Manual of the `lip` library

Maciej Malawski, Katarzyna Zając

28 May, 2001

Contents

List of Figures

Chapter 1

Runtime Support Requirements and their Solutions

1.1 Cross–platform portability

The portability of `lip` library can be guaranteed by building it on top of the Message Passing Interface [40]. `lip` follows the suggestion of MPI standard documents that provide functionality implementable on all current super–computing architectures. It should enable `lip` to be easily ported to all these systems: from SMP machines to workstation clusters.

1.2 Multi–language support

MPI delivers the possibility of efficient interoperation between languages (Java, C++, Fortran 77, Fortran 90, C). Therefore, using `lip` in other languages is easily achieved. The library was written in the C programming language according to ISO C standard specification [24], but it should be also possible to use it in Fortran90 and Java.

Using `lip` in Fortran90 applications can be done by means of a *wrapper*, that is functions written in C that call the original `lip` routines. Wrapping functions are named according to the Fortran compiler naming conventions so from the user's point of view, they are regular Fortran subroutines.

Java wrappers for `lip` can be achieved by the ***Janet*** (Java to Native Code Interfaces) translator. For more information about the creation of Java wrappers see Chapter 4.

1.3 Efficiency of performance

Reusing the communication schedule

In order to minimize communication overhead `lip` supports creation of communication patterns that enable the transfer of all needed data together by means of one communication routine. The information which data should be send is obtained during the inspector phase while analyzing the array of indices. The communication schedule object can then be stored and used whenever needed (i.e. for multiple iterations of the irregular loop). This idea was also extended for schedules that manage OOC data and should be temporarily stored on disk.

1.3.1 Data caching

As OOC data do not fit into main memory, only a part of them can be processed at a time. Therefore, there is need to exchange data between memory and disk. `lip` supports an IO buffer mapping data structure that stores information about memory contents so there is no need to read data again. The detailed description of this behavior is described below. Additionally the IO latency is minimized by means of derived data types [51] that allow reading non-contiguous data from disk via only one MPI–IO routine. This is usually implemented using the data sieving technique [51]

1.3.2 Scalability

All algorithms implemented in `lip` assume nothing about the number of processors used and therefore provide scalability. This feature was particularly important in the partitioner design. The previous algorithms provided by CHAOS were suitable only for 2^n processors which was a significant limitation.

1.4 Support for interfaces built on the top of `lip`

The `lip` interface is primarily intended to allow the easy implementation of applications and libraries. It also provides functionality needed by parallelizing compilers built on top of it in the future. This issue should be consider as a future development.

Chapter 2

The Concept of `lip`

This chapter gives the main idea of support for both in– and out–of–core irregular problems.

2.1 The Case of In–core Data

This Section presents `lip` ideas for dealing with in–core data with regular and irregular distributions. Most of the concepts presented are extensions of those of the CHAOS library.

2.1.1 Support for irregular problems

Runtime support for parallel execution of in–core irregular code is based on the *Inspector/Executor* technique introduced in CHAOS [47]. A piece of pseudocode using this technique is shown in Fig. 2.1

```
MPI_Init( /* ... */ );
LIP_Setup( /* ... */ );

/* Generate index array describing
 * relationships between user data */

/* Inspector Phase (computes
 * optimized communication pattern) */
 LIP_Localize(schedule,... );

/* Executor Phase (performs
 * communication and computation) */

/* gather non-local irregular data */
 LIP_Gather(schedule,... );

/* perform computation on data */
for ( i = /* ... */ )
{
 k = edge[i];
 y[k] = f( x[k] );
}

/* scatterer non-local
 * irregular data (results) */
LIP_Scatter(schedule,... );


LIP_Exit( /* ... */ );
MPI_Finalize( /* ... */ );
```

Figure 2.1: The use of `lip` in a simple parallel C program with an in-core data array and an index array.

In the *Inspector Phase* data references (from indirect arrays) are examined and translated either into references to the node's local data array (global to local index translation) or into references to the node's temporary buffer (referred to as *ghost area*). The latter case is tightly connected with *schedule* generation which is a data structure describing how communication is to be performed efficiently to bring

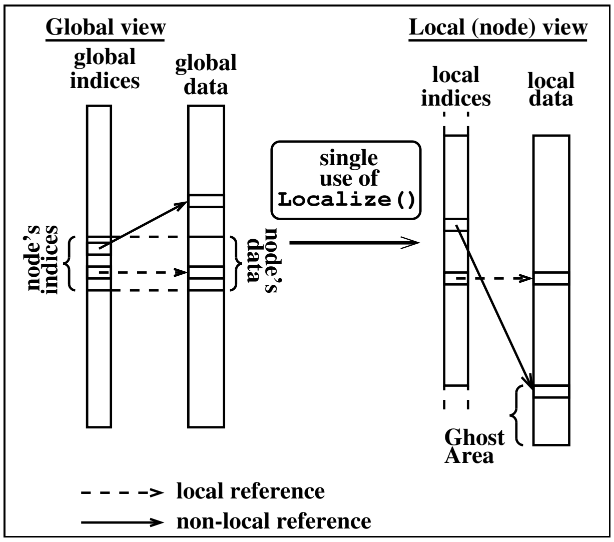

Figure 2.2: Irregular inspector in `lip`

desired data into *ghost area.* The *Inspector* is activated using LIP_localize() call and its idea is shown in Fig. 2.2.

The next step of parallelization is the *Executor Phase.* In this phase results of *Inspector phase* (including a *schedule*) are used to perform communication (that brings non-local data into *ghost area* and is done through a call to the `LIP_Gather()` routine) and then a computation is performed using translated indices and local data array with additional buffer space (*ghost area*). Following that, another communication has to be performed to scatter back results of the computation. The communication *schedule* is used again through a call to the `LIP_Scatter()` function.

2.1.2 Irregular data distribution

```
 MPI_Init( /* ... */ );
 LIP_Setup( /* ... */ );

 * Generate index array describing
 * relationships between user data */

/*Creation of data distribution*/
 LIP_create_hilbert_distribution_in_core(
                             datamap,...)

/*Creation of communication pattern used
  to move data */
 LIP_remap(bufmap,remapping_schedule,...)

/*Actual movement of data*/
 LIP_Gather(remapping_schedule,...)

/* Inspector Phase (computes
 * optimized communication pattern) */
 LIP_Localize(datamap,schedule,... );

 /* Executor Phase (performs
  * communication and computation) */

 /* gather non-local irregular data */
 LIP_Gather(schedule,... );

/* perform computation on data */
 for ( i = /* ... */ )
 {
  k = edge[i];
  y[k] = f( x[k] );
 }

/* scatterer non-local
 * irregular data (results) */
 LIP_Scatter(schedule,... );


 LIP_Exit( /* ... */ );
 MPI_Finalize( /* ... */ );
```

Figure 2.3: Use of `lip` in a simple parallel C program with irregularly distributed in-core data array and index array.

In order to achieve better load balancing and to minimize the communication phase, various techniques for creating data distribution were introduced.

At present, `lip` supports a coordinate–based partitioner using Hilbert's curve to determine what part of data should reside on the same node. The library supports both in– and out–of–core versions of the partitioner. Fig. 2.3 shows pseudocode that uses the partitioner.

