– Example Programs –

Iterative solution of the Laplace equation

Hinnerk Stüben

Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

Universität Bremen

13–18 September 2025

Contents

Overview

- motivation
  - documentation of the codes used in the programming project
- theory
  - discretisation of the Laplace equation (Section 2)
  - review of iterative methods (Section 3)
  - application to the Laplace equation (Section 4)
- implementation
  - MPI project (Sections 5 and 6)
  - OpenMP project (Section 7)

Iterative methods
for sparse linear systems

Iterative solution of systems of linear equations

- matrix notation

$$Ax = b \quad \Leftrightarrow \quad \sum_{j=1}^{N} a_{ij} x_j = b_i, \quad i = 1, \ldots, n$$

- start vector / initial guess

$$x^{(0)}$$

- iteration sequence

$$x^{(0)} \to x^{(1)} \to x^{(2)} \to \ldots$$

- iteration step

$$x^{(k)} \to x^{(k+1)}$$

Iterative solution of systems of linear equations

- methods for
  - large sparse systems
- sparse systems arise
  - e.g. from discretised differential equations
- computational effort per iteration step
  - $\textit{sparse matrix} \times \textit{vector} \approx \textit{vector} \times \textit{vector}$
- distinction
  - *direct* methods

Overview (I)

- example problem
  - iterative solution of the Laplace equation $\Delta\varphi = 0$ for given boundary values
- iterative methods
  - Jacobi iteration
  - Gauss-Seidel iteration
  - *conjugate gradient* method (*cg* method)
- focus
  - *parallelisation* of an important class of applications
  - but *not* the efficiency of
    - numerical methods
    - single processor implementation

Overview (II)

- relevance to the programming project

Jacobi iteration	$\rightarrow$	MPI and OpenMP
Gauss-Seidel iteration	$\rightarrow$	OpenMP
cg method	$\rightarrow$	OpenMP

Discretisation of the Laplace equation

Discretisation of the Laplace equation

- Laplace equation

$$\Delta\varphi = \frac{\partial^2\varphi}{\partial x^2} + \frac{\partial^2\varphi}{\partial y^2} = 0$$

- finite differences

$$\left.\frac{\partial^2\varphi}{\partial x^2}\right|_{x_i,y_j} = \frac{\varphi_{i-1,j} - 2\varphi_{i,j} + \varphi_{i+1,j}}{h^2} + O(h^2)$$

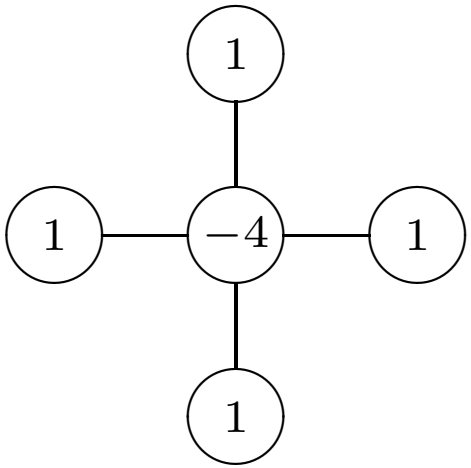
$$\left.\frac{\partial^2\varphi}{\partial y^2}\right|_{x_i,y_j} = \frac{\varphi_{i,j-1} - 2\varphi_{i,j} + \varphi_{i,j+1}}{h^2} + O(h^2)$$

- discretised equation

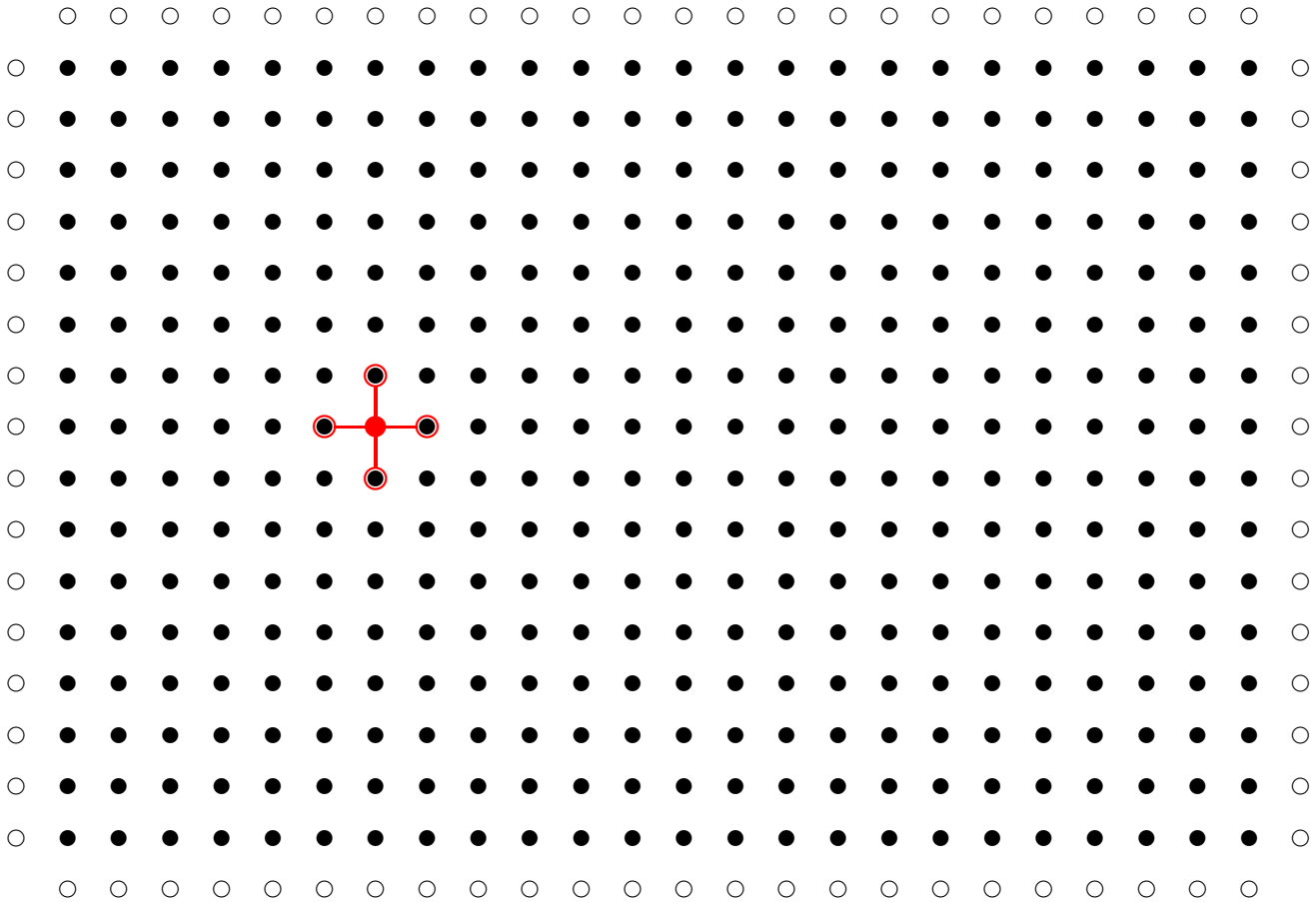
$$\Delta\varphi = \frac{\varphi_{i-1,j} + \varphi_{i,j-1} - 4\varphi_{i,j} + \varphi_{i+1,j} + \varphi_{i,j+1}}{h^2} + O(h^2) = 0$$

Geometrical representation

- five point stencil



Geometrical representation



Matrix notation

- system of equations

$$h^2 \cdot \Delta\varphi \approx \varphi_{i-1,j} + \varphi_{i,j-1} - 4\varphi_{i,j} + \varphi_{i+1,j} + \varphi_{i,j+1} = 0$$

- matrix notation

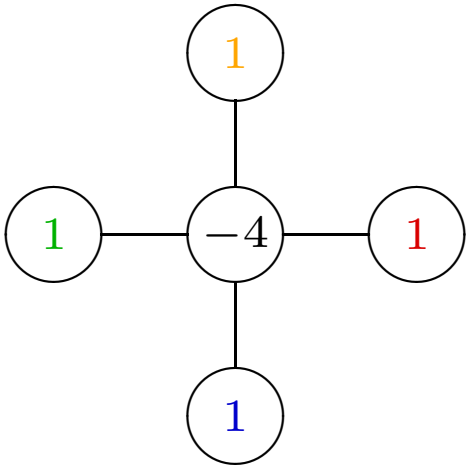
$$\begin{aligned} M\varphi + b &= 0 \\ -M\varphi &= b \end{aligned}$$

Matrix notation $-M\varphi = b$

mesh indexing:

	a	b	c	d	
T	9	10	11	12	t
S	5	6	7	8	s
R	1	2	3	4	r
	A	B	C	D	

stencil:



$$
\begin{pmatrix}
+4 & -1 & & & -1 & & & & & & & \\
-1 & +4 & -1 & & & -1 & & & & & & \\
 & -1 & +4 & -1 & & & -1 & & & & & \\
 & & -1 & +4 & & & & -1 & & & & \\
-1 & & & & +4 & -1 & & & -1 & & & \\
 & -1 & & & -1 & +4 & -1 & & & -1 & & \\
 & & -1 & & & -1 & +4 & -1 & & & -1 & \\
 & & & -1 & & & -1 & +4 & & & & -1 \\
 & & & & -1 & & & & +4 & -1 & & \\
 & & & & & -1 & & & -1 & +4 & -1 & \\
 & & & & & & -1 & & & -1 & +4 & -1 \\
 & & & & & & & -1 & & & -1 & +4
\end{pmatrix}
\cdot
\begin{pmatrix}
\varphi_1 \\ \varphi_2 \\ \varphi_3 \\ \varphi_4 \\ \varphi_5 \\ \varphi_6 \\ \varphi_7 \\ \varphi_8 \\ \varphi_9 \\ \varphi_{10} \\ \varphi_{11} \\ \varphi_{12}
\end{pmatrix}
=
\begin{pmatrix}
\varphi_A + \varphi_R \\ \varphi_B \\ \varphi_C \\ \varphi_D + \varphi_r \\ \varphi_S \\ \\ \\ \varphi_s \\ \varphi_a + \varphi_T \\ \varphi_b \\ \varphi_c \\ \varphi_d + \varphi_t
\end{pmatrix}
$$

Matrix notation

- The stencil describes the situation on the *mesh*.
  It does *not* describe the matrix.

- In order to describe the matrix one has to define the *indexing* of the mesh in addition.

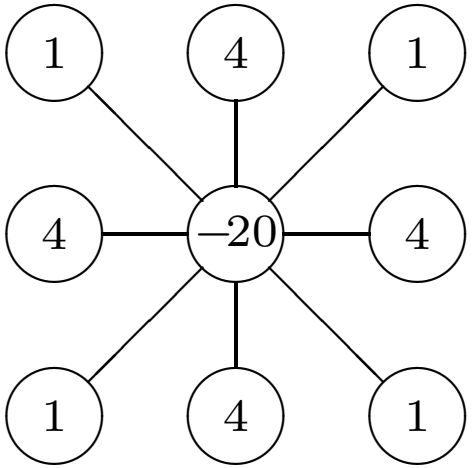
- The points of the stencil appear in *one* row of the matrix.

- mesh size: $N_x \cdot N_y$

- matrix size: $(N_x \cdot N_y)^2$

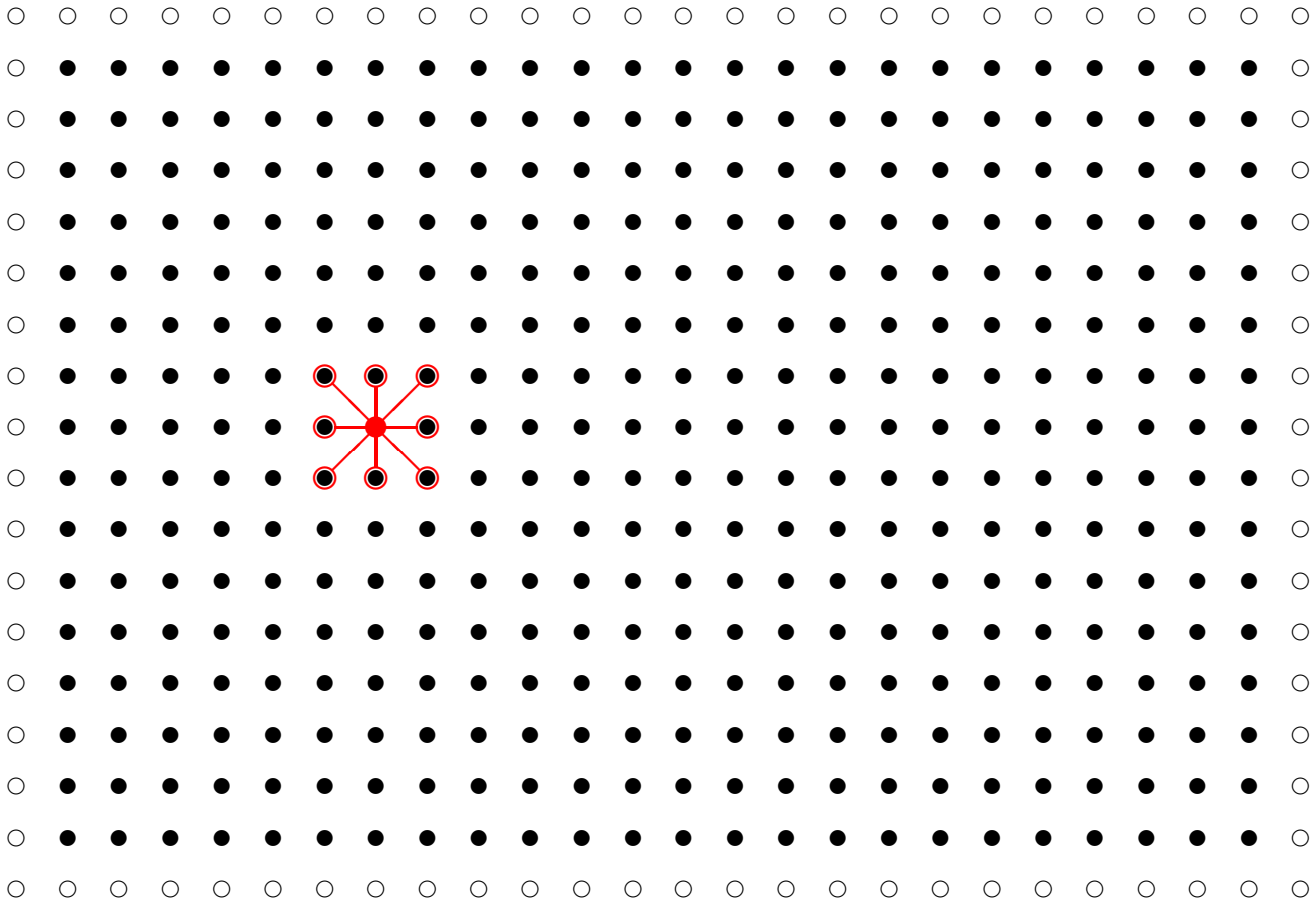
Improved discretisation

- nine point stencil



- discretisation errors are $O(h^6)$

Geometrical representation

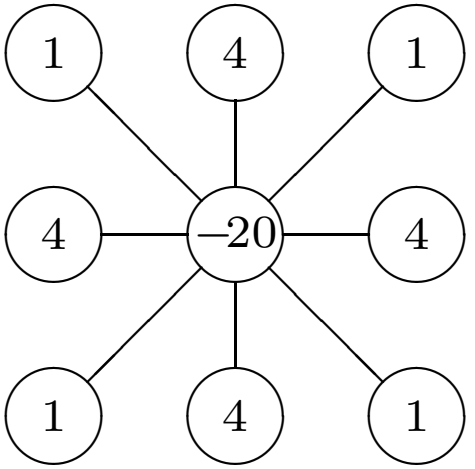


Matrix notation $-M\varphi = b$

mesh indexing:

U	a	b	c	d	u
T	9	10	11	12	t
S	5	6	7	8	s
R	1	2	3	4	r
Q	A	B	C	D	q

stencil:



$$
\left(\begin{array}{cccc|cccc|cccc}
20 & -4 & & & -4 & -1 & & & & & & \\
-4 & 20 & -4 & & -1 & -4 & -1 & & & & & \\
 & -4 & 20 & -4 & & -1 & -4 & -1 & & & & \\
 & & -4 & 20 & & & -1 & -4 & & & & \\
\hline
-4 & -1 & & & 20 & -4 & & & -4 & -1 & & \\
-1 & -4 & -1 & & -4 & 20 & -4 & & -1 & -4 & -1 & \\
 & -1 & -4 & -1 & & -4 & 20 & -4 & & -1 & -4 & -1 \\
 & & -1 & -4 & & & -4 & 20 & & & -1 & -4 \\
\hline
 & & & & -4 & -1 & & & 20 & -4 & & \\
 & & & & -1 & -4 & -1 & & -4 & 20 & -4 & \\
 & & & & & -1 & -4 & -1 & & -4 & 20 & -4 \\
 & & & & & & -1 & -4 & & & -4 & 20
\end{array}\right)
\cdot\varphi =
\left(\begin{array}{l}
4\varphi_A + 4\varphi_R + \varphi_B + \varphi_Q + \varphi_S \\
4\varphi_B + \varphi_A + \varphi_B \\
4\varphi_C + \varphi_B + \varphi_D \\
4\varphi_D + 4\varphi_r + \varphi_C + \varphi_q + \varphi_s \\
\hline
4\varphi_S + \varphi_R + \varphi_T \\
\\
\\
4\varphi_s + \varphi_r + \varphi_t \\
\hline
4\varphi_a + 4\varphi_T + \varphi_b + \varphi_U + \varphi_S \\
4\varphi_b + \varphi_a + \varphi_b \\
4\varphi_c + \varphi_b + \varphi_d \\
4\varphi_d + 4\varphi_t + \varphi_c + \varphi_u + \varphi_s
\end{array}\right)
$$

Implementation

- The matrix notation is *not* needed for an implementation.
  (For the *cg* method the right hand side b is needed explicitly.)

- One can always think in terms of the mesh.

Iterative methods
for the Laplace equation

Jacobi iteration

- iteration step

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right)$$

- the iteration step is *independent* of the mesh indexing

$$(x, y) \rightarrow i$$

- implementation
  - *two* vectors are needed (for the left and right hand sides)

Jacobi iteration – implementation

- five point stencil

```
for x := 1 to Nx do
   for y := 1 to Ny do
       v_new[x, y] := (v_old[x-1, y] + v_old[x+1, y]
                     + v_old[x, y-1] + v_old[x, y+1]) / 4
```

- nine point stencil

```
for x := 1 to Nx do
   for y := 1 to Ny do
       v_new[x, y] := (4 * (v_old[x-1, y] + v_old[x+1, y]
                          + v_old[x, y-1] + v_old[x, y+1])
                          + v_old[x-1, y-1] + v_old[x-1, y+1]
                          + v_old[x+1, y-1] + v_old[x+1, y+1]
                       ) / 20
```

Gauss-Seidel iteration

- iteration step

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right)$$

- each iteration step *depends* on the mesh indexing

$$(x, y) \rightarrow i$$

- implementation
  - the *same* vector is used on the left and right hand side

Gauss-Seidel iteration – implementation

- five point stencil

```
for x := 1 to Nx do
   for y := 1 to Ny do
       v[x, y] := (v[x-1, y] + v[x+1, y]
                   + v[x, y-1] + v[x, y+1]) / 4
```

- nine point stencil

```
for x := 1 to Nx do
   for y := 1 to Ny do
       v[x, y] := (4 * (v[x-1, y] + v[x+1, y]
                        + v[x, y-1] + v[x, y+1])
                        + v[x-1, y-1] + v[x-1, y+1]
                        + v[x+1, y-1] + v[x+1, y+1]
                    ) / 20
```

The conjugate gradient method

- notation
  - dot product: $(\,\cdot\,,\,\cdot\,)$
  - two-norm of a vector: $||\,\cdot\,||$

- algorithm

  choose $x^{(0)}$
  let $p^{(0)} = r^{(0)} = b - Ax^{(0)}$
  for $k = 0, 1, \ldots$

$$\alpha^{(k)} = \left(r^{(k)}, r^{(k)}\right) / \left(p^{(k)}, Ap^{(k)}\right)$$
$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}$$
$$r^{(k+1)} = r^{(k)} - \alpha^{(k)} Ap^{(k)}$$

if $||r^{(k+1)}|| < \epsilon$ **stop**

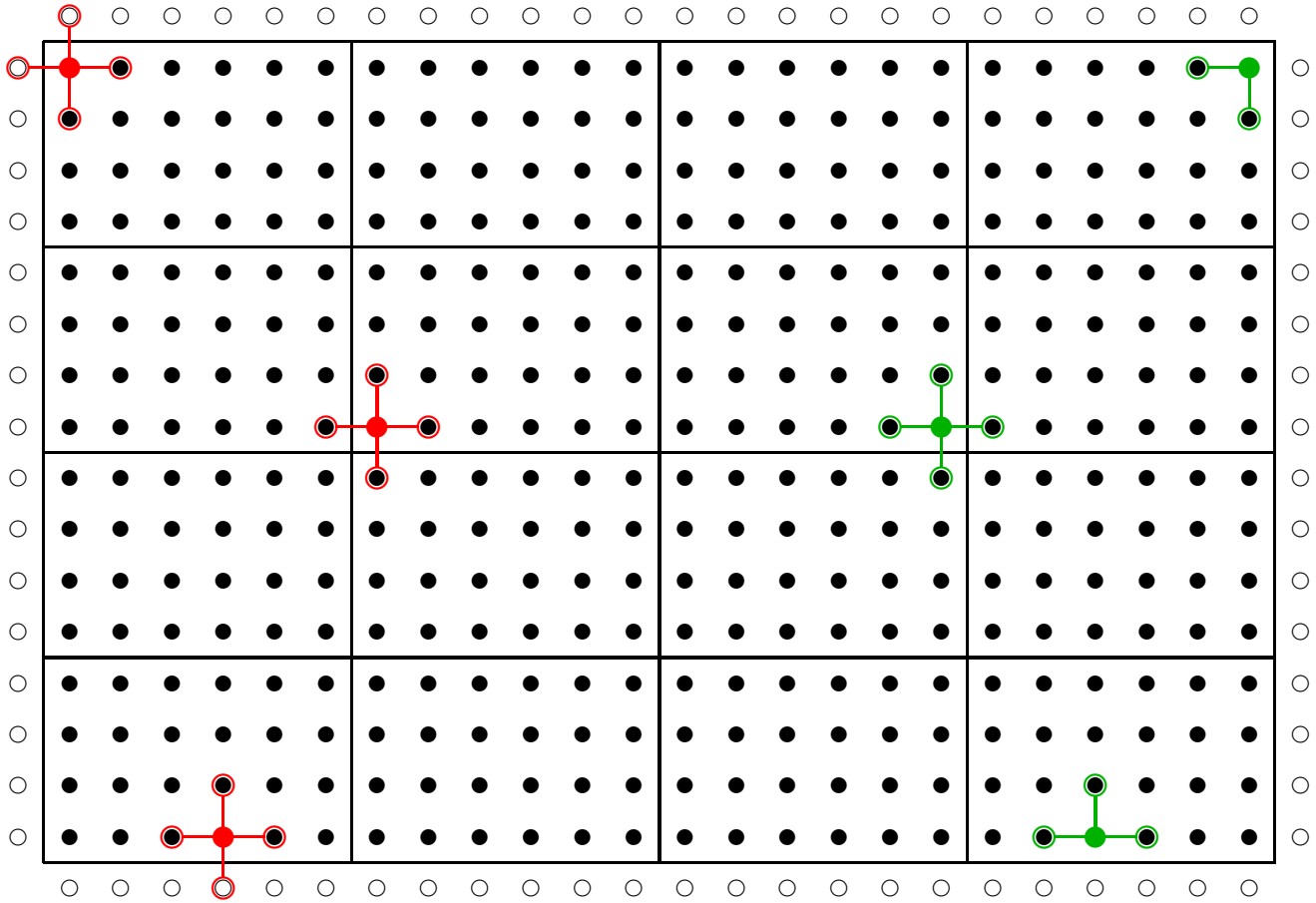
$$\beta^{(k)} = \left(r^{(k+1)}, r^{(k+1)}\right) / \left(r^{(k)}, r^{(k)}\right)$$
$$p^{(k+1)} = r^{(k+1)} + \beta^{(k)} p^{(k)}$$

The conjugate gradient method

- three auxiliary vectors: r, p, Ap
- one matrix-vector multiplication Ap per iteration
- comparison with Jacobi and Gauss-Seidel:
  - additional vector-vector operations
  - the right hand side b is needed explicitly
- parallelisation (see also: *Thinking Parallel*):
  - *cg* can be parallelised like Jacobi
    (Gauss-Seidel needs *colouring* in addition)

Comparison of data access patterns

Jacobi vs. conjugate gradient



Digression: convergence criteria

- system of equations

$$Ax = b$$

- residual after k iterations

$$r^{(k)} = b - Ax^{(k)}$$

- convergence criteria
  - $||r^{(k+1)}|| < \epsilon$
  - $||r^{(k+1)}|| < \epsilon \, ||r^{(0)}||$
  - $||r^{(k+1)}|| < \epsilon \, ||b||$
  - $||x^{(k+1)} - x^{(k)}|| < \epsilon$
  - $||x^{(k+1)} - x^{(k)}|| < \epsilon \, ||x^{(k)}||$

Digression: convergence criteria

- *cg* method: residual is calculated on the fly
- residual for the Jacobi method:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right) = \frac{1}{a_{ii}} \left(b_i - \sum_{j} a_{ij} x_j^{(k)} \right) + x_i^{(k)} = \frac{r_i^{(k)}}{a_{ii}} + x_i^{(k)}$$

$$r_i^{(k)} = a_{ii} \left(x_i^{(k+1)} - x_i^{(k)} \right)$$

– MPI project –
sequential program

source code directories:

