Message Passing Interface (MPI)

Η συνάρτηση ομαδοποίησης MPI_Pack

Η συνάρτηση απο-ομαδοποίησης MPI_Unpack

Παραγώμενοι τύποι δεδομένων (MPI Derived Data Types)

- Με τις συναρτήσεις που έχουμε δει ως τώρα, μπορούμε να ομαδοποιήσουμε σε ένα μήνυμα μόνο πολλαπλά δεδομένα ιδίου τύπου, π.χ. πολλούς ακεραίους σε έναν πίνακα ακεραίων που μπορούμε να τον στείλουμε στη συνέχεια με ένα μόνο μήνυμα, κ.ο.κ.
- Αντίθετα, δεν μπορούμε να ομαδοποιήσουμε και να στείλουμε με ένα μόνο μήνυμα πολλαπλά δεδομένα διαφορετικών μεταξύ τους τύπων ή δεδομένα τα οποία δεν είναι συνεχόμενα στη μνήμη.

Βασικές Τεχνικές

- **Χρήση Πολλαπλών Μηνυμάτων**
- **Αντιγραφή δεδομένων σε Buffer**
- **Χρήση των συναρτήσεων MPI_Pack & MPI_Unpack**

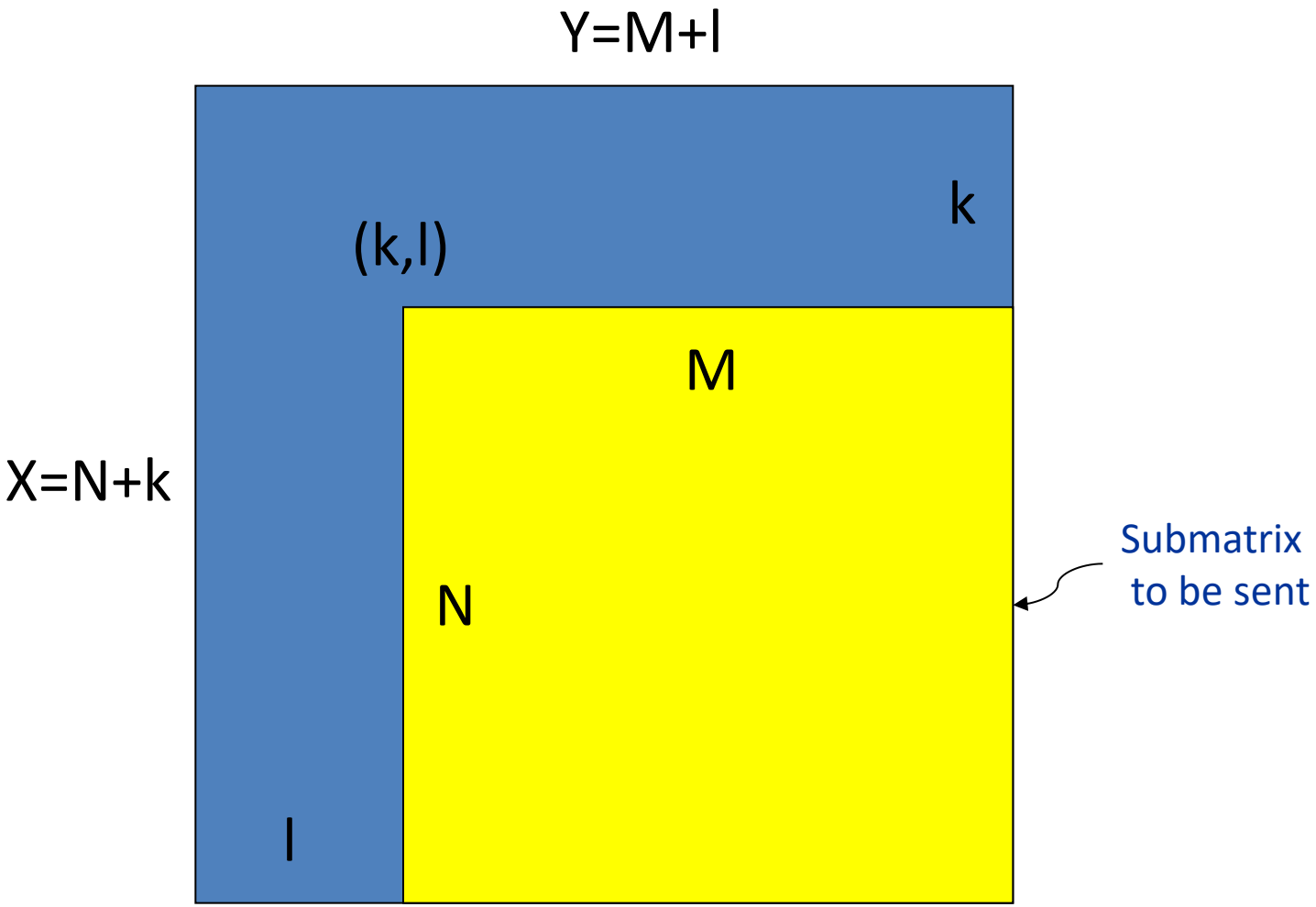
Χρήση Πολλαπλών Μηνυμάτων

Παράδειγμα:

- We have a large matrix stored in a two-dimensional array and you wish to send a rectangular submatrix to another processor.
- Since arrays are stored with the elements of rows contiguous in memory, you would send **N** messages containing the **M** elements in a row of the submatrix. If the array is declared to be ***X*** **x** ***Y*** and you are sending the **N** by **M** portion whose upper left corner is at position **(k,l)**, this might look like

```
for (i=0; i<n; ++i) {
   MPI_Send(&a[k+i][l], m, MPI_DOUBLE,  dest, tag,
   MPI_COMM_WORLD);
}
```

Χρήση Πολλαπλών Μηνυμάτων



Χρήση Πολλαπλών Μηνυμάτων

- **The values of N, M, K, or L can be sent in a separate message.**

  Advantage:

  Simlpicity.

  Disadvantage:

  Overhead. A fixed overhead is associated with the sending and receiving of a message, whatever long or short it is. If you replace one long message with multiple short messages, you slow down your program by greatly increasing your overhead.