The most significant difference between Fig. 2.1 and Fig. 2.3 is the use of the `LIP_create_hilbert_distribution_in_core()` routine. This function activates an in–core partitioner. The routine results in creation of a special object called *Datamap* that stores information about the new distribution. Additionally, a *schedule* object is created which is used by the communication function `LIP_Gather` to actually redistribute data.

2.2 The Case of OOC Data

The OOC extensions for irregular problems are the most significant feature of the `lip` library. This Section introduces the solution provided by `lip` for that purpose. Out–of–core problems can be divided taking into account the types of OOC arrays.

2.2.1 Dealing with OOC index–arrays and in–core data arrays

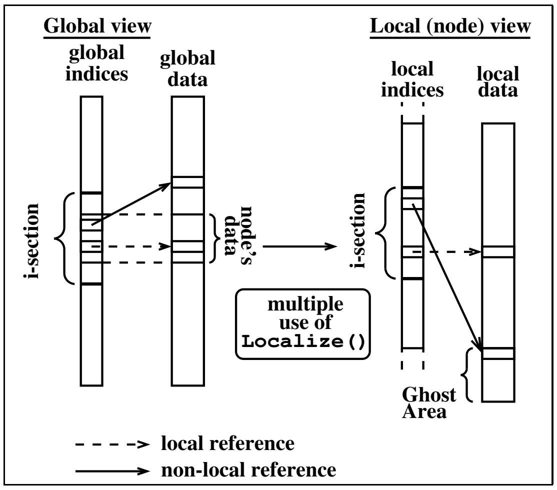

Figure 2.4: Irregular inspector in `lip` for OOC index arrays and in–core data arrays

The pseudocode in Fig. 2.5 shows the use of `lip` for OOC index arrays and in–core index arrays. Thanks to the introduction of the concept of *in-core section*, the *Inspector/Executor* approach remains almost unaltered. The whole index array is stored in a file and only a small part of it – *i-section* – is loaded into main memory to enable the previously described optimization. This is illustrated in Fig. 2.4. This step is repeated for all i–sections. In this way, the technique successfully used for irregular in-core problems has been generalized for irregular OOC problems with *in-core* data arrays and OOC index arrays.

```
MPI_Init( /* ... */ );
LIP_Setup( /* ... */ );

/* Generate index array describing
 * relationships between user data */

/* Perform irregular computation
 * on the data */
for ( /* ... */ )
{
/* read i-section's indices
 * from a file */

/* Inspector Phase (computes
 * optimized communication
 * pattern) */
 LIP_Localize(schedule,... );

  /* Executor Phase (performs
   * communication and computation) */

  /* gather non-local irregular data */
  LIP_Gather(schedule,... );

  /* perform computation on data */
  for ( i = /* ... */ )
  {
    k = edge[i];
    y[k] = f( x[k] );
  }

  /* scatterer non-local
   * irregular data (results) */
  LIP_Scatter(schedule,... );
}

LIP_Exit( /* ... */ );
MPI_Finalize( /* ... */ );
```

Figure 2.5: Use of the `lip` in a simple parallel C program with in-core data array and out-of-core index array.

2.2.2 Dealing with OOC index–arrays and data arrays

The concept presented above has to be further modified when not only index arrays but also data arrays become *out-of-core*. The previously described *Inspector phase* is insufficient because only part of the data array is stored in the memory and it is possible for the index array to refer to the data element which is currently stored on a disk. Detailed analysis leads to the conclusion that three types of non-local accesses may be observed in the index array:

- access to data which is stored in another's node memory,
- access to another node's data which is *not* present in its main memory but resides on disk,
- access to local node data which is not present in memory (it is stored on the disk).

The idea of a new parallelizing scheme is presented in Fig. 2.6. The respective pseudocode is shown in Fig. 2.7. Since reading every element from a disk just as it becomes needed would be very inefficient, the optimization has to be aimed not only at efficient communication but also at minimizing I/O accesses associated with *not-in-memory* data elements. This is done by examining and translating the references to data residing on a disk into references to the memory buffer. This is done by using the `LIP_ooc_localize()` routine after `LIP_localize()` is called. The translation process changes indices in the indirect array and in *communication schedule* so they point to the local memory buffer as shown in the Fig. 2.6. This process results in a data structure called *I/O buffer mapping*. It is updated upon examination of subsequent *i-sections* to record information about what data has to be moved between the disk and main memory as data that are in the memory are no longer needed for the processing of the next i-section. After the translation process is done, the I/O buffer mapping object may be used to create MPI

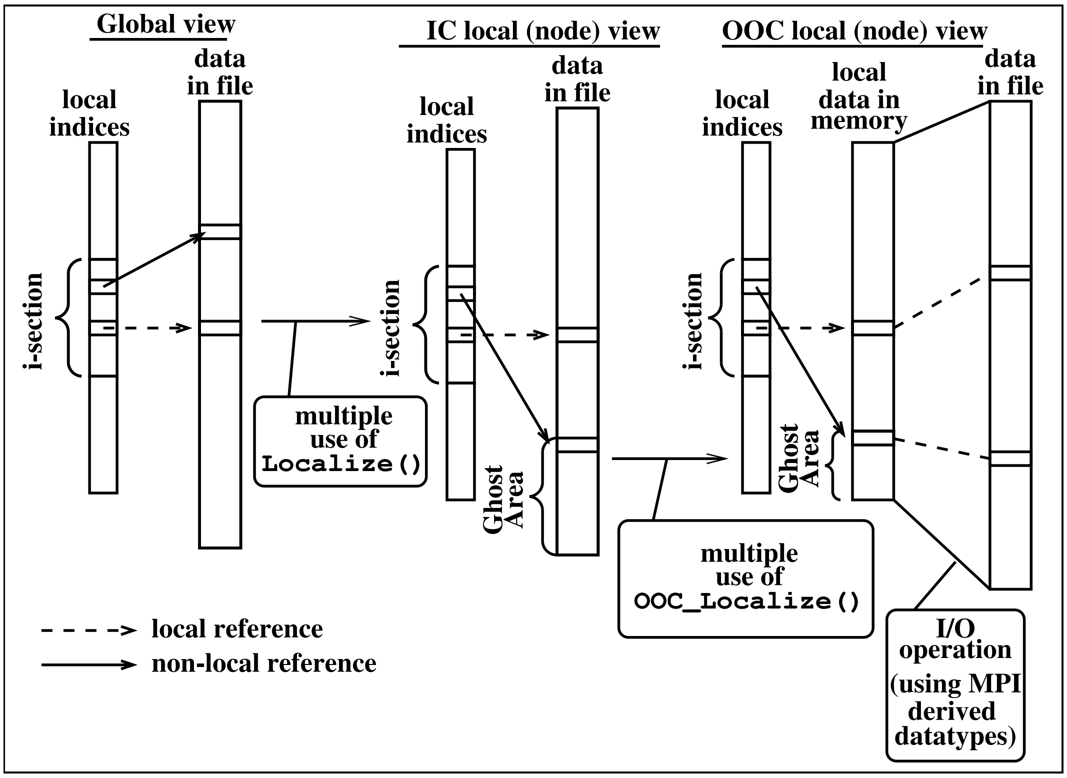

Figure 2.6: Irregular OOC inspector in `lip`

derived data types to perform actual movement of data using MPI-IO routines (like `MPI_File_read` and `MPI_File_write()`). Obtaining MPI datatypes is done via a call to`LIP_IObufmap_get_datatype()`. In this way the irregular problems for which all arrays are OOC may also be parallelized.