`./laplace-code/laplace-seq-c`

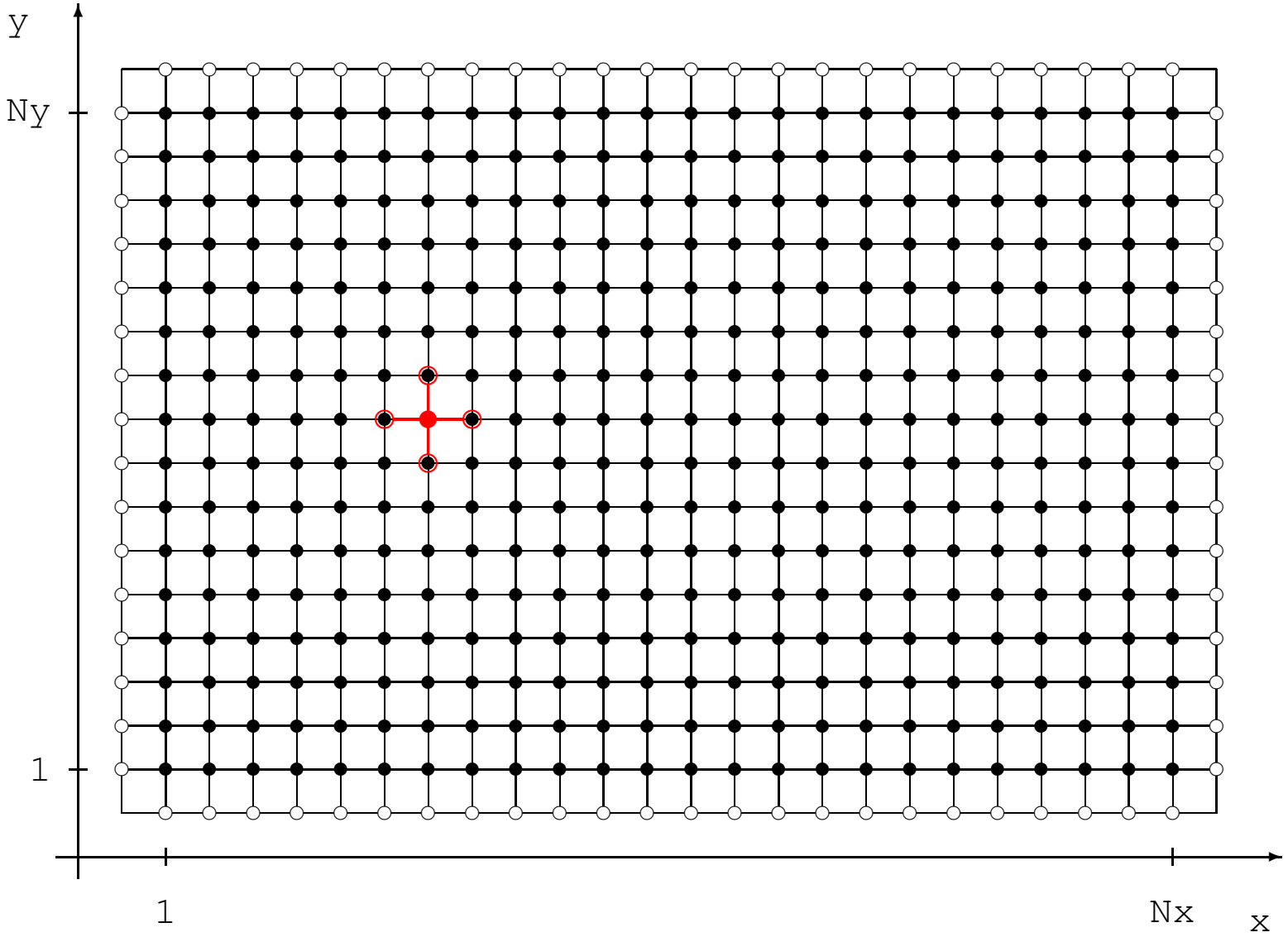
`./laplace-code/laplace-seq-f90`

MPI project – overview

- solution of the Laplace equation with a Jacobi solver
  - starting point: sequential program
  - a simple parallel version is provided
  - discussion of the process of parallelisation

- project aim: improvement of the simple program by
  - introducing reduction operations
  - introducing MPI datatypes
  - using more efficient communication calls
  - extending it for the nine point stencil

Global mesh coordinates / mesh indexing

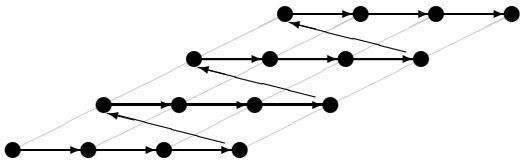


Data structures (I)

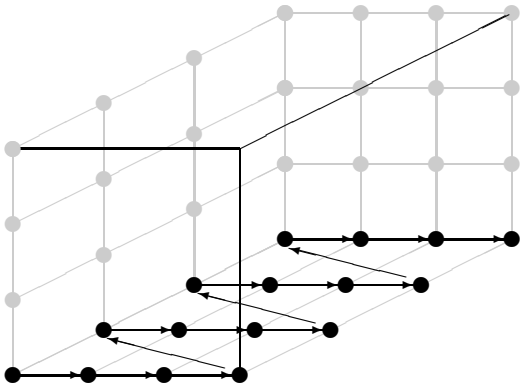
- two-dimensional arrays
- Fortran storage sequence
- definition of data structure `field` in C

Storage sequence of multi-dimensional arrays

Fortran

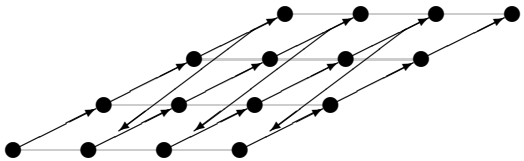


```
real(8) :: v(Nx, Ny)
```

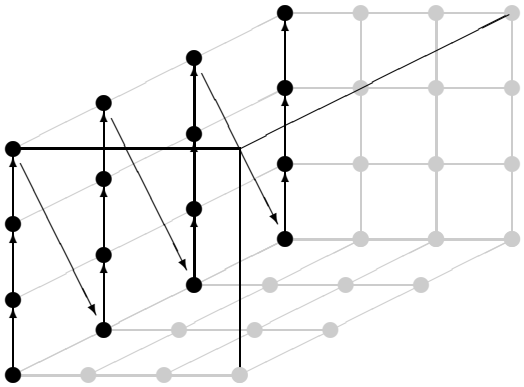


```
real(8) :: v(Nx, Ny, Nz)
```

C



```
double v[Nx][Ny];
```



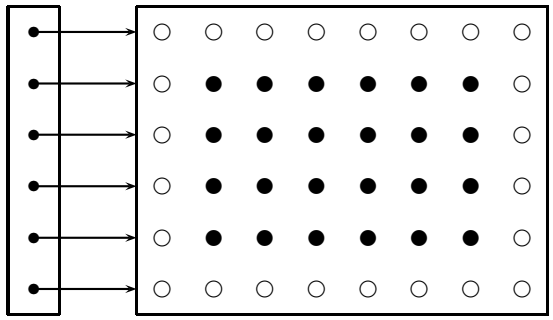
```
double v[Nx][Ny][Nz];
```

Data structures (II)

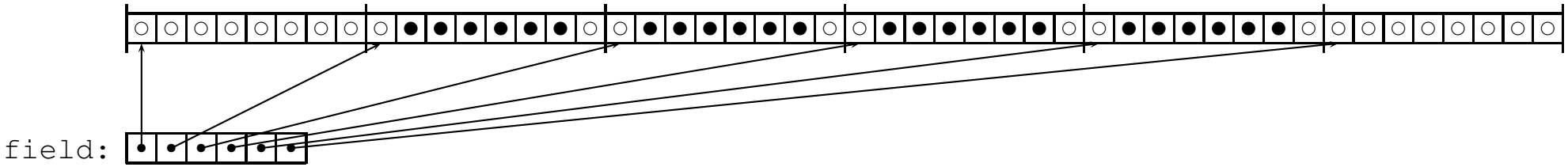
- `field` defines vectors of pointers to `double`: `typedef double **field;`

- geometric view

`field:`



- memory view



Data structures (III)

- usage of `field`:

```
void jacobi(field vnew, field vold, int Nx, int Ny)
{
    int x, y;

    for (y = 1; y <= Ny; y++)
        for (x = 1; x <= Nx; x++)
            vnew[y][x] = (vold[y][x - 1] + vold[y][x + 1]
                        + vold[y - 1][x] + vold[y + 1][x]) * 0.25;
}
```

Implementation: `field.h`

```
typedef double **field;

field field_alloc(int ny, int nx);
void field_free(field);
```

Implementation: `field.c`

```
# include <stdlib.h>
# include "field.h"

field field_alloc(int ny, int nx)
{
    ny += 2;  nx += 2;  // include boundary

    field tmp = (double **) malloc(ny * sizeof(double *));
    tmp[0] = (double *) malloc(nx * ny * sizeof(double));

    for (int i = 1; i < ny; i++)
        tmp[i] = tmp[0] + i * nx;

    return tmp;
}

void field_free(field x)
{
    free(x[0]);
    free(x);
}
```

Input

- there are 2 input files (identical for Fortran and C)

- parameters: `input.para`

```
10    Nx
20    Ny
1e-5  eps
```

- boundary values: `input.data` (format: *x*, *y*, *value*)

```
0        1        1.0
1        0        1.0
10       21       1.0
11       20       1.0
0        20      -1.0
1        21      -1.0
10       0       -1.0
11       1       -1.0
```

Output

```
#iterations: 269
  21       0.00000 -1.00000  0.00000  0.00000  0.00000  0.00000  0.00000  0.00000  0.00000  0.00000  1.00000  0.00000
  20      -1.00000 -0.60352 -0.20685 -0.07975 -0.03190 -0.00871  0.00871  0.03190  0.07975  0.20685  0.60352  1.00000
  19       0.00000 -0.20723 -0.14411 -0.08026 -0.03912 -0.01165  0.01165  0.03912  0.08026  0.14411  0.20723  0.00000
  18       0.00000 -0.08131 -0.08210 -0.05805 -0.03269 -0.01042  0.01042  0.03269  0.05805  0.08210  0.08131  0.00000
  17       0.00000 -0.03590 -0.04494 -0.03716 -0.02316 -0.00776  0.00776  0.02316  0.03716  0.04494  0.03590  0.00000
  16       0.00000 -0.01735 -0.02458 -0.02250 -0.01503 -0.00522  0.00522  0.01503  0.02250  0.02458  0.01735  0.00000
  15       0.00000 -0.00890 -0.01355 -0.01323 -0.00926 -0.00329  0.00329  0.00926  0.01323  0.01355  0.00890  0.00000
  14       0.00000 -0.00472 -0.00748 -0.00760 -0.00548 -0.00198  0.00198  0.00548  0.00760  0.00748  0.00472  0.00000
  13       0.00000 -0.00249 -0.00404 -0.00421 -0.00310 -0.00113  0.00113  0.00310  0.00421  0.00404  0.00249  0.00000
  12       0.00000 -0.00121 -0.00199 -0.00210 -0.00156 -0.00057  0.00057  0.00156  0.00210  0.00199  0.00121  0.00000
  11       0.00000 -0.00036 -0.00060 -0.00063 -0.00047 -0.00017  0.00017  0.00047  0.00063  0.00060  0.00036  0.00000
  10       0.00000  0.00036  0.00060  0.00063  0.00047  0.00017 -0.00017 -0.00047 -0.00063 -0.00060 -0.00036  0.00000
   9       0.00000  0.00121  0.00199  0.00210  0.00156  0.00057 -0.00057 -0.00156 -0.00210 -0.00199 -0.00121  0.00000
   8       0.00000  0.00249  0.00404  0.00421  0.00310  0.00113 -0.00113 -0.00310 -0.00421 -0.00404 -0.00249  0.00000
   7       0.00000  0.00472  0.00748  0.00760  0.00548  0.00198 -0.00198 -0.00548 -0.00760 -0.00748 -0.00472  0.00000
   6       0.00000  0.00890  0.01355  0.01323  0.00926  0.00329 -0.00329 -0.00926 -0.01323 -0.01355 -0.00890  0.00000
   5       0.00000  0.01735  0.02458  0.02250  0.01503  0.00522 -0.00522 -0.01503 -0.02250 -0.02458 -0.01735  0.00000
   4       0.00000  0.03590  0.04494  0.03716  0.02316  0.00776 -0.00776 -0.02316 -0.03716 -0.04494 -0.03590  0.00000
   3       0.00000  0.08131  0.08210  0.05805  0.03269  0.01042 -0.01042 -0.03269 -0.05805 -0.08210 -0.08131  0.00000
   2       0.00000  0.20723  0.14411  0.08026  0.03912  0.01165 -0.01165 -0.03912 -0.08026 -0.14411 -0.20723  0.00000
   1       1.00000  0.60352  0.20685  0.07975  0.03190  0.00871 -0.00871 -0.03190 -0.07975 -0.20685 -0.60352 -1.00000
   0       0.00000  1.00000  0.00000  0.00000  0.00000  0.00000  0.00000  0.00000  0.00000  0.00000 -1.00000  0.00000
   ^
   y   x >     0        1        2        3        4        5        6        7        8        9       10       11
```

Implementation: `output.c`

```
# include <stdio.h>
# include "laplace.h"

void output(field v, int Nx, int Ny)
{
    int x, y;

    for (y = Ny + 1; y >= 0; y--) {
        printf("%4i     ", y);
        for (x = 0; x <= Nx + 1; x++)
            printf("%9.5f", v[y][x]);
        printf("\n");
    }

    printf("    ^\n");
    printf("    y  x >");
    for (x = 0; x <= Nx + 1; x++)
        printf("%6i    ", x);
    printf("\n");
}
```

Implementation: `output.f90`

```
subroutine output(v, Nx, Ny)

  implicit none
  integer, intent(in) :: Nx, Ny
  real(8), intent(in) :: v(0:Nx + 1, 0:Ny + 1)
  integer :: x, y

  do y = Ny + 1, 0, -1
     write(6, "(i4,5x)", advance = "no") y
     do x = 0, Nx + 1
        write(6, "(f9.5)", advance = "no") v(x, y)
     enddo
     write(6,*)
  enddo

  write(6, "(a)") "    ^"
  write(6, "(a)", advance = "no") "    y   x >"
  do x = 0, Nx + 1
     write(6, "(i6,3x)", advance = "no") x
  enddo
  write(6,*)
end
```

Output

- reference output files

  `laplace.out` (5-point stencil)
  `laplace9.out` (9-point stencil)

- results are not identical because
  - the mesh is very small
  - boundary conditions are not smooth

Utility routine: `die.c`

- sequential program

```
void die(char *msg)
{
    fputs("laplace: error: ", stderr);
    fputs(msg, stderr);
    putc('\n', stderr);
    exit(1);
}
```

- MPI program

```
void die(char *msg)
{
    fputs("laplace: error: ", stderr);
    fputs(msg, stderr);
    putc('\n', stderr);
    MPI_Abort(MPI_COMM_WORLD, 1);
}
```

Unitility routine: `die.f90`

- MPI program

```
subroutine die(msg)
   implicit none
   character(len = *), intent(in) :: msg

   write(0,*) "laplace: error: ", msg
   call exit(1)
end
```

- MPI program

```
subroutine die(msg)
   implicit none
   include "mpif.h"
   character(len = *), intent(in) :: msg
   integer :: ierror

   write(0,*) "laplace: error: ", msg
   call mpi_abort(MPI_COMM_WORLD, 1, ierror)
end
```

Calling tree (sequential program)

```
main ----- laplace ---------+- init
                            |
                            +- jacobi
                            |
                            +- diff
                            |
                            +- output


die
```

Files (sequential program)

`Makefile`	`Makefile`
`field.h`	
`field.c`	
`laplace.h`	
`input.para`	`input.para`
`input.data`	`input.data`
`main.c`	`main.f90`
`laplace.c`	`laplace.f90`
`init.c`	`init.f90`
`jacobi.c`	`jacobi.f90`
`jacobi9.c`	`jacobi9.f90`
`diff.c`	`diff.f90`
`output.c`	`output.f90`
`die.c`	`die.f90`
`laplace.out`	`laplace.out`
`laplace9.out`	`laplace.9out`

Calling tree (sequential program)

```
main ----- laplace ---------+- init
                            |
                            +- jacobi
                            |
                            +- diff
                            |
                            +- output


die
```

main.c

```
1  #include <stdio.h>
2  #include "laplace.h"
3
4  int main(int argc, char *argv[])
5  {
6      int Nx, Ny;
7      double eps;
8
9      FILE *para = fopen("input.para", "r");
10
11     fscanf(para, "%d %*s", &Nx);
12     fscanf(para, "%d %*s", &Ny);
13     fscanf(para, "%lg %*s", &eps);
14
15     fclose(para);
16
17     laplace(Nx, Ny, eps);
18
19     return 0;
20 }
```

laplace.h

```
1 #include "field.h"
2
3 void die(char *msg);
4 void laplace(int Nx, int Ny, double eps);
5 void init(field v1, field v2, int Nx, int Ny);
6 void jacobi(field vnew, field vold, int Nx, int Ny);
7 void jacobi9(field vnew, field vold, int Nx, int Ny);
8 double diff(field v1, field v2, int Nx, int Ny);
9 void output(field v, int Nx, int Ny);
```

main.f90

```
1  program main
2
3    implicit none
4
5    integer :: Nx, Ny
6    real(8) :: eps
7
8    open(1, file = "input.para", status = "old", action = "read")
9
10   read(1,*) Nx
11   read(1,*) Ny
12   read(1,*) eps
13
14   close(1)
15
16   call laplace(Nx, Ny, eps)
17 end
```

Calling tree (sequential program)

```
main ----- laplace ---------+- init
                            |
                            +- jacobi
                            |
                            +- diff
                            |
                            +- output


die
```

laplace.c

```
1  #include <stdio.h>
2  #include "laplace.h"
3
4  void laplace(int Nx, int Ny, double eps)
5  {
6      field vold = field_alloc(Ny, Nx);
7      field vnew = field_alloc(Ny, Nx);
8      field tmp;
9      int   iter, max_iter = 10000;
10
11     init(vnew, vold, Nx, Ny);
12
13     for (iter = 1; iter <= max_iter; iter++) {
14         tmp = vnew;
15         vnew = vold;
16         vold = tmp;
17         jacobi(vnew, vold, Nx, Ny);  // or: jacobi9(vnew, vold, Nx, Ny)
18         if (diff(vnew, vold, Nx, Ny) < eps) break;
19     }
20
21     printf("#iterations: %i\n", iter);
22     if (iter > max_iter) die("no convergence");
23
24     output(vnew, Nx, Ny);  field_free(vold);  field_free(vnew);
25 }
```

laplace.f90

```
1  subroutine laplace(Nx, Ny, eps)
2    implicit none
3    integer, intent(in)                  :: Nx, Ny
4    real(8), intent(in)                  :: eps
5    real(8), dimension (:, :), pointer :: vnew, vold, tmp
6    real(8), external                    :: diff
7    integer                              :: iter, max_iter = 10000
8
9    allocate(vnew(0:Nx + 1, 0:Ny + 1), vold(0:Nx + 1, 0:Ny + 1))
10   call init(vnew, vold, Nx, Ny)
11
12   do iter = 1, max_iter
13      tmp => vnew
14      vnew => vold
15      vold => tmp
16      call jacobi(vnew, vold, Nx, Ny)   ! or: jacobi9(vnew, vold, Nx, Ny)
17      if (diff(vnew, vold, Nx, Ny) < eps) exit
18   enddo
19
20   write(6,"(a,i0)") "#iterations: ", iter
21   if (iter > max_iter) call die("no convergence")
22
23   call output(vnew, Nx, Ny)
24   deallocate(vnew, vold)
25 end
```

Calling tree (sequential program)

```
main ----- laplace ---------+- init
                            |
                            +- jacobi
                            |
                            +- diff
                            |
                            +- output


die
```

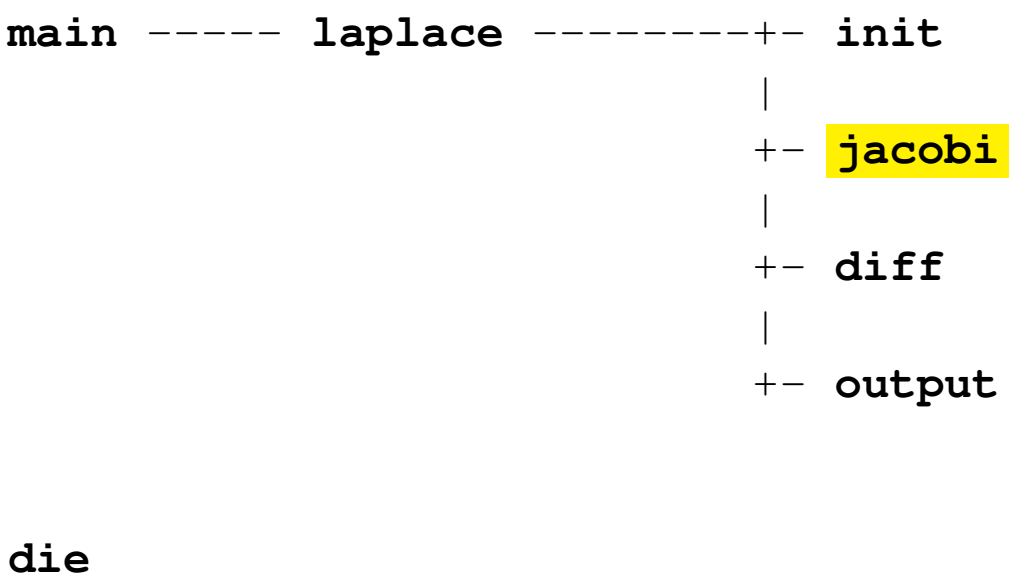
init.c

```
1  #include <stdio.h>
2  #include "laplace.h"
3
4  void init(field vnew, field vold, int Nx, int Ny)
5  {
6      int      x, y;
7      double   value;
8      FILE    *data;
9
10     for (y = 0; y <= Ny + 1; y++)
11         for (x = 0; x <= Nx + 1; x++) {
12             vnew[y][x] = 0;
13             vold[y][x] = 0;
14         }
15
16     data = fopen("input.data", "r");
17
18     while (fscanf(data, "%d %d %lg", &x, &y, &value) != EOF) {
19         vnew[y][x] = value;
20         vold[y][x] = value;
21     }
22
23     fclose(data);
24 }
```

init.f90

```
1  subroutine init(vnew, vold, Nx, Ny)
2
3    implicit none
4    integer, intent(in)  :: Nx, Ny
5    real(8), intent(out) :: vnew(0:Nx + 1, 0:Ny + 1)
6    real(8), intent(out) :: vold(0:Nx + 1, 0:Ny + 1)
7    integer              :: x, y, iostat
8    real(8)              :: value
9
10   vnew(:, :) = 0
11
12   open(2, file = "input.data", status = "old", action = "read")
13
14   do
15       read(2, *, iostat = iostat) x, y, value
16       if (iostat /= 0) exit
17       vnew(x, y) = value
18   enddo
19
20   close(2)
21
22   vold(:, :) = vnew(:, :)
23 end
```

Calling tree (sequential program)



jacobi.c

```
1  #include "laplace.h"
2
3  void jacobi(field vnew, field vold, int Nx, int Ny)
4  {
5      int x, y;
6
7      for (y = 1; y <= Ny; y++)
8          for (x = 1; x <= Nx; x++)
9              vnew[y][x] = (vold[y][x - 1] + vold[y][x + 1]
10                           + vold[y - 1][x] + vold[y + 1][x]) * 0.25;
11 }
```

jacobi9.c

```
1  #include "laplace.h"
2
3  void jacobi9(field vnew, field vold, int Nx, int Ny)
4  {
5      int x, y;
6
7      for (y = 1; y <= Ny; y++) {
8          for (x = 1; x <= Nx; x++) {
9              vnew[y][x] = 4.0 * (vold[y][x - 1] + vold[y][x + 1]
10                                 + vold[y - 1][x] + vold[y + 1][x])
11                                 + vold[y - 1][x - 1] + vold[y - 1][x + 1]
12                                 + vold[y + 1][x - 1] + vold[y + 1][x + 1];
13             vnew[y][x] *= 0.05;
14         }
15     }
16 }
```

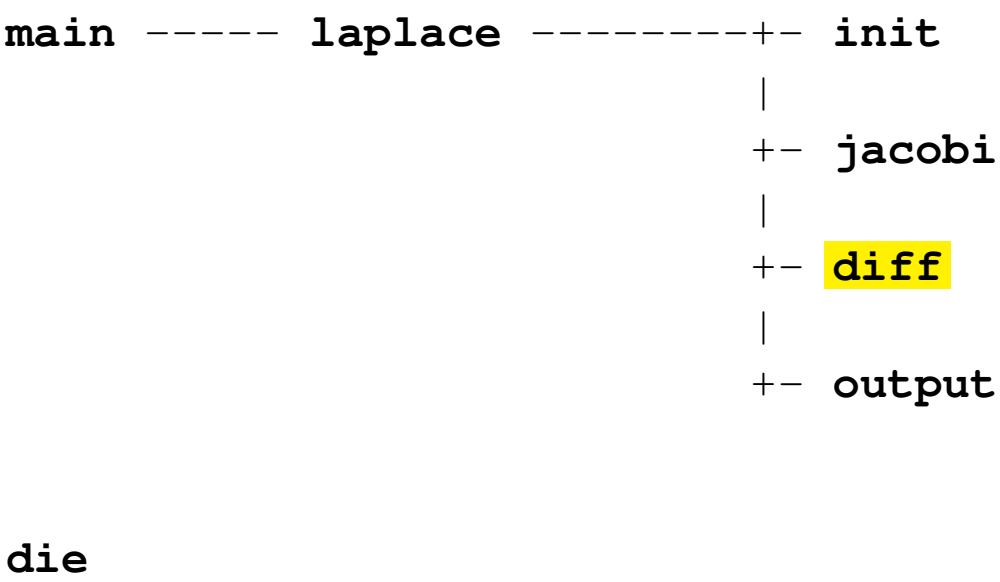
jacobi.f90

```
1  subroutine jacobi(vnew, vold, Nx, Ny)
2
3    implicit none
4    integer, intent(in)  :: Nx, Ny
5    real(8), intent(out) :: vnew(0:Nx + 1, 0:Ny + 1)
6    real(8), intent(in)  :: vold(0:Nx + 1, 0:Ny + 1)
7    integer              :: x, y
8
9    do y = 1, Ny
10      do x = 1, Nx
11         vnew(x, y) = (vold(x - 1, y) + vold(x + 1, y) &
12                     + vold(x, y - 1) + vold(x, y + 1)) * 0.25
13      end do
14   end do
15
16 end
```

jacobi9.f90

```
1  subroutine jacobi9(vnew, vold, Nx, Ny)
2
3    implicit none
4    integer, intent(in)  :: Nx, Ny
5    real(8), intent(out) :: vnew(0:Nx + 1, 0:Ny + 1)
6    real(8), intent(in)  :: vold(0:Nx + 1, 0:Ny + 1)
7    integer              :: x, y
8
9    do y = 1, Ny
10      do x = 1, Nx
11         vnew(x, y) = 4.0 * (vold(x - 1, y) + vold(x + 1, y) &
12                           + vold(x, y - 1) + vold(x, y + 1)) &
13                           + vold(x - 1, y - 1) + vold(x - 1, y + 1) &
14                           + vold(x + 1, y - 1) + vold(x + 1, y + 1)
15         vnew(x, y) = vnew(x, y) * 0.05
16      end do
17   end do
18
19 end
```

Calling tree (sequential program)



diff.c

```
1  #include <math.h>
2  #include "laplace.h"
3
4  double diff(field v1, field v2, int Nx, int Ny)
5  {
6      int    x, y;
7      double d, sum;
8
9      sum = 0;
10     for (y = 1; y <= Ny; y++)
11         for (x = 1; x <= Nx; x++) {
12             d = v1[y][x] - v2[y][x];
13             sum += d * d;
14         }
15
16     return sqrt(sum);
17 }
```

diff.f90

```
1  real(8) function diff(v1, v2, Nx, Ny)
2
3    implicit none
4    integer, intent(in) :: Nx, Ny
5    real(8), intent(in) :: v1(0:Nx + 1, 0:Ny + 1)
6    real(8), intent(in) :: v2(0:Nx + 1, 0:Ny + 1)
7    integer             :: x, y
8
9    diff = 0
10   do y = 1, Ny
11      do x = 1, Nx
12         diff = diff + (v1(x, y) - v2(x, y))**2
13      enddo
14   enddo
15   diff = sqrt(diff)
16
17 end
```