- **This technique may be used if you are working in a portion of your program that will be executed infrequently and the number of additional messages is small, the total amount of added overhead may be small enough to ignore.**

Αντιγραφή δεδομένων σε Buffer

- **Copy the data into a contiguous buffer. For example,**

2d-array → 1d array

```
p = &buffer;
for (i=0; i<n; ++i) {
    for(j=0; j<m; ++j) {
      *(p++) = a[i][j];
    }
}
MPI_Send(p, n*m, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD)
```

- **This approach eliminates the excessive messages of the previous approach, at the cost of extra memory for the buffer and extra CPU time to perform the copy into the buffer.**
- **The obvious limitation of this approach is that it still handles only one type of data at a time.**

Χρήση των συναρτήσεων MPI_Pack & MPI_Unpack

MPI_PACK :

Used to combine heterogeneous or discontiguous data into a contiguous buffer on a piecemeal basis.

- **The MPI_PACK routine allows you to fill a buffer "the right way". You call MPI_PACK with arguments that describe the buffer you are filling and with most of the arguments you would have provided to MPI_SEND in our simplest approach.**

- **MPI_PACK copies your data into the buffer. After all the data you want to send have been placed in the buffer by MPI_PACK, you can send the buffer (giving its type as MPI_PACKED).**

Η συνάρτηση ομαδοποίησης MPI_Pack

- Παρέχει τη δυνατότητα αποστολής και λήψης δεδομένων μη συνεχόμενων ή διαφορετικών τύπων τα οποία ομαδοποιεί και τα τοποθετεί σε συμπτυγμένα (packed) μηνύματα.
- Με τον μηχανισμό αυτό μπορούμε να στείλουμε τον τύπο των δεδομένων, ακολουθούμενη από τα ίδια τα δεδομένα.
- Ο τύπος δεδομένων καθώς και το ίδιο το μήνυμα βρίσκονται στην ίδια ενδιάμεση μνήμη κατά την αποστολή τους και αποθηκεύονται στην ίδια ενδιάμεση μνήμη μετά τη λήψη τους.

