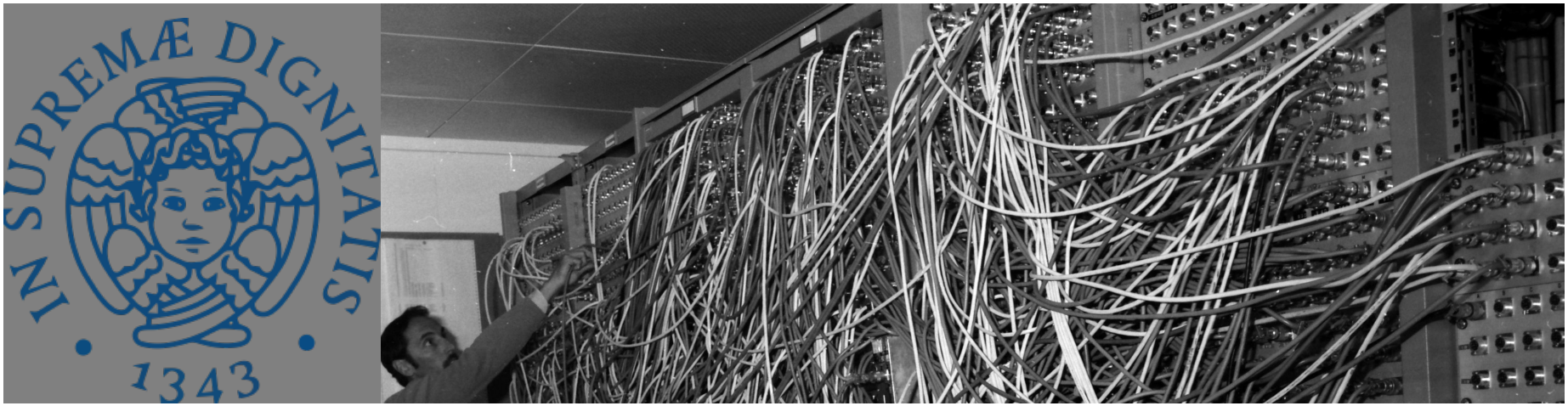


Calcolo Parallelo : Lezione 3

Fabio Durastante

Dipartimento di Matematica, Università di Pisa

Master in Scienze e Tecnologie Spaziali, 2021

1. Some advanced instructions
   - Derived datatypes
2. Numerical integration of the Hénon–Heiles model
3. “They’re moving in herds. They do move in herds.”
4. Conclusions

Derived datatypes

All the communication procedures we have seen until now are built on the assumption that

- all the buffers contain a sequence of identical *basic* datatypes.

This is constraining, sometimes we need more flexibility, we may want to

- pass messages that contain values with different datatypes,
- send noncontiguous data

Derived datatypes

All the communication procedures we have seen until now are built on the assumption that

- all the buffers contain a sequence of identical *basic* datatypes.

This is constraining, sometimes we need more flexibility, we may want to

- pass messages that contain values with different datatypes,
- send noncontiguous data

We can define a general datatype as an *opaque* object specifying

- a sequence of basic datatypes,
- a sequence of integer (byte) displacements (that may not be positive, distinct or increasing in order)

Derived datatypes

All the communication procedures we have seen until now are built on the assumption that

- all the buffers contain a sequence of identical *basic* datatypes.

This is constraining, sometimes we need more flexibility, we may want to

- pass messages that contain values with different datatypes,
- send noncontiguous data

We can define a general datatype as an *opaque* object specifying

- a sequence of basic datatypes,
- a sequence of integer (byte) displacements (that may not be positive, distinct or increasing in order)

Such **sequence of pairs** is called then a type map:

$$\text{Typemap} = \{(\text{type}_0\ ,\ \text{disp}_0\ ),\ \ldots,\ (\text{type}_{n-1}\ ,\ \text{disp}_{n-1}\ )\},$$

where type_i are basic types, and disp_i are displacements.

Derived datatypes

All the communication procedures we have seen until now are built on the assumption that

- all the buffers contain a sequence of identical *basic* datatypes.

This is constraining, sometimes we need more flexibility, we may want to

- pass messages that contain values with different datatypes,
- send noncontiguous data

We can define a general datatype as an *opaque* object specifying

- a sequence of basic datatypes,
- a sequence of integer (byte) displacements (that may not be positive, distinct or increasing in order)

Such **sequence of pairs** is called then a type map:

$$\text{Typemap} = \{(\text{type}_0\ ,\ \text{disp}_0\ ),\ \ldots,\ (\text{type}_{n-1}\ ,\ \text{disp}_{n-1}\ )\},$$

where type_i are basic types, and disp_i are displacements.

Derived datatypes

All the communication procedures we have seen until now are built on the assumption that

- all the buffers contain a sequence of identical *basic* datatypes.

This is constraining, sometimes we need more flexibility, we may want to

- pass messages that contain values with different datatypes,
- send noncontiguous data

We can define a general datatype as an *opaque* object specifying

- a sequence of basic datatypes,
- a sequence of integer (byte) displacements (that may not be positive, distinct or increasing in order)

Such **sequence of pairs** is called then a type map:

$$\text{Typemap} = \{(\text{type}_0\ ,\ \text{disp}_0\ ),\ \ldots,\ (\text{type}_{n-1}\ ,\ \text{disp}_{n-1}\ )\},$$

where type_i are basic types, and disp_i are displacements.

Derived datatypes

Given a type map

$$\text{Typemap} = \{(\text{type}_0\ ,\ \text{disp}_0\ ),\ \ldots,\ (\text{type}_{n-1}\ ,\ \text{disp}_{n-1}\ )\},$$

then

- the **extent** of a datatype is defined to be the span from the first byte to the last byte occupied by entries in this datatype, i.e.,

  $\text{lb(Typemap)} = \min_j \text{disp}_j$,
  $\text{ub(Typemap)} = \max_j(\text{disp}_j + \text{sizeof}(\text{tipe}_j)) + \varepsilon$,
  extent(Typemap) = ub(Typemap) - lb(Typemap),

  in which the ε is used to to satisfy alignment requirements.

Derived datatypes

Given a type map

$$\text{Typemap} = \{(\text{type}_0 \text{ , disp}_0 \text{ }), \ldots, (\text{type}_{n-1} \text{ , disp}_{n-1} \text{ })\},$$

then

- the **extent** of a datatype is defined to be the span from the first byte to the last byte occupied by entries in this datatype,
- The definition of extent is motivated by the assumption that the amount of *padding added* at the end of each structure in an array of structures is *the least needed* to fulfill alignment constraints.

Derived datatypes

Given a type map

$$\text{Typemap} = \{(\text{type}_0\ ,\ \text{disp}_0\ ),\ \ldots,\ (\text{type}_{n-1}\ ,\ \text{disp}_{n-1}\ )\},$$

then

- the **extent** of a datatype is defined to be the span from the first byte to the last byte occupied by entries in this datatype,
- The definition of extent is motivated by the assumption that the amount of *padding added* at the end of each structure in an array of structures is *the least needed* to fulfill alignment constraints.
- We need some *constructor* routines to build and define the new datatypes.

Datatype constructors – MPI_TYPE_CONTIGUOUS

Allows for the replication of a datatype into contiguous locations.

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
  MPI_Datatype *newtype)
```

`int count` replication count,

`MPI_Datatype oldtype` old datatype,

`MPI_Datatype *newtype` new datatype.

- `newtype` is the datatype obtained by concatenating `count` copies of `oldtype`,
- the concatenation is defined using *extent* as the size of the concatenated copies.

Datatype constructors – MPI_TYPE_CONTIGUOUS

Allows for the replication of a datatype into contiguous locations.

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
  MPI_Datatype *newtype)
```

`int count` replication count,

`MPI_Datatype oldtype` old datatype,

`MPI_Datatype *newtype` new datatype.