2.2.3 Irregular distribution

As in the case of in–core data (Subsection 2.1.2), the library also provides an out–of–core version of the partitioner . The irregular out–of-core mapping is achieved by a `LIP_create_hilbert_distribution_OOC` call as shown in the Fig. 2.8. In comparison to in–core version of data remapping, there is no need to call communication routines explicitly in order to redistribute data. Instead, one can call `LIP_remap_OOC` with a respective flag. The rest of the pseudocode is similar to that from Fig. 2.7.

```
MPI_Init( /* ... */ );
LIP_Setup( /* ... */ );

/* Generate index array describing
 * relationships between user data */

/* Perform irregular computation
 * on the data */
for ( /* ... */ )
{
  /* read i-section's indices
   * from a file */

  /* Inspector Phase (computes
   * optimized communication
   * and disk-memory mapping
     pattern)*/
  LIP_Localize(schedule,... );
  LIP_OOC_Localize(schedule,IObufmap,...);

  /* Create MPI derived datatypes
   * for moving data between
   * the memory and the disk */
  LIP_IObufmap_get_datatype(IOBufmap,...);

  /* Executor Phase (performs
     communication and computation) */

  /* Exchange data between disk and memory
   * in order to store in the memory
   * the data that will be needed
   * in this i-section */

  MPI_File_write( /* ... */ );
  MPI_File_read( /* ... */ );

  /* gather non-local irregular data */
  LIP_Gather(schedule,... );

  /* perform computation on data */
  for ( i = /* ... */ )
  {
    k = edge[i];
    y[k] = f( x[k] );
  }

  /* scatterer non-local
   * irregular data (results) */
  LIP_Scatter(schedule,... );
}

/* Get MPI derived datatypes for moving
 * the data obtained in the last
 * iteration from the memory
 * to the disk  */
LIP_IObufmap_get_datatype(IOBufmap );

/* Store the data on a disk */
MPI_File_write( /* ... */ );

LIP_Exit( /* ... */ );
MPI_Finalize( /* ... */ );
```

Figure 2.7: Simplified C code that shows the use of parallel computation on an out-of-core data array with the use of `lip` .

```
MPI_Init( /* ... */ );
LIP_Setup( /* ... */ );

/* Generate index array describing
 * relationships between user data */

/*Create of the irregular distribution*/
LIP_create_hilbert_distribution_OOC(
                        datamap,...);

/*Actually move data*/
LIP_remap_ooc(datamap,...);

/* Perform irregular computation
 * on the data */
for ( /* ... */ )
{
  /* read i-section's indices
   * from a file */

  /* Inspector Phase (computes
   * optimized communication
   * and disk-memory mapping pattern)*/
  LIP_Localize(datamap,schedule,... );
  LIP_OOC_Localize(schedule,IObufmap,...);

  /* Create MPI derived datatypes
   * for moving data between
   * the memory and the disk */
  LIP_IObufmap_get_datatype(IOBufmap,...);

  /* Executor Phase (performs
     communication and computation) */

  /* Exchange data between disk and memory
   * in order to store in the memory
   * the data that will be needed
   * in this i-section */
  MPI_File_write( /* ... */ );
  MPI_File_read( /* ... */ );

  /* gather non-local irregular data */
  LIP_Gather(schedule,...);

  /* perform computation on data */
  for ( i = /* ... */ )
  {
    k = edge[i];
    y[k] = f( x[k] );
  }

  /* scatterer non-local
   * irregular data (results) */
  LIP_Scatter(schedule,... );
}

/* Get MPI derived datatypes for moving
 * the data obtained in the last
 * iteration from the memory
 * to the disk  */
LIP_IObufmap_get_datatype(IOBufmap,... );

/* Store the data on a disk */
MPI_File_write( /* ... */ );

LIP_Exit( /* ... */ );
MPI_Finalize( /* ... */ );
```

Figure 2.8: Simplified C code that shows the use of parallel computation on irregularly distributed out-of-core data array with the use of `lip` .

Chapter 3

Detailed Examples

In this Chapter we introduce a simple irregular program that will serve as an example to show in detail how to use `lip` *to parallelize it. We will see how to move data out–of–core and how to use irregular distribution as well. To explain the practical use of library functions in as clear as possible a way numerous lines of code are accompanied by detailed descriptions.*

3.1 Generic Irregular Computation

```
#define N (2 * 3 * 2 * 5 * 7 * 2 * 9)
#define L (2 * N)

int
main(void)
{
  double *x, *y;
  int *perm;
  int n = N, l = L;
  int i;
  double sum_x, sum_y;

  x    = malloc( l * (sizeof *x) );
  y    = malloc( n * (sizeof *y) );
  perm = malloc( n * (sizeof *perm) );
  generate_data(x, perm, l, 0, l, 0, n);
  /* irregular loops */
  for (i = 0 ; i < n ; i++)
    y[i] = -x[ perm[i] ];
  for (i = 0 ; i < n ; i++)
    x[ perm[i] ] += y[i];
  for (sum_y = 0.0, i = 0 ; i < n ; i++)
  {
    sum_y += y[i];
  }
  printf( "The sum of %d elements of ‘y’ is %g\n", n, sum_y );
  for (sum_x = 0.0, i = 0 ; i < l ; i++)
  {
    sum_x += x[i];
  }
  printf( "The sum of %d elements of ‘x’ is %g\n", l, sum_x );
  free( perm );
  free( y );
  free( x );
  return 0;
} /* main */
```

In this simple program we have two data arrays: `x[]` and `y[]`. The first one is accessed through the indirect array `perm[]`, while the latter is referenced in a regular way. The irregular loops that use array `x[]` do it in two ways: one performs reading data and the second one modifies it. In the next Section we will see how to parallelize these two kinds of loops.

3.2 Parallelization of Simple Loops

The first step is to distribute data among processors. As a simpliest way we chose `BLOCK` data distribution. It means that each computing node will only own part of data and index array. To do that, we should add the following lines to our source code:

```
#define CD(j,k) (((j)-1)/(k)+1)
  MPI_Init( &argc, &argv );
  LIP_Setup( NULL, 0 ); /* all processes go to LIP */
  LIP_Rank( &rank );
  LIP_Size( &size );
  l_l = CD( l, size ); /* number of 'x' elements on single node */
  n_l = CD( n, size ); /* number of 'y' & 'perm' elements on single node */
  l_start = l_l * rank;
  n_start = n_l * rank;
  generate_data(x, perm, l, l_start, l_start+l_l, n_start, n_start+n_l);
  LIP_Datamap_create( LIP_DATAMAP_BLOCK, l, &l_l, LIP_INDEXTYPE_INT, 1, 0,
    &datamap );
```

We use ceiling division `CD(j,k)` to make sure there is enough memory allocated on all nodes in case the data length is not equally divisible by the number of nodes. Each node generates only its own data (beginning from `l_start`) and indices (from `n_start`). After that a new `Datamap` object is created with two important parameters: `l` - total length of dataset and `l_l` - size of local portion after partitioning. `Datamap` is the object that contains the description of data distribution. What we have to do next is to create the communication schedule, transform the index array and use the communication layer to get all the needed data into its place. We write the following lines:

```
  LIP_Schedule_create( l_l, &schedule );
  LIP_Localize( datamap, perm, LIP_INDEXTYPE_INT, perm_l, LIP_INDEXTYPE_INT,
    n_l, 0, &schedule, LIP_INDEXINFO_NULL );
```

In the first line we create the communication schedule and inform it that the local data array has the size `l_l`. Then we use the `LIP_Localize` function. The behavior of this function is shown in the Fig. 2.2 in Chapter 2. It reads information from the `Datamap` object and transforms the array `perm[]` into `perm_l[]`. The indices that pointed to non-local data elements now point to their places in the ghost-area, which is located in memory Section following the local array `x[]`. All the knowledge needed to bring these non-local data to their places is kept in the modified `schedule` object.