Η συνάρτηση ομαδοποίησης MPI_Pack

- **Χρήση της MPI_Pack στο προηγούμενο παράδειγμα του υποπίνακα**

```
count = 0
for (i=0; i<n; ++i) {
  MPI_Pack(&a[k+i][l], m, MPI_DOUBLE,
  buffer, bufsize, &count, MPI_COMM_WORLD);
}
MPI_Send(buffer, count, MPI_PACKED, dest,
  tag, MPI_COMM_WORLD);
```

- **COUNT is initially set to zero to indicate you are starting the construction of a new message and have an empty buffer.**
- **The successive calls to MPI_PACK update COUNT to reflect the data that have been added to the buffer. The final value of COUNT is then used in the call to MPI_SEND as the amount of data to send.**

Η συνάρτηση ομαδοποίησης MPI_Pack

int MPI_Pack (void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)

- **inbuf:** Ενδιάμεση μνήμη που περιέχει το αρχικό μήνυμα.
- **incount:** Αριθμός στοιχείων που περιέχονται στην ενδιάμεση μνήμη (δηλαδή το μέγεθος του αρχικού μηνύματος).
- **datatype:** Ο τύπος των δεδομένων που περιέχονται στην ενδιάμεση μνήμη.
- **outbuf:** Ενδιάμεση μνήμη όπου θα αποθηκευθεί το τελικό μήνυμα.
- **outsize:** Μέγεθος της ενδιάμεσης μνήμης του τελικού μηνύματος.
- **position:** Η τρέχουσα θέση στην ενδιάμεση μνήμη, σε bytes.
- **comm:** Ο communicator για το συμπτυγμένο μήνυμα.

Η συνάρτηση ομαδοποίησης MPI_Pack

- **Η συνάρτηση συμπτύσσει το μήνυμα που καθορίζεται από τις τιμές των inbuf, incount στην ενδιάμεση μνήμη που καθορίζεται από τις outbuf και outsize. Η inbuf μπορεί να έχει οποιοδήποτε τύπο δεδομένων επιτρέπει η MPI_Send. Η outbuf είναι ενδιάμεση μνήμη που κατέχει συνεχόμενες διευθύνσεις και έχει μέγεθος outsize bytes, αρχίζοντας από τη διεύθυνση outbuf.**
- **Η position δείχνει στην πρώτη θέση του outbuf από την οποία θα αρχίσει η αποθήκευση των συμπτυγμένων δεδομένων. Η τιμή της αυξάνει όσο είναι το μέγεθος του συμπτυγμένου μηνύματος έτσι ώστε, να μπορεί να χρησιμοποιηθεί και σε επόμενες κλήσεις της MPI_Pack.**
- **Το συμπτυγμένο μήνυμα μεταδίδεται με μια συνηθισμένη συνάρτηση αποστολής μηνύματος (π.χ. MPI_Send).**

Η συνάρτηση απο-ομαδοποίησης MPI_Unpack

int MPI_Unpack (void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)

- **inbuf:** Η διεύθυνση αρχής της ενδιάμεσης μνήμης που περιέχει το συμπιεσμένο μήνυμα.
- **insize:** Το μέγεθος της ενδιάμεσης μνήμης σε bytes.
- **position:** Η τρέχουσα θέση στην ενδιάμεση μνήμη, σε bytes.
- **outbuf:** Η ενδιάμεση μνήμη που θα αποθηκεύσει τελικά το αποσυμπιεσμένο μήνυμα.
- **outcount:** Ο αριθμός των στοιχείων που θα αποσυμπιεστούν.
- **datatype:** Ο τύπος δεδομένων της outbuf.
- **comm:** Ο communicator για το συμπιεσμένο μήνυμα.

Η συνάρτηση απο-ομαδοποίησης MPI_Unpack

- Η συνάρτηση MPI_Unpack, κάνει την αντίστροφη εργασία από την MPI_Pack. Δηλαδή, αποσυμπτύσσει ένα μήνυμα έτσι ώστε η διεργασία-παραλήπτης να μπορεί να χρησιμοποιήσει τα δεδομένα που περιέχονται σ’ αυτό.
- Η συνάρτηση αποσυμπτύσσει το μήνυμα που βρίσκεται στην inbuf, στην ενδιάμεση μνήμη που καθορίζεται από τις outbuf, outcount και datatype. Η outbuf μπορεί να είναι οποιουδήποτε τύπου επιτρέπει η MPI_Recv. Η inbuf είναι μια ενδιάμεση μνήμη που καταλαμβάνει συνεχόμενες διευθύνσεις και έχει μέγεθος insize bytes, αρχίζοντας από τη διεύθυνση inbuf.

