06 <<<<<<<<<<<<<<<<<<<<<<
1083
1084 !****************** print routines ******************************
1085 SUBROUTINE appl_print_vec(name, v, mprint, nprint)
1086     CHARACTER, INTENT(IN)              :: name
1087     REAL, DIMENSION(0:,0:), INTENT(IN) :: v
1088     INTEGER, INTENT(IN)                :: mprint, nprint
1089     INTEGER                            :: i,j
1090     IF (my_rank == 0) THEN
1091         WRITE(*,'(A, A)', ADVANCE='NO') name, ':   j='
1092         DO j=0, n - 1
1093             IF ( (n <= nprint) .OR. ( (n > nprint) .AND. ((nprint-1)*(j+n-2)/(n-1) /= (nprint-1)*(j+n-1)/(n-1)))
    ) THEN
1094                 WRITE(*,'(I6)', ADVANCE='NO') j
1095             END IF
1096         END DO
1097         WRITE(*,*)
1098         DO i=0, m - 1
1099             IF ( (m <= mprint) .OR. ( (m > mprint) .AND. ((mprint-1)*(i+m-2)/(m-1) /= (mprint-1)*(i+m-1)/(m-1))
     ) ) THEN
1100                 WRITE(*,'(A, A, I3, A)', ADVANCE='NO') name, ': i=', i, ' '
1101                 DO j=0, n - 1
1102                    IF ( (n <= nprint) .OR. ((n > nprint) .AND. ((nprint-1)*(j+n-2)/(n-1) /= (nprint-1)*(j+n-1)/(
    n-1)))) THEN
1103                        WRITE(*, '(A, F5.3)', ADVANCE='NO') ' ', v(i, j)
1104                    END IF
1105                 END DO
1106                 WRITE(*,*)
1107             END IF
1108         END DO
1109     END IF
1110 END SUBROUTINE appl_print_vec
1111
1112 SUBROUTINE recv_token()
1113 #ifndef serial
1114     INTEGER :: dummy, status(MPI_STATUS_SIZE)
1115     IF (my_rank > 0) THEN
1116         CALL MPI_RECV(dummy, 0, MPI_INTEGER, my_rank-1, 333, comm_cart, status, ierror)
1117     END IF
1118 #endif
1119 END SUBROUTINE recv_token
1120
1121 SUBROUTINE flush_and_send_token()
1122 #ifndef serial
1123     INTEGER    :: dummy, status(MPI_STATUS_SIZE)
1124     CALL MPI_SEND(dummy, 0, MPI_INTEGER, MODULO((my_rank+1), numprocs), 333, comm_cart, ierror)
1125     ! Process 0 must wait until the last process (numprocs-1) has finished its output
1126     IF (my_rank == 0) THEN
1127         CALL MPI_RECV(dummy, 0, MPI_INTEGER, numprocs-1, 333, comm_cart, status, ierror)
1128     END IF
1129 #endif
1130 END SUBROUTINE flush_and_send_token
1131
1132 SUBROUTINE print_halo()
1133 #ifndef serial
1134     INTEGER :: i
1135
1136     CALL recv_token()
1137
1138     WRITE(*,*)
1139     WRITE(*,*) 'Halo of rank ', my_rank
1140     WRITE(*,*) '----------------'
1141     IF (halo%north_rank == MPI_PROC_NULL) THEN
1142         WRITE(*,*) 'North: NO halo'
1143     ELSE
1144         WRITE(*,*) 'North:'
1145         WRITE(*,*) '------'
1146         WRITE(*,*) 'values: ', halo%north_size, ', neighbor rank: ', halo%north_rank
1147         DO i = 1, halo%north_size
1148             WRITE(*,*) 'halo[', i, ']: ', halo%north(i)
1149         END DO
1150     END IF
1151     IF (halo%east_rank == MPI_PROC_NULL) THEN
1152         WRITE(*,*) 'East: NO halo'
1153     ELSE
1154         WRITE(*,*) 'East:'
1155         WRITE(*,*) '------'
1156         WRITE(*,*) 'values: ', halo%east_size, ', neighbor rank: ', halo%east_rank
1157         DO i = 1, halo%east_size
1158             WRITE(*,*) 'halo[', i, ']: ', halo%east(i)
1159         END DO
1160     END IF
1161     IF (halo%south_rank == MPI_PROC_NULL) THEN
1162         WRITE(*,*) 'South: NO halo'
1163     ELSE
1164         WRITE(*,*) 'South:'
1165         WRITE(*,*) '------'