After localizing indices we are ready to perform computation of the loops. The first loop reads data from array `x[]` in an irregular way so before it can be computed, all the needed data must be gathered from other nodes. We write:

```
  LIP_Schedule_commit( &schedule );
  LIP_Gather( x_l, x_l + l_l, MPI_DOUBLE, schedule );
  for (i = 0 ; i < n_l ; i++)
    y[i] = -x_l[ perm_l[i] ];
```

The schedule has to be committed to ensure synchronization with other nodes. We gather data and put imported ones to the ghost-area.

The second loop is more complicated: we modify array `x[]` by adding to it some values. So after calculating the loop we have to scatter data to their owners, where their original values have to be modified, i.e. the summation has to be performed. Before we can start calculating this loop, we have to clear the ghost-area - otherwise wrong values might be added what would give errornous results.

```
for (i = 0 ; i < n_l ; i++)
  x_l[ i + l_l ] = 0.0;
for (i = 0 ; i < n_l ; i++)
   x_l[perm_l[i] ] += y[i];
LIP_Scatter( x_l + l_l, x_l, MPI_DOUBLE, schedule, LIP_OP_SUM );
LIP_Schedule_free( &schedule );
```

After scattering data and updating the local values, we can compute the total sum of arrays. Each node has to obtain the local sum and the final result is given by the `MPI_Reduce`.

```
for (sum_y_l = 0.0, i = 0 ; i < n_l ; i++)
  sum_y_l += y[i];

MPI_Reduce( &sum_y_l, &sum_y, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD );

for (sum_x_l = 0.0, i = 0 ; i < l_l ; i++)
  sum_x_l += x_l[i];

MPI_Reduce( &sum_x_l, &sum_x, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD );
```

3.3 Moving to Out–of–core Data

Let's consider the following situation: our data array is big enough to fill all our local memory. We don't want to use the paging mechanism and allocate big memory buffer to read the whole array from disk, since we want to use only these elements, that are referenced by our index array, which is quite small. We will see how to use `lip` in this case.

First, we suppose our data array is located in a disk file, called `"data.file"`. As a local buffer we allocate the array `x_l` of the size 2*`n_l` - identical to that of the index array with additional space for *ghost-area*.

```
x_l    = malloc( (n_l + n_l) * (sizeof *x_l) );

MPI_File_open(MPI_COMM_WORLD, "data.file", MPI_MODE_RDWR, MPI_INFO_NULL,
              &file);
```

In the inspector phase we have to create an extra structure, called `IOBufmap`, to provide mapping between disk and memory data placement. As a parameter we give the size of the local memory buffer. To create the schedule we provide the size of the array located on disk. Functions `LIP_Schedule_create` and `LIP_Localize` work in the same way for both in-core and out-of core versions. The new thing is the `LIP_OOC_localize` function, that performs the next step of index translation. The newly created `tperm_l` array consists of indices that point to a local memory buffer of size `n_l`, as declared in the `IOBufmap` structure, and to *ghost-area* located after it.

```
LIP_Datamap_create( LIP_DATAMAP_BLOCK, 1, &l_l, LIP_INDEXTYPE_INT, 1, 0,
  &datamap );
/* create IO bufmap which maps local data buffer on the data array
 * on disk */
LIP_IOBufmap_create(n_l, &bufm);

LIP_Schedule_create( l_l, &schedule );

LIP_Localize( datamap, perm, LIP_INDEXTYPE_INT, perm_l, LIP_INDEXTYPE_INT,
  n_l, 0, &schedule, LIP_INDEXINFO_NULL );

LIP_OOC_localize(perm_l, LIP_INDEXTYPE_INT, tperm_l, LIP_INDEXTYPE_INT,
                n_l, 0, x_l, MPI_DOUBLE, &bufm, &schedule);
```

After this we can start the executor phase. We use our `IOBufmap` structure to get two `MPI_Datatypes` needed to read data from disk: `filetype` to set view on a file and `memtype` to place data in proper memory places in the read function. As a displacement parameter to `MPI_File_set_view` function we have to give the position of the data block associated with the computing node, as the file is logically distributed (and may also be physically distributed on filesystems that support such distribution).

```
 LIP_IOBufmap_get_datatype(bufm,MPI_DOUBLE,&memtype,&filetype,
                           LIP_DATA_NEW);
 MPI_File_set_view(file,l_l*rank*8,MPI_DOUBLE,filetype,"native",info);
 MPI_File_read_at(file,0, x_l, 1, memtype, &status);

 LIP_Gather( x_l, x_l + n_l, MPI_DOUBLE, schedule );
```

The irregular loops are computed using the `tperm_l` array.

```
for (i = 0 ; i < n_l ; i++)
  y[i] = -x_l[ tperm_l[i] ];
for (i = 0 ; i < n_l ; i++)
  x_l[ i + n_l ] = 0.0;
for (i = 0 ; i < n_l ; i++)
   x_l[tperm_l[i] ] += y[i];

LIP_Scatter( x_l + n_l, x_l, MPI_DOUBLE, schedule, LIP_OP_SUM );
```

After that, the data on disk has to be updated. Once again we receive the required datatypes from `IOBufmap` and call I/O routines.

```
LIP_IOBufmap_get_datatype(bufm,MPI_DOUBLE,&memtype,&filetype,
                          LIP_DATA_ALL);
MPI_File_set_view(file,l_l*rank*8,MPI_DOUBLE,filetype,"native",info);
MPI_File_write_at(file,0, x_l, 1, memtype, &status);
```

Now the arrays on the disk are updated and the final sums can be computed in the same way as in the in-core version.

3.4 Indices and Data Out–of–core

In the following example we will see how to deal with the case when both the index and the data arrays are too large to fit into memory. We store these arrays in two files "`data.file`" and "`index.file`" respectively. To solve our problem we have to divide our out-of-core arrays into in-core sections and perform calculations step by step, each time reading new portions of data from the disk and writing them back after the calculation ends.

```
isec_l_l = l_l/ISEC; /* sizes of arrays in memory */
isec_n_l = n_l/ISEC;

y = malloc( isec_n_l * (sizeof *y) );
x_l1 = malloc( (isec_n_l + isec_n_l) * (sizeof *x) );
x_l2 = malloc( (isec_n_l + isec_n_l) * (sizeof *x) );
perm = malloc( isec_n_l * (sizeof *perm) );
perm_l = malloc( isec_n_l * (sizeof *perm) );
tperm_l = malloc( isec_n_l * (sizeof *perm) );

MPI_File_open(MPI_COMM_SELF, fname1, MPI_MODE_RDWR, MPI_INFO_NULL,
              &file1);
MPI_File_open(MPI_COMM_SELF, fname2, MPI_MODE_RDWR, MPI_INFO_NULL,
              &file2);
MPI_File_open(MPI_COMM_SELF, fname3, MPI_MODE_RDWR, MPI_INFO_NULL,
              &ifile);
```