Παράδειγμα (pack.c)

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char** argv)
{
int my_rank, p, k, size;
int root, position, namelen;
char buffer[50];
int matrixA[100];
int loc_num, loc_matrix[100];
float b, loc_res[100];
float final_res[100];
char proc_name[MPI_MAX_PROCESSOR_NAME];

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Get_processor_name(proc_name, &namelen);
```

```
if (my_rank == 0)
 {
  printf("YPOLOGISMOS THS PARASTASHS b * A \n\n");
  printf("DOSE THN TIMH TOY B:");
  scanf("%f", &b);
  printf("DOSE TO MHKOS TOY PINAKA A:");
  scanf("%d", &size);
  printf("DOSE TA STOIXEIA TOY PINAKA A MHKOYS %d : ", size);
  for (k=0; k<size; k++)
          scanf("%d", &matrixA[k]);
  position = 0;
  MPI_Pack (&size, 1, MPI_INT, buffer, 50, &position,
      MPI_COMM_WORLD);
  MPI_Pack (&b, 1, MPI_FLOAT, buffer, 50, &position,
      MPI_COMM_WORLD);   }
```

```
root = 0;
MPI_Bcast(buffer, 50, MPI_PACKED, root, MPI_COMM_WORLD);
position = 0;
MPI_Unpack(buffer, 50, &position, &size, 1, MPI_INT,
        MPI_COMM_WORLD);
MPI_Unpack(buffer, 50, &position, &b, 1, MPI_FLOAT,
        MPI_COMM_WORLD);

loc_num = size/p; root = 0;
MPI_Scatter(matrixA, loc_num, MPI_INT, loc_matrix,
         loc_num, MPI_INT, root, MPI_COMM_WORLD);

for (k=0; k<loc_num; k++)
   loc_res[k] = b*loc_matrix[k];
```

```
printf("\n Process %d on %s : local results are :  ",
      my_rank, proc_name);
for (k=0; k<loc_num; k++)
  printf("%f ", loc_res[k]);
printf("\n\n");
root = 0;
MPI_Gather(loc_res, loc_num, MPI_INT, final_res,
loc_num,
         MPI_FLOAT, root, MPI_COMM_WORLD);

if (my_rank == 0)
 {
   printf("\n TELIKO APOTELESMA  %f *  A =\n", b);
   for (k=0; k<size; k++) printf("%f ", final_res[k]);
   printf("\n\n");
  }
MPI_Finalize();  }
```

Datatypes

- MPI datatypes have two main purposes:
  - Heterogeneity --- parallel programs between different processors
  - Noncontiguous data --- structures, vectors with non-unit stride, etc.
- Basic/primitive datatypes, corresponding to the underlying language, are predefined
- The user can construct new datatypes at run time; these are called *derived* datatypes
- Avoids explicit packing/unpacking of data by user
- A derived datatype can be used in any communication operation instead of primitive datatype
  - **MPI_Send (buf, 1, mytype, …..)**
  - **MPI_Recv (buf, 1, mytype, …..)**

Derived Data Types

MPI_Type_contiguous (count,oldtype,&newtype) Produces a new data type by making count copies of an existing data type.
MPI_Type_vector (count,blocklength,stride,oldtype,&newtype) Similar to contiguous, but allows for regular gaps (stride) in the displacements.
MPI_Type_indexed (count,blocklens[],offsetsp[],oldtype,&newtype) An array of displacements of the input data type is provided as the map for the new data type.
MPI_Type_struct (count,blocklens[],offsets[],oldtypes[],&newtype) The new data type is formed according to completely defined map of the component data types.
MPI_Type_extent (datatype,&extent) Returns the size in bytes of the specified data type.
MPI_Type_commit (&datatype) Commits new datatype to the system.
MPI_Type_free (&datatype) Deallocates the specified datatype object.

Defining MPI Derived Types

- **Basic Steps**
  - **Create new derived datatypes: define shape, obtain new type handle**
  - **Commit new datatype: pass to system**
  - **Free type: if no longer needed**
- **Before a derived datatype can be used in a communication, the program must create it. This is done in two stages.**
  - ***Construct the datatype (Step 1).***

    New datatype definitions are built up from existing datatypes (either derived or basic) using a call, or a recursive series of calls, to the following routines:

    MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_HVECTOR, MPI_TYPE_INDEXED MPI_TYPE_HINDEXED, MPI_TYPE_STRUCT.
  - ***Commit the datatype (Step 2).***

    The new datatype is "committed" with a call to MPI_TYPE_COMMIT. It can then be used in any number of communications.
- **Finally *(Step 3),* there is a complementary routine to MPI_TYPE_COMMIT, namely MPI_TYPE_FREE, which marks a datatype for de-allocation.**

  MPI_TYPE_FREE (datatype)
- **Any datatypes derived from datatype are unaffected when it is freed, as are any communications which are using the datatype at the time of freeing. datatype is returned as MPI_DATATYPE_NULL.**

MPI Subroutines for Derived Datatypes

- **Return a new datatype that represents the concatenation of count instances of old datatype**

