Topics in C
for Programming with MPI and OpenMP

Hinnerk Stüben

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Universität Bremen

13–18 September 2025

What is important for programming with MPI and OpenMP?

- MPI is a subprogram library

  $\rightarrow$ functions

  $\hookrightarrow$ parameters

  $\hookrightarrow$ data types

- OpenMP is a language extension (mostly)
  - storage classes
  - scope

- for the programming project (MPI and OpenMP) we need
  - two-dimensional arrays

Other C language topics

- expressions and operators
- statements and flow control
- I/O
  - standard *library*
  - will be addressed in conjunction with parallel (MPI) I/O
- preprocessor
  - inclusion of header files
  - conditional compilation

Using functions

```
double f(double x, double y);   // function prototype declarations . . .
                                // . . . usually collected in header files

int main(void)                  // in C the main program is a function as well
{
    double x;
    double a = 1.0;
    double b = 2.0;
    ...
    x = f(a, b);                // function call
    ...                         // data types of parameters must match
    return 0;
}

double f(double x, double y)    // function definition
{
    return x + y;
}
```

Using functions from a library (example: `MPI_Send`)

- step 1

  look up the function declaration (e.g. by `man mpi_send`):

```
#include <mpi.h>
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
```

- step 2

  understand the meaning of the parameters (this course)

- step 3

  call the function with data objects that have
  - matching data types (correct *syntax*)
  - appropriate contents (right *semantics*)

Using `MPI_Send`

```
int i;                          // scalar (variable)
double x[10];                   // vector (array)

MPI_Send(&i,                    // send the contents of i
         1,                     // send 1 data item
         MPI_INT,               // data item is of type int
         5,                     // send data to process 5
         0,                     // message tag is 0
         MPI_COMM_WORLD);       // communication context is: all processes

MPI_Send(x,                     // send contents of x
         2,                     // send first 2 data items
         MPI_DOUBLE,            // data items are of type double
         4,                     // send data to process 4
         0,                     // message tag is 0
         MPI_COMM_WORLD);       // communication context is: all processes
```

Common MPI syntax errors (I)

```
int i;
double x[10];
```

```
MPI_Send(i, 1, MPI_INT, 5, 0, MPI_COMM_WORLD);        // i is not a pointer
                                                      // the address operator & is needed
```

```
MPI_Send(&x, 2, MPI_DOUBLE, 4,0,MPI_COMM_WORLD); // x is already a pointer
                                                 // the address operator & is not needed
```

```
MPI_Send(&i, 1, int, 5, 0, MPI_COMM_WORLD);           // int is a keyword
                                          // int is not an object of type MPI_Datatype
```

```
MPI_Send(x, 2, double, 4, 0, MPI_COMM_WORLD);     // double is a keyword
                                         // double is not an object of type MPI_Datatype
```

Common MPI syntax errors (II)

```
MPI_INT i;                  // MPI_INT is a named constant, not a type

MPI_DOUBLE x[10];           // MPI_DOUBLE is a named constant, not a type

MPI_Send(&i, 1, MPI_INT, 5, 0, MPI_COMM_WORLD);

MPI_Send(x, 2, MPI_DOUBLE, 4, 0, MPI_COMM_WORLD);
```

printf error

```
int i;
float f;
double d;

printf("%i", i);        // correct
printf("%d", i);        // correct

printf("%f", f);        // correct
printf("%d", d);        // wrong
printf("%f", d);        // correct
```

Using `struct`

- a `struct` is a data type defined by the programmer
- example:

```
struct point;                           // declaration (not necessary here)
struct point { double x, y; };          // definition

int main(void)
{
    struct point p;                     // structure variable declaration ...
    struct point q = { 1.0, 1.5 };      // ... and initialisation

    p.x = 2.0;                          // accessing structure members
    p.y = 2.5;
    ...
}
```

Using pointers to `struct`

- there is a special syntax for accessing members via a pointer to a structure
- example:

```
struct point { double x, y; };
struct point p;
struct point *ptr = &p;

ptr->x = 2.0;
ptr->y = 2.5;
```

- type casting

```
void my_function(void *p)
{
    struct point *q = (struct point *) p;

    q->x = 2.0;
    q->y = 2.5;
}
```

typedef

- `typedef` can be used to define new names
- examples:

```
// definition ...
    typedef double Temperature;   // Temperature is an alias for double
    typedef struct point Point;                  // Point is a new type
    typedef void MPI_User_function(void *invec, void *inoutvec,
                                   int *len, MPI_Datatype *datatype);
                                // MPI_User_function is a function declaration

// ... and usage of new names
    Temperature T;
    Point p, *q;
    MPI_User_function my_fun;

    void my_fun(void *in, void *inout, int *n, MPI_Datatype *mytype)
    {
        ...
    }
```

MPI datatypes vs. C data types

- an MPI datatype
  - is a name that refers to or describes a C data object
  - is a data object in C which has C data type `MPI_Datatype`

- `MPI_Datatype`
  - is a (derived) C data type
  - is defined via `typedef` in `<mpi.h>`

- example: declaration of `MPI_Datatype` in Open-MPI

```
struct ompi_datatype_t;
typedef struct ompi_datatype_t *MPI_Datatype;
```

- example: part of definition of `MPI_INT` in Open-MPI

```
#define OMPI_PREDEFINED_GLOBAL(type, global) ((type) ((void *) &(global)
#define MPI_INT OMPI_PREDEFINED_GLOBAL(MPI_Datatype, ompi_mpi_int)
```