- `newtype` is the datatype obtained by concatenating `count` copies of `oldtype`,
- the concatenation is defined using *extent* as the size of the concatenated copies.
- Example: If the old datatype has *type map* {(double, 0), (char, 8)}, and we choose `count=2;`, then the new datatype has *type map*

Datatype constructors – MPI_TYPE_CONTIGUOUS

Allows for the replication of a datatype into contiguous locations.

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
  MPI_Datatype *newtype)
```

`int count` replication count,

`MPI_Datatype oldtype` old datatype,

`MPI_Datatype *newtype` new datatype.

- `newtype` is the datatype obtained by concatenating `count` copies of `oldtype`,
- the concatenation is defined using *extent* as the size of the concatenated copies.
- Example: If the old datatype has *type map* {(double, 0), (char, 8)}, and we choose `count=2;`, then the new datatype has *type map* {(double, 0), (char, 8), (double, 16), (char, 24)}.

Datatype constructors – Commit and Free

Can we use now our new *datatype* in a communication?

Datatype constructors – Commit and Free

Can we use now our new *datatype* in a communication?

- **No!** A datatype object has to be committed before it can be used in a communication,

Datatype constructors – Commit and Free

Can we use now our new *datatype* in a communication?

- **No!** A datatype object has to be committed before it can be used in a communication,
- we need to commit the formal description of a communication buffer having this datatype:

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

Datatype constructors – Commit and Free

Can we use now our new *datatype* in a communication?

- **No!** A datatype object has to be committed before it can be used in a communication,
- we need to commit the formal description of a communication buffer having this datatype:

  ```
  int MPI_Type_commit(MPI_Datatype *datatype)
  ```

- after a datatype has been committed, it can be repeatedly reused to communicate *any* buffer of that datatype.

Datatype constructors – Commit and Free

Can we use now our new *datatype* in a communication?

- **No!** A datatype object has to be committed before it can be used in a communication,
- we need to commit the formal description of a communication buffer having this datatype:

  ```
  int MPI_Type_commit(MPI_Datatype *datatype)
  ```

- after a datatype has been committed, it can be repeatedly reused to communicate *any* buffer of that datatype.
- When we have ended using the new datatype we can free (deallocate) it by doing

  ```
  int MPI_Type_free(MPI_Datatype *datatype)
  ```

  observe that, any communication that is currently using this datatype will complete normally, and that derived datatype depending on the one you are deallocating will not be touched.

Datatype constructors – An example

We want to modify the simple send and receive program from Lecture 2:

The fundamental part of the code:

```
char message[20];
if (myrank == 0){
strcpy(message,"Hello, there");
MPI_Send(message, strlen(message)+1,
  MPI_CHAR, 1, 99, MPI_COMM_WORLD);
}
else if (myrank == 1){
MPI_Recv(message, 20, MPI_CHAR, 0,
  99, MPI_COMM_WORLD, &status);
printf("received :%s:\n", message);
}
```

Datatype constructors – An example

We want to modify the simple send and receive program from Lecture 2:

The fundamental part of the code:

```
char message[20];
if (myrank == 0){
strcpy(message,"Hello, there");
MPI_Send(message, strlen(message)+1,
  MPI_CHAR, 1, 99, MPI_COMM_WORLD);
}
else if (myrank == 1){
MPI_Recv(message, 20, MPI_CHAR, 0,
  99, MPI_COMM_WORLD, &status);
printf("received :%s:\n", message);
}
```

We want to introduce a new type for strings made of 20 `char`, and use it in our communications.

Note: a string is exactly a contiguous set of `char`.

Datatype constructors – An example

We want to modify the simple send and receive program from Lecture 2:

The fundamental part of the code:

```
char message[20];
if (myrank == 0){
strcpy(message,"Hello, there");
MPI_Send(message, strlen(message)+1,
  MPI_CHAR, 1, 99, MPI_COMM_WORLD);
}
else if (myrank == 1){
MPI_Recv(message, 20, MPI_CHAR, 0,
  99, MPI_COMM_WORLD, &status);
printf("received :%s:\n", message);
}
```

We want to introduce a new type for strings made of 20 `char`, and use it in our communications.

Note: a string is exactly a contiguous set of `char`.

We define the new type by doing:

```
MPI_Datatype mystring;
MPI_Type_contiguous(20,MPI_CHAR,
 &mystring);
```

then we commit it by

```
MPI_Type_commit(&mystring);
```

and rewrite the send/receive as

```
 if (myrank == 0){
strcpy(message,"Hello, there");
MPI_Send(message, 1, mystring, 1, 99,
 MPI_COMM_WORLD);
}
else if (myrank == 1){
MPI_Recv(message, 1, mystring, 0, 99,
 MPI_COMM_WORLD, &status);
printf("received :%s:\n", message);
}
```

before finalizing we deallocate the type

```
 MPI_Type_free(&mystring);
```

Datatype constructors – MPI_TYPE_VECTOR

There is also a more general construct available in MPI that allows replication of a datatype into locations of equally spaced blocks.

```
int MPI_Type_vector(int count, int blocklength, int stride,
  MPI_Datatype oldtype, MPI_Datatype *newtype)
```

`int count` number of blocks

`int blocklength` number of elements in each block

`int stride` number of elements between start of each block

`MPI_Datatype oldtype` old datatype

`MPI_Datatype *newtype` new datatype

- Each block is obtained by concatenating the same number of copies of the old datatype.
- The spacing between blocks is a multiple of the extent of the old datatype.

Datatype constructors – Example

We try now to use the `MPI_TYPE_VECTOR` by starting from a `MPI_TYPE_CONTIGUOUS` type.

1. we define an `MPI_TYPE_CONTIGUOUS` made up of 3 `int`:

   ```
   MPI_Type_contiguous(3, MPI_INT, &my3int);
   ```

2. we commit the new type

   ```
   MPI_Type_commit(&my3int);
   ```

3. Then we define (and commit) a vector type consisting in 3 blocks, of size 2 and a stride of 3 elements:

   ```
   MPI_Type_vector(3, 2, 3, my3int, &myvector);
   MPI_Type_commit(&myvector);
   ```

4. On `rank` 0 we create an array `int buffer[24];` and populate it with the number $i = 0, \ldots, 23$. Then we send it to `rank` 1 as an element of type `myvector`

   ```
   MPI_Send(buffer, 1, myvector, 1, 666, MPI_COMM_WORLD);
   ```

Datatype constructors – Example

5. We receive the message on `rank` 1 on a buffer initialized to -1

```
for (i=0; i<24; i++)
 buffer[i] = -1;
MPI_Recv(buffer, 1, myvector, 0, 666, MPI_COMM_WORLD,
    &status);
```

We can depict the communication as:

0	1	2	3	4	5	6	7	8	9	10	...	21	22	23

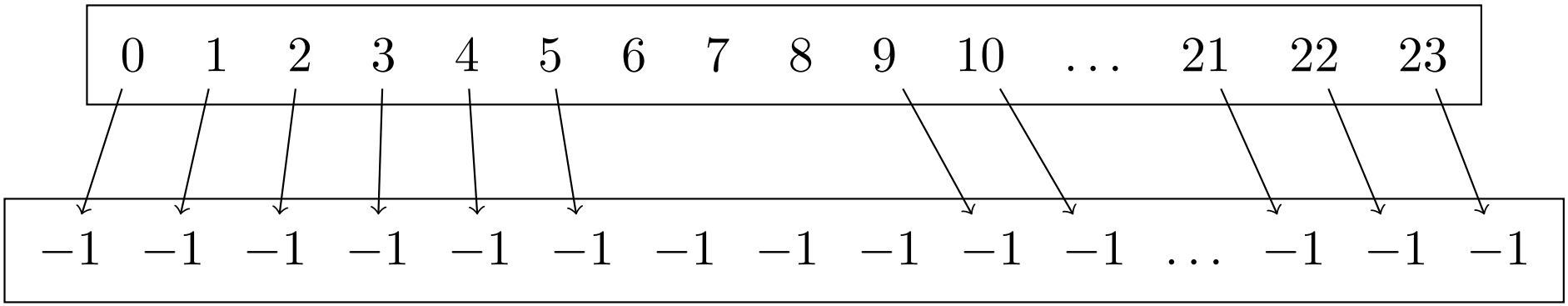
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	...	-1	-1	-1

Datatype constructors – Example

5. We receive the message on `rank` 1 on a buffer initialized to -1