```
int MPI_Type_contiguous(count,oldtype,newtype);
```

- **Return a new datatype that represents equally spaced blocks. The spacing between the start of each block is given in:**
  - **unit of extent from oldtype(count)**

```
int MPI_Type_vector(count,blocklength,stride,
                    oldtype,newtype);
```

  - bytes

```
int MPI_Type_hvector(count,blocklength,
                     stride,oldtype,newtype);
```

- **Extension of vector: varying block length and strides. Return a new datatype that represents count block. Each block is defined by an entry of *array_of_bocklength* and *array_of_displacement*. Displacement between successive blocks are expressed in:**
  - **unit of oldtype.**

```
intMPI_Type_indexed(count,array_of_blocklength,
                    array_of_displacement, oldtype,newtype);
```

  - **bytes**

```
int MPI_Type_hindexed(count,array_of_blocklength,
                      array_of_displacement, oldtype,newtype);
```

Contiguous Derived Data Type

Create a type representing a row of an array. Distribute a different row to each proces

```
#include "mpi.h“
 #include <stdio.h>
#define SIZE 4
main(int argc, char *argv[]) {
int numtasks, rank, source=0, dest, tag=1, i;
float a[SIZE][SIZE] = { 1.0, 2.0, 3.0, 4.0,  5.0, 6.0, 7.0, 8.0,
                        9.0, 10.0, 11.0, 12.0,  13.0, 14.0, 15.0, 16.0};
float b[SIZE];
MPI_Status stat;
MPI_Datatype rowtype;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank); MPI_Comm_size(MPI_COMM_WORL
    &numtasks);
MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);
```

```
if (numtasks == SIZE) {
    if (rank == 0) {
            for (i=0; i<numtasks; i++)
               MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
    }
    MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD,
                           &stat);
    printf("rank= %d b= %3.1f %3.1f %3.1f %3.1f\n",
                                          rank,b[0],b[1],b[2],b[3]);
   }
else
    printf("Must specify %d processors. Terminating.\n",SIZE);
MPI_Type_free(&rowtype);
MPI_Finalize(); }
```

Sample program output:

```
rank= 0  b= 1.0  2.0  3.0  4.0
rank= 1  b= 5.0  6.0  7.0  8.0
rank= 2  b= 9.0   10.0  11.0  12.0
rank= 3  b= 13.0 14.0  15.0  16.0
```

Η συνάρτηση MPI_Type_vector

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

- **count** number of blocks (nonnegative integer)
- **blocklength** number of elements in each block (nonnegative integer)
- **stride** number of elements between start of each block (integer)
- **oldtype** old datatype (handle)
- **newtype** new datatype (handle)

Example: Vector Derived Data Type

Create a type representing a column of an array. Distribute a different column to each process.