The sizes of arrays in memory are now described by `isec_n_l` and `isec_l_l`. We have to create one additional file for writing data and one extra array as a data buffer. The reason for this is that we cannot update the original file after computing each *i-section*, since in the next one we may need to read this data again from the file and of course we don't want to have this data altered. Instead we write updated values to the second file so that after having finished computing all *i-sections* we get all our data properly updated.
In the same way as in previous example we create all the necessary data structures and start the main loop covering all *i-sections*

```
LIP_IOBufmap_create(isec_n_l, &bufm);

LIP_Datamap_create( LIP_DATAMAP_BLOCK, 1, &l_l, LIP_INDEXTYPE_INT, 1, 0,
    &datamap );

LIP_Schedule_create( l_l, &schedule ); /* l_l - size of array on disk */

/*** for each i-section ***/
for(isec_cnt=0;isec_cnt<ISEC; isec_cnt++)
{
```

In each i-section we read one portion of the index array and translate indices using localize functions as illustrated in the Fig 2.6 in the Chapter 2.

```
  offset = isec_cnt * isec_n_l;

  MPI_File_read_at(ifile,offset,perm,isec_n_l,MPI_INT,&status);

  LIP_Localize( datamap, perm, LIP_INDEXTYPE_INT, perm_l,
               LIP_INDEXTYPE_INT, isec_n_l, 0, &schedule,
               LIP_INDEXINFO_NULL );

  LIP_Schedule_commit( &schedule );

  LIP_OOC_localize(perm_l, LIP_INDEXTYPE_INT, tperm_l, LIP_INDEXTYPE_INT,
                  isec_n_l, 0, x_l1, MPI_DOUBLE, &bufm, &schedule)
```

Next have to update disk arrays before we read new data into memory buffers. To do it we get MPI datatypes from the `IOBufmap` structure with the `LIP_DATA_OLD` flag and after writing to the file we can use flag `LIP_DATA_NEW` to get the data required for the next step of calculation.

```
 LIP_IOBufmap_get_datatype(bufm,MPI_DOUBLE,&memtype,&filetype,
                          LIP_DATA_OLD);

 if (MPI_DATATYPE_NULL!=memtype)
 {
   MPI_File_set_view(file2,0,MPI_DOUBLE,filetype,"native",info);
   MPI_File_write_at(file2,0, x_l2, 1, memtype, &status);
 }
 LIP_IOBufmap_get_datatype(bufm,MPI_DOUBLE,&memtype,&filetype,
                          LIP_DATA_NEW);
 if (MPI_DATATYPE_NULL!=memtype)
 {
   MPI_File_set_view(file1,0,MPI_DOUBLE,filetype,"native",info);
   MPI_File_set_view(file2,0,MPI_DOUBLE,filetype,"native",info);

   MPI_File_read_at(file1,0, x_l1, 1, memtype, &status);
   MPI_File_read_at(file2,0, x_l2, 1, memtype, &status);
 }
```

After the IO phase we enter executor routines consisting of communication and computation.

```
 LIP_Gather( x_l1, x_l1 + isec_n_l, MPI_DOUBLE, schedule );

  for (i = 0 ; i < isec_n_l ; i++)
    y[i] = -x_l1[ tperm_l[i] ];

  for (i = 0 ; i < isec_n_l ; i++)
    x_l2[ i + isec_n_l ] = 0.0;

  for (i = 0 ; i < isec_n_l ; i++)
     x_l2[tperm_l[i] ] += y[i];

  LIP_Scatter( x_l2 + isec_n_l, x_l2, MPI_DOUBLE, schedule, LIP_OP_SUM );
} /*end for each i-section*/
```

That is all for the *i-sections* loop. After finishing it we should finally update the disk arrays using the datatypes obtained with the `LIP_DATA_NEW` flag.

```
LIP_Schedule_free( &schedule );
/* save all data to the disk */
LIP_IOBufmap_get_datatype(bufm,MPI_DOUBLE,&memtype,&filetype,
                          LIP_DATA_ALL);
if (MPI_DATATYPE_NULL!=memtype)
{
    MPI_File_set_view(file2,0,MPI_DOUBLE,filetype,"native",info);
    MPI_File_write_at(file2,0, x_l2, 1, memtype, &status);
    MPI_Type_free(&memtype);
    MPI_Type_free(&filetype);
}
```

At the end we can compute the final sum of array `x` by reading it from `file2`.

3.5 Using Irregular Distribution

Until now we have been using block distribution which meant we divided the arrays into blocks of equal size without taking into account any potential relationships between the data. In many cases, however, we can expect such relationships to exist and we can use them to distribute data in a better way. The case when each array element can be located in any of the nodes is called irregular distribution. A procedure called the partitioner creates such a distribution by using additional information about the data provided by the user. When computation is performed on data located in some 2D or 3D area and we know that the loop represents the computing of some local interaction between geometrically non-distant points, then we can use the coordinate based partitioner. It will distribute data in such a way that it will minimize communication during the executor phase, since most of the data will be available locally.
The next example shows how to use `lip` to deal with irregular distribution. We take the same loop as before, but we regard all data elements as points in a 2D area. The coordinates of these points are stored in arrays `xx[]` and `yy[]`. To do the remapping, we add the following lines:

```
LIP_Datamap_create( LIP_DATAMAP_BLOCK, l, &l_loc, LIP_INDEXTYPE_INT, 1, 0,
  &blockdata );

LIP_create_hilbert_distribution_in_core(blockdata,&irrdata,
xx,yy,NULL,vertex,edge,minedge,sample);

LIP_Schedule_create( l_l, &sch_rem );

LIP_remap_in_core(blockdata,irrdata,&sch_rem);
LIP_Datamap_free(&blockdata,0);
LIP_Schedule_commit( &sch_rem );

LIP_Gather(x,x2,MPI_DOUBLE,sch_rem);

LIP_Schedule_free(&sch_rem);
```

The `LIP_remap_in_core` function creates a communication schedule that should be used in the `LIP_Gather` function to move data according to new mapping. The rest of the program is the same as the regular in-core version using the new `x2[]` data array.