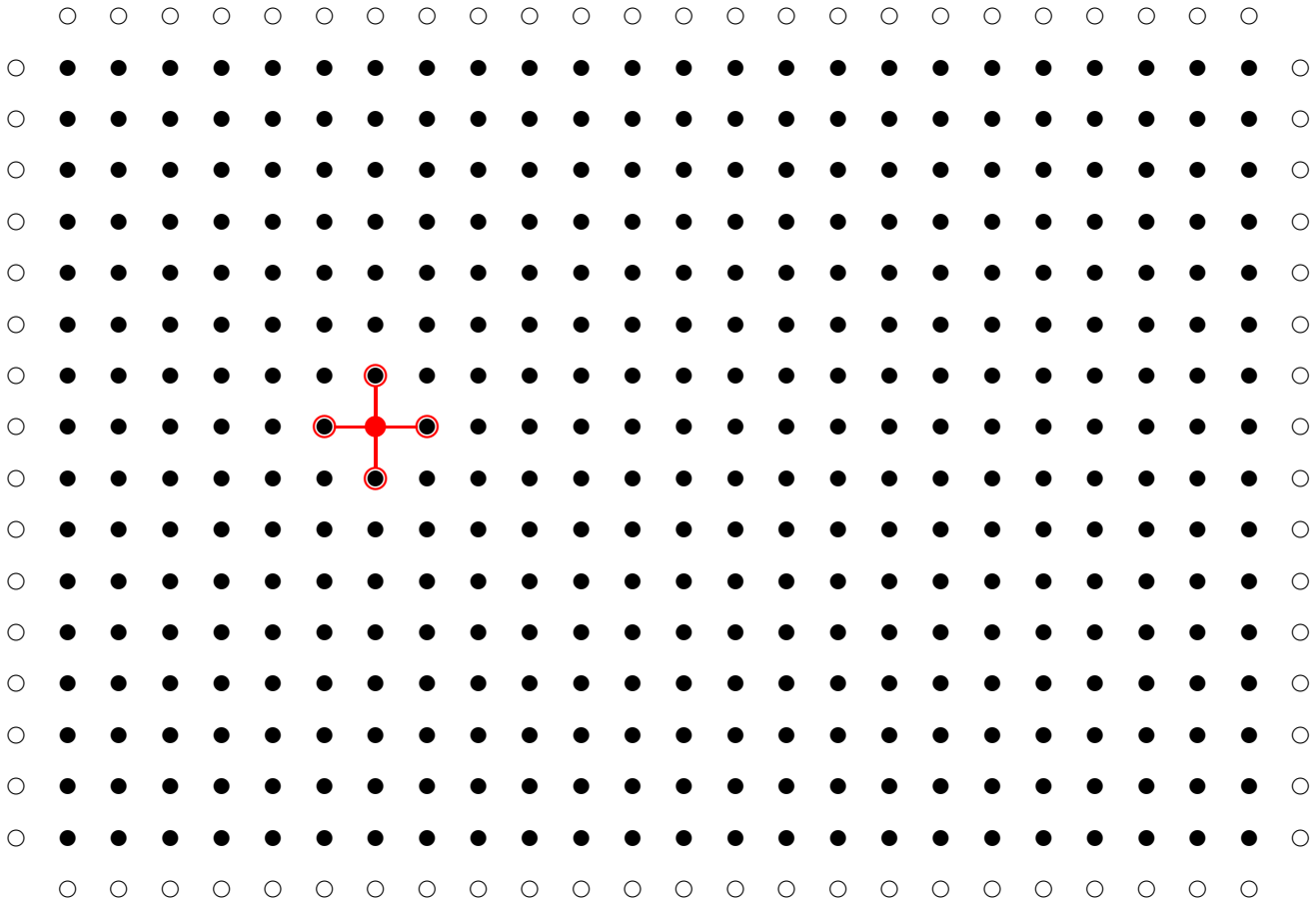
– MPI project –
parallel program

source code directories:

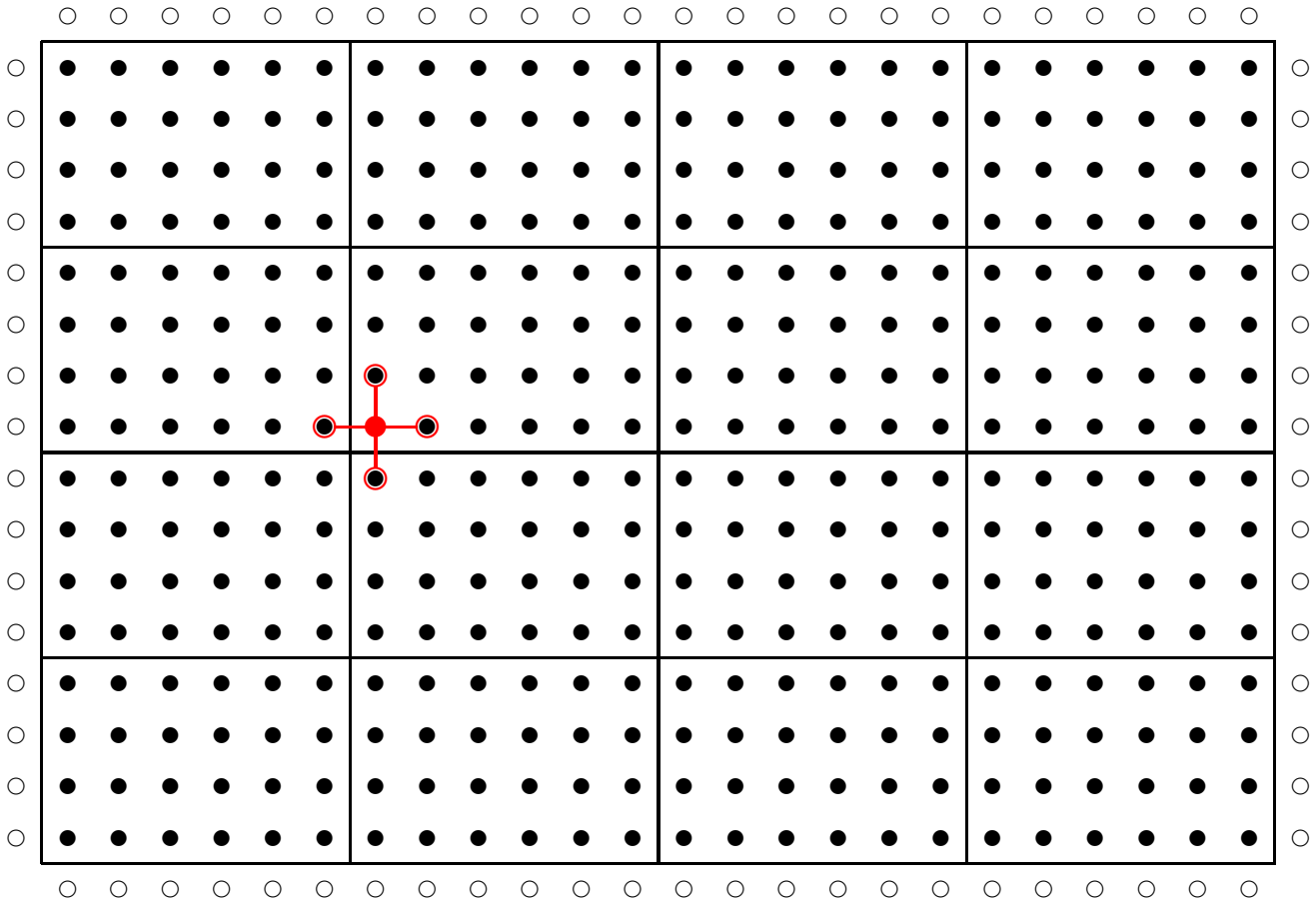
`./laplace-code/laplace-par-c`

`./laplace-code/laplace-par-f90`

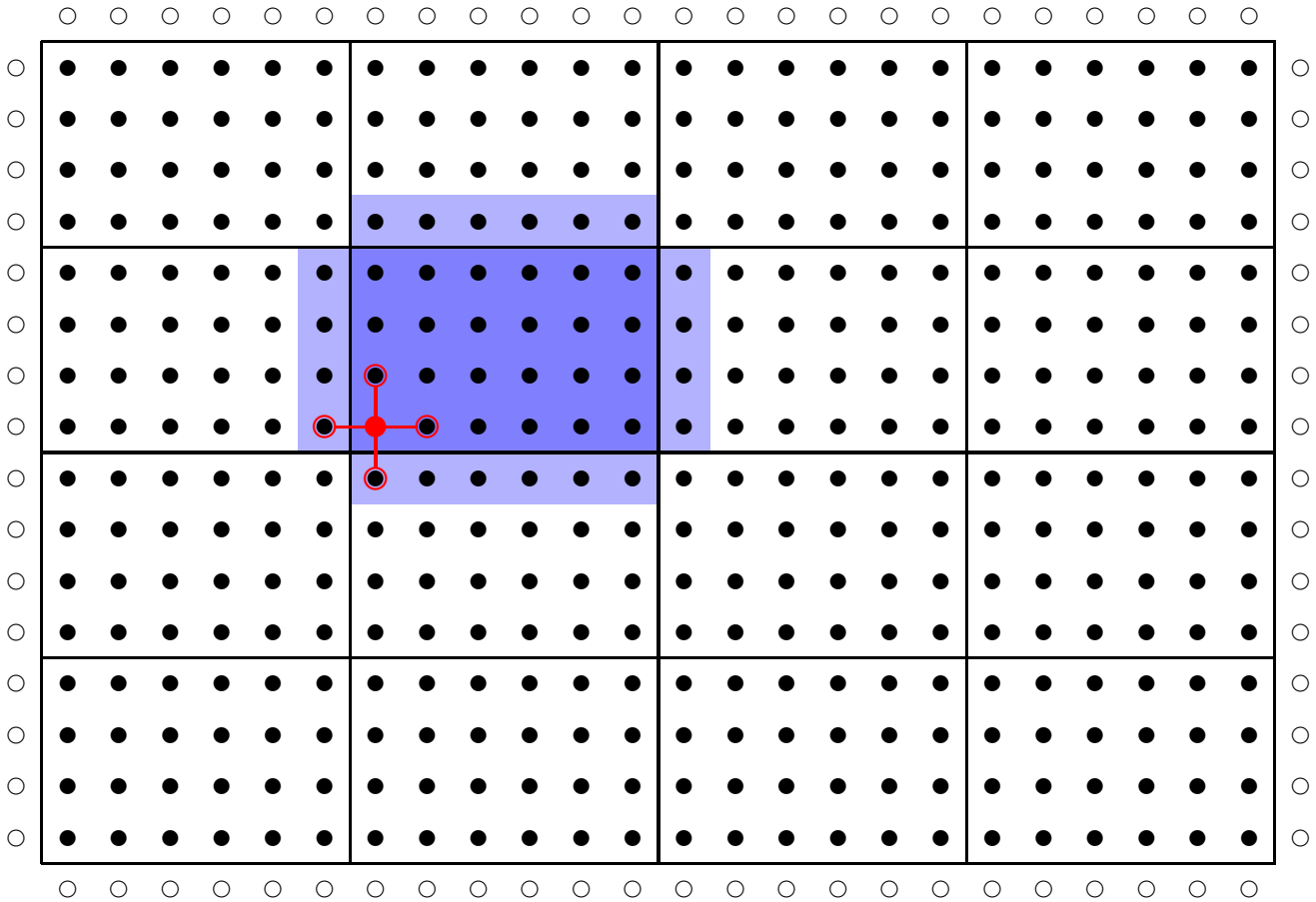
Laplace equation



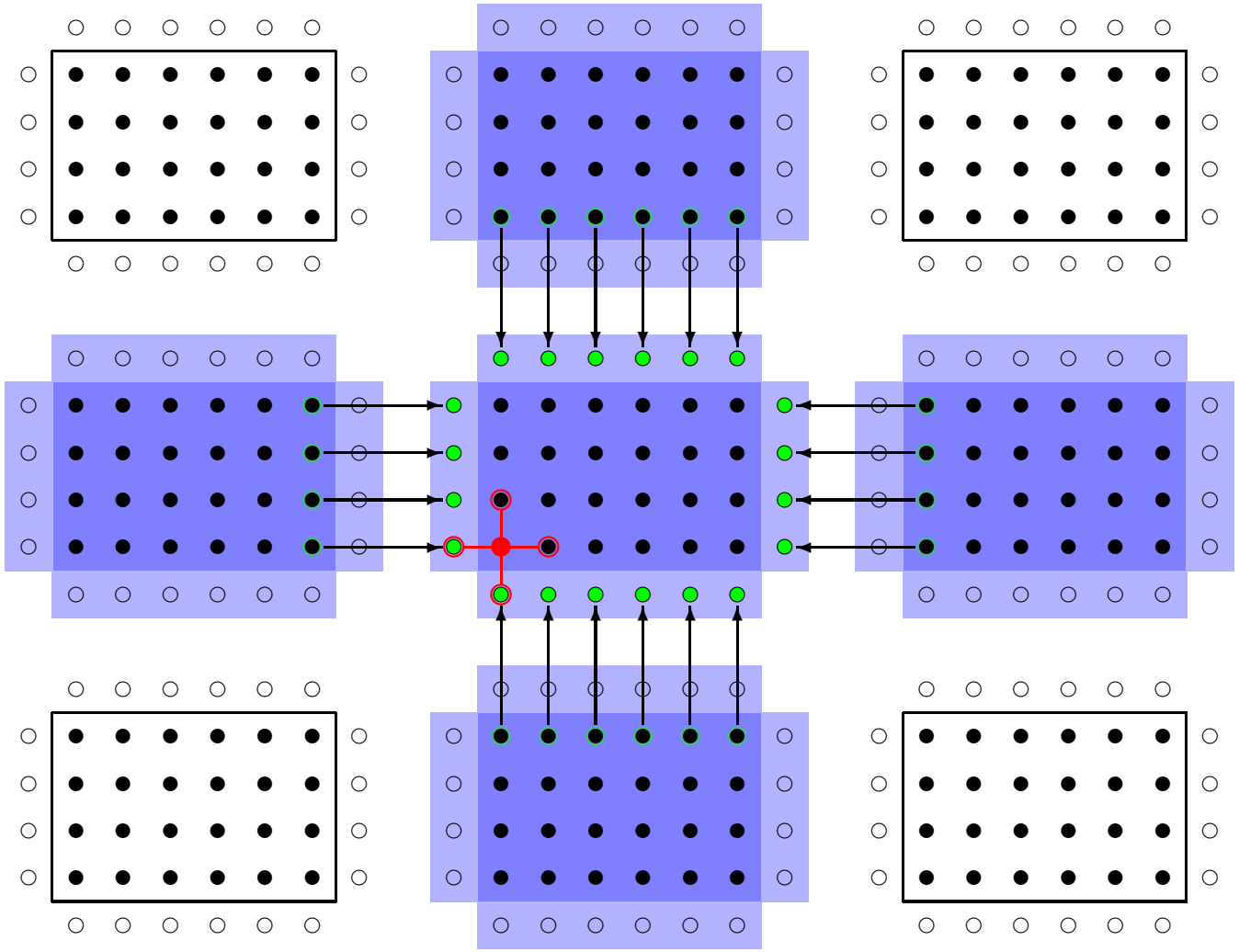
Laplace equation



Laplace equation



Laplace equation



Pseudo code

- Parallel program:

```
repeat

   v_old := v_new

   exchange boundary of v_old

   for x := 1 to N_x do
      for y := 1 to N_y do
         v_new[x, y] = (v_old[x-1, y] + v_old[x+1, y]
                      + v_old[x, y-1] + v_old[x, y+1]) / 4

until (convergence)
```

(N_x und N_y are adjusted according to the decomposition.)

Overview of tasks for parallelisation

- description of the decomposition / bookkeeping
  - process mapping
    - introduction of process coordinates
    - mapping: *process coordinates* $\leftrightarrow$ *ranks*
- input and output of data
  - mapping: *local* $\leftrightarrow$ *global* mesh coordinates
  - (parallel I/O $\rightarrow$ MPI-2)
- exchange of boundary data
  - definition of MPI datatypes ($\rightarrow$ programming project)
  - efficient communication ($\rightarrow$ programming project)
- global reduction operations ($\rightarrow$ programming project)

decomp.h

```
1  struct {
2
3    int Nx, Ny;     // global mesh size
4    int Lx, Ly;     // local mesh size
5
6    int procs_x;    // #processes in x-direction
7    int procs_y;    // #processes in y-direction
8    int processes; // total #processes
9
10   int coord_x;    // x-coordinate of this process
11   int coord_y;    // y-coordinate of this process
12
13   int my_rank;    // rank of this process
14   int north;      // rank of process neighbour in +y direction
15   int south;      // rank of process neighbour in -y direction
16   int east;       // rank of process neighbour in +x direction
17   int west;       // rank of process neighbour in -x direction
18
19 } decomp;
```

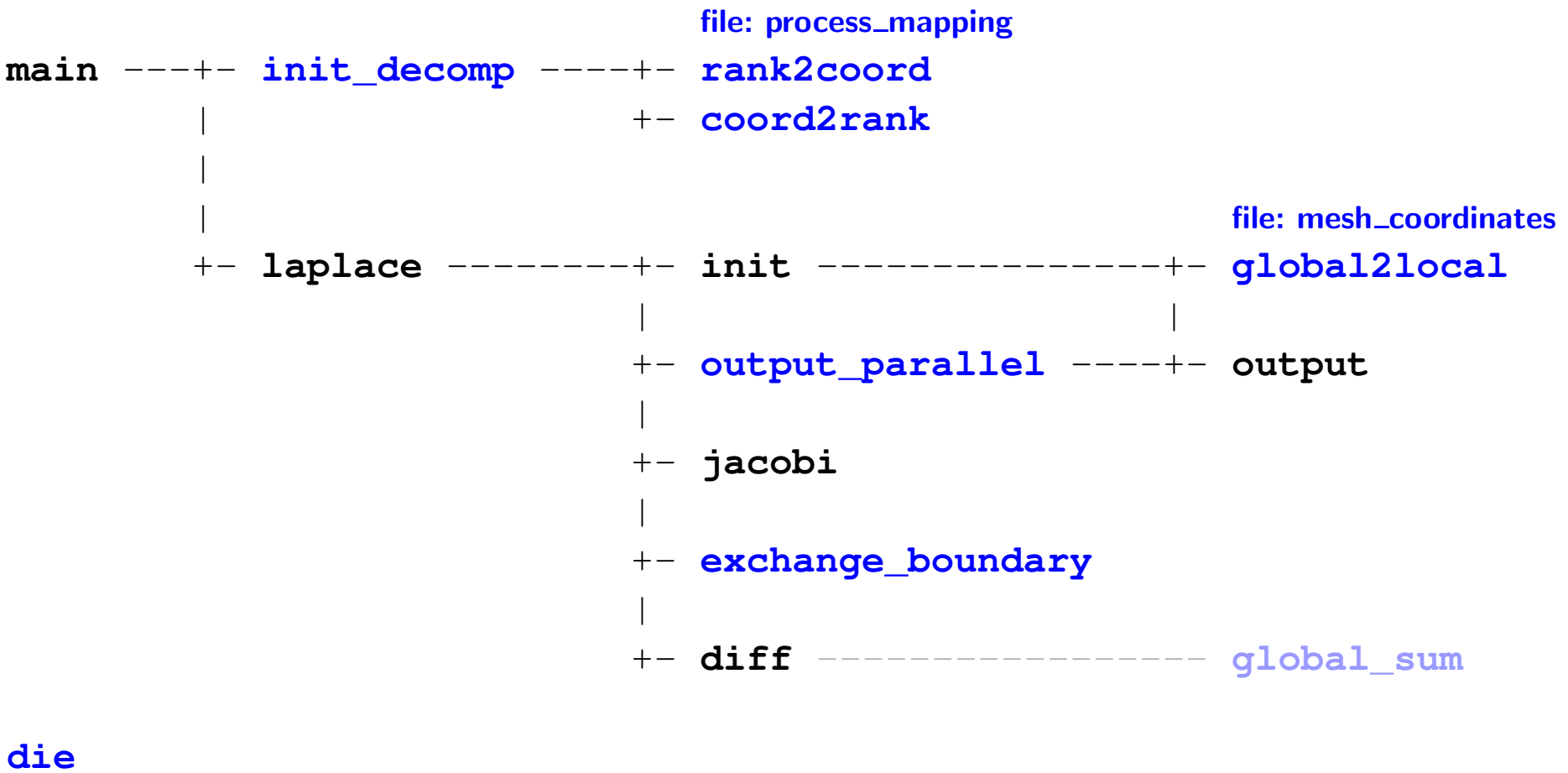
decomp.h

```
20
21 void init_decomp(void);
22 void coord2rank(int coord_x, int coord_y, int *rank);
23 void rank2coord(int rank, int *coord_x, int *coord_y);
24 void global2local(int x_global, int y_global,
25                   int *x_local, int *y_local, int *home_of_xy);
26 void global2local_x(int x_global, int *x_local, int *coord_x);
27 void global2local_y(int y_global, int *y_local, int *coord_y);
28 void exchange_boundary(field v, int Lx, int Ly);
29 double global_sum(double local_sum);
30 void output_parallel(field v, int Nx, int Ny);
```

module_decomp.f90

```
1  module module_decomp
2
3    type type_decomp
4        integer :: Nx, Ny    ! global mesh size
5        integer :: Lx, Ly    ! local mesh size
6
7        integer :: procs_x   ! #processes in x-direction
8        integer :: procs_y   ! #processes in y-direction
9        integer :: processes ! total #processes
10
11       integer :: coord_x   ! x-coordinate of this process
12       integer :: coord_y   ! y-coordinate of this process
13
14       integer :: my_rank   ! rank of this process
15       integer :: north     ! rank of process neighbour in +y direction
16       integer :: south     ! rank of process neighbour in -y direction
17       integer :: east      ! rank of process neighbour in +x direction
18       integer :: west      ! rank of process neighbour in -x direction
19   end type type_decomp
20
21   type (type_decomp), save :: decomp
22
23 end module module_decomp
```

Calling tree (parallel program)



Usage

- command line

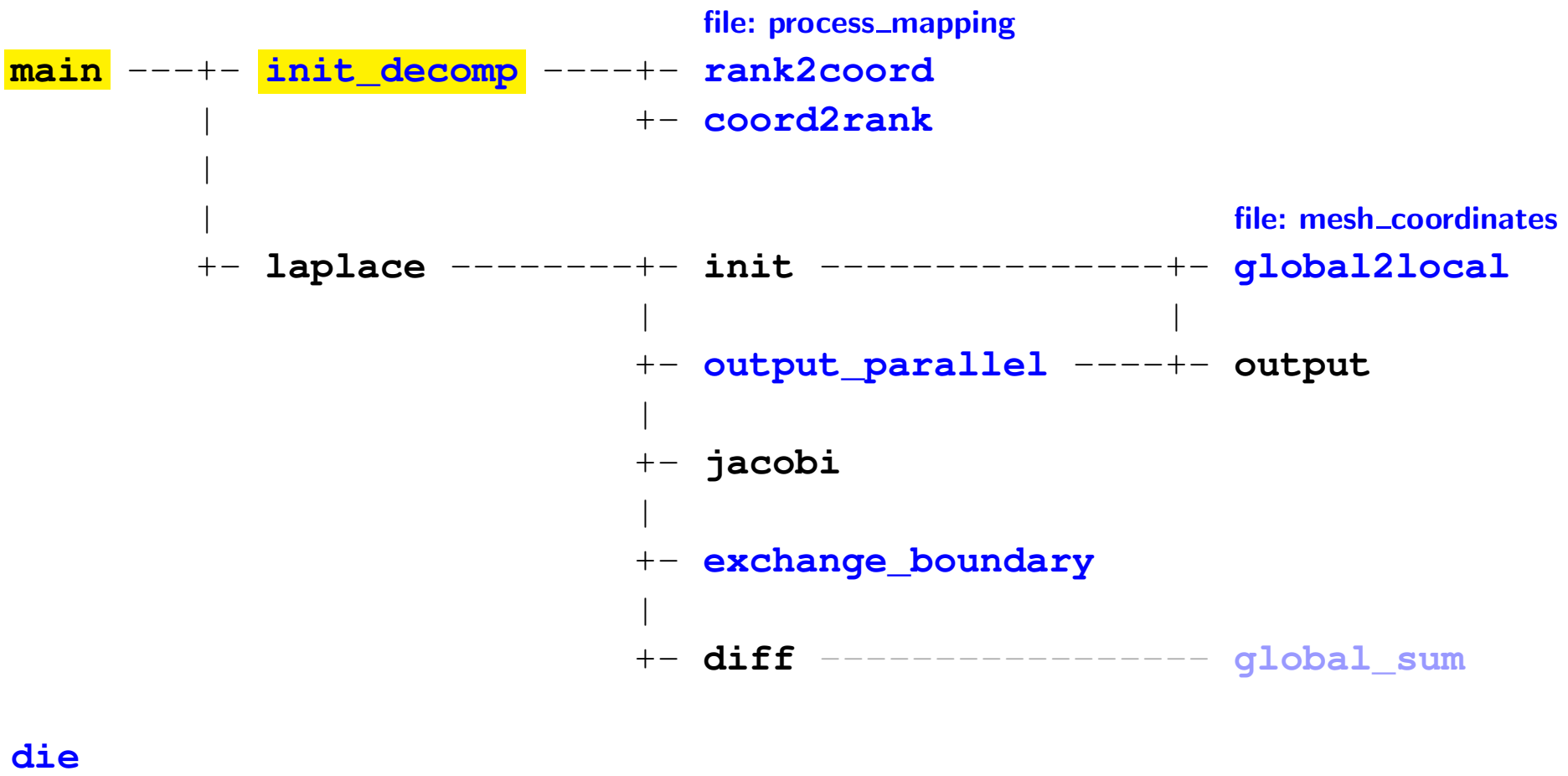
  ```
  mpirun -np N ./laplace procs_x procs_y
  ```

  where $N = procs\_x \times procs\_y$

- testing

  ```
  mpirun -np N ./laplace procs_x procs_y | diff - laplace.out
  mpirun -np N ./laplace procs_x procs_y | diff - laplace9.out
  ```

Calling tree



main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4 #include "laplace.h"
5 #include "decomp.h"
6
```

main.c

```
7  int main(int argc, char *argv[])
8  {
9      double eps;
10
11     MPI_Init(&argc, &argv);
12
13     if (argc != 3) die("Usage: laplace procs_x procs_y");
14
15     decomp.procs_x = atoi(argv[1]);
16     decomp.procs_y = atoi(argv[2]);
17
18     FILE *para = fopen("input.para", "r");
19     fscanf(para, "%d %*s", &decomp.Nx);
20     fscanf(para, "%d %*s", &decomp.Ny);
21     fscanf(para, "%lg %*s", &eps);
22     fclose(para);
23
24     init_decomp();
25     laplace(decomp.Lx, decomp.Ly, eps);
26
27     MPI_Finalize();
28     return 0;
29 }
```

main.f90