1166         WRITE(*,*) 'values: ', halo%south_size, ', neighbor rank: ', halo%south_rank
1167         DO i = 1, halo%south_size
1168             WRITE(*,*) 'halo[', i, ']: ', halo%south(i)
1169         END DO
1170     END IF
1171     IF (halo%west_rank == MPI_PROC_NULL) THEN
1172         WRITE(*,*) 'West: NO halo'
1173     ELSE
1174         WRITE(*,*) 'West:'
1175         WRITE(*,*) '------'
```

```
1176         WRITE(*,*) 'values: ', halo%west_size, ', neighbor rank: ', halo%west_rank
1177         DO i = 1, halo%west_size
1178             WRITE(*,*) 'halo[', i, ']: ', halo%west(i)
1179         END DO
1180     END IF
1181     WRITE(*,*) ''
1182     CALL flush_and_send_token()
1183 #endif
1184 END SUBROUTINE print_halo
1185
1186 SUBROUTINE print_halo_struct()
1187 #ifndef serial
1188     CALL recv_token()
1189
1190     WRITE(*,*) ''
1191     WRITE(*,*) 'Halo of rank ', my_rank
1192     WRITE(*,*) '----------------'
1193     IF (halo%north_rank == MPI_PROC_NULL) THEN
1194         WRITE(*,*) 'North: NO halo'
1195     ELSE
1196         WRITE(*,*) 'North: values: ', halo%north_size ,', neighbor rank: ', halo%north_rank
1197     END IF
1198     IF (halo%east_rank == MPI_PROC_NULL) THEN
1199         WRITE(*,*) 'East: NO halo'
1200     ELSE
1201         WRITE(*,*) 'East: values: ', halo%east_size ,', neighbor rank: ', halo%east_rank
1202     END IF
1203     IF (halo%south_rank == MPI_PROC_NULL) THEN
1204         WRITE(*,*) 'South: NO halo'
1205     ELSE
1206         WRITE(*,*) 'South: values: ', halo%south_size ,', neighbor rank: ', halo%south_rank
1207     END IF
1208     IF (halo%west_rank == MPI_PROC_NULL) THEN
1209         WRITE(*,*) 'West: NO halo'
1210     ELSE
1211         WRITE(*,*) 'West: values: ', halo%west_size ,', neighbor rank: ', halo%west_rank
1212     END IF
1213     WRITE(*,*) ''
1214
1215     CALL flush_and_send_token()
1216 #endif
1217 END SUBROUTINE print_halo_struct
1218
1219 SUBROUTINE print_vec2d(name, v2, mprint, nprint)
1220     CHARACTER, INTENT(IN)          :: name
1221     REAL, DIMENSION(0:), INTENT(IN) :: v2
1222     INTEGER, INTENT(IN)            :: mprint, nprint
1223     INTEGER :: i,j
1224     REAL    :: whole_v2(0:m-1, 0:n-1)
1225 #ifndef serial
1226     INTEGER    :: rank
1227     REAL       :: buff(0:(n + 2) * (m + 2) - 1)
1228     INTEGER    :: extent(0:3), new_offset
1229     INTEGER    :: status(MPI_STATUS_SIZE), status2(MPI_STATUS_SIZE)
1230
1231     IF (my_rank > 0) THEN
1232         ! send the start_i, end1-i, start_j, end1_j
1233         extent(0) = start_i
1234         extent(1) = end1_i
1235         extent(2) = start_j
1236         extent(3) = end1_j
1237         CALL MPI_SEND(extent, 4, MPI_INTEGER, 0, 666, comm_cart, ierror)
1238
1239         ! send the vector data to rank 0
1240         CALL MPI_SEND(v2, chunksize, MPI_REAL, 0, 333, comm_cart, ierror)
1241     ELSE
1242         ! copy the rank 0 chunk into the whole physical vector
1243         DO i = start_i, end1_i - 1
1244             DO j = start_j, end1_j - 1
1245                 whole_v2(i, j) = v2(SIDX((i - start_i + 1), (j - start_j + 1)))
1246             END DO
1247         END DO
1248
1249         ! receive the halo data and data extent from all other ranks into the buffer
1250         ! and copy the buffer to where it belongs into the whole physical vector
1251         DO rank = 1, numprocs - 1
1252             CALL MPI_RECV(extent, 4, MPI_INTEGER, rank, 666, comm_cart, status, ierror)