```
#include "mpi.h“
 #include <stdio.h>
#define SIZE 4
main(int argc, char *argv[]) {
int numtasks, rank, source=0, dest, tag=1, i;
float a[SIZE][SIZE] = { 1.0, 2.0, 3.0, 4.0,  5.0, 6.0, 7.0, 8.0,
                        9.0, 10.0, 11.0, 12.0,  13.0, 14.0, 15.0, 16.0};
float b[SIZE];
MPI_Status stat;
MPI_Datatype  columntype;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Type_vector(SIZE, SIZE, SIZE, MPI_FLOAT, &columntype);
MPI_Type_commit(&columntype);
```

```
if (numtasks == SIZE) {
    if (rank == 0) {
            for (i=0; i<numtasks; i++)
               MPI_Send(&a[0][i], 1, columntype, i, tag,
    MPI_COMM_WORLD);
    }
    MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD,
                              &stat);
    printf("rank= %d b= %3.1f %3.1f %3.1f %3.1f\n",
                                              rank,b[0],b[1],b[2],b[3]);
   }
else
    printf("Must specify %d processors. Terminating.\n",SIZE);
MPI_Type_free(&columntype);
MPI_Finalize(); }
```

Sample program output:

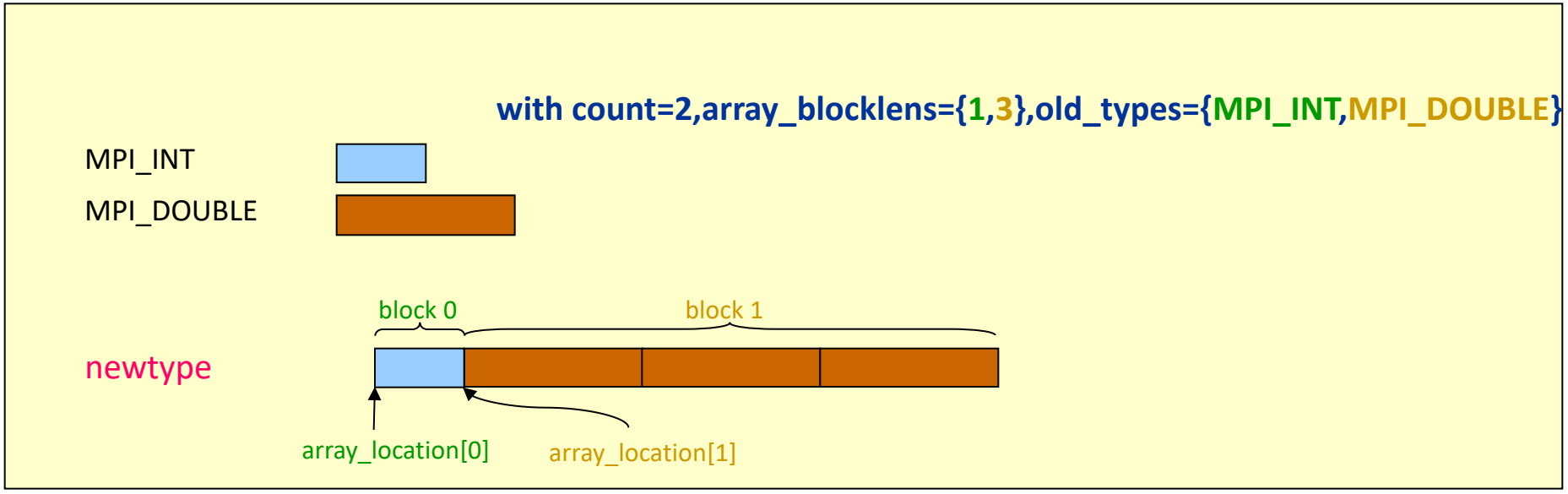
```
rank= 0   b= 1.0   5.0   9.0    13.0
rank= 1   b= 2.0   6.0   10.0  14.0
rank= 2   b= 3.0   7.0   11.0   15.0
rank= 3   b= 4.0   8.0   12.0   16.0
```

Structure Derived Data Type

`MPI_Type_struct:`

Βασική συνάρτηση δημιουργίας ενός παραγώμενου τύπου για μία δομή τύπου «struct» της C. Οι παράμετροι της συνάρτησης περιέχουν τον αριθμό και χαρακτηριστικά (αριθμός μεταβλητών (block length), offsets, τύποι δεδομένων) των μπλοκς διαφορετικών τύπων που είναι δηλωμένα στη συγκεκριμένη δομή τύπου 'struct' .