```
1  program main
2    use module_decomp
3    implicit none
4    include "mpif.h"
5    real(8) :: eps
6    integer :: ierror
7    character(32) :: arg
8
9    call mpi_init(ierror)
10
11   if (iargc() /= 2) call die("Usage: laplace procs_x procs_y")
12   call getarg(1, arg) ;  read(arg, *) decomp%procs_x
13   call getarg(2, arg) ;  read(arg, *) decomp%procs_y
14
15   open(1, file = "input.para", status = "old", action = "read")
16   read(1,*) decomp%Nx
17   read(1,*) decomp%Ny
18   read(1,*) eps
19   close(1)
20
21   call init_decomp()
22   call laplace(decomp%Lx, decomp%Ly, eps)
23
24   call mpi_finalize(ierror)
25 end
```

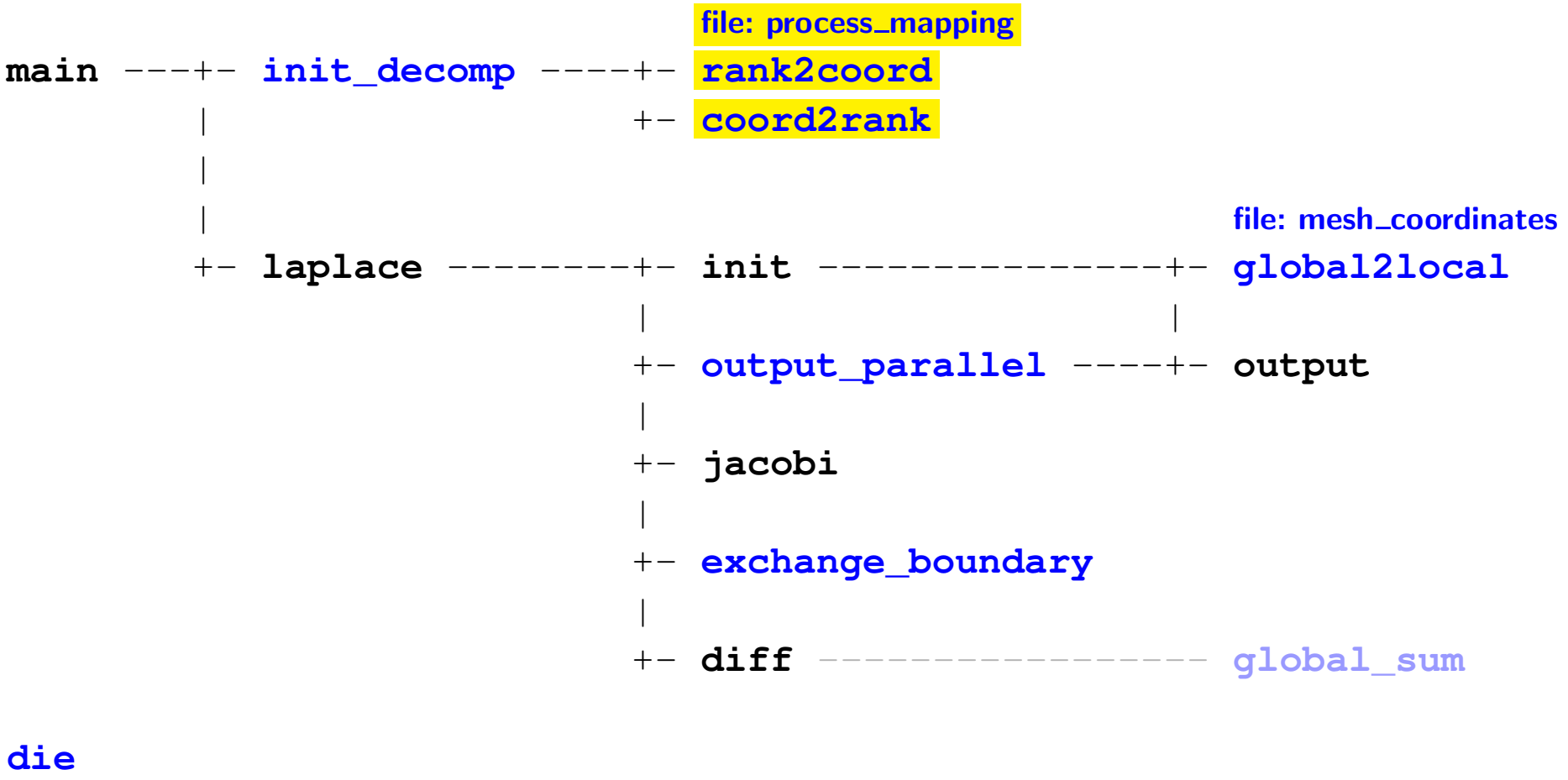
init_decomp.c

```
1  #include <mpi.h>
2  #include "laplace.h"
3  #include "decomp.h"
4
5  void init_decomp(void)
6  {
7      MPI_Comm_size(MPI_COMM_WORLD, &decomp.processes);
8      MPI_Comm_rank(MPI_COMM_WORLD, &decomp.my_rank);
9
10     decomp.Lx = decomp.Nx / decomp.procs_x;
11     decomp.Ly = decomp.Ny / decomp.procs_y;
12
13     if (decomp.procs_x * decomp.procs_y != decomp.processes)
14         die("procs_x * procs_y != processes");
15
16     if (decomp.Nx % decomp.procs_x) die("mod(Nx, procs_x) != 0");
17     if (decomp.Ny % decomp.procs_y) die("mod(Ny, procs_y) != 0");
18
19     rank2coord(decomp.my_rank, &decomp.coord_x, &decomp.coord_y);
20     coord2rank(decomp.coord_x, decomp.coord_y + 1, &decomp.north);
21     coord2rank(decomp.coord_x, decomp.coord_y - 1, &decomp.south);
22     coord2rank(decomp.coord_x + 1, decomp.coord_y, &decomp.east);
23     coord2rank(decomp.coord_x - 1, decomp.coord_y, &decomp.west);
24 }
```

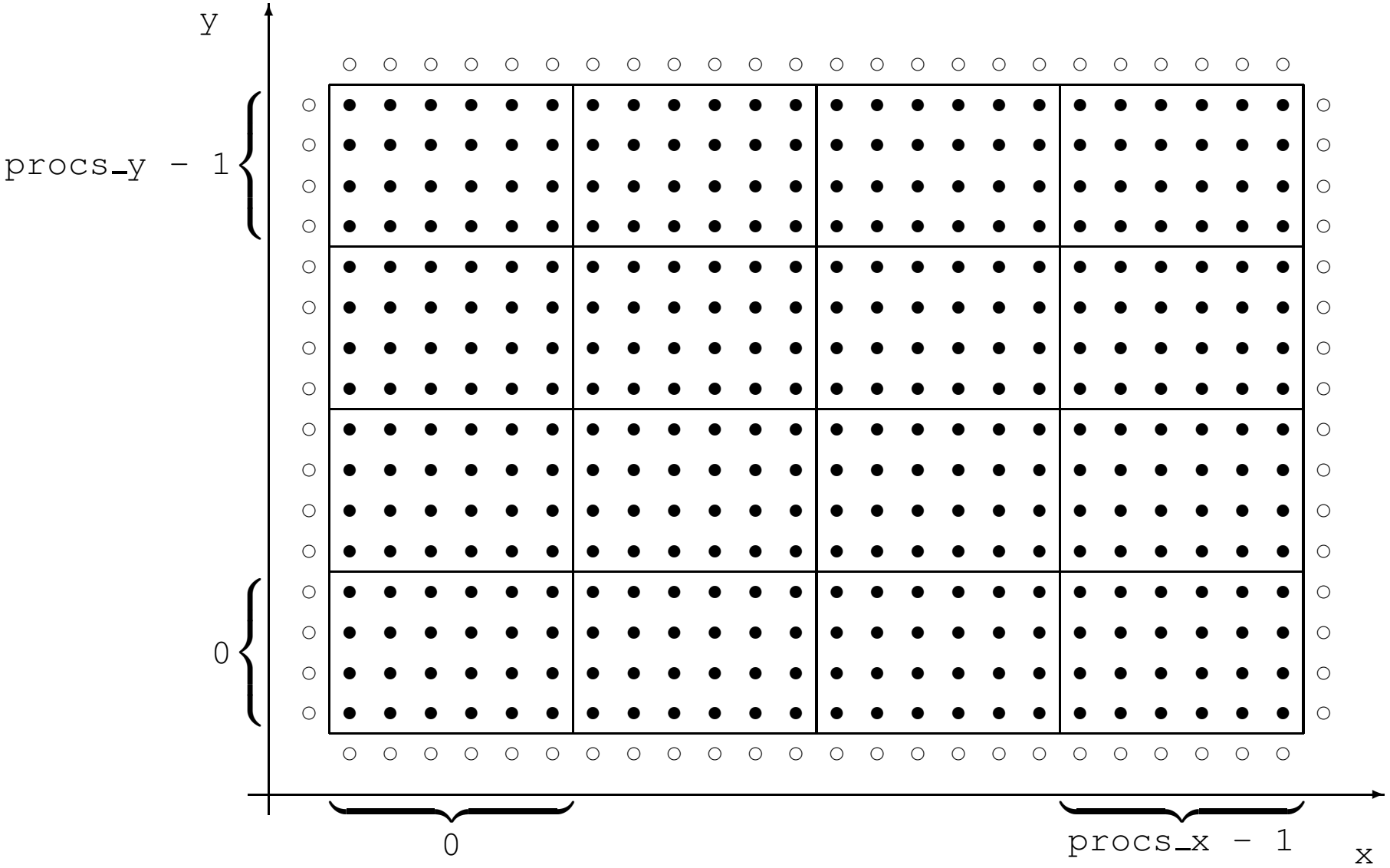
init_decomp.f90

```
1  subroutine init_decomp()
2    use module_decomp
3    implicit none
4    include "mpif.h"
5    integer :: ierror
6
7    call mpi_comm_size(MPI_COMM_WORLD, decomp%processes, ierror)
8    call mpi_comm_rank(MPI_COMM_WORLD, decomp%my_rank, ierror)
9
10   decomp%Lx = decomp%Nx / decomp%procs_x
11   decomp%Ly = decomp%Ny / decomp%procs_y
12
13   if (decomp%procs_x * decomp%procs_y /= decomp%processes) then
14      call die("procs_x * procs_y /= #processes")
15   endif
16
17   if (mod(decomp%Nx, decomp%procs_x) /= 0) call die("mod(Nx, procs_x) /= 0'
18   if (mod(decomp%Ny, decomp%procs_y) /= 0) call die("mod(Ny, procs_y) /= 0'
19
20   call rank2coord(decomp%my_rank, decomp%coord_x, decomp%coord_y)
21   call coord2rank(decomp%coord_x, decomp%coord_y + 1, decomp%north)
22   call coord2rank(decomp%coord_x, decomp%coord_y - 1, decomp%south)
23   call coord2rank(decomp%coord_x + 1, decomp%coord_y, decomp%east)
24   call coord2rank(decomp%coord_x - 1, decomp%coord_y, decomp%west)
25 end
```

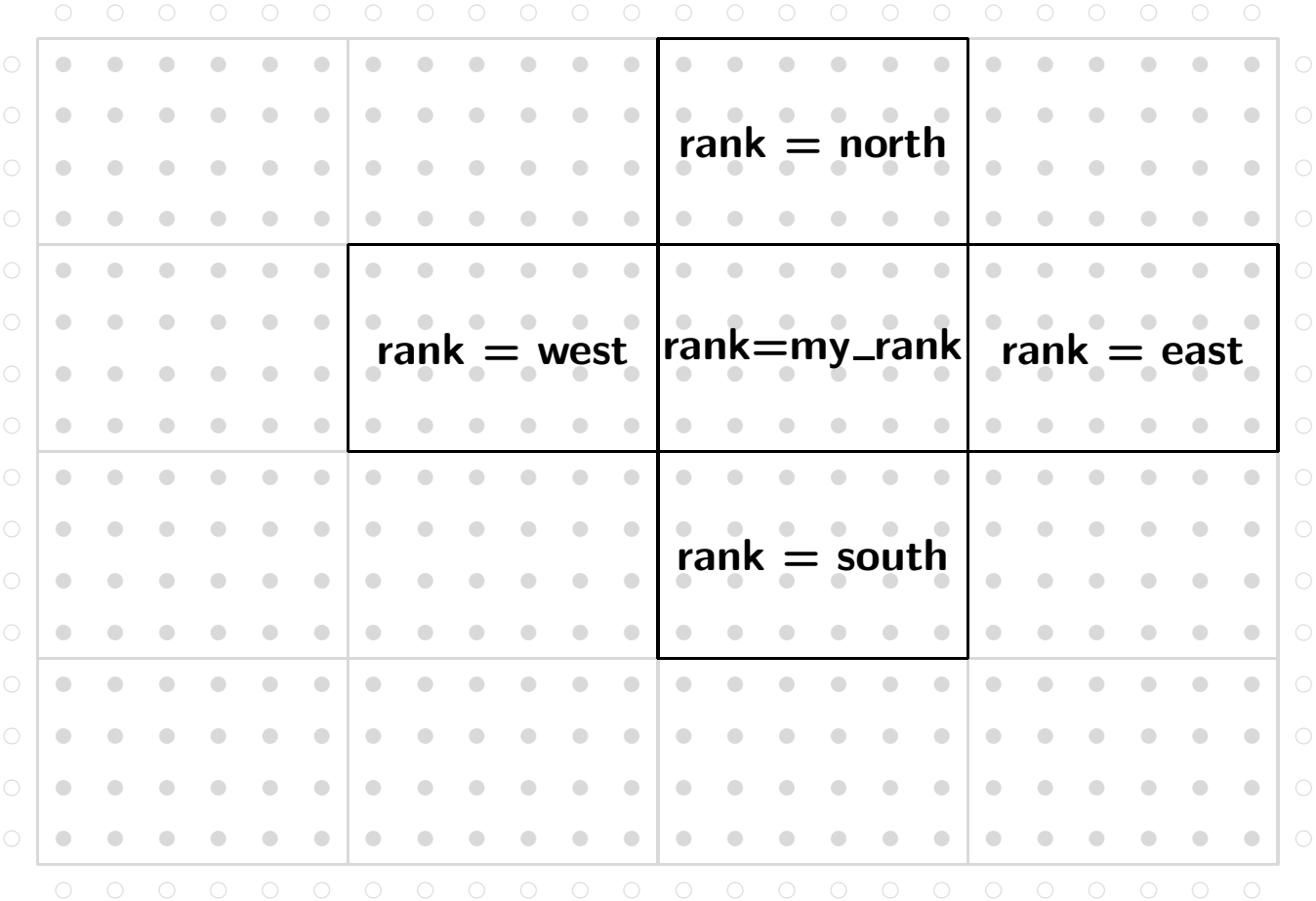
Calling tree



Process mapping (I)



Process mapping (II)



Process mapping (III)

coords = (0,3) rank = 12	coords = (1,3) rank = 13	coords = (2,3) rank = 14	coords = (3,3) rank = 15
coords = (0,2) rank = 8	coords = (1,2) rank = 9	coords = (2,2) rank = 10	coords = (3,2) rank = 11
coords = (0,1) rank = 4	coords = (1,1) rank = 5	coords = (2,1) rank = 6	coords = (3,1) rank = 7
coords = (0,0) rank = 0	coords = (1,0) rank = 1	coords = (2,0) rank = 2	coords = (3,0) rank = 3

```
rank = coord_x + coord_y * procs_x
```

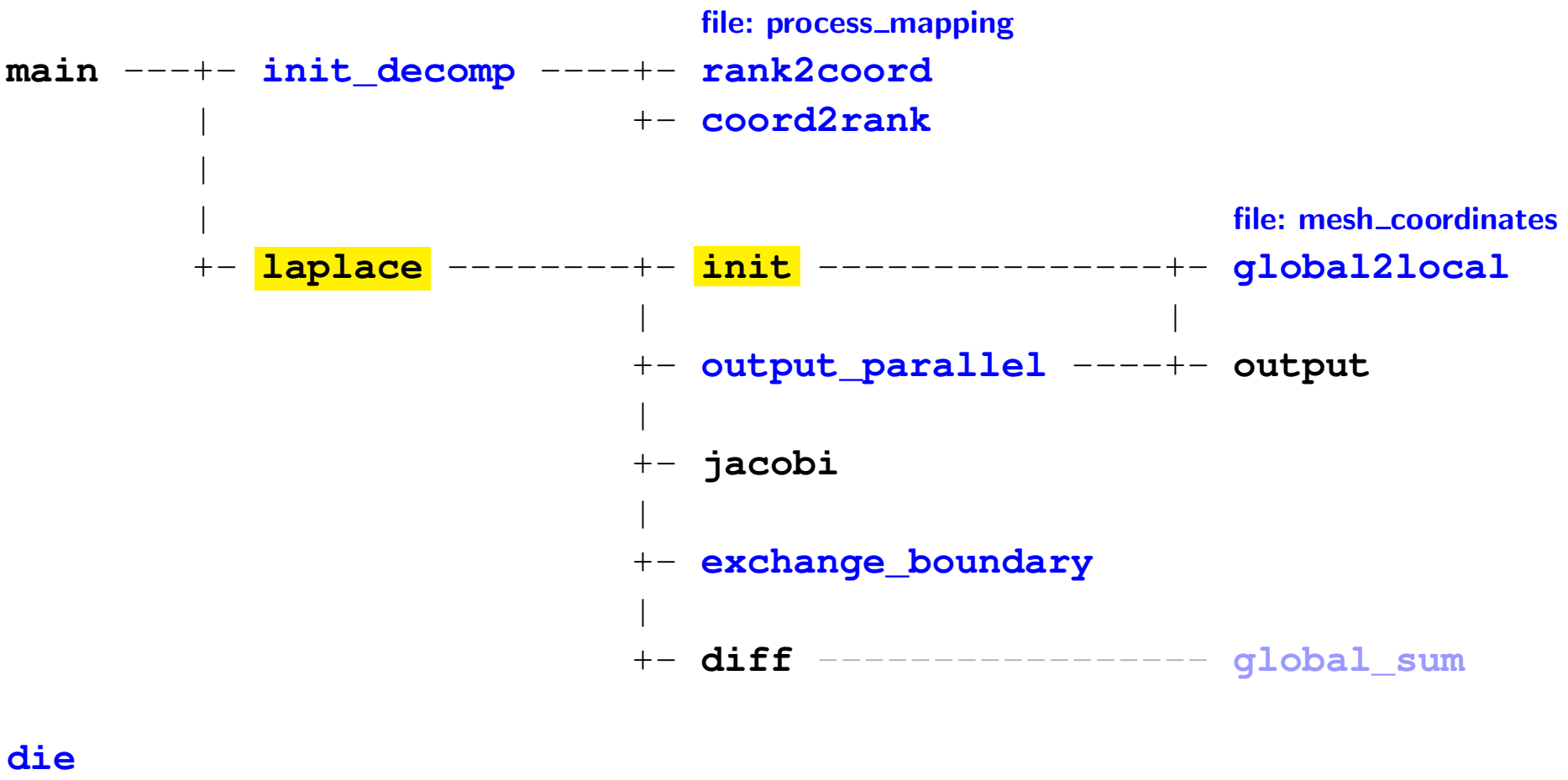
process_mapping.c

```
1  #include <mpi.h>
2  #include "laplace.h"
3  #include "decomp.h"
4
5  void coord2rank(int coord_x, int coord_y, int *rank)
6  {
7      if (coord_x < 0 || coord_x >= decomp.procs_x ||
8          coord_y < 0 || coord_y >= decomp.procs_y
9          )
10         *rank = -1;
11     else
12         *rank = coord_x + coord_y * decomp.procs_x;
13 }
14
15 void rank2coord(int rank, int *coord_x, int *coord_y)
16 {
17     *coord_x = rank % decomp.procs_x;
18     *coord_y = rank / decomp.procs_x;
19 }
```

process_mapping.f90

```
1  subroutine coord2rank(coord_x, coord_y, rank)
2    use module_decomp
3    implicit none
4    integer, intent(in)  :: coord_x, coord_y
5    integer, intent(out) :: rank
6
7    if (coord_x < 0 .or. coord_x >= decomp%procs_x .or. &
8        coord_y < 0 .or. coord_y >= decomp%procs_y) then
9       rank = -1
10   else
11      rank = coord_x + coord_y * decomp%procs_x
12   endif
13 end
14
15 subroutine rank2coord(rank, coord_x, coord_y)
16   use module_decomp
17   implicit none
18   integer, intent(in)  :: rank
19   integer, intent(out) :: coord_x, coord_y
20
21   coord_x = mod(rank, decomp%procs_x)
22   coord_y = rank / decomp%procs_x
23 end
```

Calling tree



laplace.c

```
1  #include <stdio.h>
2  #include <mpi.h>
3  #include "laplace.h"
4  #include "decomp.h"
5
6  void laplace(int Nx, int Ny, double eps)
7  {
8      field vold = field_alloc(Ny, Nx);
9      field vnew = field_alloc(Ny, Nx);
10     field tmp;
11     const int max_iter = 10000;
12     int iter;
13     double time;
14
```

laplace.c

```
15      init(vnew, vold, Nx, Ny);
16
17      time = MPI_Wtime();
18      for (iter = 1; iter <= max_iter; iter++) {
19          tmp = vnew;
20          vnew = vold;
21          vold = tmp;
22          exchange_boundary(vold, Nx, Ny);
23          jacobi(vnew, vold, Nx, Ny);  // or: jacobi9(vnew, vold, Nx, Ny)
24          if (diff(vnew, vold, Nx, Ny) < eps) break;
25      }
26      time = MPI_Wtime() - time;
27
28      if (decomp.my_rank == 0) {
29          printf("#iterations: %i\n", iter);
30          fprintf(stderr, "time(iterations) = %8.2e s\n", time);
31      }
32      if (iter > max_iter) die("no convergence");
33
34      output_parallel(vnew, Nx, Ny);
35
36      field_free(vold);
37      field_free(vnew);
38  }
```

laplace.f90

```
1  subroutine laplace(Nx, Ny, eps)
2
3    use module_decomp
4    implicit none
5    include "mpif.h"
6    integer, intent(in)                  :: Nx, Ny
7    real(8), intent(in)                  :: eps
8    real(8), dimension (:, :), pointer :: vnew, vold, tmp
9    integer, parameter                   :: max_iter = 10000
10   integer                              :: iter
11   real(8), external                    :: diff
12   real(8)                              :: time
13
14   allocate(vnew(0:Nx + 1, 0:Ny + 1), vold(0:Nx + 1, 0:Ny + 1))
15
```

laplace.f90

```
16    call init(vnew, vold, Nx, Ny)
17
18    time = mpi_wtime()
19    do iter = 1, max_iter
20       tmp => vnew
21       vnew => vold
22       vold => tmp
23       call exchange_boundary(vold, Nx, Ny)
24       call jacobi(vnew, vold, Nx, Ny)  ! or: jacobi9(vnew, vold, Nx, Ny)
25       if (diff(vnew, vold, Nx, Ny) < eps) exit
26    enddo
27    time = mpi_wtime() - time
28
29    if (decomp%my_rank == 0) then
30       write(6,"(a,i0)") "#iterations: ", iter
31       write(0,"(a,e8.2,a)") "time(iterations) = ", time, " s"
32    endif
33    if (iter > max_iter) call die("no convergence")
34
35    call output_parallel(vnew, Nx, Ny)
36    deallocate(vnew, vold)
37 end
```

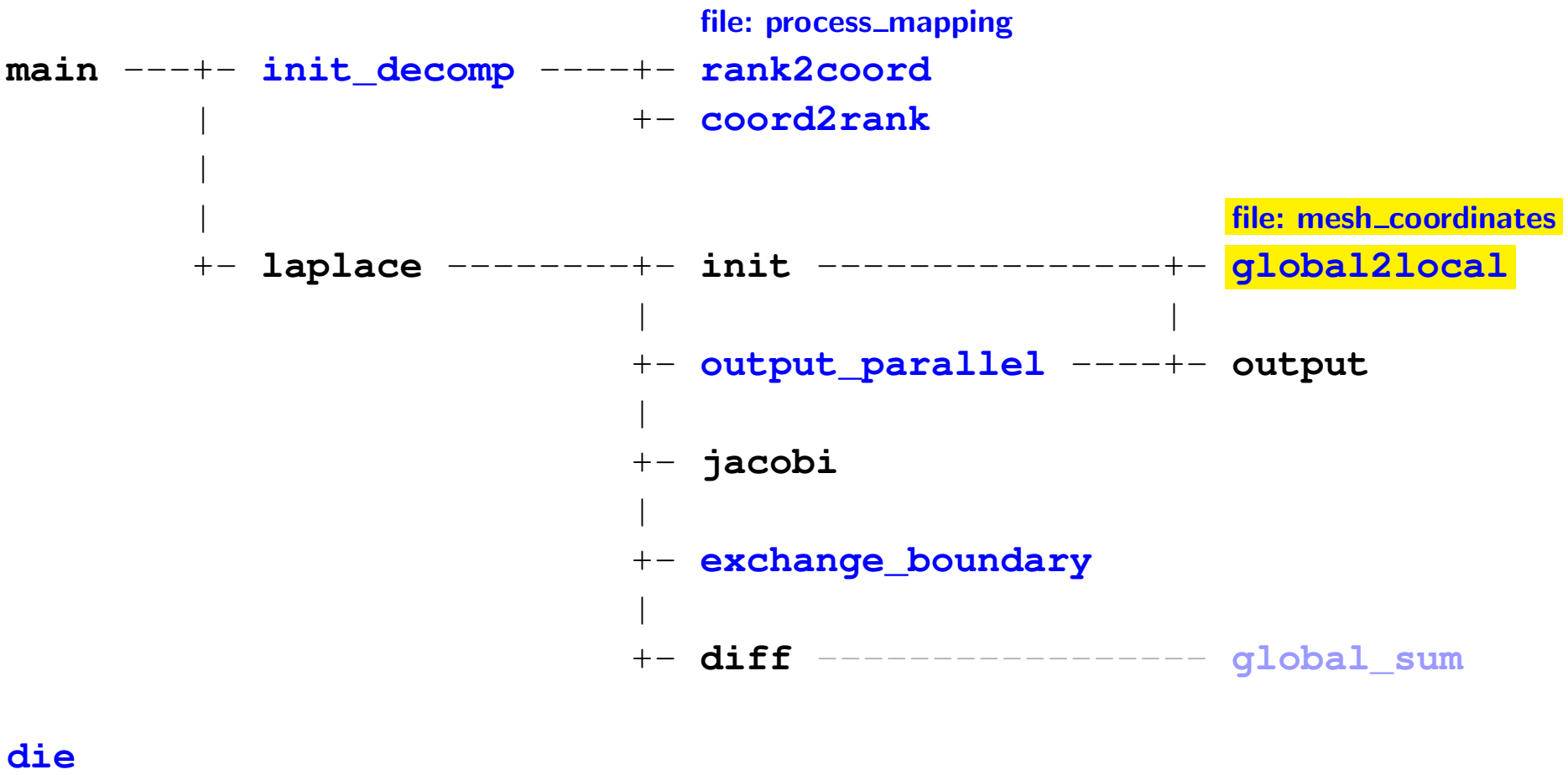
init.c