3.6 Irregular Out-of-core Distribution

Now we will look at the combination of two previous examples – the case when in an out-of-core problem the data is distributed in irregular way. We will need three files: the first one called "`data_reg.file`" with the original data regularly distributed and two files with irregular distribution for computation.

```
MPI_File_open(MPI_COMM_WORLD, "data_reg.file", MPI_MODE_RDWR |
              MPI_MODE_CREATE, MPI_INFO_NULL, &file1_reg);
MPI_File_open(MPI_COMM_WORLD, "data.file", MPI_MODE_RDWR |
              MPI_MODE_CREATE, MPI_INFO_NULL, &file1);
MPI_File_open(MPI_COMM_WORLD, "data-out.file", MPI_MODE_RDWR |
              MPI_MODE_CREATE, MPI_INFO_NULL, &file2);
```

In this example we assume the global file placement mode. It means the files are opened collectively by all computing nodes and all nodes have access to all of the files. In `data_reg.file` the data is logically distributed in a `BLOCK` fashion, i.e. each node is responsible for one block of data. To partition the data using the Hilbert's curve method we use the files containing coordinates.

```
LIP_Datamap_create( LIP_DATAMAP_BLOCK, 1, &l_l, LIP_INDEXTYPE_INT, 1, 0,
    &datamap );

LIP_create_hilbert_distribution_ooc(datamap, &datamap1, fx, fy, fz,
l_l, ps, sample, vertex,edge,minedge);

LIP_remap_ooc(datamap,&datamap1,file1_reg,file1,l_l, isec_l_l,MPI_DOUBLE,
              LIP_CREATE_LOCAL | LIP_MOVE_DATA);

LIP_remap_ooc(datamap,&datamap1,file1_reg,file2,l_l,isec_l_l,MPI_DOUBLE,
              LIP_MOVE_DATA);
LIP_Datamap_free(&datamap, 0);
```

Creation of the irregular mapping in `datamap1` proceeds in two steps. In the first step the `LIP_create_hilbert_distribution_ooc` function assigns data to processors; in the second, `LIP_remap_ooc` with `LIP_CREATE_LOCAL` flag finds the actual location on the node for each data element. The `LIP_MOVE_DATA` flag tells the function to copy data into its new locations in the new file. The remapping function can be called more times to remap more files according to one distribution.
The rest of the computation looks in the same way as for a regular OOC example, using files with remapped data and the newly created `datamap1` object.

Chapter 4

Java Bindings to `lip`

This Chapter presents the Java language bindings to `lip` *and gives a short description of the* ***Janet*** *tool that was used to create function wrappers. The concept of class design is described as well.*

4.1 General Concepts

4.1.1 Introduction to *Janet*

As Java is becoming increasingly popular in distributed and parallel computing [28, 6], there is a need to provide Java bindings to the `lip` library. The approach to this was to use Java Native Interface (JNI) [29] as a tool for creating Java wrappers to `lip` functions. JNI enables native code to use Java objects, call Java methods, access Java arrays and take full advantage of Java–specific features such as exceptions handling and threads. All this can be accessed by calling appropriate functions from JNI API.

During our work it transpired that the interface creation process using bare JNI is time consuming and error–prone. As a solution of this problem the Java to Native Interface Extension called ***Janet*** was created by Dawid Kurzyniec and described in his M.Sc. thesis and papers [33, 15, 16]. The ***Janet*** tool, now available on the Web [27], is an extension of the Java language that allows inserting native (i.e. C) language statements directly into Java source code files. The files with such extended Java syntax are the input for ***Janet*** translator which creates automatically Java and native source files with all the required JNI calls. It frees the programmer from calling low level JNI API and makes interface creation more flexible and clearer.

Java bindigs to `lip` were created using ***Janet*** (described above). The contribution of the authors of this thesis was to complete Java wrappers for the implemented functions and to integrate them with the library. Performance tests were conducted as well.

4.1.2 Java–Specific Features

Since Java was originally conceived as an object–oriented language, it encourages taking advantage of an OO programming model. As a higher level language in comparison to C it enables the expression of the following features in this manner:

- classes instead of structures and handles
- arrays instead of pointers
- exceptions instead of error return values

In `lip` there are three main data structures: `LIP_Schedule`, `LIP_Datamap` and `LIP_IOBufmap` that can naturally be represented as distinct Java classes. ***Janet*** introduces special utilities for creating classes from such structures. They are implemented in the `DataEncapsulator` class which stores the handle to the original structure and that handle can be passed to and modified by Java routines.

Many `lip` routines take as a parameter a pointer to data or an index array, that may refer to some portion of such an array. Since Java lacks pointers, ***Janet*** provides an utility called *virtual array* that represents Java array or a portion of one. It may be passed to a native function that converts it easily into a pointer and uses it as an argument for a native library call.

As all `lip` routines return status (i.e. error) values, the Java bindings can easily detect such errors and raise proper Java exceptions. The `LipException` class implements all possible errors generated by the `lip` library.

4.1.3 Sample function wrapper in *Janet*

The short example taken from file `Datamap.janet`, outlined below, shows the basic features of the ***Janet*** syntax used to create a wrapper for the `LIP_remap_ooc` routine. The file with the "`.janet`" extension is the source for the ***Janet*** translator. It can combine both Java and C languages, i.e. define Java a class that makes calls to C library functions.

```
public class Datamap extends DataEncapsulator {

native "C" {
#include <lip.h>
#include <mpi.h>
#include <mpio.h>
}
...

   public native "C" void remap_ooc (Datamap datamap,
             mpi.File datafile, mpi.File dataoutfile, int count,
             int pagesize, Datatype datatype, int flags)
   {
        LIP_Datamap new_datamap = (LIP_Datamap) `this.data`;
        int result = LIP_remap_ooc(`datamap.data`, &new_datamap,
            `datafile.data`, `dataoutfile.data`, `count`,
            `pagesize`, `datatype.handle`, `flags`);
        if (result != LIP_SUCCESS) `throw new LipException(#(result));`
   }
...
}
```

The `Datamap` java class includes the `remap_ooc` native method which is written in C language. The code can refer to Java objects and throw Java exceptions using statements enclosed in special characters “ ‘ ”.

4.2 Class Design Details

According to Java–specific features described in Sec. 4.1 the flat set of `lip` library functions was transformed into a group of classes implementing relevant methods. Such aggregation resulted in a clearer hierarchy of functions with reasonably reduced parameter lists.

The class hierarchy design is presented below, for further details on methods please refer to Appendix A.

4.2.1 Main classes

All `lip` classes are stored in the package `pl.edu.agh.icsr.lip`.

- Class `lip`
  The main library class that collects all basic functions for index translation and communication.
  - *Methods:*
    - `Setup` – implements: LIP_SETUP function;
    - `Exit` – implements: LIP_EXIT function;
    - `Rank` – implements: LIP_RANK function;
    - `Size` – implements: LIP_SIZE function;
    - `Localize` – implements: LIP_LOCALIZE function;
    - `OOC_localize` – implements: LIP_OOC_LOCALIZE function;
    - `Gather` – implements: LIP_GATHER function;
    - `Scatter` – implements: LIP_SCATTER function;
  - *Fields:*
    - `ContIntIndexer` class – the class representing the contiguous integer indexing scheme (see `Indexer` interface).
- Class `Schedule`
  Implements the functionality of the `Schedule` structure.
  - *Methods:*
    - `Schedule` constructor –
      implements: LIP_SCHEDULE_CREATE function;
    - `getNullValue` – returns LIP_SCHEDULE_NULL handle;
    - `commit` – implements: LIP_SCHEDULE_COMMIT function;
    - `free` – implements: LIP_SCHEDULE_FREE function;
- Class `Datamap`
  Contains all functions connected with data distribution and mapping.
  - *Methods:*

    * `Datamap` constructor –
      implements: LIP_DATAMAP_CREATE function;
    * `create_hilbert_distribution_in_core` –
      implements: LIP_CREATE_HILBERT_DISTRIBUTION_IN_CORE function;
    * `create_hilbert_distribution_ooc` –
      implements: LIP_CREATE_HILBERT_DISTRIBUTION_OOC function;
    * `remap_in_core` – implements: LIP_REMAP_IN_CORE function;
    * `remap_ooc` – implements: LIP_REMAP_OOC function;
  - *Fields:*
    * `BlockDistribution` class - the class represent data distribution specified by the LIP_BLOCK flag;

- Class `IOBufmap`
  Encapsulates data stored in the `IOBufmap` structure.
  - *Methods:*
    * `IOBufmap` constructor –
      implements: LIP_IOBUFMAP_CREATE function;
    * `getNullValue` – returns LIP_IOBUFMAP_NULL handle;
    * `getDatatype` –
      implements: LIP_IOBUFMAP_GET_DATATYPE function;
    * `free` – implements: LIP_IOBUFMAP_FREE function;

4.2.2 Other classes and interfaces

Java bindings to `lip` provide additional classes to encapsulate additional features.