```
for (i=0; i<24; i++)
 buffer[i] = -1;
MPI_Recv(buffer, 1, myvector, 0, 666, MPI_COMM_WORLD,
    &status);
```

We can depict the communication as:



Datatype constructors – Example

5. We receive the message on `rank` 1 on a buffer initialized to -1

```
for (i=0; i<24; i++)
 buffer[i] = -1;
MPI_Recv(buffer, 1, myvector, 0, 666, MPI_COMM_WORLD,
    &status);
```

We can depict the communication as:

0	1	2	3	4	5	6	7	8	9	10	...	21	22	23

0	1	2	3	4	5	-1	-1	-1	9	10	...	21	22	23

Datatype constructors – MPI_TYPE_CREATE_SUBARRAY

This constructor creates a datatype describing an n-dimensional subarray of an n-dimensional array.

```
int MPI_Type_create_subarray(int ndims,
 const int array_of_sizes[],
 const int array_of_subsizes[],
 const int array_of_starts[],
 int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

where

`int ndims` number of array dimensions

`const int array_of_sizes[]` number of `oldtype` elements in each dimension of the full array

`const int array_of_subsizes[]` number of `oldtype` elements in each dimension of the subarray

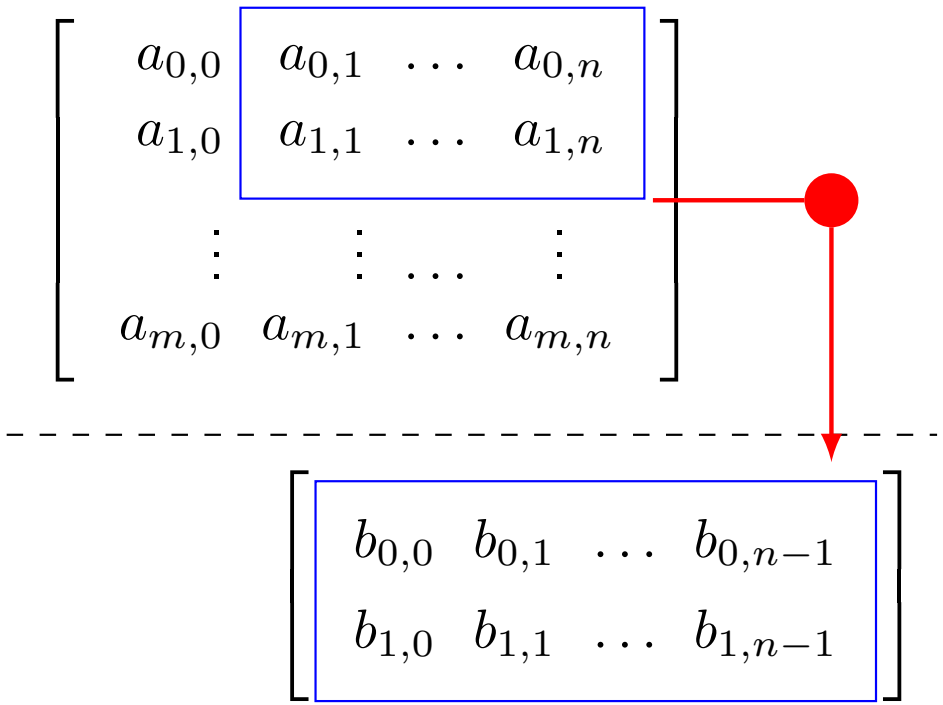
`const int array_of_starts[]` starting coordinates of the subarray in each dimension

`int order` array storage order, either `MPI_ORDER_C` (row–major order), `MPI_ORDER_FORTRAN` (column-major order)

Datatype constructors – MPI_TYPE_CREATE_SUBARRAY

This constructor creates a datatype describing an n-dimensional subarray of an n-dimensional array.

```
int MPI_Type_create_subarray(int ndims,
 const int array_of_sizes[],
 const int array_of_subsizes[],
 const int array_of_starts[],
 int order, MPI_Datatype oldtype, MPI_Datatype *newtype)
```



- The subarray may be situated anywhere within the full array,
- The subarray may be of any (nonzero) size up to the size of the larger array (it has to be confined within the original array!)
- Note that a C program may use Fortran order and a Fortran program may use C order.

Datatype constructors – Example

As a first example we extract and communicate a subarray of a 1D array.

- We start by producing a subarray for a 1D array `int array[9]`

```
int myrank;
MPI_Status status;
MPI_Datatype subarray;
int array[9] = { -1, 1, 2, 3, -2,
-3, -4, -5, -6 };
int array_size[] = {9};
int array_subsize[] = {3};
int array_start[] = {1};
```

Datatype constructors – Example

As a first example we extract and communicate a subarray of a 1D array.

```
int myrank;
MPI_Status status;
MPI_Datatype subarray;
int array[9] = { -1, 1, 2, 3, -2,
-3, -4, -5, -6 };
int array_size[] = {9};
int array_subsize[] = {3};
int array_start[] = {1};

MPI_Init(&argc, &argv);
MPI_Type_create_subarray(1, array_size,
 array_subsize, array_start,
 MPI_ORDER_C, MPI_INT, &subarray);
MPI_Type_commit(&subarray);
```

- We start by producing a subarray for a 1D array `int array[9]`
- We select the subarray made of 3 elements (`int array_subsize[]={3};`), starting from the first position (`int array_start[]={1}`) from an array of 9 elements (`int array_size[]={9}`);

Datatype constructors – Example

As a first example we extract and communicate a subarray of a 1D array.

```
int myrank;
MPI_Status status;
MPI_Datatype subarray;
int array[9] = { -1, 1, 2, 3, -2,
-3, -4, -5, -6 };
int array_size[] = {9};
int array_subsize[] = {3};
int array_start[] = {1};

MPI_Init(&argc, &argv);
MPI_Type_create_subarray(1, array_size,
 array_subsize, array_start,
 MPI_ORDER_C, MPI_INT, &subarray);
MPI_Type_commit(&subarray);

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0){
 MPI_Send(array, 1, subarray, 1, 123,
  MPI_COMM_WORLD);
}
```

- We start by producing a subarray for a 1D array `int array[9]`
- We select the subarray made of 3 elements (`int array_subsize[]={3};`), starting from the first position (`int array_start[]={1}`) from an array of 9 elements (`int array_size[]={9}`);
- From the process with `myrank = 0;` we send the subarray taken from the variable `array` to the process with rank 1.

Datatype constructors – Example

As a first example we extract and communicate a subarray of a 1D array.

```
else if (myrank == 1){
 for (int i=0; i<9; i++)
  array[i] = 0;
 MPI_Recv(array, 1, subarray, 0, 123,
  MPI_COMM_WORLD, &status);
 for (int i=0; i<9; i++)
  printf("array[%d] = %d\n", i, array[i]);
 fflush(stdout);
}
```

- The process with `myrank = 1;` receives the variable and stores it into its version of the variable `array`.