```
1  #include <stdio.h>
2  #include <mpi.h>
3  #include "laplace.h"
4  #include "decomp.h"
5
6  void init(field vnew, field vold, int Nx, int Ny)
7  {
8      int x, y, x_local, y_local, home_of_xy;
9      double value;
10
11     for (y = 0; y <= Ny + 1; y++)
12         for (x = 0; x <= Nx + 1; x++) {
13             vnew[y][x] = 0; vold[y][x] = 0; }
14
15     FILE *data = fopen("input.data", "r");
16
17     while (fscanf(data, "%d %d %lg", &x, &y, &value) != EOF) {
18         global2local(x, y, &x_local, &y_local, &home_of_xy);
19         if (home_of_xy == decomp.my_rank) {
20             vnew[y_local][x_local] = value;
21             vold[y_local][x_local] = value;
22         }
23     }
24     fclose(data);
25 }
```

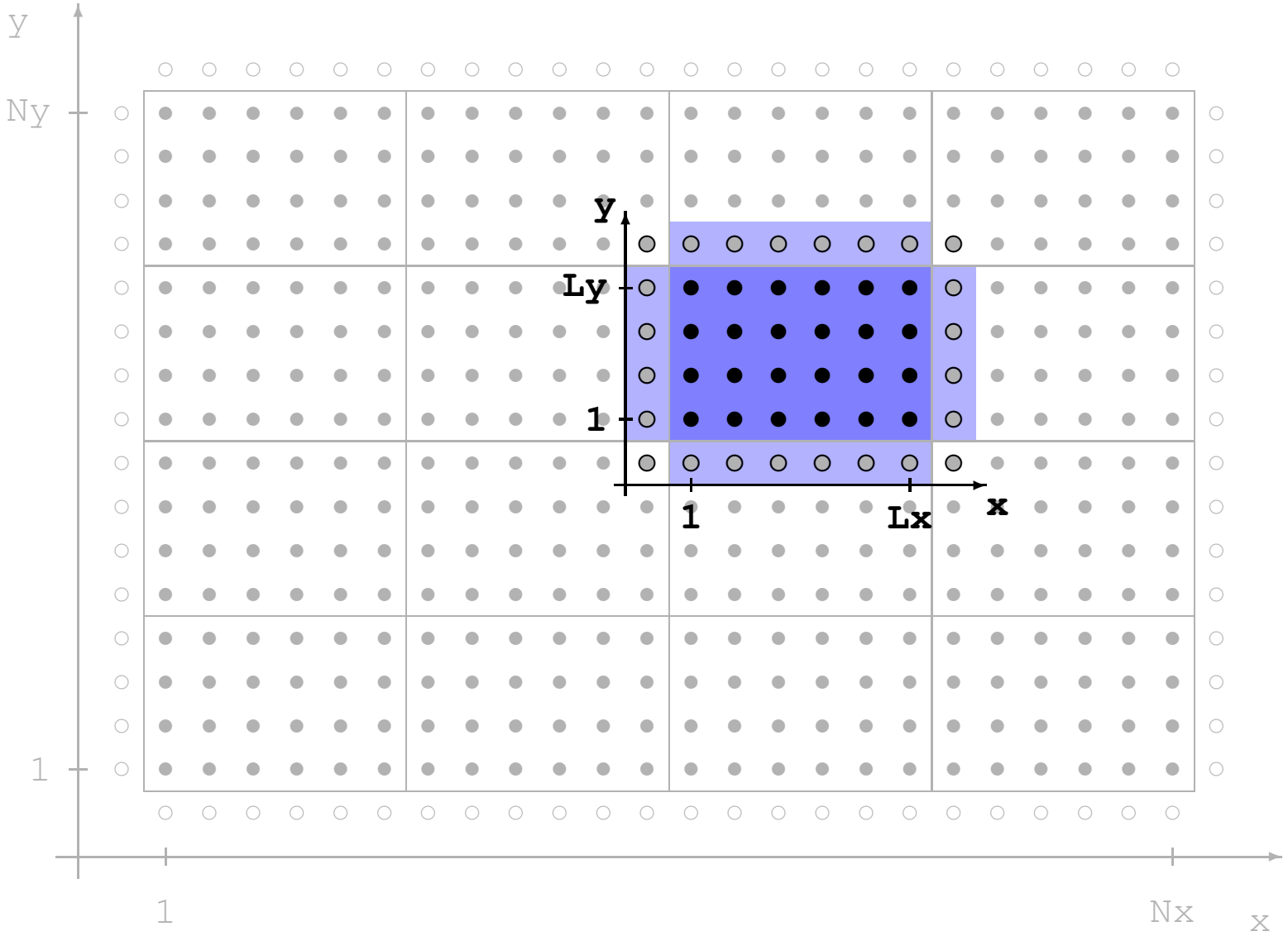
init.f90

```
1  subroutine init(vnew, vold, Nx, Ny)
2    use module_decomp
3    implicit none
4    integer, intent(in)  :: Nx, Ny
5    real(8), intent(out) :: vnew(0:Nx + 1, 0:Ny + 1)
6    real(8), intent(out) :: vold(0:Nx + 1, 0:Ny + 1)
7    integer              :: x, y, iostat, x_local, y_local, home_of_xy
8    real(8)              :: value
9
10   vnew(:, :) = 0
11   vold(:, :) = 0
12
13   open(2, file = "input.data", status = "old", action = "read")
14   do
15       read(2, *, iostat = iostat) x, y, value
16       if (iostat /= 0) exit
17
18       call global2local(x, y, x_local, y_local, home_of_xy)
19       if (home_of_xy == decomp%my_rank) then
20          vnew(x_local, y_local) = value
21          vold(x_local, y_local) = value
22       endif
23   enddo
24   close(2)
25 end
```

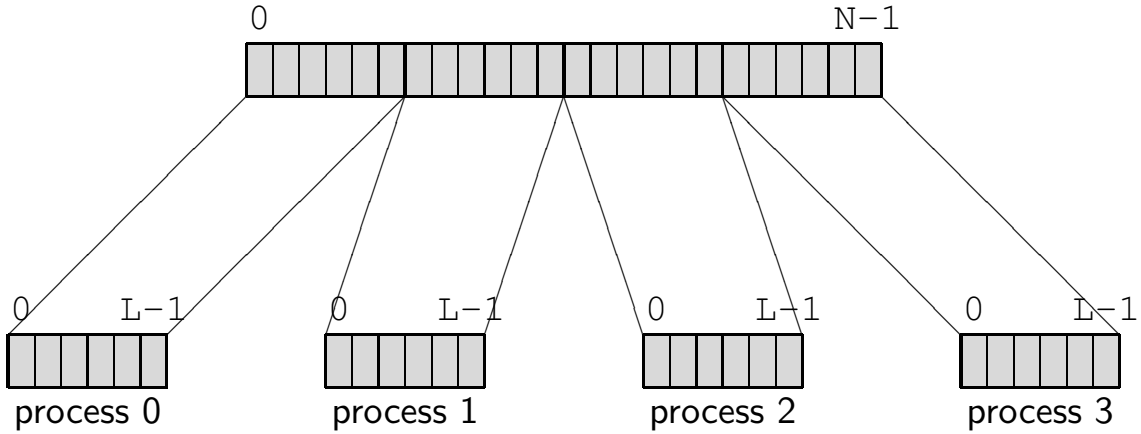
Calling tree (parallel program)



Local mesh coordinates / mesh indexing



One-dimensional decomposition / index calculations in C



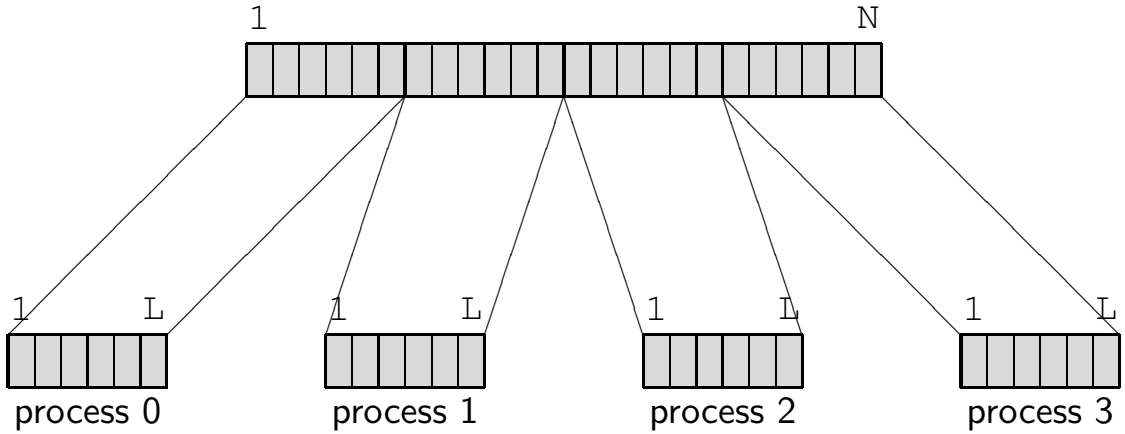
global index: `i = 0..(N-1)`
local index: `j = 0..(L-1)`
process ID: `p = 0..(Nproc-1)`

`(j, p)` $\longrightarrow$ `i`: `i = p * L + j`

`i` $\longrightarrow$ `(j, p)`: `j = i % L`

`p = i / L` (integer division!)

One-dimensional decomposition / index calculations in Fortran



global index: `i = 1..N`
local index: `j = 1..L`
process ID: `p = 0..(Nproc-1)`

`(j, p)` $\longrightarrow$ `i`: `i = p * L + j`

`i` $\longrightarrow$ `(j, p)`: `j = mod(i - 1, L) + 1`

`p = (i - 1) / L` (integer division!)

mesh_coordinates.c

```
1  #include "laplace.h"
2  #include "decomp.h"
3
4  void global2local(int x, int y,
5                    int *x_local, int *y_local, int *home_of_xy)
6  {
7      int home_x, home_y;
8
9      global2local_x(x, x_local, &home_x);
10     global2local_y(y, y_local, &home_y);
11     coord2rank(home_x, home_y, home_of_xy);
12 }
13
```

mesh_coordinates.c

```
14 void global2local_x(int x_global, int *x_local, int *coord_x)
15 {
16     if (x_global == 0) {
17         *x_local = 0;
18         *coord_x = 0;
19     } else if (x_global == decomp.Nx + 1) {
20         *x_local = decomp.Lx + 1;
21         *coord_x = decomp.procs_x - 1;
22     } else {
23         *x_local = (x_global - 1) % decomp.Lx + 1;
24         *coord_x = (x_global - 1) / decomp.Lx;
25     }
26 }
27
```

mesh_coordinates.c

```
28 void global2local_y(int y_global, int *y_local, int *coord_y)
29 {
30     if (y_global == 0) {
31         *y_local = 0;
32         *coord_y = 0;
33     } else if (y_global == decomp.Ny + 1) {
34         *y_local = decomp.Ly + 1;
35         *coord_y = decomp.procs_y - 1;
36     } else {
37         *y_local = (y_global - 1) % decomp.Ly + 1;
38         *coord_y = (y_global - 1) / decomp.Ly;
39     }
40 }
```

mesh_coordinates.f90

```
1  subroutine global2local(x, y, x_local, y_local, home_of_xy)
2
3    implicit none
4    integer, intent(in)  :: x, y
5    integer, intent(out) :: x_local, y_local, home_of_xy
6    integer              :: home_x, home_y
7
8    call global2local_x(x, x_local, home_x)
9    call global2local_y(y, y_local, home_y)
10   call coord2rank(home_x, home_y, home_of_xy)
11 end
12
```

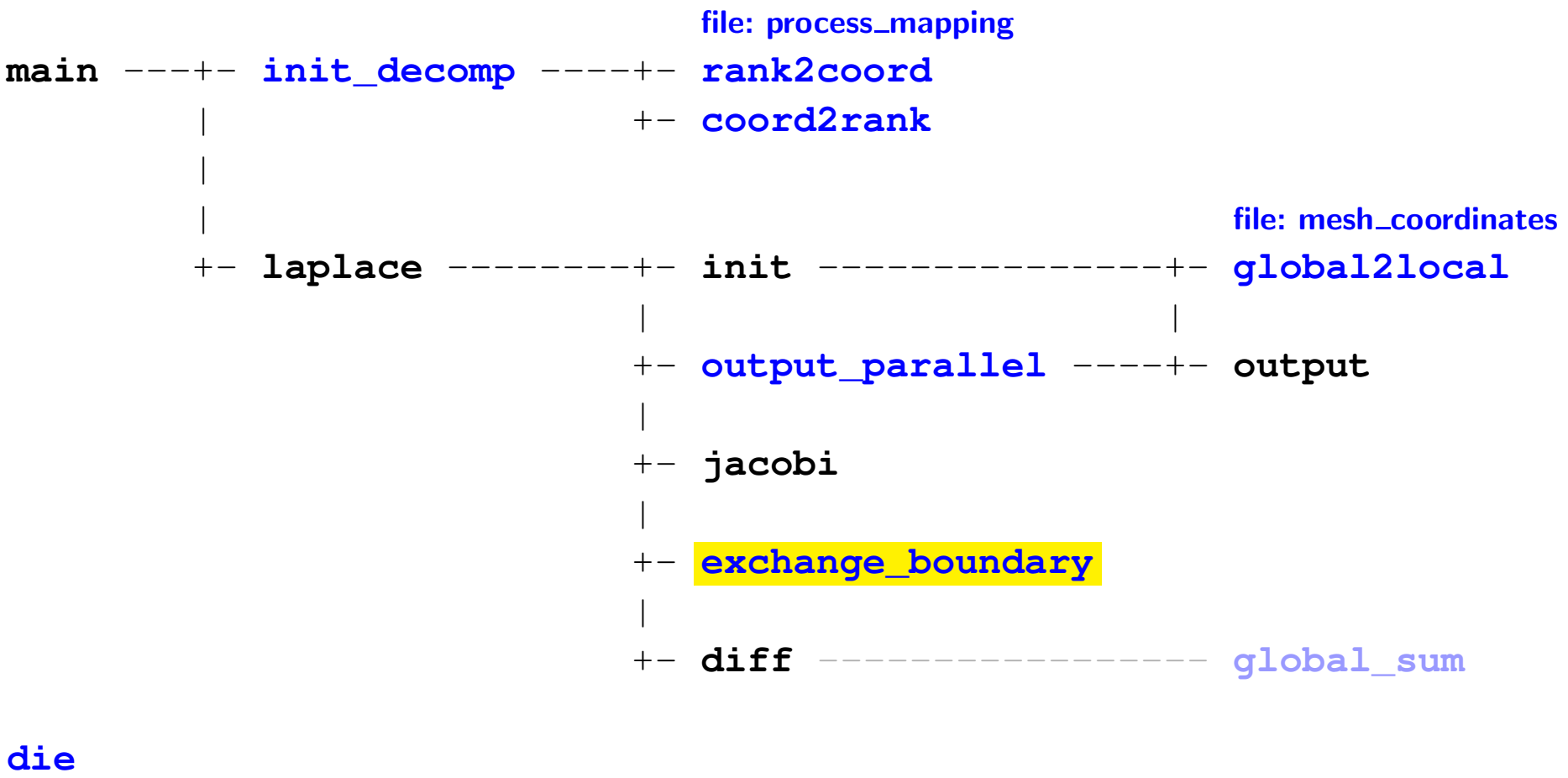
mesh_coordinates.f90

```
13 subroutine global2local_x(x_global, x_local, coord_x)
14
15   use module_decomp
16   implicit none
17   integer, intent(in)  :: x_global
18   integer, intent(out) :: x_local, coord_x
19
20   if (x_global == 0) then
21      x_local = 0
22      coord_x = 0
23   else if (x_global == decomp%Nx + 1) then
24      x_local = decomp%Lx + 1
25      coord_x = decomp%procs_x - 1
26   else
27      x_local = mod(x_global - 1, decomp%Lx) + 1
28      coord_x = (x_global - 1) / decomp%Lx
29   endif
30 end
31
32
```

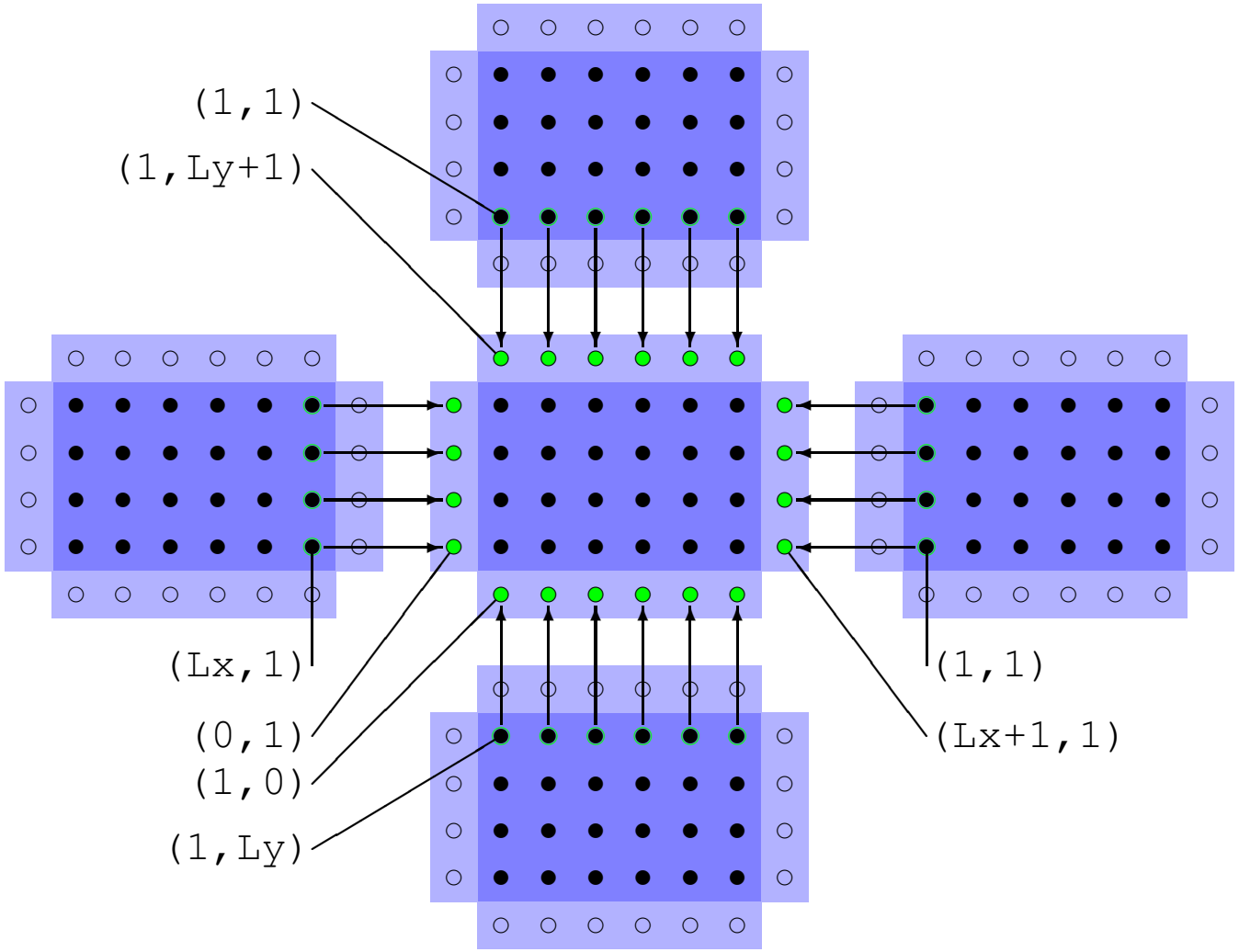
mesh_coordinates.f90

```
33 subroutine global2local_y(y_global, y_local, coord_y)
34
35   use module_decomp
36   implicit none
37   integer, intent(in)  :: y_global
38   integer, intent(out) :: y_local, coord_y
39
40   if (y_global == 0) then
41      y_local = 0
42      coord_y = 0
43   else if (y_global == decomp%Ny + 1) then
44      y_local = decomp%Ly + 1
45      coord_y = decomp%procs_y - 1
46   else
47      y_local = mod(y_global - 1, decomp%Ly) + 1
48      coord_y = (y_global - 1) / decomp%Ly
49   endif
50 end
51
52
```

Calling tree



Boundary exchange – local coordinates



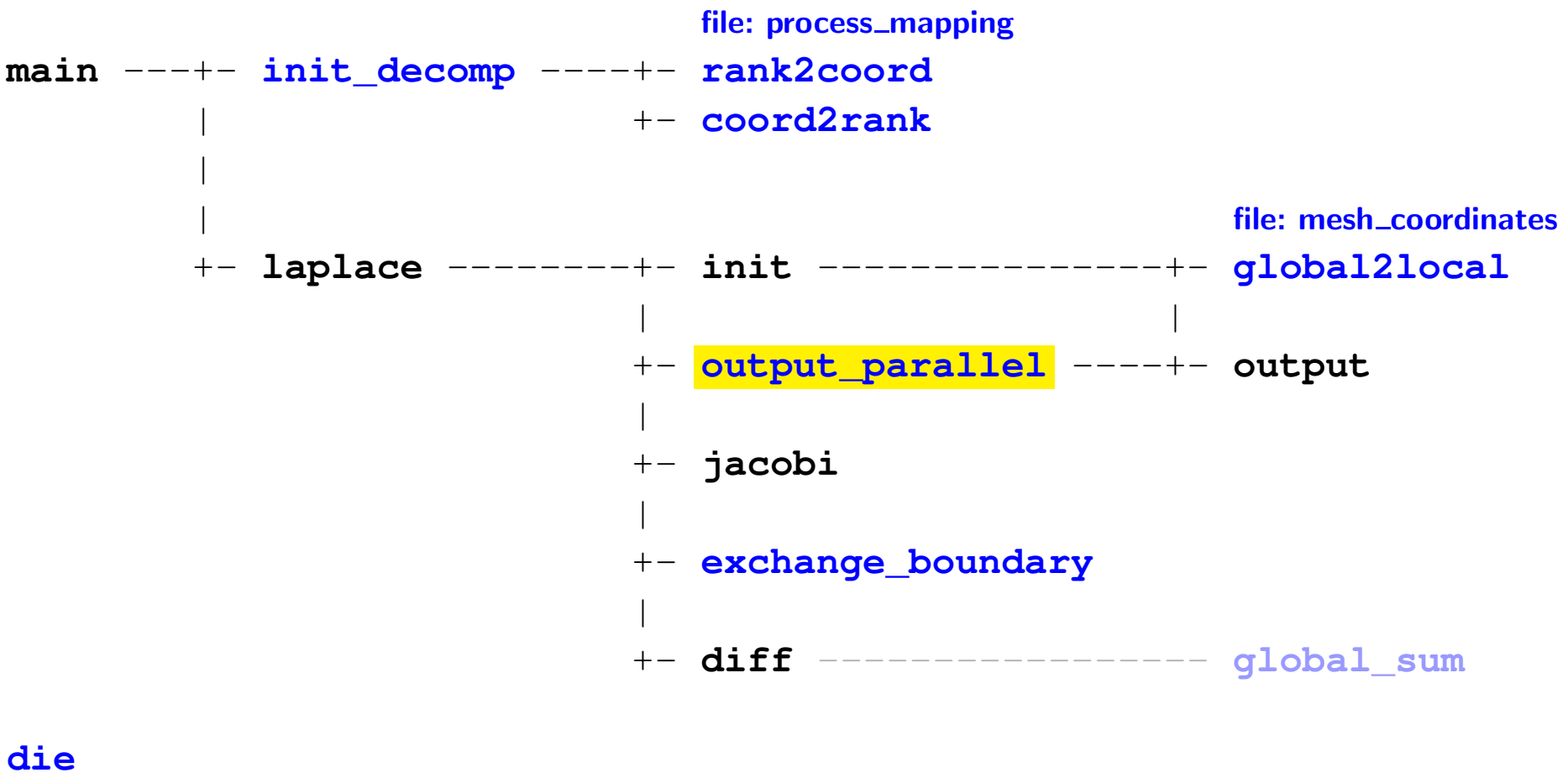
exchange_boundary.c

```
1  #include <mpi.h>
2  #include "laplace.h"
3  #include "decomp.h"
4
5  void exchange_boundary(field v, int Lx, int Ly)
6  {
7   int y;
8   MPI_Status status;
9
10  if (decomp.north >= 0) MPI_Ssend(&v[Ly][1],  Lx, MPI_DOUBLE, decomp.north,
11  if (decomp.south >= 0) MPI_Recv(&v[0][1],    Lx, MPI_DOUBLE, decomp.south,
12
13  if (decomp.south >= 0) MPI_Ssend(&v[1][1],   Lx, MPI_DOUBLE, decomp.south,
14  if (decomp.north >= 0) MPI_Recv(&v[Ly+1][1], Lx, MPI_DOUBLE, decomp.north,
15
16  for (y = 1; y <= Ly; y++) {
17    if (decomp.east >= 0) MPI_Ssend(&v[y][Lx],  1, MPI_DOUBLE, decomp.east,
18    if (decomp.west >= 0) MPI_Recv(&v[y][0],    1, MPI_DOUBLE, decomp.west,
19
20    if (decomp.west >= 0) MPI_Ssend(&v[y][1],   1, MPI_DOUBLE, decomp.west,
21    if (decomp.east >= 0) MPI_Recv(&v[y][Lx+1], 1, MPI_DOUBLE, decomp.east,
22  }
23 }
```

exchange_boundary.f90

```
1  subroutine exchange_boundary(v, Lx, Ly)
2
3   use module_decomp
4   implicit none
5   include "mpif.h"
6   integer, intent(in)    :: Lx, Ly
7   real(8), intent(inout) :: v(0:Lx + 1, 0:Ly + 1)
8   integer                :: y, status(MPI_STATUS_SIZE), ierror
9
10  if (decomp%north >= 0) call mpi_ssend(v(1, Ly),  Lx, MPI_REAL8, decomp%nor
11  if (decomp%south >= 0) call mpi_recv(v(1, 0),    Lx, MPI_REAL8, decomp%sou
12
13  if (decomp%south >= 0) call mpi_ssend(v(1, 1),   Lx, MPI_REAL8, decomp%sou
14  if (decomp%north >= 0) call mpi_recv(v(1, Ly+1), Lx, MPI_REAL8, decomp%nor
15
16  do y = 1, Ly
17     if (decomp%east >= 0) call mpi_ssend(v(Lx, y), 1, MPI_REAL8, decomp%eas
18     if (decomp%west >= 0) call mpi_recv(v(0, y),   1, MPI_REAL8, decomp%wes
19
20     if (decomp%west >= 0) call mpi_ssend(v(1, y),  1, MPI_REAL8, decomp%wes
21     if (decomp%east >= 0) call mpi_recv(v(Lx+1,y), 1, MPI_REAL8, decomp%eas
22  enddo
23 end
```

Calling tree



Writing output

- In order to keep the program simple output data is gathered in a big array on process 0.
- One can imagine that the array represents a direct access file.
- The main task, i.e. writing data in correct order, is the same for writing into a file and an array.

output_parallel.c

```
1  #include <stdio.h>
2  #include <mpi.h>
3  #include "laplace.h"
4  #include "decomp.h"
5
6  void output_parallel(field v, int Lx, int Ly)
7  {
8    field      vv = field_alloc(decomp.Ny, decomp.Nx);
9    int        x, y, x_local, y_local, home_of_xy;
10   MPI_Status status;
11   double     time;
12
```

output_parallel.c

```
13    time = MPI_Wtime();
14    for (y = 0; y <= decomp.Ny + 1; y++) {
15      for (x = 0; x <= decomp.Nx + 1; x++) {
16        global2local(x, y, &x_local, &y_local, &home_of_xy);
17
18        if (decomp.my_rank == 0) {
19          if (home_of_xy == 0)
20            vv[y][x] = v[y_local][x_local];
21          else
22            MPI_Recv(&vv[y][x], 1, MPI_DOUBLE, home_of_xy, 0, MPI_COMM_WORLD,
23        } else {
24          if (decomp.my_rank == home_of_xy)
25            MPI_Ssend(&v[y_local][x_local], 1, MPI_DOUBLE, 0, 0, MPI_COMM_WOR
26        }
27      }
28    }
29    time = MPI_Wtime() - time;
30
31    if (decomp.my_rank == 0) {
32      fprintf(stderr, "time(filling vv) = %8.2e s\n", time);
33      output(vv, decomp.Nx, decomp.Ny);
34    }
35
36    field_free(vv);
37  }
```

output_parallel.f90

```
1  subroutine output_parallel(v, Lx, Ly)
2
3    use module_decomp
4    implicit none
5    include "mpif.h"
6    integer, intent(in) :: Lx, Ly
7    real(8), intent(in) :: v(0:Lx + 1, 0:Ly + 1)
8    real(8)             :: vv(0:decomp%Nx + 1, 0:decomp%Ny + 1)
9    integer             :: x, y, x_local, y_local, home_of_xy
10   integer             :: status(MPI_STATUS_SIZE), ierror
11   real(8)             :: time
12
```

output_parallel.f90

```
13    time = mpi_wtime()
14    do y = 0, decomp%Ny + 1
15       do x = 0, decomp%Nx + 1
16          call global2local(x, y, x_local, y_local, home_of_xy)
17          if (decomp%my_rank == 0) then
18             if (home_of_xy == 0) then
19                vv(x, y) = v(x_local, y_local)
20             else
21               call mpi_recv(vv(x, y), 1, MPI_REAL8, &
22                             home_of_xy, 0, MPI_COMM_WORLD, status, ierror)
23             endif
24          else
25             if (decomp%my_rank == home_of_xy) &
26                call mpi_ssend(v(x_local, y_local), 1, MPI_REAL8, &
27                               0, 0, MPI_COMM_WORLD, ierror)
28          endif
29       enddo
30    enddo
31    time = mpi_wtime() - time
32
33    if (decomp%my_rank == 0) then
34       write(0,"(a,e8.2,a)") "time(filling vv) = ", time, " s"
35       call output(vv, decomp%Nx, decomp%Ny)
36    endif
37 end
```

– MPI project –
tasks

Task 1

Complete the parallelisation of `diff.c/f90`.

Use the completed version of `diff.c/f90` in tasks 2–7.

Before this task is completed the parallel program is incorrect.
It will deadlock for most decompositions.

diff.c