- Interface `DataDistribution`
  It can represent various distributions handled by `lip` and it aggregates several parameters from the LIP_DATAMAP_CREATE function;
  - *Methods:*
    * `getGlobalSize` – global size of data array (global);
    * `getType` – type of datamap (type);
    * `getDescType` – index type (desctype);
    * `getDesc` – detailed description of datamap (desc);

- Class `IndicesDatatypes`
  Serves as a parameter to `IOBufmap.getDatatype` function, aggregating both required MPI datatypes.
  - *Fields:*
    * `memtype` – represents the memtype parameter;
    * `filetype` – represents the filetype parameter;

- Interface `Indexer`
  Represents the index array with the index layout.
  - *Methods:*
    - `getIndexData` – returns the array of indices;
    - `getIndexType` – returns an INDEXTYPE handle;
    - `getCount` – returns the size of the index array;
- Class `ContIntIndexer`
  Implements the `Indexer` interface for the integer array with contiguous index layout (LIP_INDEXTYPE_INT).
- Class `LipException` extends `RuntimeException` – the exception thrown by `lip` functions.

4.3 Sample Java Irregular Program

The Java code presented below shows the irregular out–of–core program that uses the Hilbert partitioner as described in Section 3.6. The declaration of `lip` structures in the Java class is convenient since they are implemented as objects:

```
import pl.edu.agh.icsr.lip.*;
import mpi.*;
import pl.edu.agh.icsr.janet.tools.*;

public class Data_ooc_irr {

   /* ... */

   public static void main(String[] args) throws MPIException {

       /* ... */

       Datamap datamap, filetype, memtype;
       Schedule schedule;
       IOBufmap bufm;
       Datamap datamap_irr;

       MPI.Init(args);
       LIP.Setup(null, 0);
       info = new mpi.Info();

       rank = LIP.Rank();
       size = LIP.Size();

       datamap = new Datamap(new Datamap.BlockDistribution(l, l_l), 1, 0);
       datamap_irr = new Datamap();
       datamap_irr.create_hilbert_distribution_ooc(datamap, xfile, yfile,
            null, l_l, ps, sample, new VDouble(vertex),edge,minedge);
```

Calling `lip` methods in Java also comes naturally by using virtual arrays `VDouble` with the method `subArray` to reference a portion of array data. It replaces the pointers that are commonly used in C library calls to pass arrays to functions.

```
mpi.File file_irr = new mpi.File();
file_irr.open(MPI.COMM_WORLD, "data_irr.file",
        mpi.File.MODE_RDWR | mpi.File.MODE_CREATE, null);

datamap_irr.remap_ooc(datamap,file,file_irr,l_l, ps,
        MPI.DOUBLE, Datamap.CREATE_LOCAL | Datamap.MOVE_DATA);

bufm = new IOBufmap(l_l);

schedule = new Schedule(l_l);
```

The index layout is encapsulated in the `ContIntIndexer` class which eliminates the need for using many flags to localize functions, as was the case in C.

```
LIP.Localize(datamap_irr,
       new LIP.ContIntIndexer(perm),
       new LIP.ContIntIndexer(perm_l),
       0, schedule);
LIP.OOC_Localize(new LIP.ContIntIndexer(perm_l),
           new LIP.ContIntIndexer(tperm_l),
           0, new VDouble(x_l), MPI.DOUBLE, bufm, schedule);
/* ... */

LIP.Gather(new VDouble(x_l), new VDouble(x_l).subArray(l_l),
     MPI.DOUBLE, schedule );

/* irregular loops
...
*/

LIP.Scatter(new VDouble(x_l).subArray(l_l), new VDouble(x_l),
      MPI.DOUBLE, schedule, LIP.OP_SUM);

schedule.free();
schedule = null;
```

4.4 Summary

The Java bindings to `lip` make it possible to deal with irregular and out–of–core applications in the Java language. `lip` proves the functionality of the ***Janet*** tool for creating such bindings. The performance results of sample Java application are presented in the next Chapter.

Bibliography

[1] Amor, M., Arguello, F., Lopez, J., Plata, O., Zapata, E.L.: A Data Parallel Formulation of the Barnes–Hut Method for N–body Simulations. *Workshop on Applied Parallel Computing (PARA'2000)*, Bergen, Norway, June 18-21, 2000.

[2] Bach, M.: The Design of the UNIX® Operating System. Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1986)

[3] Baiardi, F., Chiti, S., Mori, P., Ricci, L.: Parallelization of irregular problems based on Herarchical Domain Representation in: *Proc. Int. Conf. High Performance Computing and Networking*, Amsterdam, May 2000, Lecture Notes in Computer Science **1823**, Springer, 2000, pp. 71-80.

[4] Basney, J., Livny, M.: Deploying a High Throughput Computing Cluster. in: Rajkumar Buyya, Editor, *High Performance Cluster Computing*, Volume 1, Chapter 5, Prentice Hall PTR, May 1999.

[5] Bennett, R., Bryant, K., Sussman, A., Das, R., Saltz, J.: Jovian: A Framework for Optimizing Parallel I/O in: *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pages 10-20. IEEE Computer Society Press, October 1994.

[6] Boisvert, R.F., Dongarra, J.J., Pozo, R., Remington, K.A., Stewart, G.W.: Developing Numerical Libraries in Java. *ACM-1998 Workshop on Java for High-Performance Network Computing*, Stanford University, Palo Alto, California, February-March 1998.
`http://www.cs.ucsb.edu/conferences/java98/papers/jnt.ps`

[7] Bouhmala, N., Cai, X.: Extended Abstract Partition of Unstructured Finite Elements Mesches by a Multilevel Approach. March 15, 2000. in:*Proceedings of PARA'2000.*

[8] Brezany, P., Choudhary A.: Techniques and Optimizations for Developing Irregular Out-of-Core Applications on Distributed-Memory Systems. Technical Report Version 1.0, Institute for Software Technology and Parallel Systems, University of Vienna, November 1996.

[9] Brezany, P., Mueck, T.A., and Schikuta, E.: A Software Architecture for Massively Parallel Input/Output. In Waśniewski. J., et al.: *Applied Parallel Computing, Proc. PARA'96*, Lyngby, August 1996, Lecture Notes in Computer Science **1184**, Springer, pp. 85-96.

[10] Brezany P., Sipkova V., Chapman B., Griemel R.: Automatic Parallelization of the AVL FIRE Benchmark for a Distributed-Memory System In: *PARA'95*, Lyngby, Denmark, Springer Verlag, LNCS 1041, pp. 50-60

[11] Bubak, M., Łuszczek, P., and Wierzbowska, A.: Porting CHAOS Library to MPI. In Alexandrov, V., Dongarra, J., (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface, *Proc. 5th European PVM/MPI Users' Group Meeting*, Liverpool, UK, September 7-9, 1998, Lecture Notes in Computer Science **1497**, Springer, pp. 131-136.