Datatype constructors – Example

As a first example we extract and communicate a subarray of a 1D array.

```
else if (myrank == 1){
 for (int i=0; i<9; i++)
  array[i] = 0;
 MPI_Recv(array, 1, subarray, 0, 123,
  MPI_COMM_WORLD, &status);
 for (int i=0; i<9; i++)
  printf("array[%d] = %d\n", i, array[i]);
 fflush(stdout);
}

MPI_Type_free(&subarray);
MPI_Finalize();
```

- The process with `myrank = 1;` receives the variable and stores it into its version of the variable `array`.
- In conclusion we deallocate the `subarray` type, and finalize the MPI environment.

Datatype constructors – Example

As a first example we extract and communicate a subarray of a 1D array.

```
else if (myrank == 1){
 for (int i=0; i<9; i++)
  array[i] = 0;
 MPI_Recv(array, 1, subarray, 0, 123,
  MPI_COMM_WORLD, &status);
 for (int i=0; i<9; i++)
  printf("array[%d] = %d\n", i, array[i]);
 fflush(stdout);
}

MPI_Type_free(&subarray);
MPI_Finalize();
```

- The process with `myrank = 1;` receives the variable and stores it into its version of the variable `array`.
- In conclusion we deallocate the `subarray` type, and finalize the MPI environment.

- When we run the code we print in output the `array` with values

  `array[0] = 0` `array[1] = 1` `array[2] = 2` `array[3] = 3`

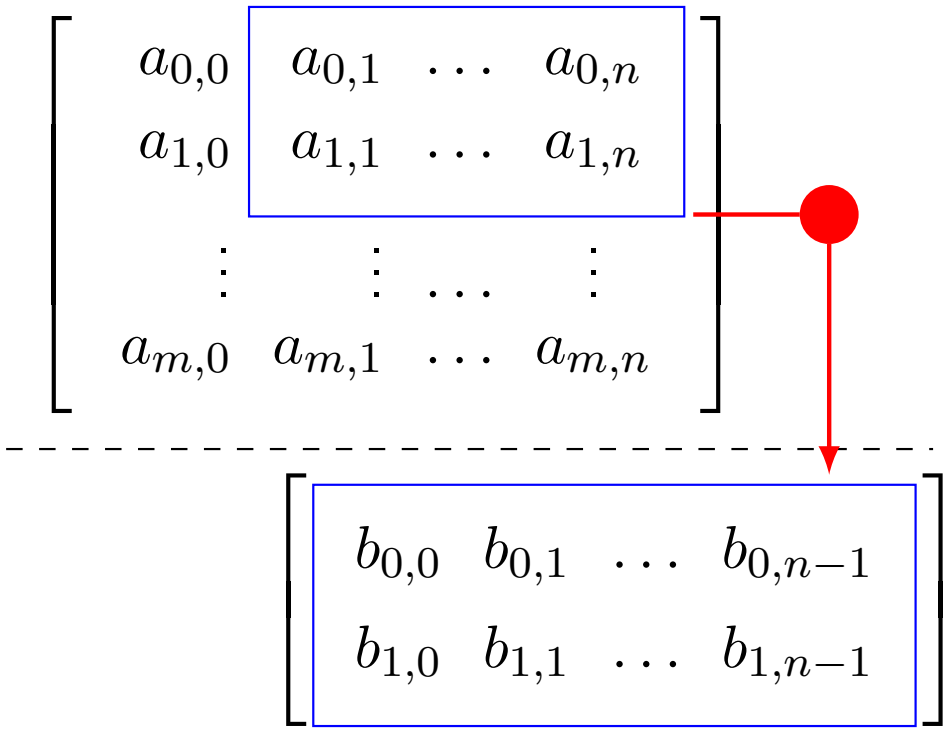
  `array[4] = 0` ...

  that are the elements in position 2 to 4 of

  `array[9] = { -1, 1, 2, 3, -2,-3, -4, -5, -6 };`

Datatype constructors – Example

- We could have been more clever in the receveing phase:



- If we look back at the figure the receveing array was instead an array object of the received size,
- Nevertheless, if we expect to use the the new datatype `subarray` on the receiving end we have to use an array with global size equal to the one used to build the type
- We can do it in a slightly different way to match what is depicted in the figure:

```
int receivearray[3];
MPI_Recv(receivearray, 3, MPI_INT, 0, 123,
 MPI_COMM_WORLD, &status);
```

Datatype constructors – Exercise

Let us use the `MPI_Type_create_subarray` routine to extract a 2D subarray from a 2D array.

1. We want to be a little more general than we have been for the 1D array case, thus let us define first two functions:
   - a function `int **allocarray(int n)` that allocates a $n \times n$ array;
   - a function `void printarr(int **array, int n, char *str)` that takes as input the address of a 2D array `int **array`, its size `int n`, and a string with the name of the array for printing everything to screen.
2. Then processor 0 initializes (and populates) the `bigarray`, build the new data type with the `MPI_Type_create_subarray` routine, prints it in output and sends the data to a (smaller) array on processor 1.
3. On the other side, processor 1 initializes the memory for the `subarray`, receives the message from process 0, prints it in output and terminates.

- You have an "skeleton code" in the `subarray2Darray.c` file in the code folder.

Datatype constructors – Exercise

The function to dynamically allocate a 2D array can be written as:

```
int **allocarray(int n) {
int *rows = malloc(n*n*sizeof(int));
int **arr2D = malloc(n*sizeof(int *));
for (int i=0; i<n; i++)
  arr2D[i] = &(rows[i*n]);
return arr2D;
}
```

while the function to print a given $n \times n$ array can be obtained as

```
void printarr(int **array, int n, char *str) {
 printf("-- %s --\n", str);
 for (int i=0; i<n; i++) {
  for (int j=0; j<n; j++) {
   printf("%3d ", array[i][j]);
  }
  printf("\n");
 }
}
```

Datatype constructors – Exercise

On the processor with `rank = 0;` we initialize (and populate) the `bigarray`, build the new data type with the `MPI_Type_create_subarray` routine, then we print it in output and sends the data to a (smaller) array on processor 1.

```
int **bigarray = allocarray(bigsize);
for (int i=0; i<bigsize; i++)
 for (int j=0; j<bigsize; j++)
  bigarray[i][j] = i*bigsize+j;
printarr(bigarray, bigsize, " Sender: Big array ");
```

and then the main part:

```
MPI_Datatype mysubarray;
// We choose the coordinate from which we start the communication:
int starts[ ] = {};
int subsizes[ ]  = {}; // The size of the square subarray to
                       // communicate
int bigsizes[ ]  = {}; // The size of the big square array
MPI_Type_create_subarray(); // We create the subarray type
MPI_Type_commit(); // We commit it
MPI_Send(); // We send the subarray to process 1
MPI_Type_free(); // We deallocate the type
```

Datatype constructors – Exercise

On the processor with `rank = 0;` we initialize (and populate) the `bigarray`, build the new data type with the `MPI_Type_create_subarray` routine, then we print it in output and sends the data to a (smaller) array on processor 1.

```
int **bigarray = allocarray(bigsize);
for (int i=0; i<bigsize; i++)
 for (int j=0; j<bigsize; j++)
  bigarray[i][j] = i*bigsize+j;
printarr(bigarray, bigsize, " Sender: Big array ");
```

and then the main part:

```
MPI_Datatype mysubarray;
int starts[2] = {5,3};
int subsizes[2]  = {subsize,subsize};
int bigsizes[2]  = {bigsize,bigsize};
MPI_Type_create_subarray(2, bigsizes, subsizes, starts,
MPI_ORDER_C, MPI_INT, &mysubarray);
MPI_Type_commit(&mysubarray);
MPI_Send(&(bigarray[0][0]),1,mysubarray,1,tag,MPI_COMM_WORLD);
MPI_Type_free(&mysubarray);
```