```
1  #include <math.h>
2  #include "laplace.h"
3
4  double diff(field v1, field v2, int Nx, int Ny)
5  {
6      int    x, y;
7      double d, sum;
8
9      sum = 0;
10     for (y = 1; y <= Ny; y++)
11         for (x = 1; x <= Nx; x++) {
12             d = v1[y][x] - v2[y][x];
13             sum += d * d;
14         }
15
16     return sqrt(sum);
17 }
```

diff.f90

```
1  real(8) function diff(v1, v2, Nx, Ny)
2
3    implicit none
4    integer, intent(in) :: Nx, Ny
5    real(8), intent(in) :: v1(0:Nx + 1, 0:Ny + 1)
6    real(8), intent(in) :: v2(0:Nx + 1, 0:Ny + 1)
7    integer             :: x, y
8
9    diff = 0
10   do y = 1, Ny
11      do x = 1, Nx
12         diff = diff + (v1(x, y) - v2(x, y))**2
13      enddo
14   enddo
15
16   diff = sqrt(diff)
17
18 end
```

Task2

Replace the `Send` and `Recv` calls in `exchange_boundary.c/f90` by appropriate calls of `Sendrecv`.

Steps (not necessarily in `exchange_boundary.c/f90`):

- Introduce `MPI_PROC_NULL`.
- Introduce two datatypes that describe data defined on the boundaries in x- and y-directions.
- Use these datatypes in your `Sendrecv` calls.

Tasks 3–5

3) Take your program from task 2 and replace all `Sendrecv` calls by `Irecv` and `Isend` and a single call of `Waitall`.

4) Take your program from task 2 and modify it such that the calculation is performed with the nine point stencil.

5) Take your program from task 3 and modify it such that the calculation is performed with the nine point stencil.

Tasks 6 and 7

6) Introduce a new datatype in `output_parallel.c/f90`
by the use of which a whole line $\{(x,y)_{\mathrm{local}} \,|\, 1 \le x \le L_x \wedge y = \mathrm{const}\}$
can be copied in one communication step.

Treat the boundary $\{(x,y)_{\mathrm{global}} \,|\, x = 0 \vee x = N_x + 1 \vee y = 0 \vee y = N_y + 1\}$
separately.

7) Introduce two new datatypes in `output_parallel.c/f90`
such that all points $\{(x,y)_{\mathrm{local}} \,|\, 1 \le x \le L_x \wedge 1 \le y \le L_y\}$
can be copied in one communication step.

Treat the boundary $\{(x,y)_{\mathrm{global}} \,|\, x = 0 \vee x = N_x + 1 \vee y = 0 \vee y = N_y + 1\}$
separately.

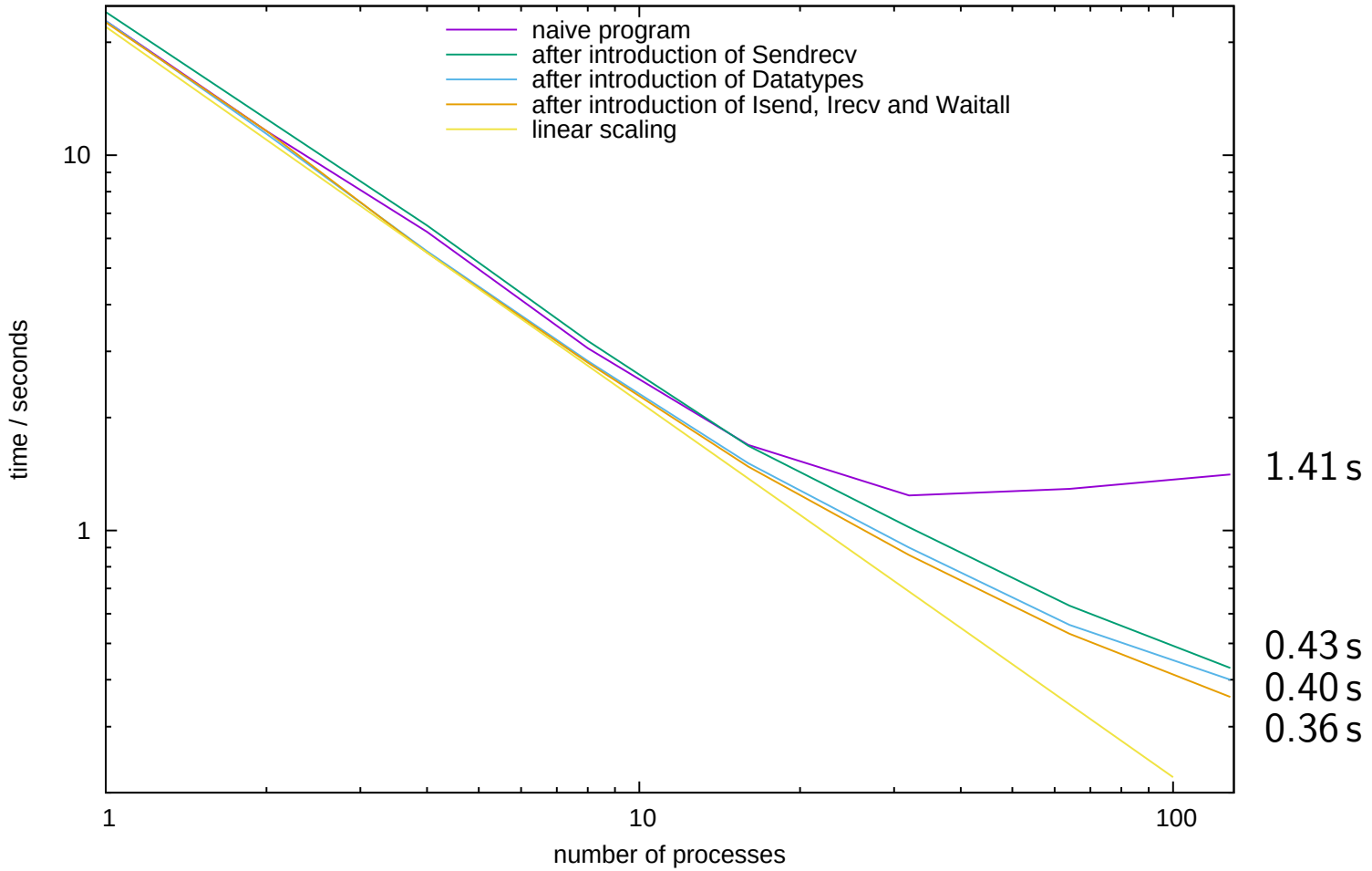
Testing

Check your implementations with several decompositions, e.g.

$$(\mathit{procs\_x}, \mathit{procs\_y}) = (1,1), (1,5), (1,20), (2,1), (10,1), (2,2), (5,4).$$

Timing results / Motivation

- times in seconds for 10000 iterations on a 1024×1024 mesh
- HPC cluster at Universität Hamburg (installed in 2015, 1 node has 2×8 cores)



– OpenMP project –
(sequential program)

source code directories:

`./laplace-code/laplace-omp-c`

`./laplace-code/laplace-omp-f90`

OpenMP project – overview

- parallel solution of the Laplace equation with OpenMP
- iterative methods
  - Jacobi iteration
  - Gauss-Seidel iteration
  - *conjugate gradient* method
- discretisations
  - five point stencil
  - nine point stencil
- starting point is a given sequential program
- the task is to parallelise it loop by loop

OpenMP project – parallelising loops

- steps (for every loop)
  - is the loop a candidate for parallelisation? (does it have enough parallelism?)
  - data dependence analysis: is the loop parallelisable?
  - if necessary and possible: reformulate loop
  - insert OpenMP directives or a comment explaining why the loop cannot be parallelised

- define data-sharing attributes correctly
  - `shared`
  - `private`
  - `reduction`

Program usage

- input files
  - mesh size and stopping criterion: `input.para`
  - boundary values: `input.data`

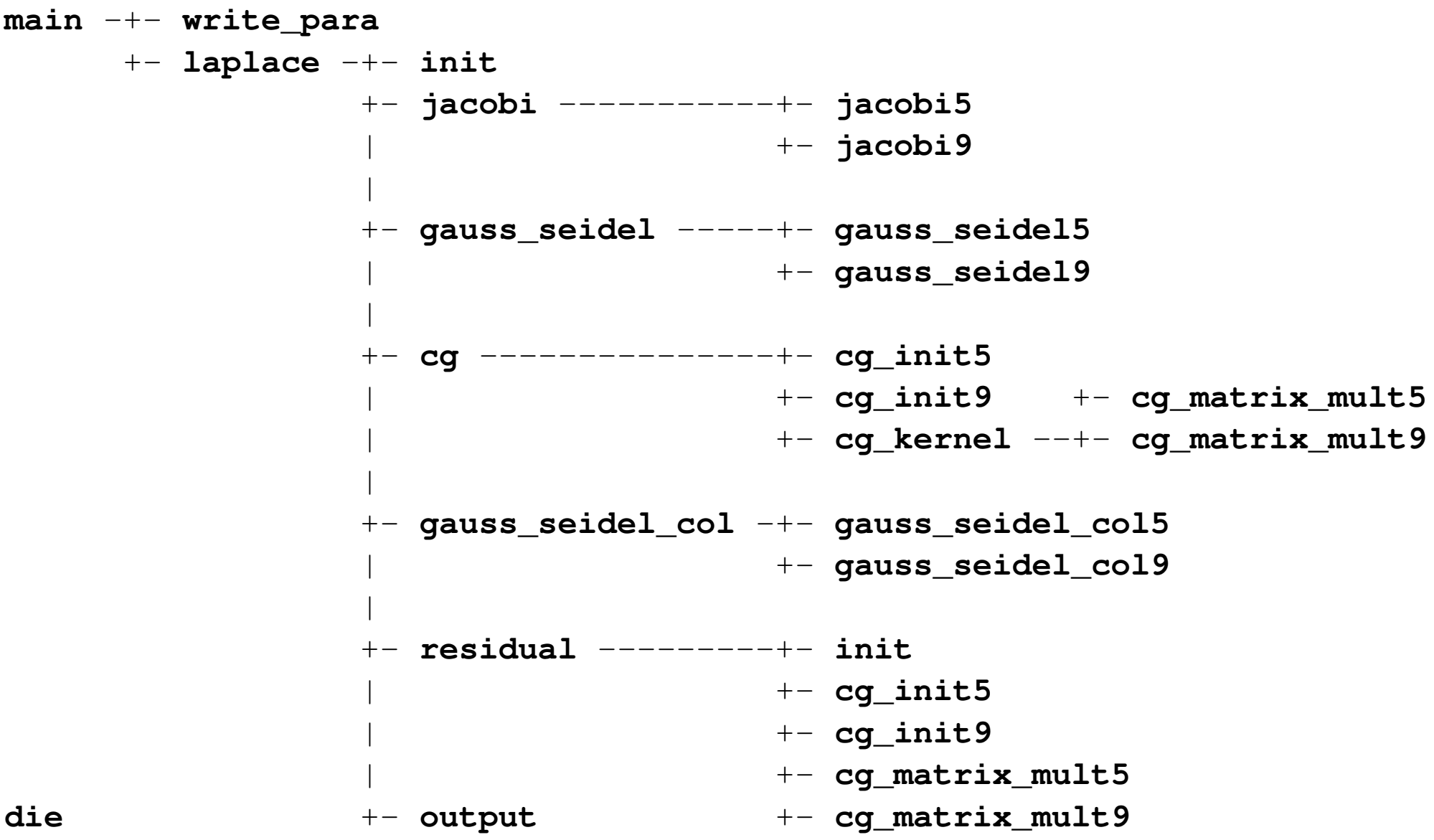
- command line

  `./laplace` *`solver stencil`*

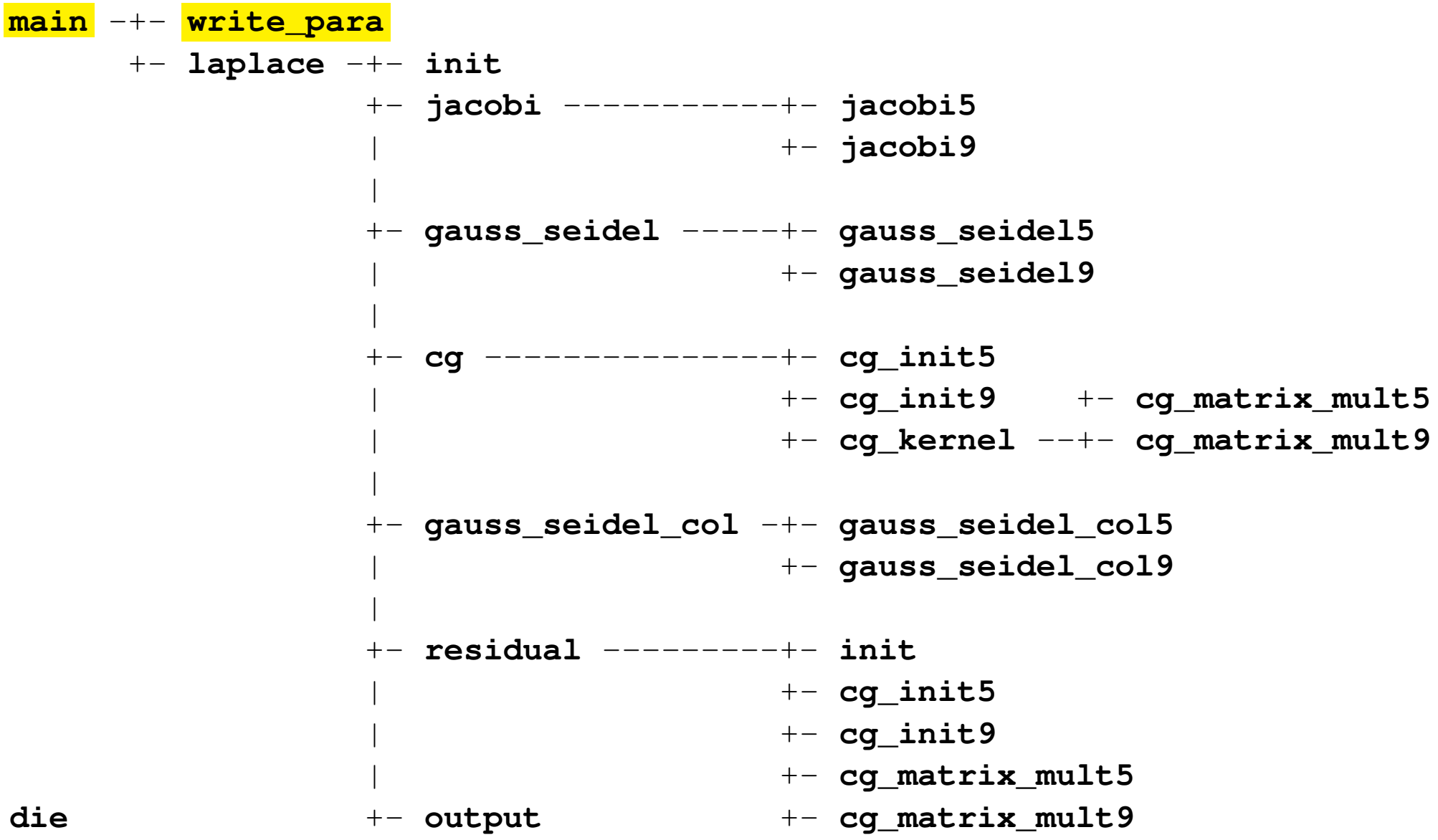
- parameters

`solver`	method	*`stencil`*	
0	Jacobi	5	five point stencil
1	Gauss-Seidel	9	nine point stencil
2	*conjugate gradient*		
3	Gauss-Seidel with colouring		

Calling tree



Calling tree



main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "laplace.h"
4
5  char parafile[] = "input.para";
6  char datafile[] = "input.data";
7
8  int main(int argc, char *argv[])
9  {
10     if (argc != 3) die("Usage: laplace solver stencil");
11
12     int solver = atoi(argv[1]);
13     int stencil = atoi(argv[2]);
14
15     FILE  *para = fopen(parafile, "r");
16     int Nx, Ny;
17     double eps;
18
19     fscanf(para, "%d %*s", &Nx);
20     fscanf(para, "%d %*s", &Ny);
21     fscanf(para, "%lg %*s", &eps);
22     fclose(para);
23
24     write_para(solver, stencil, eps);
25
```

```
26      Solver *the_solver;
27      switch (solver) {
28          case 0: the_solver = jacobi; break;
29          case 1: the_solver = gauss_seidel; break;
30          case 2: the_solver = cg; break;
31          case 3: the_solver = gauss_seidel_col; break;
32      }
33
34      laplace(datafile, Nx, Ny, eps, the_solver, stencil);
35      return 0;
36  }
37
```

main.c

```
38 void write_para(int solver, int stencil, double eps)
39 {
40     puts("Iterative solution of the Laplace equation");
41     puts("-----------------------------------------");
42
43     switch (solver) {
44         case 0:
45             printf("solver:            %d (Jacobi)\n", solver);
46             break;
47         case 1:
48             printf("solver:            %d (Gauss-Seidel)\n", solver);
49             break;
50         case 2:
51             printf("solver:            %d (conjugate gradient)\n", solver);
52             break;
53         case 3:
54             printf("solver:            %d (Gauss-Seidel with colouring)\n",
55             break;
56         default:
57             die("unknown solver");
58             break;
59     }
60
61     printf("stencil:           %d point\n", stencil);
62     printf("eps:               %8.2e\n", eps);
63 }
```

laplace.h

```
1  #include "field.h"
2
3  void die(char *msg);
4  void write_para(int solver, int stencil, double eps);
5  void laplace(char *datafile, int Nx, int Ny, double eps,
6               Solver solver, int stencil);
7  void init(char *datafile, field v, int Nx, int Ny);
8
9  void residual(char *datafile, field solution, int stencil, int Nx, int Ny,
10                double *diff);
11 void output(field v, int Nx, int Ny);
12
```

```
13 typedef void Solver(field solution,
14                      int stencil, int Nx, int Ny, double eps, int max_iter,
15                      int *iterations, double *diff);
16
17 typedef void MatrixMult(field out, field in, int Nx, int Ny);
18
19 Solver jacobi;
20 void jacobi5(field vnew, field vold, int Nx, int Ny, double *diff);
21 void jacobi9(field vnew, field vold, int Nx, int Ny, double *diff);
22
23 Solver gauss_seidel;
24 void gauss_seidel5(field v, int Nx, int Ny, double *diff);
25 void gauss_seidel9(field v, int Nx, int Ny, double *diff);
26
27 Solver gauss_seidel_col;
28 void gauss_seidel_col5(field v, int Nx, int Ny, double *diff);
29 void gauss_seidel_col9(field v, int Nx, int Ny, double *diff);
30
31 Solver cg;
32 void cg_init5(field solution, field rhs, int Nx, int Ny);
33 void cg_init9(field solution, field rhs, int Nx, int Ny);
34 MatrixMult cg_matrix_mult5;
35 MatrixMult cg_matrix_mult9;
36 void cg_kernel(MatrixMult matrix_mult, field x, field b, int Nx, int Ny,
37                double eps, int max_iter, int *iterations, double* diff);
```

laplace.h

main.f90

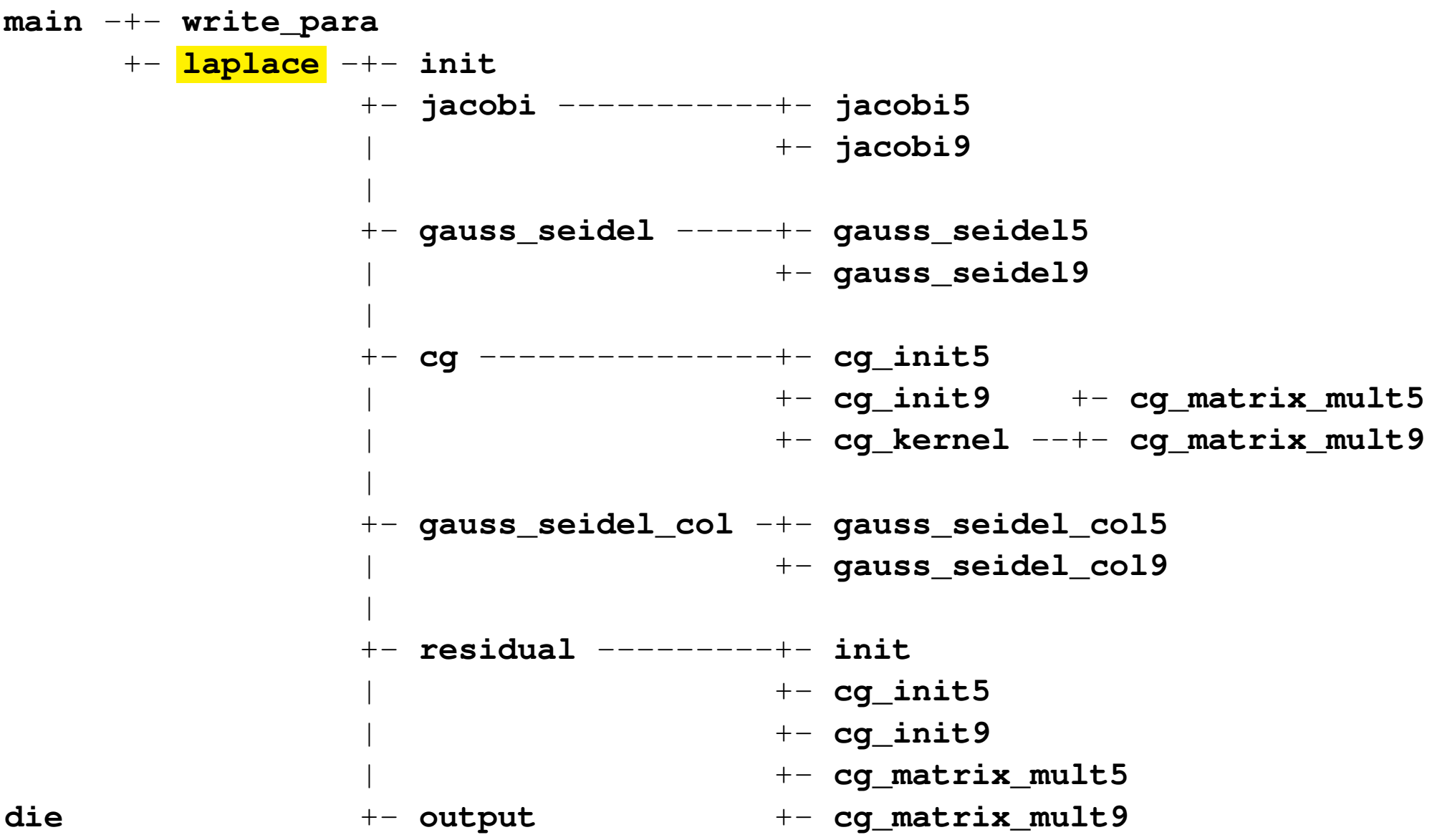
```
1  program main
2
3    implicit none
4    integer :: Nx, Ny
5    real(8) :: eps
6    integer :: solver, stencil
7    procedure() :: jacobi, gauss_seidel, cg, gauss_seidel_col
8    procedure(), pointer :: the_solver
9    character(32) :: arg
10   character(*), parameter :: parafile = "input.para"
11   character(*), parameter :: datafile = "input.data"
12
13   if (iargc() /= 2) call die("Usage: laplace solver stencil")
14
15   call getarg(1, arg) ;  read(arg, *) solver
16   call getarg(2, arg) ;  read(arg, *) stencil
17
18   open(1, file = parafile, status = "old", action = "read")
19   read(1,*) Nx
20   read(1,*) Ny
21   read(1,*) eps
22   close(1)
23
24   call write_para(solver, stencil, eps)
```

```
25
26    select case (solver)
27       case(0); the_solver => jacobi
28       case(1); the_solver => gauss_seidel
29       case(2); the_solver => cg
30       case(3); the_solver => gauss_seidel_col
31    end select
32
33    call laplace(datafile, Nx, Ny, eps, the_solver, stencil)
34 end
35
```

main.f90

```
36 subroutine write_para(solver, stencil, eps)
37
38   implicit none
39   integer, intent(in)       :: solver, stencil
40   real(8), intent(in)       :: eps
41
42   write(6,*) "Iterative solution of the Laplace equation"
43   write(6,*) "-----------------------------------------"
44
45   select case (solver)
46      case(0)
47         write(6,*) "solver:     ", solver, "(Jacobi)"
48      case(1)
49         write(6,*) "solver:     ", solver, "(Gauss-Seidel)"
50      case(2)
51         write(6,*) "solver:     ", solver, "(conjugate gradient)"
52      case(3)
53         write(6,*) "solver:     ", solver, "(Gauss-Seidel with colouring)"
54      case default
55         call die("unknown solver")
56   end select
57
58   write(6,*)              "stencil:    ", stencil, "point"
59   write(6,"(x,a,e8.2)") "eps:                ", eps
60 end
```

Calling tree



laplace.c

```
1  #include <stdio.h>
2  #include <omp.h>
3  #include "laplace.h"
4
5  void laplace(char *datafile, int Nx, int Ny, double eps,
6               Solver solver, int stencil)
7  {
8      field solution = field_alloc(Ny, Nx);
9      double idiff, rdiff;
10     const int max_iter = 10000;
11     int iterations;
12     double time;
13
14     init(datafile, solution, Nx, Ny);
15
16     time = omp_get_wtime();
17     solver(solution, stencil, Nx, Ny, eps, max_iter, &iterations, &idiff);
18     time = omp_get_wtime() - time;
19
20     if (iterations > max_iter) die("no convergence");
21
22     residual(datafile, solution, stencil, Nx, Ny, &rdiff);
23
```

laplace.c

```
24      printf("iterated residual: %8.2e\n", idiff);
25      printf("real residual:     %8.2e\n", rdiff);
26      printf("iterations:        %d\n", iterations);
27      printf("time(solver):      %8.2e s\n", time);
28      printf("\n");
29
30      output(solution, Nx, Ny);
31
32      field_free(solution);
33  }
```

laplace.f90

```
1  subroutine laplace(datafile, Nx, Ny, eps, solver, stencil)
2
3    use omp_lib
4    implicit none
5    character(*)          :: datafile
6    integer, intent(in)   :: Nx, Ny, stencil
7    real(8), intent(in)   :: eps
8    procedure()           :: solver
9
10   real(8), allocatable :: solution(:, :)
11   real(8)              :: idiff, rdiff, time
12   integer, parameter   :: max_iter = 10000
13   integer              :: iterations
14
15   allocate(solution(0:Nx + 1, 0:Ny + 1))
16
17   call init(datafile, solution, Nx, Ny)
18
19   time = omp_get_wtime()
20   call solver(solution, stencil, Nx, Ny, eps, max_iter, iterations, idiff)
21   time = omp_get_wtime() - time
22
23   if (iterations > max_iter) call die("no convergence")
24
25   call residual(datafile, solution, stencil, Nx, Ny, rdiff)
```

laplace.f90

```
26
27    write(6,"(x,a,e8.2)")   "iterated residual: ", idiff
28    write(6,"(x,a,e8.2)")   "real residual:     ", rdiff
29    write(6,*)              "iterations: ", iterations
30    write(6,"(x,a,e8.2,a)") "time(solver):      ", time, " s"
31    write(6,*)
32
33    call output(solution, Nx, Ny)
34
35    deallocate(solution)
36  end
```