```
MPI_Type_struct(count,blocklens[],offsets[],old_types,newtype);
```



Η συνάρτηση MPI_Type_struct

int MPI_Type_struct (int count, int blocklens[], MPI_Aint offsets[], MPI_Datatype old_types[], MPI_Datatype *newtype)

- **count:** Ο # των διαδοχικών blocks διαφορετικού τύπου που έχει η δομή.
- **blocklens:** Πίνακας count θέσεων – μίας θέσης για κάθε μπλοκ – σε κάθε θέση του περιέχει τον # των μεταβλητών του αντίστοιχου μπλοκ.
- **offsets:** Πίνακας count θέσεων – μίας θέσης για κάθε μπλοκ – σε κάθε θέση του περιέχει την απόσταση του μπλοκ από την αρχή.
- **old_types:** Πίνακας count θέσεων – μίας θέσης για κάθε μπλοκ – σε κάθε θέση του περιέχει τον τύπο των μεταβλητών του μπλοκ.
- **newtype:** Παράμετρος εξόδου – το λεκτικό του νέου τύπου.

Παράδειγμα

- **Έστω η ακόλουθη δομή αναπαράστασης ενός κινούμενου σημείου στο χώρο :**

  struct { double x, y, z; float speed; int origin, type; }

- **Ζητείται να φτιάξουμε έναν αντίστοιχο παραγώμενο τύπο (π.χ. pointtype) και στη συνέχεια να εκτελέσουμε μία ενδεικτική αποστολή-παραλαβή ενός πίνακα από τέτοιες δομές (με χρήση του νέου τύπου του MPI που φτιάξαμε) σε όλες τις διεργασίες.**

```
#include <stdio.h>
#include "mpi.h"
#define N 20
 int main(argc,argv)
int argc;
char *argv[];
{
int numtasks, rank, source=0, dest, tag=1, i;
 typedef struct {
  double x, y, z;
  float speed;
  int   origin, type;
  }     Point;
 Point points[N];
MPI_Datatype pointtype, blocktypes[3];
int blockcounts[3];
 /* ο τύπος MPI_Aint χρησιμοποιείται εδώ για να υπάρχει πλήρης συμβατότητα
με τη σύνταξη και την επιστροφή παραμέτρων της MPI_Type_extent */
 MPI_Aint  blockoffsets[3], extent;
MPI_Status stat;
```

```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

/* Layout του πρώτου μπλοκ, δηλαδή των 3 MPI_DOUBLE – ενημερώνουμε τις πρώτες θέσεις των τριών πινάκων */
 blockoffsets[0] = 0;
blocktypes[0] = MPI_DOUBLE;
blockcounts[0] = 3;

/* Layout του δεύτερου μπλοκ, δηλαδή, του 1 MPI_FLOAT    - Για να καθορίσουμε το offset, χρησιμοποιήσουμε την MPI_Type_extent */
MPI_Type_extent(MPI_DOUBLE, &extent);
blockoffsets[1] = 3 * extent;
blocktypes[1] = MPI_FLOAT;
blockcounts[1] = 1;

/* Layout του τρίτου μπλοκ */
MPI_Type_extent(MPI_FLOAT, &extent);
blockoffsets[2] = blockoffsets[1] + extent;
blocktypes[2] = MPI_INT;
blockcounts[2] = 2;
```

```
MPI_Type_struct (3, blockcounts, blockoffsets, blocktypes,
&pointtype);
  MPI_Type_commit (&pointtype);

/* Στη συνέχεια, γεμίζουμε μία δομή πίνακα από structs τύπου
Point με τυχαίες τιμές */
 if (rank == 0) {
  for (i=0; i<N; i++) {
     points[i].x = i * 0.5;
     points[i].y = i * -0.5;
     points[i].z = i * 1.5;
     points[i].speed = 0.5;
     points[i].origin = i * 5;
     points[i].type = i % 2;
     }
}
```

```
/* αποστέλεται με ένα μόνο μήνυμα-εντολή ένας ολόκληρος πίνακας από N δομές αρχικού τύπου Point */
MPI_Bcast(points, N, pointtype, 0, MPI_COMM_WORLD);

if (rank == numtasks-1)
 for (i=5; i<10; i++)
  printf("rank=%d %f %f %f %f %d %d\n", rank, points[i].x,
points[i].y,points[i].z,points[i].speed,points[i].origin, points[i].type);

MPI_Finalize();
}
```