Datatype constructors – Exercise

Processor 1 initializes the memory for the `subarray`, receives the message from process 0, prints it in output and terminates.

```
// Now we are rank 1: we first allocate the array for
// receiving the message from rank 0 (it has to be a
// square array of size subsize^2):
int **subarray = allocarray();
// with a double for loop we put to zero the entries of the
// receiveing array:
for (int i=0; i<subsize; i++)
 for (int j=0; j<subsize; j++)
  subarray[i][j] = 0;
// We receive the data in the subarray we have allocated
MPI_Recv();
// We print what we have received:
printarr(subarray, subsize,"Receiver: Subarray -- after receive");
```

Datatype constructors – Exercise

Processor 1 initializes the memory for the subarray, receives the message from process 0, prints it in output and terminates.

```
// Now we are rank 1: we first allocate the array for
// receiving the message from rank 0 (it has to be a
// square array of size subsize^2):
int **subarray = allocarray(subsize);
// with a double for loop we put to zero the entries of the
// receiveing array:
for (int i=0; i<subsize; i++)
 for (int j=0; j<subsize; j++)
  subarray[i][j] = 0;
// We receive the data in the subarray we have allocated
MPI_Recv(&(subarray[0][0]), subsize*subsize, MPI_INT, 0,
  tag, MPI_COMM_WORLD, &status);
// We print what we have received:
printarr(subarray, subsize,"Receiver: Subarray -- after receive");
```

Datatype constructors – (Further) Exercise(s)

Observe that:

- we have freed the `mysubarray` (`MPI_Type_free(&mysubarray);`) right after sending a data with this type, this operation is *safe* since it does not interfere with instantiated communications.
- to free the dynamically allocated arrays we do it in the order:

  ```
  free(bigarray[0]);
  free(bigarray);
  ```

  i.e., we first free the memory allocated for the rows, then we free the pointer to the whole array. We need to free the memory from inside/out, otherwise we loose the inner pointers!

Datatype constructors – (Further) Exercise(s)

Observe that:

- we have freed the `mysubarray` (`MPI_Type_free(&mysubarray);`) right after sending a data with this type, this operation is *safe* since it does not interfere with instantiated communications.
- to free the dynamically allocated arrays we do it in the order:

  ```
  free(bigarray[0]);
  free(bigarray);
  ```

  i.e., we first free the memory allocated for the rows, then we free the pointer to the whole array. We need to free the memory from inside/out, otherwise we loose the inner pointers!

Some extensions of the previous exercise could be:

- Generalize the construction of the array to a rectangular array of size $n \times m$,
- Try allocating and sending array with more than 2D dimensions,

Numerical integration of the Hénon–Heiles model

As a concluding example we consider the problem of integrating an (a system of) ODE(s) for different values of the parameters defining it.

Numerical integration of the Hénon–Heiles model

As a concluding example we consider the problem of integrating an (a system of) ODE(s) for different values of the parameters defining it.

Parallel integration of ODEs

There exists algorithms (and libraries) that permits to actually integrate in parallel an ODE, they exploit

- parallelism across the method, e.g., substantially rewrite the sequential nature of integration procedure: exploit independent stages of multi–stage algorithms,
- parallelism across the system, e.g., different evolution time for the ODEs: waveform relaxation.
- parallelism across the steps, e.g., we have a huge time domain that we want to divide into parallel processes: parallel in time algorithm.

Numerical integration of the Hénon–Heiles model

As a concluding example we consider the problem of integrating an (a system of) ODE(s) for different values of the parameters defining it.

Parallel integration of ODEs

There exists algorithms (and libraries) that permits to actually integrate in parallel an ODE, they exploit

- parallelism across the method, e.g., substantially rewrite the sequential nature of integration procedure: exploit independent stages of multi–stage algorithms,
- parallelism across the system, e.g., different evolution time for the ODEs: waveform relaxation.
- parallelism across the steps, e.g., we have a huge time domain that we want to divide into parallel processes: parallel in time algorithm.

We are going to treat a much simpler usage of parallelism: having to integrate the same ODE(s) for different values of some parameters.

Numerical integration of the Hénon–Heiles model

Let us remain on the concrete, the ODEs we want to integrate is obtained from the Hamiltonian

$$H(\mathbf{p},\mathbf{q}) = \frac{\omega_1}{2}(q_1^2+p_1^2) + \frac{\omega_2}{2}(q_2^2+p_2^2) + p_1^2 q_2 - \frac{1}{3}p_2^3, \quad \omega_1,\omega_2 \in \mathbb{R},$$

where the q_i denotes the position coordinates, the p_i the momentum coordinates, and from which we obtain the Hamilton's equations

$$\begin{cases} \dot{q}_1 = \frac{\partial H}{\partial p_1} = 2p_1q_2 + \omega_1 p_1, \\ \dot{q}_2 = \frac{\partial H}{\partial p_2} = \omega_2 p_2 - \frac{2p_2}{3}, \\ \dot{p}_1 = -\frac{\partial H}{\partial q_1} = -\omega_1 q_1, \\ \dot{p}_2 = -\frac{\partial H}{\partial q_2} = -p_1^2 - \omega_2 q_2. \end{cases}$$

and we are interested in integrating it for different values of the constants ω_1, ω_2, and different values of the energy E.

Numerical integration of the Hénon–Heiles model

Let us remain on the concrete, the ODEs we want to integrate is obtained from the Hamiltonian

$$H(\mathbf{p},\mathbf{q}) = \frac{\omega_1}{2}(q_1^2+p_1^2) + \frac{\omega_2}{2}(q_2^2+p_2^2) + p_1^2 q_2 - \frac{1}{3}p_2^3, \quad \omega_1,\omega_2 \in \mathbb{R},$$

where the q_i denotes the position coordinates, the p_i the momentum coordinates, and from which we obtain the Hamilton's equations

$$\begin{cases} \dot{q}_1 = \frac{\partial H}{\partial p_1} = 2p_1q_2 + \omega_1 p_1, \\ \dot{q}_2 = \frac{\partial H}{\partial p_2} = \omega_2 p_2 - \frac{2p_2}{3}, \\ \dot{p}_1 = -\frac{\partial H}{\partial q_1} = -\omega_1 q_1, \\ \dot{p}_2 = -\frac{\partial H}{\partial q_2} = -p_1^2 - \omega_2 q_2. \end{cases}$$

and we are interested in integrating it for different values of the constants ω_1, ω_2, and different values of the energy E.

What are the operation that our (parallel) software should perform?

Numerical integration of the Hénon–Heiles model

The "general idea" for performing such task can be summarized as:

1. Processor 0 reads the list of parameters in input for each instance of the problem,
2. Processor 0 *scatters* such data to every other process in the pool,
3. Each processor from `0,...,size-1` executes its own integration,
4. Processor 0 *gathers* the results from all the processors,
5. Processor 0 writes the result on a file in a format that can be used to plot the `size` different solutions.

Numerical integration of the Hénon–Heiles model

The "general idea" for performing such task can be summarized as:

1. Processor 0 reads the list of parameters in input for each instance of the problem,
2. Processor 0 *scatters* such data to every other process in the pool,
3. Each processor from `0,...,size-1` executes its own integration,
4. Processor 0 *gathers* the results from all the processors,
5. Processor 0 writes the result on a file in a format that can be used to plot the `size` different solutions.

Now, we need to precise this statements. . . and the devil is in the details

Numerical integration of the Hénon–Heiles model

The "general idea" for performing such task can be summarized as:

1. Processor 0 reads the list of parameters in input for each instance of the problem,
2. Processor 0 *scatters* such data to every other process in the pool,
3. Each processor from `0,...,size-1` executes its own integration,
4. Processor 0 *gathers* the results from all the processors,
5. Processor 0 writes the result on a file in a format that can be used to plot the `size` different solutions.