Calling tree

```
main -+- write_para
      +- laplace -+- init
                  +- jacobi ------------+- jacobi5
                  |                     +- jacobi9
                  |
                  +- gauss_seidel -----+- gauss_seidel5
                  |                    +- gauss_seidel9
                  |
                  +- cg ---------------+- cg_init5
                  |                    +- cg_init9    +- cg_matrix_mult5
                  |                    +- cg_kernel --+- cg_matrix_mult9
                  |
                  +- gauss_seidel_col -+- gauss_seidel_col5
                  |                    +- gauss_seidel_col9
                  |
                  +- residual ---------+- init
                  |                    +- cg_init5
                  |                    +- cg_init9
                  |                    +- cg_matrix_mult5
die               +- output            +- cg_matrix_mult9
```

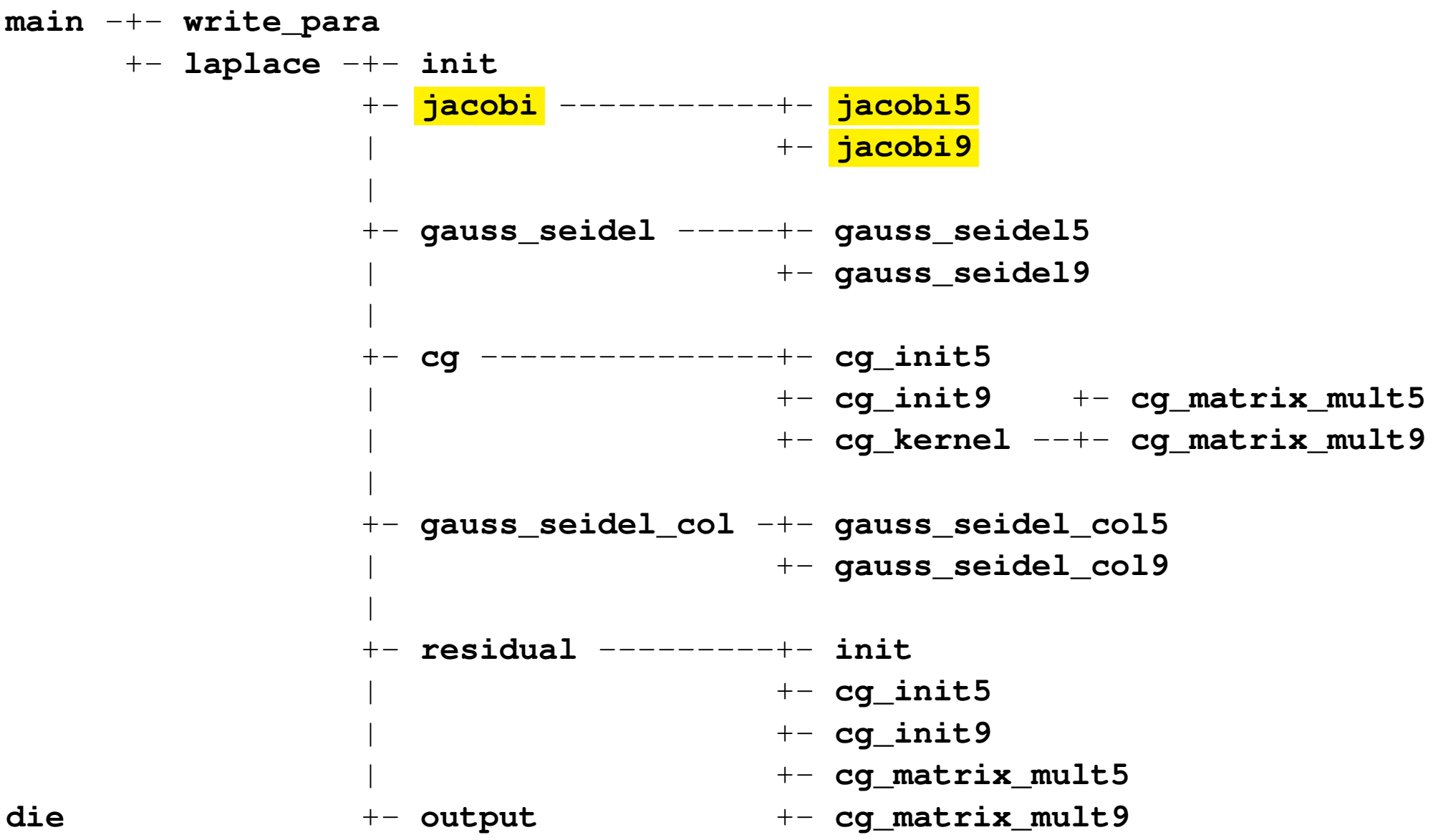
init.c

```
1  #include <stdio.h>
2  #include "laplace.h"
3
4  void init(char *datafile, field v, int Nx, int Ny)
5  {
6      int x, y;
7      double value;
8
9      for (y = 0; y <= Ny + 1; y++) {
10         for (x = 0; x <= Nx + 1; x++) {
11             v[y][x] = 0;
12         }
13     }
14
15     FILE *data = fopen(datafile, "r");
16
17     while (fscanf(data, "%d %d %lg", &x, &y, &value) != EOF) {
18         v[y][x] = value;
19     }
20
21     fclose(data);
22 }
```

init.f90

```
1  subroutine init(datafile, v, Nx, Ny)
2
3    implicit none
4    character(*)           :: datafile
5    integer, intent(in)    :: Nx, Ny
6    real(8), intent(out)   :: v(0:Nx + 1, 0:Ny + 1)
7    integer                :: x, y, iostat
8    real(8)                :: value
9
10   do y = 0, Ny + 1
11      do x = 0, Nx + 1
12         v(x, y) = 0
13      enddo
14   enddo
15
16   open(2, file = datafile, status = "old", action = "read")
17
18   do
19       read(2, *, iostat = iostat) x, y, value
20       if (iostat /= 0) exit
21       v(x, y) = value
22   enddo
23
24   close(2)
25 end
```

Calling tree



jacobi.c

```
1 #include <math.h>
2 #include "laplace.h"
3
4 void jacobi(field solution, int stencil, int Nx, int Ny, double eps,
5             int max_iter, int *iterations, double *diff)
6 {
7     int x, y, niter;
8     field v = field_alloc(Ny, Nx);
9
```

jacobi.c

```
10      for (niter = 1; niter <= max_iter; niter++) {
11
12          for (y = 0; y <= Ny + 1; y++)
13              for (x = 0; x <= Nx + 1; x++)
14                  v[y][x] = solution[y][x];
15
16          switch (stencil) {
17              case 5:
18                  jacobi5(solution, v, Nx, Ny, diff);
19                  break;
20              case 9:
21                  jacobi9(solution, v, Nx, Ny, diff);
22                  break;
23              default:
24                  die("unknown stencil");
25                  break;
26          }
27
28          if (*diff < eps) break;
29      }
30
31      *iterations = niter;
32      field_free(v);
33  }
34
```

jacobi.c

```
35 void jacobi5(field vnew, field vold, int Nx, int Ny, double *diff)
36 {
37     int x, y;
38     double d, sum;
39
40     sum = 0;
41     for (y = 1; y <= Ny; y++) {
42         for (x = 1; x <= Nx; x++) {
43             vnew[y][x] = (vold[y][x - 1]
44                         + vold[y][x + 1]
45                         + vold[y - 1][x]
46                         + vold[y + 1][x]) * 0.25;
47             d = vnew[y][x] - vold[y][x];
48             sum += d * d;
49         }
50     }
51
52     *diff = 4.0 * sqrt(sum);
53 }
54
```

jacobi.c

```
55 void jacobi9(field vnew, field vold, int Nx, int Ny, double *diff)
56 {
57     int x, y;
58     double d, sum;
59
60     sum = 0;
61     for (y = 1; y <= Ny; y++) {
62         for (x = 1; x <= Nx; x++) {
63             vnew[y][x] = 4.0 * (vold[y][x - 1]
64                                 + vold[y][x + 1]
65                                 + vold[y - 1][x]
66                                 + vold[y + 1][x])
67                                 + vold[y - 1][x - 1]
68                                 + vold[y + 1][x - 1]
69                                 + vold[y - 1][x + 1]
70                                 + vold[y + 1][x + 1];
71             vnew[y][x] *= 0.05;
72             d = vnew[y][x] - vold[y][x];
73             sum += d * d;
74         }
75     }
76
77     *diff = 20.0 * sqrt(sum);
78 }
```

jacobi.f90

```
1  subroutine jacobi(solution, stencil, Nx, Ny, eps, max_iter, iterations, dif
2
3    integer, intent(in)    :: stencil, Nx, Ny, max_iter
4    real(8), intent(inout) :: solution(0:Nx + 1, 0:Ny + 1)
5    real(8), intent(in)    :: eps
6    integer, intent(out)   :: iterations
7    real(8), intent(out)   :: diff
8
9    integer                :: x, y
10   real(8), dimension(:, :), allocatable :: v
11
12   allocate(v(0:Nx + 1, 0:Ny + 1))
13
```

jacobi.f90

```
14    do iterations = 1, max_iter
15
16       do y = 0, Ny + 1
17          do x = 0, Nx + 1
18             v(x, y) = solution(x, y)
19          enddo
20       enddo
21
22       select case (stencil)
23          case(5)
24             call jacobi5(solution, v, Nx, Ny, diff)
25          case(9)
26             call jacobi9(solution, v, Nx, Ny, diff)
27          case default
28             call die("unknown stencil")
29       end select
30
31       if (diff < eps) exit
32    enddo
33
34    deallocate(v)
35 end
36
```

jacobi.f90

```
37 subroutine jacobi5(vnew, vold, Nx, Ny, diff)
38
39   implicit none
40   integer, intent(in)  :: Nx, Ny
41   real(8), intent(out) :: vnew(0:Nx + 1, 0:Ny + 1)
42   real(8), intent(in)  :: vold(0:Nx + 1, 0:Ny + 1)
43   real(8), intent(out) :: diff
44   real(8)              :: d, sum
45   integer              :: x, y
46
47   sum = 0
48   do y = 1, Ny
49      do x = 1, Nx
50         vnew(x, y) = (vold(x - 1, y) &
51                     + vold(x + 1, y) &
52                     + vold(x, y - 1) &
53                     + vold(x, y + 1)) * 0.25
54         d = vnew(x, y) - vold(x, y)
55         sum = sum + d**2
56      enddo
57   enddo
58
59   diff = 4.0 * sqrt(sum)
60 end
61
```

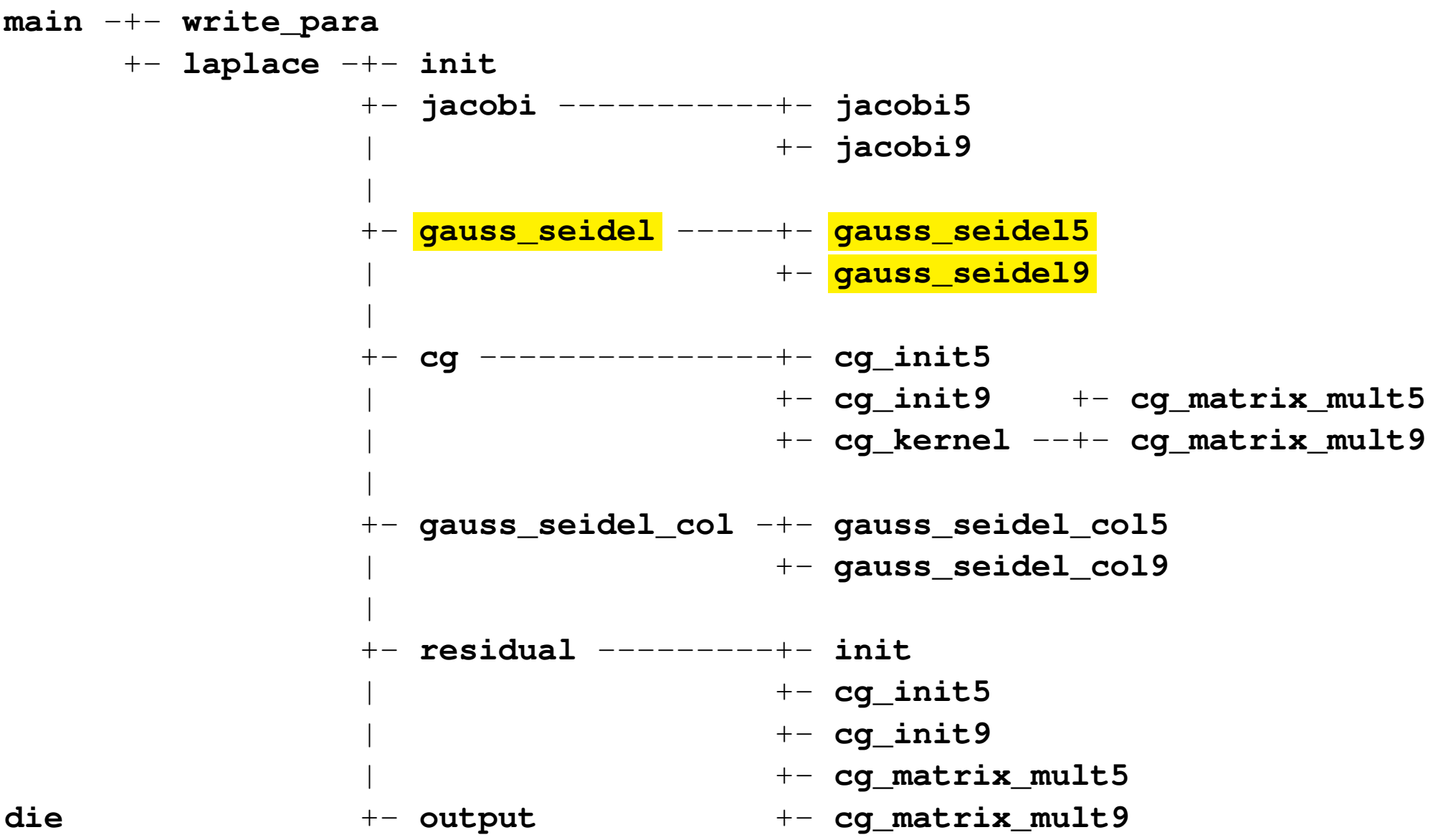
jacobi.f90

```
62 subroutine jacobi9(vnew, vold, Nx, Ny, diff)
63
64   implicit none
65   integer, intent(in)  :: Nx, Ny
66   real(8), intent(out) :: vnew(0:Nx + 1, 0:Ny + 1)
67   real(8), intent(in)  :: vold(0:Nx + 1, 0:Ny + 1)
68   real(8), intent(out) :: diff
69   real(8)              :: d, sum
70   integer              :: x, y
71
```

jacobi.f90

```
72    sum = 0
73    do y = 1, Ny
74       do x = 1, Nx
75          vnew(x, y) = 4.0 * (vold(x - 1, y) &
76                              + vold(x + 1, y) &
77                              + vold(x, y - 1) &
78                              + vold(x, y + 1)) &
79                              + vold(x - 1, y - 1) &
80                              + vold(x - 1, y + 1) &
81                              + vold(x + 1, y - 1) &
82                              + vold(x + 1, y + 1)
83          vnew(x, y) = vnew(x, y) * 0.05
84
85          d = vnew(x, y) - vold(x, y)
86          sum = sum + d**2
87       enddo
88    enddo
89
90    diff = 20.0 * sqrt(sum)
91 end
```

Calling tree



gauss_seidel.c

```
#include <math.h>
#include "laplace.h"

void gauss_seidel(field solution, int stencil, int Nx, int Ny, double eps,
                  int max_iter, int *iterations, double *diff)
{
    int niter;
    for (niter = 1; niter <= max_iter; niter++) {

        switch (stencil) {
            case 5:
                gauss_seidel5(solution, Nx, Ny, diff);
                break;
            case 9:
                gauss_seidel9(solution, Nx, Ny, diff);
                break;
            default:
                die("unknown stencil");
                break;
        }
        if (*diff < eps) break;
    }

    *iterations = niter;
}
```

gauss_seidel.c

```
27 void gauss_seidel5(field v, int Nx, int Ny, double *diff)
28 {
29     int x, y;
30     double vold, d, sum;
31
32     sum = 0;
33     for (y = 1; y <= Ny; y++) {
34         for (x = 1; x <= Nx; x++) {
35             vold = v[y][x];
36             v[y][x] = (v[y][x - 1]
37                      + v[y][x + 1]
38                      + v[y - 1][x]
39                      + v[y + 1][x]) * 0.25;
40             d = v[y][x] - vold;
41             sum += d * d;
42         }
43     }
44
45     *diff = 4.0 * sqrt(sum);
46 }
47
48
```

gauss_seidel.c

```
49 void gauss_seidel9(field v, int Nx, int Ny, double *diff)
50 {
51     int x, y;
52     double vold, d, sum;
53
54     sum = 0;
55     for (y = 1; y <= Ny; y++) {
56         for (x = 1; x <= Nx; x++) {
57             vold = v[y][x];
58             v[y][x] = 4.0 * (v[y][x - 1]
59                              + v[y][x + 1]
60                              + v[y - 1][x]
61                              + v[y + 1][x])
62                              + v[y - 1][x - 1]
63                              + v[y + 1][x - 1]
64                              + v[y - 1][x + 1]
65                              + v[y + 1][x + 1];
66             v[y][x] *= 0.05;
67             d = v[y][x] - vold;
68             sum += d * d;
69         }
70     }
71
72     *diff = 20.0 * sqrt(sum);
73 }
```

gauss_seidel.f90

```
1  subroutine gauss_seidel(solution, stencil, Nx, Ny, eps, max_iter, &
2                          iterations, diff)
3
4    integer, intent(in)    :: stencil, Nx, Ny, max_iter
5    real(8), intent(inout) :: solution(0:Nx + 1, 0:Ny + 1)
6    real(8), intent(in)    :: eps
7    integer, intent(out)   :: iterations
8    real(8), intent(out)   :: diff
9
10   do iterations = 1, max_iter
11
12      select case (stencil)
13         case(5)
14            call gauss_seidel5(solution, Nx, Ny, diff)
15         case(9)
16            call gauss_seidel9(solution, Nx, Ny, diff)
17         case default
18            call die("unknown stencil")
19      end select
20
21      if (diff < eps) exit
22   enddo
23
24 end
25
```

gauss_seidel.f90

```
26 subroutine gauss_seidel5(v, Nx, Ny, diff)
27
28   implicit none
29   integer, intent(in)  :: Nx, Ny
30   real(8), intent(out) :: v(0:Nx + 1, 0:Ny + 1)
31   real(8), intent(out) :: diff
32   real(8)              :: vold
33   real(8)              :: d, sum
34   integer              :: x, y
35
36   sum = 0
37   do y = 1, Ny
38      do x = 1, Nx
39         vold = v(x, y)
40         v(x, y) = (v(x - 1, y) &
41                  + v(x + 1, y) &
42                  + v(x, y - 1) &
43                  + v(x, y + 1)) * 0.25
44         d = v(x, y) - vold
45         sum = sum + d**2
46      end do
47   end do
48
49   diff = 4.0 * sqrt(sum)
50 end
```

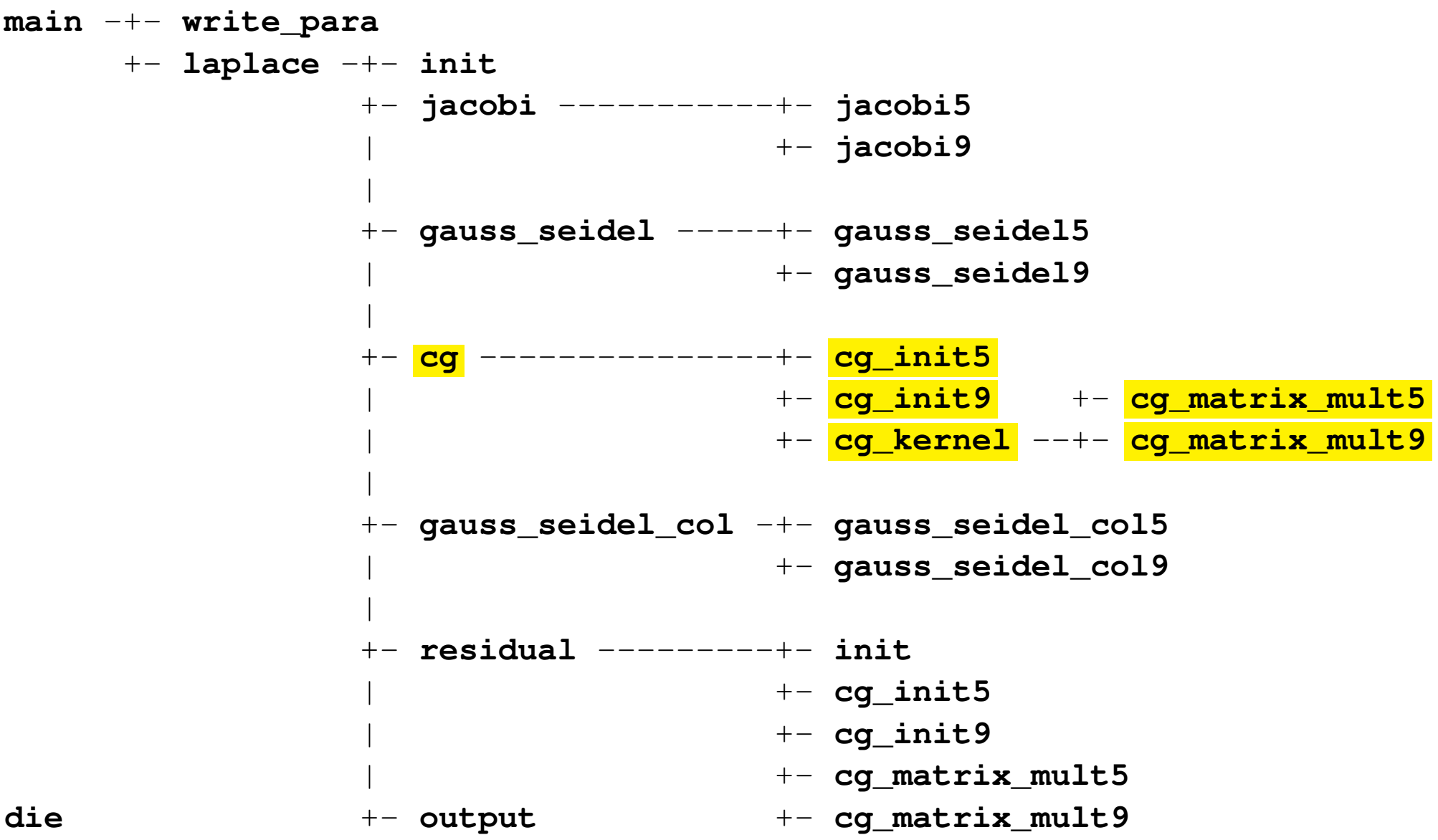
gauss_seidel.f90

```
51
52 subroutine gauss_seidel9(v, Nx, Ny, diff)
53
54   implicit none
55   integer, intent(in)  :: Nx, Ny
56   real(8), intent(out) :: v(0:Nx + 1, 0:Ny + 1)
57   real(8), intent(out) :: diff
58   real(8)              :: vold, d, sum
59   integer              :: x, y
60
```

gauss_seidel.f90

```
61    sum = 0
62    do y = 1, Ny
63       do x = 1, Nx
64          vold = v(x, y)
65          v(x, y) = 4.0 * (v(x - 1, y) &
66                           + v(x + 1, y) &
67                           + v(x, y - 1) &
68                           + v(x, y + 1)) &
69                           + v(x - 1, y - 1) &
70                           + v(x - 1, y + 1) &
71                           + v(x + 1, y - 1) &
72                           + v(x + 1, y + 1)
73          v(x, y) = v(x, y) * 0.05
74          d = v(x, y) - vold
75          sum = sum + d**2
76       enddo
77    enddo
78
79    diff = 20.0 * sqrt(sum)
80 end
```