Storage classes and scope

- *storage class* refers to the lifespan of a datum
  - *automatic* variables are allocated and deallocated automatically according to the program flow
  - *static* variables exist over the whole runtime of the program
- *scope* refers to the visibility of a datum
  - *local* variables are only visible in a block or function
  - *global* variables are visible in the whole program
- remarks
  - *automatic* variables are usually *local*
  - *global* variables are usually *static*
  - in C there is also *file scope*

Storage classes and scope in C – example

```
extern global_info;          // global (no instance exists yet)
struct Global global_info;   // static, global

static int status;           // static, file scope

void sub(void)
{
    double x;                // automatic, local
    int j = 1;               // automatic, local
    static int count = 0;    // static, local

    ...

    {
        int i;               // automatic, local
        ...
    }
}
```

Multi-dimensional arrays

- C has no multi-dimensional arrays with unknown dimensions at compile-time
- Fortran (correct)

```
subroutine sub(n, m, x)
  integer n
  integer m
  real(8) x(n, m)
  ...
end
```

- C (wrong)

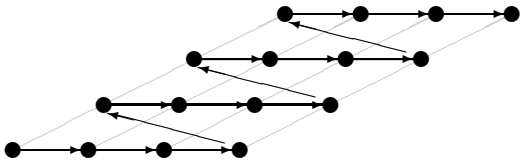
```
void sub(int n, int m, double x[n][m])
{
  ...
}
```

Data structures (I)

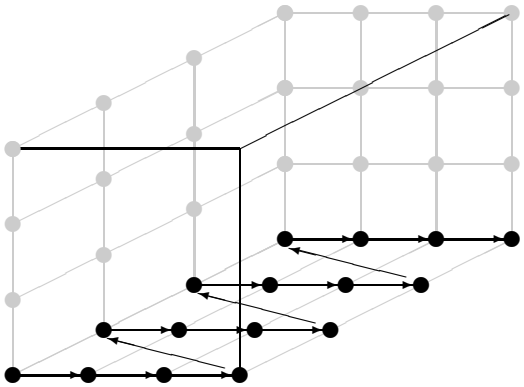
- two-dimensional arrays
- Fortran storage sequence
- definition of data structure `field` in C

Storage sequence of multi-dimensional arrays

Fortran

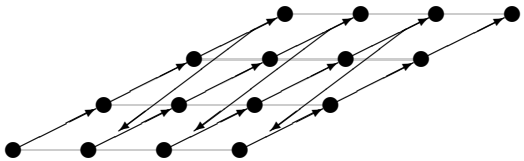


```
real(8) :: v(Nx, Ny)
```

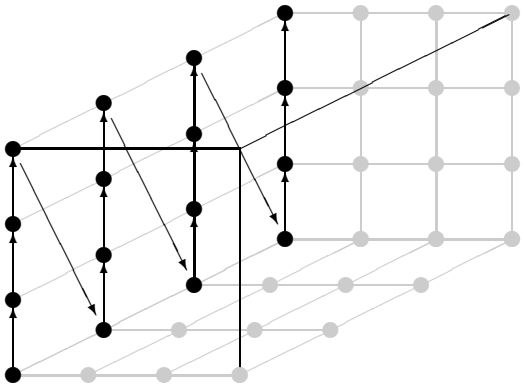


```
real(8) :: v(Nx, Ny, Nz)
```

C



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



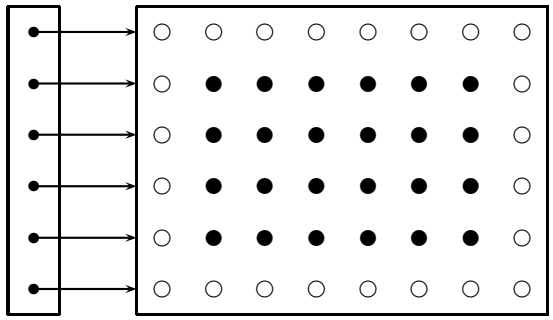
```
double v[Nx][Ny][Nz];
```

Data structures (II)

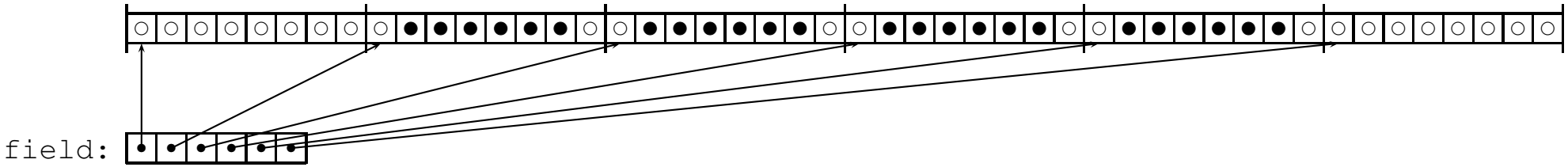
- `field` defines vectors of pointers to `double`: `typedef double **field;`

- 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);
}
```

Address and offset calculations

- address of an element of a 2-dimensional array

```
double a[N][M];
int i, j;
intptr_t address1, address2;
intptr_t base_address, offset;

address1 = (intptr_t) &a[i][j];

base_address = (intptr_t) a;
offset = (i * M + j) * sizeof(double);

address2 = base_address + offset;
```

→ `address1` and `address2` are identical

References and further reading

- Rothwell, *The GNU C Reference Manual*
  `http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf`
- `http://en.wikipedia.org`
- Kernighan and Ritchie, *The C Programming Language*
- Klemens, *21st Century C*