Now, we need to precise this statements. . . and the devil is in the details

- To decide what parameters we need, we first need to decide what method we want to use for the integration,

Numerical integration of the Hénon–Heiles model

The "general idea" for performing such task can be summarized as:

1. Processor 0 reads the list of parameters in input for each instance of the problem,
2. Processor 0 *scatters* such data to every other process in the pool,
3. Each processor from `0,...,size-1` executes its own integration,
4. Processor 0 *gathers* the results from all the processors,
5. Processor 0 writes the result on a file in a format that can be used to plot the `size` different solutions.

Now, we need to precise this statements. . . and the devil is in the details

- To decide what parameters we need, we first need to decide what method we want to use for the integration,
- What type of method do we need?

Numerical integration of the Hénon–Heiles model

The "general idea" for performing such task can be summarized as:

1. Processor 0 reads the list of parameters in input for each instance of the problem,
2. Processor 0 *scatters* such data to every other process in the pool,
3. Each processor from `0,...,size-1` executes its own integration,
4. Processor 0 *gathers* the results from all the processors,
5. Processor 0 writes the result on a file in a format that can be used to plot the `size` different solutions.

Now, we need to precise this statements. . . and the devil is in the details

- To decide what parameters we need, we first need to decide what method we want to use for the integration,
- What type of method do we need? We need conservation of energy, so we need a *symplectic* method.

Symplectic integrators

A symplectic integrator is a numerical integration scheme for Hamiltonian systems, i.e., it is used to integrate equations of the form:

$$\dot{q}_i = \frac{\partial H}{\partial p_i}, \quad \dot{p}_i = -\frac{\partial H}{\partial q_i}, \quad i = 1, \ldots, n.$$

This is a method for which, under opportune constraints on its parameters, a solution on a symplectic manifold is computed.

Symplectic integrators

A symplectic integrator is a numerical integration scheme for Hamiltonian systems, i.e., it is used to integrate equations of the form:

$$\dot{q}_i = \frac{\partial H}{\partial p_i}, \quad \dot{p}_i = -\frac{\partial H}{\partial q_i}, \quad i = 1, \ldots, n.$$

This is a method for which, under opportune constraints on its parameters, a solution on a symplectic manifold is computed.

- a symplectic integrator conserves the value of H along the computed trajectories,

Symplectic integrators

A symplectic integrator is a numerical integration scheme for Hamiltonian systems, i.e., it is used to integrate equations of the form:

$$\dot{q}_i = \frac{\partial H}{\partial p_i}, \quad \dot{p}_i = -\frac{\partial H}{\partial q_i}, \quad i = 1, \ldots, n.$$

This is a method for which, under opportune constraints on its parameters, a solution on a symplectic manifold is computed.

- a symplectic integrator conserves the value of H along the computed trajectories,
- it conserves the two–from $d\mathbf{p} \wedge d\mathbf{q}$.

Symplectic integrators

A symplectic integrator is a numerical integration scheme for Hamiltonian systems, i.e., it is used to integrate equations of the form:

$$\dot{q}_i = \frac{\partial H}{\partial p_i}, \quad \dot{p}_i = -\frac{\partial H}{\partial q_i}, \quad i = 1, \ldots, n.$$

This is a method for which, under opportune constraints on its parameters, a solution on a symplectic manifold is computed.

- a symplectic integrator conserves the value of H along the computed trajectories,
- it conserves the two–from $d\mathbf{p} \wedge d\mathbf{q}$.
- we are doing it *numerically* thus "conserves" means up to the accuracy of method, i.e., it conserves a *numerical* Hamiltonian (. . . usually a perturbation of the original one)

Numerical integration of the Hénon–Heiles model

For our test program we will use the leapfrog method[1]. We divide the integration interval $[0, T_{\text{max}}]$ in n intervals, i.e., $dt = T_{\text{max}}/n$, and consider the evaluation of $\{p_i^{(n)}, q_i^{(n)}\}_{i=1}^2$, for $n = 0, \ldots, n$, on the points $t_i = idt$.
The method is then obtained as:

$$\begin{cases} p_1^{(n+1/2)} = p_1^{(n-1)} + \frac{\omega_1}{2} q_1^{(n-1)} dt, \\ p_2^{(n+1/2)} = p_2^{(n-1)} + \frac{\omega_2}{2} q_2^{(n-1)} dt, \\ q_1^{(n)} = q_1^{(n-1)} - (\omega_1 p_1^{(n+1/2)} + 2 p_1^{(n+1/2)} p_2^{(n+1/2)}) dt, \\ q_2^{(n)} = q_2^{(n-1)} - (\omega_2 p_2^{(n+1/2)} + [p_1^{(n+1/2)}]^2 + [p_2^{(n+1/2)}]^2) dt, \\ p_1^{(n)} = p_1^{(n+1/2)} + \frac{\omega_1}{2} q_1^{(n)} dt, \\ p_2^{(n)} = p_2^{(n+1/2)} + \frac{\omega_2}{2} q_2^{(n)} dt, \end{cases}$$

This is a symplectic method of order 1!

[1] See, e.g., J. Laskar and P. Robutel, High order symplectic integrators for perturbed Hamiltonian systems, Celestial Mech. Dynam. Astronom. **80** (2001), no. 1, 39–62.

Numerical integration of the Hénon–Heiles model

- Observe that the first and the last block are the same computation on different inputs, therefore we can perform it as:

```
void kin_flow(double p[2], double q[2],
 double omega[2], double dt){
  p[0] += omega[0] * q[0] * 0.5 * dt;
  p[1] += omega[1] * q[1] * 0.5 * dt;}
```

- then the central block for updating the q_i can be coded as

```
void pot_flow(double p[2], double q[2], double omega[2],
  double dt){
 q[0] -= ( omega[0] * p[0] + 2 * p[0] * p[1] ) * dt;
 q[1] -= ( omega[1] * p[1] + pow(p[0],2) - pow(p[1],2) ) * dt;}
```

- A complete step of the integration is then given by

```
void leapfrog(double p[2], double q[2], double omega[2],
 double dt){
kin_flow(p, q, omega, dt);
pot_flow(p, q, omega, dt);
kin_flow(p, q, omega, dt);}
```

Numerical integration of the Hénon–Heiles model

Then to integrate our Hamiltonian system we repeat the routine performing one step n times (`int numberofsteps` times)

```
for(int i=1; i<=numberofsteps; i++) {
 leapfrog(p, q, omega, dt);
 energy_at_time_t = compute_energy(p, q, omega);
 delta_energy[i] = energy_at_time_t - initialenergy;
}
```

- we have coded the algorithm in a way that does not store the values of $\{p_i^{(j)}, q_i^{(j)}\}_{i=1,2}^{j=1,\ldots,n}$, therefore we need to compute the energy at each integration step,
- the *energy* is nothing more than the value of the Hamiltonian on the current values of p_i and q_j, i.e.,

```
double compute_energy(double p[2], double q[2], double omega[2]){
int i; double ris = 0;
for(i=0; i<2; i++)
 ris += omega[i] * ( pow(p[i],2) + pow(q[i],2) ) / 2;
ris += p[1] * ( pow(p[0],2) - pow(p[1],2) / 3 );
return ris;
}
```

Numerical integration of the Hénon–Heiles model

The last part of the sequential algorithm we need to precise regards the initial conditions and the initial energy:

```
delta_energy[0] = initialenergy - omega[1]*(pow(p0[1],2)
 + pow(q0[1],2))/2.0 + pow(p0[1],3)/3.0;
q0[0] = sqrt(2.0*delta_energy[0]/omega[0]);
p0[0] = 0;
for(int i=0; i<2; i++) {
 p[i] = p0[i];
 q[i] = q0[i];
}
```

in which we are assuming to know:

- the values of `p0[1]`, and `q0[1]`,
- the initial value of the energy `initialenergy`,
- to complete the information we need as input also the values `omega[0]`, `omega[1]`, the maximum time of integration `tmax`, and either the value of `dt` or the number of intervals `n`.