Calling tree



cg.c

```
1  #include <math.h>
2  #include "laplace.h"
3
4  void cg(field solution, int stencil, int Nx, int Ny, double eps,
5          int max_iter, int *iterations, double *diff)
6  {
7      field rhs = field_alloc(Ny, Nx);
8
9      switch (stencil) {
10         case 5:
11             cg_init5(solution, rhs, Nx, Ny);
12             cg_kernel(cg_matrix_mult5, solution, rhs, Nx, Ny, eps,
13                       max_iter, iterations, diff);
14             break;
15         case 9:
16             cg_init9(solution, rhs, Nx, Ny);
17             cg_kernel(cg_matrix_mult9, solution, rhs, Nx, Ny, eps,
18                       max_iter, iterations, diff);
19             break;
20         default:
21             die("unknown stencil");
22             break;
23     }
24     field_free(rhs);
25 }
```

cg.c

```
27 void cg_init5(field solution, field rhs, int Nx, int Ny)
28 {
29     int x, y;
30
31     for (y = 0; y <= Ny + 1; y++) {
32         for (x = 0; x <= Nx + 1; x++) {
33             rhs[y][x] = solution[y][x];
34         }
35     }
36
37     for (x = 1; x <= Nx; x++) {
38         rhs[1][x] += rhs[0][x];
39         rhs[Ny][x] += rhs[Ny + 1][x];
40         solution[1][x] = rhs[1][x];
41         solution[Ny][x] = rhs[Ny][x];
42     }
43
44     for (y = 1; y <= Ny; y++) {
45         rhs[y][1] += rhs[y][0];
46         rhs[y][Nx] += rhs[y][Nx + 1];
47         solution[y][1] = rhs[y][1];
48         solution[y][Nx] = rhs[y][Nx];
49     }
50 }
51
```

cg.c

```
52 void cg_init9(field solution, field rhs, int Nx, int Ny)
53 {
54     int x, y;
55
56     for (x = 1; x <= Nx; x++) {
57         solution[1][x] += 4.0 * solution[0][x]
58                               + solution[0][x - 1]
59                               + solution[0][x + 1];
60         solution[Ny][x] += 4.0 * solution[Ny + 1][x]
61                                + solution[Ny + 1][x - 1]
62                                + solution[Ny + 1][x + 1];
63     }
64     for (y = 1; y <= Ny; y++) {
65         solution[y][1] += 4.0 * solution[y][0]
66                               + solution[y - 1][0]
67                               + solution[y + 1][0];
68         solution[y][Nx] += 4.0 * solution[y][Nx + 1]
69                                + solution[y - 1][Nx + 1]
70                                + solution[y + 1][Nx + 1];
71     }
72     solution[1][1]  -= solution[0][0];
73     solution[Ny][1] -= solution[Ny + 1][0];
74     solution[1][Nx] -= solution[0][Nx + 1];
75     solution[Ny][Nx] -= solution[Ny + 1][Nx + 1];
```

```
76
77      for (y = 0; y <= Ny + 1; y++)
78          for (x = 0; x <= Nx + 1; x++)
79              rhs[y][x] = solution[y][x];
80  }
81
```

```
82 void cg_matrix_mult5(field out, field in, int Nx, int Ny)
83 {
84     int x, y;
85     double a, b, c, d;
86
87     for (y = 1; y <= Ny; y++) {
88         for (x = 1; x <= Nx; x++) {
89            if (x ==  1) a = 0;  else  a = in[y][x - 1];
90            if (x == Nx) b = 0;  else  b = in[y][x + 1];
91            if (y ==  1) c = 0;  else  c = in[y - 1][x];
92            if (y == Ny) d = 0;  else  d = in[y + 1][x];
93
94            out[y][x] = 4.0 * in[y][x] - a - b - c - d;
95         }
96     }
97 }
98
```

cg.c

```
 99 void cg_matrix_mult9(field out, field in, int Nx, int Ny)
100 {
101     int x, y;
102     double n, s, e, w, ne, nw, se, sw;
103 
104     for (y = 1; y <= Ny; y++) {
105         for (x = 1; x <= Nx; x++) {
106             n = in[y + 1][x];
107             s = in[y - 1][x];
108             e = in[y][x + 1];
109             w = in[y][x - 1];
110             ne = in[y + 1][x + 1];
111             nw = in[y + 1][x - 1];
112             se = in[y - 1][x + 1];
113             sw = in[y - 1][x - 1];
114 
115             if (x ==  1) { w = 0; nw = 0; sw = 0; }
116             if (x == Nx) { e = 0; ne = 0; se = 0; }
117             if (y ==  1) { s = 0; se = 0; sw = 0; }
118             if (y == Ny) { n = 0; ne = 0; nw = 0; }
119 
120             out[y][x] = 20.0 * in[y][x]
121                         - 4.0 * (n + s + e + w) - ne - nw - se - sw;
122         }
123     }
124 }
```

```
125
126 void cg_kernel(MatrixMult matrix_mult, field x, field b, int Nx, int Ny,
127                 double eps, int max_iter, int *iterations, double* diff)
128 {
129     // solves "matrix_mult * x = b", returns #iterations and residual
130
131     field r = field_alloc(Ny, Nx);
132     field p = field_alloc(Ny, Nx);
133     field aap = field_alloc(Ny, Nx);
134     double ak, bk, rtr, rtrold, paap;
135     int i, j, niter;
```

```
136
137     // initialisation
138
139     for (j = 0; j <= Ny + 1; j++) {
140         for (i = 0; i <= Nx + 1; i++) {
141             r[j][i] = 0;
142             p[j][i] = 0;
143             aap[j][i] = 0;
144         }
145     }
146
147     matrix_mult(r, x, Nx, Ny);
148
149     rtrold = 0;
150     for (j = 1; j <= Ny; j++) {
151         for (i = 1; i <= Nx; i++) {
152             r[j][i] = b[j][i] - r[j][i];
153             p[j][i] = r[j][i];
154             rtrold += r[j][i] * r[j][i];
155         }
156     }
157
```

cg.c

```
158     // iteration
159     for (niter = 1; niter <= max_iter; niter++) {
160         matrix_mult(aap, p, Nx, Ny);
161         paap = 0;
162         for (j = 1; j <= Ny; j++)
163             for (i = 1; i <= Nx; i++)
164                 paap += p[j][i] * aap[j][i];
165 
166         ak = rtrold / paap;
167         rtr = 0;
168         for (j = 1; j <= Ny; j++) {
169             for (i = 1; i <= Nx; i++) {
170                 x[j][i] += ak * p[j][i];
171                 r[j][i] -= ak * aap[j][i];
172                 rtr += r[j][i] * r[j][i];
173             }
174         }
175         if (rtr <= eps * eps) break;
176 
177         bk = rtr / rtrold;
178         rtrold = rtr;
179         for (j = 1; j <= Ny; j++)
180             for (i = 1; i <= Nx; i++)
181                 p[j][i] = bk * p[j][i] + r[j][i];
182     }
```

```
183
184       // return values
185       *iterations = niter;
186       *diff = sqrt(rtr);
187
188       // cleanup
189       field_free(r);
190       field_free(p);
191       field_free(aap);
192   }
```

cg.f90

```
1  subroutine cg(solution, stencil, Nx, Ny, eps, max_iter, iterations, diff)
2    implicit none
3    integer, intent(in)    :: stencil, Nx, Ny, max_iter
4    real(8), intent(inout) :: solution(0:Nx + 1, 0:Ny + 1)
5    real(8), intent(in)    :: eps
6    integer, intent(out)   :: iterations
7    real(8), intent(out)   :: diff
8    real(8), allocatable   :: rhs(:, :)
9    procedure()            :: cg_matrix_mult5, cg_matrix_mult9
10
11   allocate(rhs(0:Nx + 1, 0:Ny + 1))
12   select case (stencil)
13      case(5)
14         call cg_init5(solution, rhs, Nx, Ny)
15         call cg_kernel(cg_matrix_mult5, solution, rhs, Nx, Ny, eps, max_ite
16                        iterations, diff)
17      case(9)
18         call cg_init9(solution, rhs, Nx, Ny)
19         call cg_kernel(cg_matrix_mult9, solution, rhs, Nx, Ny, eps, max_ite
20                        iterations, diff)
21      case default
22         call die("unknown stencil")
23   end select
24   deallocate(rhs)
25 end
```

```
26
27 subroutine cg_init5(solution, rhs, Nx, Ny)
28
29   implicit none
30   integer, intent(in)    :: Nx, Ny
31   real(8), intent(inout) :: solution(0:Nx + 1, 0:Ny + 1)
32   real(8), intent(out)   :: rhs(0:Nx + 1, 0:Ny + 1)
33
34   integer :: x, y
```

cg.f90

```
35
36    do y = 0, Ny + 1
37       do x = 0, Nx + 1
38          rhs(x, y) = solution(x, y)
39       enddo
40    enddo
41
42    do x = 1, Nx
43       rhs(x, 1) = rhs(x, 1) + rhs(x, 0)
44       rhs(x, Ny) = rhs(x, Ny) + rhs(x, Ny + 1)
45       solution(x, 1) = rhs(x, 1)
46       solution(x, Ny) = rhs(x, Ny)
47    enddo
48
49    do y = 1, Ny
50       rhs(1, y) = rhs(1, y) + rhs(0, y)
51       rhs(Nx, y) = rhs(Nx, y) + rhs(Nx + 1, y)
52       solution(1, y) = rhs(1, y)
53       solution(Nx, y) = rhs(Nx, y)
54    enddo
55 end
56
```

```
57 subroutine cg_init9(solution, rhs, Nx, Ny)
58
59   implicit none
60   integer, intent(in)    :: Nx, Ny
61   real(8), intent(inout) :: solution(0:Nx + 1, 0:Ny + 1)
62   real(8), intent(out)   :: rhs(0:Nx + 1, 0:Ny + 1)
63
64   integer :: x, y
```

cg.f90

```
65
66    do x = 1, Nx
67       solution(x, 1) = solution(x, 1) + 4.0 * solution(x, 0) &
68                                             + solution(x - 1, 0) &
69                                             + solution(x + 1, 0)
70       solution(x, Ny) = solution(x, Ny) + 4.0 * solution(x, Ny + 1) &
71                                               + solution(x - 1, Ny + 1) &
72                                               + solution(x + 1, Ny + 1)
73    enddo
74
75    do y = 1, Ny
76       solution(1, y) = solution(1, y) + 4.0 * solution(0, y) &
77                                             + solution(0, y - 1) &
78                                             + solution(0, y + 1)
79       solution(Nx, y) = solution(Nx, y) + 4.0 * solution(Nx + 1, y) &
80                                               + solution(Nx + 1, y - 1) &
81                                               + solution(Nx + 1, y + 1)
82    enddo
83
84    solution(1,  1)  = solution(1, 1) - solution(0, 0)
85    solution(1,  Ny) = solution(1, Ny) - solution(0, Ny + 1)
86    solution(Nx, 1)  = solution(Nx, 1) - solution(Nx + 1 , 0)
87    solution(Nx, Ny) = solution(Nx , Ny) - solution(Nx + 1, Ny + 1)
```

```
88
89    do y = 0, Ny + 1
90       do x = 0, Nx + 1
91          rhs(x, y) = solution(x, y)
92       enddo
93    enddo
94 end
95
```

cg.f90

```
96  subroutine cg_matrix_mult5(out, in, Nx, Ny)
97
98    implicit none
99    integer, intent(in)  :: Nx
100   integer, intent(in)  :: Ny
101   real(8), intent(out) :: out(0:Nx + 1, 0:Ny + 1)
102   real(8), intent(in)  :: in(0:Nx + 1, 0:Ny + 1)
103
104   integer              :: x, y
105   real(8)              :: a, b, c, d
106
107   do y = 1, Ny
108      do x = 1, Nx
109         if (x ==  1) then;  a = 0;  else;  a = in(x - 1, y);  endif
110         if (x == Nx) then;  b = 0;  else;  b = in(x + 1, y);  endif
111         if (y ==  1) then;  c = 0;  else;  c = in(x, y - 1);  endif
112         if (y == Ny) then;  d = 0;  else;  d = in(x, y + 1);  endif
113
114         out(x, y) = 4.0 * in(x, y) - a - b - c - d
115      enddo
116   enddo
117 end
118
```

```
119 subroutine cg_matrix_mult9(out, in, Nx, Ny)
120
121   implicit none
122   integer, intent(in)  :: Nx
123   integer, intent(in)  :: Ny
124   real(8), intent(out) :: out(0:Nx + 1, 0:Ny + 1)
125   real(8), intent(in)  :: in(0:Nx + 1, 0:Ny + 1)
126
127   integer              :: x, y
128   real(8)              :: n, s, e, w, ne, nw, se, sw
129
```

cg.f90

```
130   do y = 1, Ny
131      do x = 1, Nx
132         n = in(x, y + 1)
133         s = in(x, y - 1)
134         e = in(x + 1, y)
135         w = in(x - 1, y)
136
137         ne = in(x + 1, y + 1)
138         nw = in(x - 1, y + 1)
139         se = in(x + 1, y - 1)
140         sw = in(x - 1, y - 1)
141
142         if (x ==  1) then;  w = 0; nw = 0; sw = 0; endif
143         if (x == Nx) then;  e = 0; ne = 0; se = 0; endif
144         if (y ==  1) then;  s = 0; se = 0; sw = 0; endif
145         if (y == Ny) then;  n = 0; ne = 0; nw = 0; endif
146
147         out(x, y) = 20.0 * in(x, y) &
148                   - 4.0 * (n + s + e + w) - ne - nw - se - sw
149      enddo
150   enddo
151 end
152
```

cg.f90

```
153 subroutine cg_kernel(matrix_mult, x, b, Nx, Ny, eps, &
154                      max_iter, iterations, diff)
155
156   ! solves "matrix_mult * x = b", returns #iterations and residual
157
158   implicit none
159   procedure()              :: matrix_mult
160   real(8), intent(inout) :: x(0:Nx + 1, 0:Ny + 1)
161   real(8), intent(in)    :: b(0:Nx + 1, 0:Ny + 1)
162   integer, intent(in)    :: Nx
163   integer, intent(in)    :: Ny
164   real(8), intent(in)    :: eps
165   integer, intent(in)    :: max_iter
166   integer, intent(out)   :: iterations
167   real(8), intent(out)   :: diff
168
169   real(8), dimension(:, :), allocatable :: r, p, aap
170   real(8)                  :: ak, bk, rtr, rtrold, paap
171   integer                  :: i, j, niter
172
173   allocate(r(0:Nx + 1, 0:Ny + 1))
174   allocate(p(0:Nx + 1, 0:Ny + 1))
175   allocate(aap(0:Nx + 1, 0:Ny + 1))
```

cg.f90

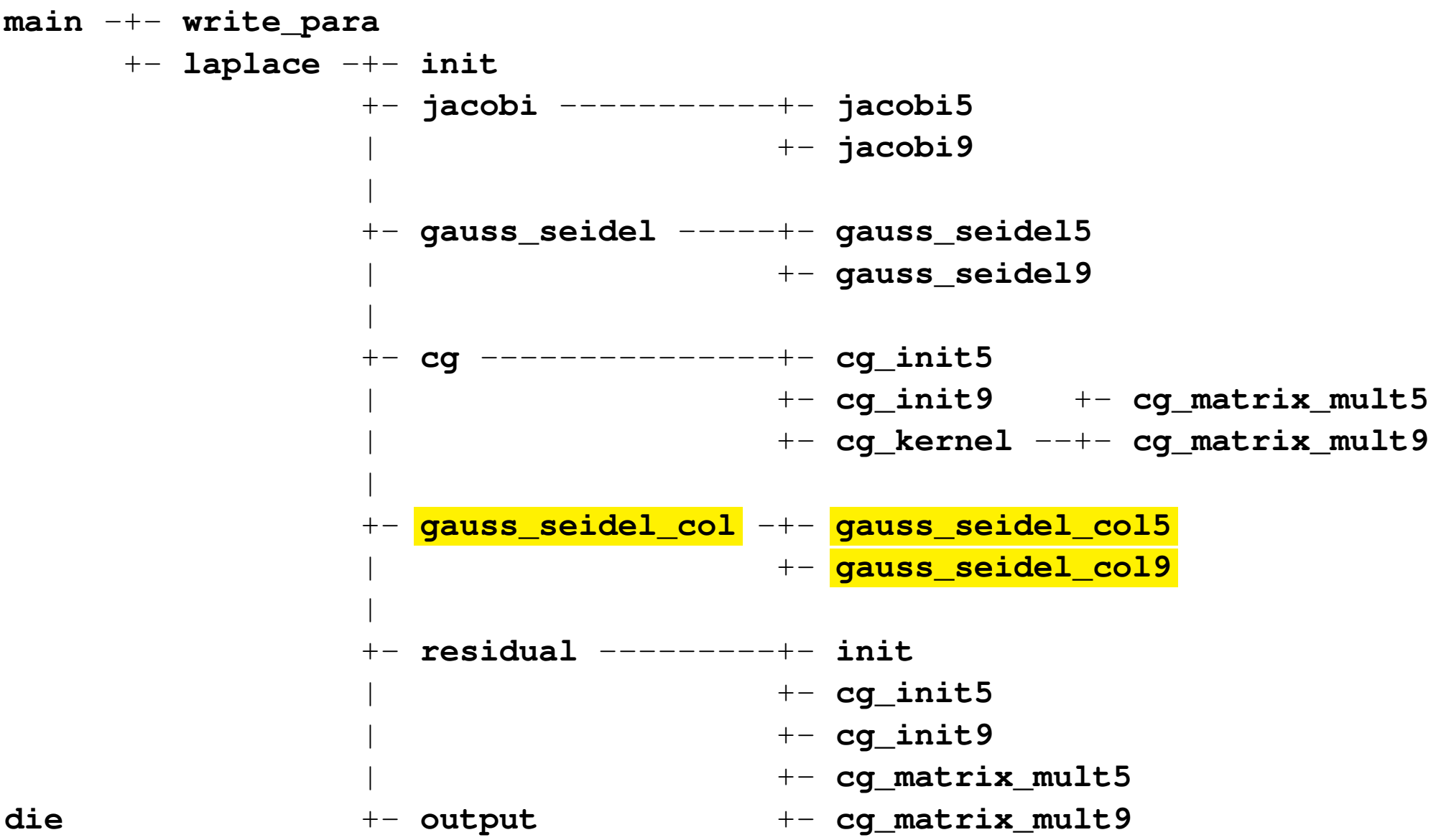
```
176
177   ! initialisation
178
179   do j = 0, Ny + 1
180      do i = 0, Nx + 1
181         r(i, j) = 0
182         p(i, j) = 0
183         aap(i, j) = 0
184      enddo
185   enddo
186
187   call matrix_mult(r, x, Nx, Ny)
188
189   rtrold = 0
190   do j = 1, Ny
191      do i = 1, Nx
192         r(i, j) = b(i, j) - r(i, j)
193         p(i, j) = r(i, j)
194         rtrold = rtrold + r(i, j)**2
195      enddo
196   enddo
197
```

cg.f90

```
198   ! iteration
199   do niter = 1, max_iter
200      call matrix_mult(aap, p, Nx, Ny)
201
202      paap = 0
203      do j = 1, Ny
204         do i = 1, Nx
205            paap = paap + p(i, j) * aap(i, j)
206         enddo
207      enddo
208
209      ak = rtrold / paap
210      rtr = 0
211      do j = 1, Ny
212         do i = 1, Nx
213            x(i, j) = x(i, j) + ak * p(i, j)
214            r(i, j) = r(i, j) - ak * aap(i, j)
215            rtr = rtr + r(i, j)**2
216         enddo
217      enddo
218
219      if (rtr < eps**2) exit
```

```
220
221       bk = rtr / rtrold
222       rtrold = rtr
223       do j = 1, Ny
224          do i = 1, Nx
225             p(i, j) = bk * p(i, j) + r(i, j)
226          enddo
227       enddo
228    enddo
229
230    ! return values
231    iterations = niter
232    diff = sqrt(rtr)
233
234    ! cleanup
235    deallocate(r, p, aap)
236 end
```

Calling tree



gauss_seidel_col.c

```
1  #include <math.h>
2  #include "laplace.h"
3
4  void gauss_seidel_col(field solution, int stencil, int Nx, int Ny,
5            double eps, int max_iter, int *iterations, double *diff)
6  {
7      int niter;
8
9      for (niter = 1; niter <= max_iter; niter++) {
10
11         switch (stencil) {
12             case 5:
13                 gauss_seidel_col5(solution, Nx, Ny, diff);
14                 break;
15             case 9:
16                 gauss_seidel_col9(solution, Nx, Ny, diff);
17                 break;
18             default:
19                 die("unknown stencil");
20                 break;
21         }
22         if (*diff < eps) break;
23     }
24     *iterations = niter;
25 }
```

gauss_seidel_col.c

```
27 void gauss_seidel_col5(field v, int Nx, int Ny, double *diff)
28 {
29     int x, y, eo;
30     double vold, d, sum;
31
32     sum = 0;
33     for (eo = 0; eo <= 1; eo++) {  // eo = "even/odd" = colour
34         for (y = 1; y <= Ny; y++) {
35             for (x = 1; x <= Nx; x++) {
36                 if ((x + y) % 2 == eo) {
37                     vold = v[y][x];
38                     v[y][x] = (v[y][x - 1]
39                                + v[y][x + 1]
40                                + v[y - 1][x]
41                                + v[y + 1][x]) * 0.25;
42                     d = v[y][x] - vold;
43                     sum += d * d;
44                 }
45             }
46         }
47     }
48
49     *diff = 4.0 * sqrt(sum);
50 }
```

gauss_seidel_col.c

```
52 void gauss_seidel_col9(field v, int Nx, int Ny, double *diff)
53 {
54     int x, y, col, colx, coly, colour[2][2];
55     double vold, d, sum;
56 
57     // introduce 4 colours in a 2x2 colour matrix
58 
59     colour[1][1] = 0;
60     colour[0][0] = 1;
61     colour[1][0] = 2;
62     colour[0][1] = 3;
63 
```

gauss_seidel_col.c

```
64      sum = 0;
65      for (col = 0; col <= 3; col++) { // loop over colours
66          for (y = 1; y <= Ny; y++) {
67              coly = y % 2;
68              for (x = 1; x <= Nx; x++) {
69                  colx= x % 2;
70                  if (colour[coly][colx] == col) {
71                      vold = v[y][x];
72                      v[y][x] = 4.0 * (v[y][x - 1]
73                                       + v[y][x + 1]
74                                       + v[y - 1][x]
75                                       + v[y + 1][x])
76                                       + v[y - 1][x - 1]
77                                       + v[y + 1][x - 1]
78                                       + v[y - 1][x + 1]
79                                       + v[y + 1][x + 1];
80                      v[y][x] *= 0.05;
81                      d = v[y][x] - vold;
82                      sum += d * d;
83                  }
84              }
85          }
86      }
87      *diff = 20.0 * sqrt(sum);
88  }
```

gauss_seidel_col.f90

```
1  subroutine gauss_seidel_col(solution, stencil, Nx, Ny, eps, &
2                              max_iter, iterations, diff)
3
4    integer, intent(in)    :: stencil, Nx, Ny, max_iter
5    real(8), intent(inout) :: solution(0:Nx + 1, 0:Ny + 1)
6    real(8), intent(in)    :: eps
7    integer, intent(out)   :: iterations
8    real(8), intent(out)   :: diff
9
10   do iterations = 1, max_iter
11
12      select case (stencil)
13         case(5)
14            call gauss_seidel_col5(solution, Nx, Ny, diff)
15         case(9)
16            call gauss_seidel_col9(solution, Nx, Ny, diff)
17         case default
18            call die("unknown stencil")
19      end select
20
21      if (diff < eps) exit
22   enddo
23 end
24
```

gauss_seidel_col.f90

```
25 subroutine gauss_seidel_col5(v, Nx, Ny, diff)
26   implicit none
27   integer, intent(in)  :: Nx, Ny
28   real(8), intent(out) :: v(0:Nx + 1, 0:Ny + 1)
29   real(8), intent(out) :: diff
30   real(8)              :: vold, d, sum
31   integer              :: x, y, eo
32   sum = 0
33   do eo = 0, 1  ! eo = "even/odd" = colour
34      do y = 1, Ny
35         do x = 1, Nx
36            if (mod(x + y, 2) == eo) then
37               vold = v(x, y)
38               v(x, y) = (v(x - 1, y) &
39                        + v(x + 1, y) &
40                        + v(x, y - 1) &
41                        + v(x, y + 1)) * 0.25
42               d = v(x, y) - vold
43               sum = sum + d**2
44            endif
45         enddo
46      enddo
47   enddo
48   diff = 4.0 * sqrt(sum)
49 end
```

gauss_seidel_col.f90

```
50
51 subroutine gauss_seidel_col9(v, Nx, Ny, diff)
52
53   implicit none
54   integer, intent(in)  :: Nx, Ny
55   real(8), intent(out) :: v(0:Nx + 1, 0:Ny + 1)
56   real(8), intent(out) :: diff
57   real(8)              :: vold
58   real(8)              :: d, sum
59   integer              :: x, y, col, colx, coly, colour(0:1, 0:1)
60
61   ! introduce 4 colours in a 2x2 colour matrix
62
63   colour(1, 1) = 0
64   colour(0, 0) = 1
65   colour(0, 1) = 2
66   colour(1, 0) = 3
67
```

gauss_seidel_col.f90

```
68    sum = 0
69    do col = 0, 3  ! loop over colours
70       do y = 1, Ny
71          coly = mod(y, 2)
72          do x = 1, Nx
73             colx = mod(x, 2)
74             if (colour(colx, coly) == col) then
75                vold = v(x, y)
76                v(x, y) = 4.0 * (v(x - 1, y) &
77                                 + v(x + 1, y) &
78                                 + v(x, y - 1) &
79                                 + v(x, y + 1)) &
80                                 + v(x - 1, y - 1) &
81                                 + v(x - 1, y + 1) &
82                                 + v(x + 1, y - 1) &
83                                 + v(x + 1, y + 1)
84                v(x, y) = v(x, y) * 0.05
85                d = v(x, y) - vold
86                sum = sum + d**2
87             endif
88          enddo
89       enddo
90    enddo
91    diff = 20.0 * sqrt(sum)
92 end
```

Calling tree

```
main -+- write_para
      +- laplace -+- init
                  +- jacobi ------------+- jacobi5
                  |                     +- jacobi9
                  |
                  +- gauss_seidel -----+- gauss_seidel5
                  |                    +- gauss_seidel9
                  |
                  +- cg ---------------+- cg_init5
                  |                    +- cg_init9    +- cg_matrix_mult5
                  |                    +- cg_kernel --+- cg_matrix_mult9
                  |
                  +- gauss_seidel_col -+- gauss_seidel_col5
                  |                    +- gauss_seidel_col9
                  |
                  +- residual ---------+- init
                  |                    +- cg_init5
                  |                    +- cg_init9
                  |                    +- cg_matrix_mult5
die               +- output            +- cg_matrix_mult9
```

residual.c

```
1  #include <math.h>
2  #include "laplace.h"
3
4  void residual(char *datafile, field solution, int stencil,
5                int Nx, int Ny, double *diff)
6  {
7      field rhs = field_alloc(Ny, Nx);
8      field aax = field_alloc(Ny, Nx);
9
10     init(datafile, aax, Nx, Ny);  // aax[] is used to initialise rhs[]
11                                   // later: aax = A * solution
12     switch (stencil) {
13         case 5:
14             cg_init5(aax, rhs, Nx, Ny);
15             cg_matrix_mult5(aax, solution, Nx, Ny);
16             break;
17         case 9:
18             cg_init9(aax, rhs, Nx, Ny);
19             cg_matrix_mult9(aax, solution, Nx, Ny);
20             break;
21     }
22
```

residual.c

```
23      double sum = 0;
24      for (int j = 1; j <= Ny; j++) {
25          for (int i = 1; i <= Nx; i++) {
26              double d = aax[j][i] - rhs[j][i];
27              sum += d * d;
28          }
29      }
30      *diff = sqrt(sum);
31
32      field_free(rhs);
33      field_free(aax);
34  }
```

residual.f90

```
1  subroutine residual(datafile, solution, stencil, Nx, Ny, diff)
2
3    implicit none
4    character(*)          :: datafile
5    integer, intent(in)   :: stencil, Nx, Ny
6    real(8), intent(in)   :: solution(0:Nx + 1, 0:Ny + 1)
7    real(8), intent(out)  :: diff
8    real(8)               :: sum, d
9
10   real(8), dimension(:, :), allocatable :: rhs, aax
11   integer               :: i, j
12
13   allocate(rhs(0:Nx + 1, 0:Ny + 1))
14   allocate(aax(0:Nx + 1, 0:Ny + 1))
15
```

```
16    call init(datafile, aax, Nx, Ny)  ! aax is used to initialise rhs
17                                      ! later: aax = A * solution
18                                                          residual.f90
19    select case (stencil)
20       case(5)
21          call cg_init5(aax, rhs, Nx, Ny)
22          call cg_matrix_mult5(aax, solution, Nx, Ny)
23       case(9)
24          call cg_init9(aax, rhs, Nx, Ny)
25          call cg_matrix_mult9(aax, solution, Nx, Ny)
26    end select
27
28    sum = 0
29    do j = 1, Ny
30       do i = 1, Nx
31          d = aax(i, j) - rhs(i, j)
32          sum = sum + d**2
33       enddo
34    enddo
35
36    diff = sqrt(sum)
37
38    deallocate(rhs)
39    deallocate(aax)
40 end
```

– OpenMP project –
tasks

Task

Parallelise the program discussed with OpenMP.

Procedure:

- Recall the necessary steps from slide 127.
- Parallelise loop by loop by employing the directive:

  C: `#pragma omp parallel for`

  Fortran: `!$omp parallel do`

- For nested loops the outer loop shall be parallelised.
- If a (nested) loop cannot be parallelised note this as a comment in the program. Give a reason why the (nested) loop cannot be parallelised.

Hints (I)

- candidates for parallelisation are the following kinds of loops:

C:

```
for (i = 0; i <= Nx + 1; i++)
for (i = 1; i <= Nx; i++)
for (j = 0; j <= Ny + 1; j++)
for (j = 1; j <= Ny; j++)
for (x = 0; x <= Nx + 1; x++)
for (x = 1; x <= Nx; x++)
for (y = 0; y <= Ny + 1; y++)
for (y = 1; y <= Ny; y++)
```

Fortran:

```
do i = 0, Nx + 1
do i = 1, Nx
do j = 0, Ny + 1
do j = 1, Ny
do x = 0, Nx + 1
do x = 1, Nx
do y = 0, Ny + 1
do y = 1, Ny
```

Hints (II)

- generating a list of all loops (including file names and line numbers)
  - C code

    ```
    grep -n -w for *.c > loops.txt
    ```

  - Fortran code

    ```
    grep -n -w do *.f90 > loops.txt
    ```

Hints on testing

- generating and saving output from the unmodified program

```
#!/bin/bash
for A in 0 1 2 3
do
    for B in 5 9
    do
        ./laplace $A $B > laplace$A$B.out
    done
done
```

- testing parallel versions

```
OMP_NUM_THREADS=n ./laplace A B | diff - laplaceAB.out
```

$\Rightarrow$ be aware of *race conditions*, which might be undiscovered during testing $\Leftarrow$

Timing results (for the MPI program parallelised with OpenMP)

- times in seconds for 10000 iterations on a 1024×1024 mesh
- HPC cluster at Universität Hamburg (installed in 2015, 1 node has 2×8 cores)