1253             CALL MPI_RECV(buff, ((extent(1) - extent(0) + 2) * (extent(3) - extent(2) + 2)), MPI_REAL, &
1254                           & rank, 333, comm_cart, status2, ierror)
1255
1256             new_offset = extent(3) - extent(2) + 2
1257
1258             DO i = extent(0), extent(1) - 1
1259                 DO j = extent(2), extent(3) - 1
1260                     whole_v2(i, j) = buff((i - extent(0) + 1) * new_offset + (j - extent(2) + 1))
1261                 END DO
1262             END DO
1263         END DO
1264     END IF
1265
1266     CALL appl_print_vec(name, whole_v2, mprint, nprint)
1267 #endif
1268 #ifdef serial
1269     ! copy the data without halo into the whole physical vector
1270     DO i = start_i, end1_i - 1
1271         DO j = start_j, end1_j - 1
1272             whole_v2(i, j) = v2(SIDX((i - start_i + 1), (j - start_j + 1)))
```

```
1273          END DO
1274      END DO
1275      CALL appl_print_vec(name, whole_v2, mprint, nprint)
1276  #endif
1277  END SUBROUTINE print_vec2d
1278
1279  SUBROUTINE print_vec(name, v1)
1280      CHARACTER, INTENT(IN)           :: name
1281      REAL, DIMENSION(0:), INTENT(IN) :: v1
1282      INTEGER                         :: i, j
1283      CALL recv_token()
1284      DO i = 1, (colsize + 1) - 1
1285          DO j = 1, (rowsize + 1) - 1
1286              WRITE(*,'(A, A, I2, A, F6.3)') name, '[', ((i * rowsize) + j), '] = ', v1(SIDX(i, j))
1287          END DO
1288      END DO
1289      CALL flush_and_send_token()
1290  END SUBROUTINE print_vec
1291
1292  SUBROUTINE print_decomp()
1293      CALL recv_token()
1294      WRITE(*, '(A,I3, A, I, A, I, A, I, A, I)') 'Rank ', my_rank, ': start_i, end1_i, start_j, end1_j: ', &
1295               & start_i, ',', end1_i, ',', start_j, ',', end1_j
1296      CALL flush_and_send_token()
1297  END SUBROUTINE print_decomp
1298
1299  !>>>>>>>>>>>>>>>>>>>>>> -end- of task 06 <<<<<<<<<<<<<<<<<<<<<<
1300  !>>>>>>>>>>>>>>>>>>>>>> begin of task 07 <<<<<<<<<<<<<<<<<<<<<<
1301
1302  !*********************** CG solver ****************************
1303
1304  !------------------------------
1305  ! CG_solve
1306  !   OUT INTEGER num_iter : return the number of iterations
1307  !   OUT REAL    norm_r   : return the norm of r
1308  !
1309  ! TASK:
1310  !   solve Laplace equation
1311  !
1312  ! GLOBAL DATA:
1313  !   IN     INTEGER     iter_max : maximal number of iterations
1314  !   IN     REAL        epsilon  : epsilon for maximal error
1315  !   IN     REAL(0:)    b        : local chunk of vector b
1316  !   IN/OUT REAL(0:)    x        : local chunk of vector x
1317  !   IN     INTEGER      my_rank  : rank of the local process
1318  ! USED FUNCTIONS:
1319  !   APPL:
1320  !     comm_vec_halo, print_vec, add_vec_vec
1321  !     sqr_norm_vec, print_halo, mult_A_vec, dot_product_vec_vec
1322  !   MPI-lib:
1323  !     MPI_WTIME
1324  SUBROUTINE CG_solve(num_iter, norm_r)
1325      INTEGER, INTENT(INOUT) :: num_iter
1326      REAL, INTENT(INOUT)    :: norm_r
1327
1328      ! only p has a halo
1329      REAL, DIMENSION(0:chunksize-1) :: p, r, v
1330      REAL                           :: alpha, alpha_new, lambda
1331      INTEGER                        :: iter
1332
1333      ! set the content of the vectors to zero
1334      p = 0
1335      r = 0
1336      v = 0
1337
1338  #ifndef serial
1339      ! !!! determine communication partners for vector p !!!