Numerical integration of the Hénon–Heiles model

Let us look again at our list:

1. Processor 0 reads the list of parameters in input for each instance of the problem,
2. Processor 0 *scatters* such data to every other process in the pool,
3. Each processor from `0,...,size-1` executes its own integration,
4. Processor 0 *gathers* the results from all the processors,
5. Processor 0 writes the result on a file in a format that can be used to plot the `size` different solutions.

we have solved how to perform the integration at step 3, and we know what parameters in input we need, namely:

- `omega[0]`, `omega[1]`, `initialenergy`, `p0[1]`, `q0[1]`, `tmax`, `dt`
- we want to read the the input for all the processors on `rank 0`, therefore we decide to produce our input file in the following form:
  - the first line will tell us how many sets of parameters we have,
  - the following lines, in groups by seven, will give us all the various parameters in the order above, one for each line.

Numerical integration of the Hénon–Heiles model

Input file:

```
henonheiles.inp

3
1
0.61803398874989482
0.03
0.35
0.
1.
0.1
1
0.61803398874989482
0.03
0.35
0.
1.
0.1
1
0.61803398874989482
0.03
0.35
0.
1.
0.01
```

```
if(myrank == 0){
 FILE *ifp; char line[300];
 int numberoftests, i=0;
 ifp = fopen("henonheiles.inp","r");
 if (ifp == NULL) {
  fprintf(stderr,
   "Can't open input file henonheiles.inp!\n");
  MPI_Finalize(); return 1;
 }
 fgets(line, 300, ifp);
 sscanf(line,"%d",&numberoftests);
 readdata = (double *)
   malloc( sizeof(double)*7*numberoftests );
 while(fgets(line, 300, ifp)!=NULL) {
 readdata[i]=atof(line);
 i++;
 }
 fclose(ifp);
}
```

Numerical integration of the Hénon–Heiles model

At this point the array `readdata` contains all the input value, we need now to send to each processor in `MPI_COMM_RANK` for `0,...,size-1` the relative seven values:

```
double input[7];
MPI_Scatter(readdata,7,MPI_DOUBLE,input,7,MPI_DOUBLE,0,
  MPI_COMM_WORLD);
```

- this function has to be called by each processor,
- after the completion of the communication processor j has the jth group of 7 inputs (`omega[0]`, `omega[1]`, `initialenergy`, `p0[1]`, `q0[1]`, `tmax`, `dt`),

Numerical integration of the Hénon–Heiles model

At this point the array `readdata` contains all the input value, we need now to send to each processor in `MPI_COMM_RANK` for `0,...,size-1` the relative seven values:

```
double input[7];
MPI_Scatter(readdata,7,MPI_DOUBLE,input,7,MPI_DOUBLE,0,
  MPI_COMM_WORLD);
```

- this function has to be called by each processor,
- after the completion of the communication processor j has the jth group of 7 inputs (`omega[0]`, `omega[1]`, `initialenergy`, `p0[1]`, `q0[1]`, `tmax`, `dt`),

On each processor we can populate the (local) inputs with the values:

```
double omega[2],initialenergy,p0[2],q0[2],tmax,dt;
omega[0] = input[0]; omega[1] = input[1];
initialenergy = input[2]; p0[1] = input[3]; q0[1] = input[4];
tmax = input[5]; dt = input[6];
```

and compute the number of steps by doing:

```
int numberofsteps = tmax/dt;
```

Numerical integration of the Hénon–Heiles model

We need now to gather the array `delta_energy` containing the error on the energy from each processor.

- Note: the size of `delta_energy` may be different on each processor!

Numerical integration of the Hénon–Heiles model

We need now to gather the array `delta_energy` containing the error on the energy from each processor.

- Note: the size of `delta_energy` may be different on each processor!
- we can use the `MPI_Gatherv` function to accommodate this issue, but this requires knowing the length of each array!

Numerical integration of the Hénon–Heiles model

We need now to gather the array `delta_energy` containing the error on the energy from each processor.

- Note: the size of `delta_energy` may be different on each processor!
- we can use the `MPI_Gatherv` function to accommodate this issue, but this requires knowing the length of each array!
- we can first share this information between all the processors by using an `MPI_Allgather` that will store this information on the array

  ```
  int *intervals_for_process;
  ```

  by calling:

  ```
  intervals_for_process=(int *) malloc(sizeof(int)*size);
  MPI_Allgather(&numberofsteps,1,MPI_INT,
    intervals_for_process,1,MPI_INT,MPI_COMM_WORLD);
  ```

  after this call every processor will have the information regarding the number of nodes computed by each processor.

Numerical integration of the Hénon–Heiles model

We put together the ingredients needed for the `MPI_Gatherv`

```
if (myrank == 0){
 stride = (int *)
   malloc(sizeof(int)*size);
 for(int i = 0; i < size; i++){
  grandtotal +=
    intervals_for_process[i];
 if(i==0){
  stride[i] = 0;
 }else{
  stride[i] = stride[i-1]
   +intervals_for_process[i-1];
 }
}
every_delta_energy = (double *)
 malloc(sizeof(double)*
  (grandtotal+size));
}
```

- Observe that we have allocated the memory for the receive only on the root processor, it would have been a waste of resources allocating it everywhere!
- We will need to put the received vectors with the correct spacing in the receiving buffer (`every_delta_energy`), thus we need to define the opportune stride (`recevbuf + stride[i]*extent[recevtype]`)
- all this information are needed only on the `root` process, so it is safe to define their value only on this rank.

Numerical integration of the Hénon–Heiles model

We have all the pieces needed for the `MPI_Gatherv`:

```
MPI_Gatherv(delta_energy,numberofsteps,MPI_DOUBLE,
 every_delta_energy, intervals_for_process,stride,
 MPI_DOUBLE,0,MPI_COMM_WORLD);
```

- each process will contribute with its own local `delta_energy` (that is an array of `numberofsteps` `doubles`),
- everything will end up in the `every_delta_energy`, it is going to receive from jth processor `intervals_for_process[j]` `doubles`,
- the received values will be placed contiguously as dictated by the `stride` array.

Numerical integration of the Hénon–Heiles model

We have all the pieces needed for the `MPI_Gatherv`:

```
MPI_Gatherv(delta_energy,numberofsteps,MPI_DOUBLE,
 every_delta_energy, intervals_for_process,stride,
 MPI_DOUBLE,0,MPI_COMM_WORLD);
```

- each process will contribute with its own local `delta_energy` (that is an array of `numberofsteps` `doubles`),
- everything will end up in the `every_delta_energy`, it is going to receive from jth processor `intervals_for_process[j]` `doubles`,
- the received values will be placed contiguously as dictated by the `stride` array.

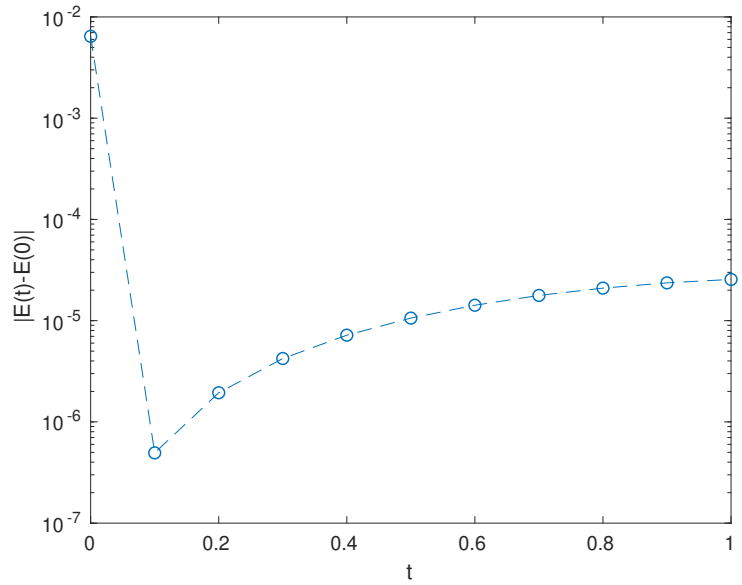
To conclude our code, we need only to print every result in its own file.

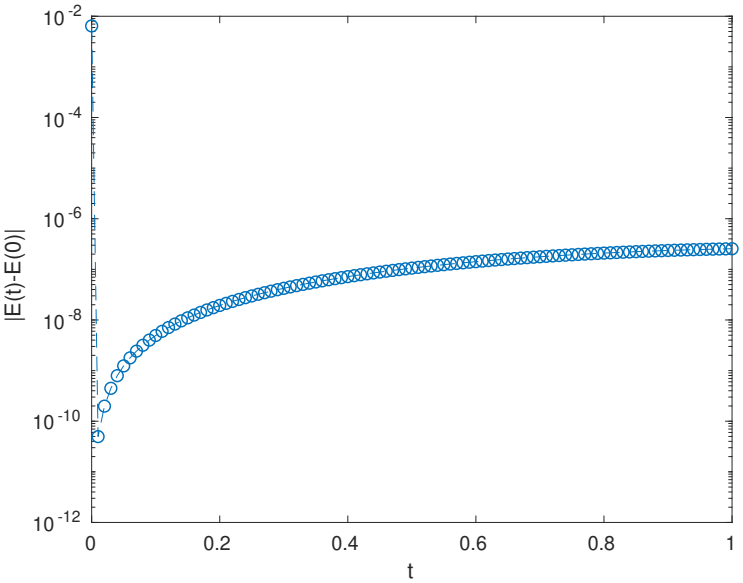
Numerical integration of the Hénon–Heiles model

The writing is completely mechanical at this point:

```
if (myrank == 0){
 FILE *ofp;
 int glob_counter = 0;
 char filename[200];
 for(int i = 0; i < size;i++){
  sprintf(filename,"energy_process_%d.dat",i);
  ofp = fopen(filename,"w+");
  for(int j=0; j < intervals_for_process[i]; j++){
   fprintf(ofp, "     %le      %le\n",
   j * readdata[i*7+6],
   every_delta_energy[glob_counter]);
   glob_counter++;
  }
  fclose(ofp);
 }
 free(readdata); free(every_delta_energy);
}
free(intervals_for_process);
```

Numerical integration of the Hénon–Heiles model





As usual, this code is not perfect, and there are several possible improvement

- Written in this way we need to pay attention to the ratio between the number of processors and parameter sets,
- We could use a reduce operation (`MPI_Reduce`) to get (instead of all the array of the errors) a norm or some other meaningful statistics,
- We may modify it to get instead the value of the positions p_i, or the moments q_i,
- ...

“They’re moving in herds. They do move in herds.”

"They're moving in herds. They do move in herds."

There exists many libraries with parallel capabilities for scientific computing.

LA Parallel Basic Linear Algebra Subprograms

- `ScaLAPACK`: www.netlib.org/scalapack
- `PSBLAS`: github.com/sfilippone/psblas3
- `PETSc`: www.mcs.anl.gov/petsc

Solvers Solvers for Linear and Nonlinear equations

- `Hypre`: computing.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods
- `AMG4PSBLAS`: github.com/sfilippone/amg4psblas
- `MUMPS`: mumps.enseeiht.fr/
- `Trilinos`: trilinos.github.io/
- `SUNDIALS`: computing.llnl.gov/projects/sundials

Repos

- software.llnl.gov
- developer.nvidia.com/gpu-accelerated-libraries
- en.wikipedia.org/wiki/List_of_numerical_libraries

"They're moving in herds. They do move in herds."

There exists many libraries with parallel capabilities for scientific computing.

LA Parallel Basic Linear Algebra Subprograms

- `ScaLAPACK`: www.netlib.org/scalapack
- `PSBLAS`: github.com/sfilippone/psblas3
- `PETSc`: www.mcs.anl.gov/petsc

Solvers Solvers for Linear and Nonlinear equations

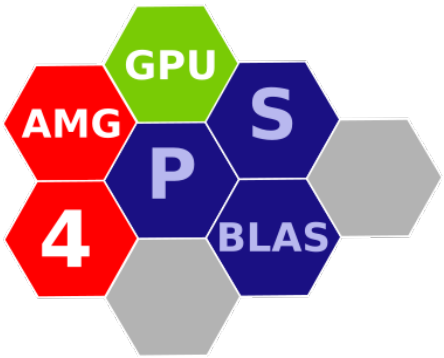
- `Hypre`: computing.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods
- `AMG4PSBLAS`: github.com/sfilippone/amg4psblas
- `MUMPS`: mumps.enseeiht.fr/
- `Trilinos`: trilinos.github.io/
- `SUNDIALS`: computing.llnl.gov/projects/sundials

Repos

- software.llnl.gov
- developer.nvidia.com/gpu-accelerated-libraries
- en.wikipedia.org/wiki/List_of_numerical_libraries

“They’re moving in herds. They do move in herds.”

Shameless advertisement



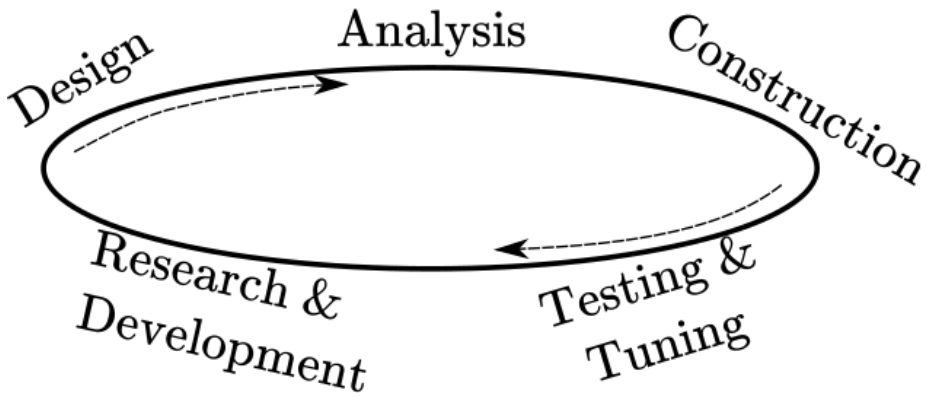
I am a collaborator on the `PSCToolkit` project. . . so give it a try!
If you have either to solve large linear systems in parallel, or have programs based on BLAS, then these tools can be useful.

- For an introduction on how to use the libraries there are several recorded webinars: psctoolkit.github.io/talks/
- The website psctoolkit.github.io contains up-to-date information on software release, webinars, and new developments.

Conclusions

What do we do now?

- Find a problem you want to solve *computationally* and in *parallel*,
- Look for an MPI-enabled library implementing the basic routine you may need,
- Enter the never-ending cycle of scientific software development:



Things to read:

- Rouson, D., Xia, J., & Xu, X. (2011). Scientific software design: the object-oriented way. Cambridge University Press.
- Leiserson, Charles E., et al. There's plenty of room at the Top: What will drive computer performance after Moore's law?. Science, 2020, 368.6495.