1340      CALL Init_Halo_Struct(p)
1341      IF (print_level>=4) THEN
1342          CALL print_halo_struct()
1343      END IF
1344  #endif
1345
1346      ! x is initialized with 0
1347
1348      ! instead of p = Ax
1349      ! p := x
1350      p = x
1351
1352  #ifndef serial
1353      ! update halo
1354      commtime_start = MPI_WTIME()
1355      CALL comm_vec_halo(p)
1356      commtime_end = MPI_WTIME()
1357      commtime_sum = commtime_sum + commtime_end - commtime_start
1358  #endif
1359
1360      ! v = Ap
1361      CALL mult_A_vec(p, v)
1362      IF (print_level>=4) THEN
1363          CALL print_vec('v', v)
1364      END IF
1365
1366      ! p = b - v
1367      CALL add_vec_vec(b, -1.0, v, p)
1368      IF (print_level>=4) THEN
1369          CALL print_vec('p', p)
```

```
1370      END IF
1371
1372      ! r := p
1373      r = p
1374      IF (print_level>=4) THEN
1375          CALL print_vec('r', r)
1376      END IF
1377
1378      ! alpha = ||r||**2,   Caution: sqr_norm_vec() computes already ||.||**2
1379      alpha = sqr_norm_vec(r)
1380
1381      iter = 1
1382      DO WHILE ((iter < iter_max) .AND. (alpha > epsilon))
1383          iter = iter + 1
1384          IF ((print_level>=5) .OR. ( (print_level>=4) .AND. (iter==0) )) THEN
1385              WRITE(*,'(A, I3, A, I, A, F9.6)') 'CG - my_rank =', my_rank, ', i=', iter, ', alpha=', alpha
1386          END IF
1387
1388  #ifndef serial
1389          ! update halo
1390          commtime_start = MPI_WTIME()
1391          CALL comm_vec_halo(p)
1392          commtime_end = MPI_WTIME()
1393          commtime_sum = commtime_sum + commtime_end - commtime_start
1394  #endif
1395
1396          IF ((print_level>=5) .OR. ( (print_level>=4) .AND. (iter==0) )) THEN
1397              CALL print_vec('p', p)
1398              CALL print_halo()
1399          END IF
1400
1401          ! v = Ap
1402          CALL mult_A_vec(p, v)
1403          IF ((print_level>=5) .OR. ( (print_level>=4) .AND. (iter==0) )) THEN
1404              CALL print_vec('v', v)
1405          END IF
1406
1407          ! lambda = alpha / (v, p)2
1408          lambda = alpha / dot_product_vec_vec(v, p)
1409
1410          IF ((print_level>=5) .OR. ( (print_level>=4) .AND. (iter==0) )) THEN
1411              WRITE(*,'(A, I3, A, I, A, F9.6, A, F9.6)') 'CG - my_rank =', my_rank, ', i=', iter, ', vp=', &
1412                        & dot_product_vec_vec(v, p), ', lambda=', lambda
1413          END IF
1414
1415          ! x := x + lambda * p
1416          CALL add_vec_vec(x, lambda, p, x)
1417          IF ((print_level>=5) .OR. ( (print_level>=4) .AND. (iter==0) )) THEN
1418              CALL print_vec('x', x)
1419          END IF
1420
1421          ! r := r - lambda * v
1422          CALL add_vec_vec(r, -lambda, v, r)
1423          IF ((print_level>=5) .OR. ( (print_level>=4) .AND. (iter==0) )) THEN
1424              CALL print_vec('r', r)
1425          END IF
1426
1427          ! alpha_new = ||r||**2,   Caution: sqr_norm_vec() computes already ||.||**2
1428          alpha_new = sqr_norm_vec(r)
1429
1430          IF ((print_level>=5) .OR. ( (print_level>=4) .AND. (iter==0) )) THEN
1431              WRITE(*,'(A, I3, A, I, A, F9.6)') 'CG - my_rank =', my_rank, ', i=', iter, ', alpha_new=', alpha_new
1432          END IF
1433
1434          ! p := r + (alpha_new/alpha) * p
1435          CALL add_vec_vec(r, alpha_new/alpha, p, p)
1436          IF ((print_level>=5) .OR. ( (print_level>=4) .AND. (iter==0) )) THEN
1437              CALL print_vec('p', p)
1438          END IF
1439
1440          ! alpha := alpha_new
1441          alpha = alpha_new
1442
1443          IF ((print_level>=2) .AND. (my_rank==0)) THEN
1444              WRITE (*,'(A, I, A, E8.2, A)', ADVANCE='NO') 'CG i=', iter, ', ||r||**2 = ', alpha_new, ' '
1445              IF (alpha_new <= epsilon) THEN
1446                  WRITE (*,'(A)', ADVANCE='NO') '<='
1447              ELSE
1448                  WRITE (*,'(A)', ADVANCE='NO') '>'
1449              ENDIF
1450              WRITE (*,'(A, E8.2, A, E8.2, A)', ADVANCE='NO') ' ', epsilon, ', ||r|| = ', sqrt(alpha_new), ' '
1451              IF (SQRT(alpha_new) <= SQRT(epsilon)) THEN
1452                  WRITE (*,'(A)', ADVANCE='NO') '<='
1453              ELSE
1454                  WRITE (*,'(A)', ADVANCE='NO') '>'
1455              ENDIF
1456              WRITE (*,'(A, E8.2)') ' ', sqrt(epsilon)
1457          END IF
1458      END DO
1459
1460      ! update output-arguments
1461      num_iter = iter
1462      norm_r = SQRT(alpha)
1463  END SUBROUTINE CG_solve
1464
1465  !>>>>>>>>>>>>>>>>>>>>>> -end- of task 07 <<<<<<<<<<<<<<<<<<<<<<
1466  END PROGRAM cgv_fortran
```

1467