[12] Bubak, M., Łuszczek, P.: Out-of-core Computations on HP S2000. in: *Proc. to HiPer 99 Conference*, Tromsø, Norway, June 27-30 1999.

[13] Bubak, M., Łuszczek, P.: Towards Portable Runtime Support for Irregular and Out-of-Core Computations. in: Dongarra, J., Luque, E., Margalef, T., (Eds.): Recent Advances in Parallel Virtual Machine and Message Passing Interface. *Proceedings of 6th European PVM/MPI Users' Group Meeting*, Barcelona, Spain, September 1999.

[14] Bubak, M., Łuszczek, P., Malawski, M., Zając, K.: Irregular and Out–of–core Computing on SGI Origin 2000 in: *Proceedings of 1st SGI Users Conference*, Cracow, Poland, October 2000, ACC Cyfronet UMM, 2000, pp. 185-193.

[15] Bubak M., Kurzyniec D., and Łuszczek P.: A versatile support of binding native code to Java. In M. Bubak, H. Afsarmanesh, R. Williams, and B. Hertzberger, editors, *Proceedings of the HPCN Conference*, pages 373-384, Amsterdam, May 2000.

[16] Bubak, M., Kurzyniec D., Łuszczek, P.: Creating Java to native code interfaces with Janet extension. in: *Proceedings of 1st SGI Users Conference*, Cracow, Poland, October 2000, ACC Cyfronet UMM, 2000, pp. 283-294.

[17] Caron, E., Cozette, O., Lazure, D., and Utard, G.: Virtual Memory Management in Data Parallel Applications In: Sloot, P., Bubak, M., Hoekstra, A., Hertzberger, B., (eds.): *Proc. Int. Conf. High Performance Computing and Networking*, Amsterdam, April 12-14, 1999, Lecture Notes in Computer Science **1593**, Springer, 1999, pp. 1107-1116.

[18] Das, R., Uysal, M., Saltz, J., Hwang, Y.-S.: Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. in:*Journal of Parallel and Distributed Computing*, **22**(3):462–479, September 1994.

[19] Datagrid project
`http://www.eu-datagrid.org/`

[20] Kitowski, J.: Współczesne systemy komputerowe.
Wydownictwo CCNS, Kraków 2000.

[21] Choudhary, A., Bordawekar, R., Harry, M., Krishnaiyer, R., Ponnusany, R., Singh, T., Thakur, R.: PASSION: parallel and scalable software for input–output. Technical Report SCCS–636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[22] Goudin, D., Roman, J.: A scalable parallel assembly for irregular meshes based on a block distribution for a parallel block direct solver. in: *Proceedings of PARA'2000*, LNCS 1947. - 2000.

[23] Hwang, Y-S., Das, R., Saltz, J., Brooks, B., and Hodoscek, M.: Parallelizing Molecular Dynamics Programs for Distributed Memory Machines: An Application of the CHAOS Runtime Support Library. *IEEE Computational Science and Engineering*, **2** (2) (1995) 18-29.

[24] International Organization for Standardization, Geneva. ISO/IEC DIS 9899 Information technology – Programming languages – Programming language C `http://www.lysator.liu.se/c/index.html`

[25] International Organization for Standardization, Geneva. Information technology — Portable Operating System Interface (POSIX) `http://anubis.dkuug.dk/JTC1/SC22/WG15/`

[26] Jagielski, R.,: Tablice rozproszone. WNT, Warszawa 1982.

[27] Janet homepage: `http://www.icsr.agh.edu.pl/janet`

[28] Java Grande Forum `http://www.javagrande.org`

[29] JavaSoft: Java Native Interface. `http://java.sun.com/products/jdk/1.2/docs/guide/jni/`

[30] Karypis, G., Schloegel, K., Kumar, V.: Parmetis: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 2.0., September 1999. `http://www-users.cs.umn.edu/ karypis/metis/partmetis/`

[31] Kernighan, B., Ritchie, D.: The C programming Language. *Second Edition.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1988)

[32] Kotz, D.: Disk-directed I/O for MIMD Multiprocessors. *ACM Transactions on Computer Systems*, **15**(1):41-74, February 1997.

[33] Kurzyniec D.: Creating Java to native code interfaces with Janet extension. M. Sc. Thesis, Department of Computer Science at University of Mining and Metallurgy. Cracow, Poland, August 2000.

[34] Laudon, J., Lenoski, D.: The SGI Origin: A ccNUMA Highly Scalable Server `http://www.sgi.com/origin/2000/`

[35] `lip` library distribution `http://www.icsr.agh.edu.pl/ malawski/lip/`

[36] Łuszczek, P.: Runtime Support for Irregular Out–of–core Problems. Master of Science Thesis, Institute of Computer Science, University of Mining and Metallurgy, Kraków, Poland.

[37] MPICH – A Portable Implementation of MPI,
`http://www.mcs.anl.gov/mpi/mpich/`

[38] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, **8**, 1994.

[39] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard (version 1.1), Technical report, 1995;
`http://www.mpi-forum.org/`

[40] Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface, July 18, 1997;
`http://www.mpi-forum.org/`

[41] Moon, B., Saltz, J.: Adaptive Runtime Support for Direct Simulation Monte Carlo Methoda on Distributed Memory Architectures. in: *Proceedings of the Scalable High Performance Computing Conference (SHPCC94)*, IEEE Computer Society Press, Knoxville, TN, May 1994, pp. 176183.

[42] Nieplocha, J., Foster, I.: Disk Resident Arrays: An Array–Oriented I/O Library for Out–of–Core Computations. In *Proc. of the Sixth Symposium of the Frontiers of Massively Parallel Computation* , IEEE Computing Society Press, October 1996,pp.196-204.

[43] Ou, C.-W.,Ranka, S.: SPRINT: Scalable Partitioning, Refinement, and INcremental partitioning Techniques.
`http://citeseer.nj.nec.com/169110.html`

[44] Ponnusamy, R.: Runtime and Compilation Methods for Irregular Computations on Distributed Memory Multiprocessors. Doctor of Philosophy Dissertation in Computer Science in The Graduate School of Syracuse University, May 1994

[45] Preis, R., Diekmann, R.: The PARTY Partitioning – Library, User Guide – Version 1.1 Technical Report, University of Paderborn, Sep. 1996.
`http://www.upb.de/fachbereich/AG/monien/RESEARCH/PART/party.html`

[46] Salmon, J., Warren, M.S.: Parallel, Out–of–core methods for N-body simulation. in: *Proceedings of the Eight Conference on Parallel Processing for Scientific Computing*, SIAM, 1997

[47] Saltz, J., Ponnusamy, R., Sharma, S., Moon, B., Hwang, Y.-S., Uysal, M., and Das, R.:
A Manual for the CHAOS Runtime Library.
UMIACS Technical Reports CS-TR-3437 and UMIACS-TR-95-34, University of Maryland: Department of Computer Science, March 1995;
`http://www.cs.umd.edu/users/uysal/papers/chaos.manual.html`

[48] Seamons, K. E., Chen, Y., Jones, P., Jozwiak, J., and Winslett, M.: Server Directed Collective I/O in Panda. In *Proc. Supercomputing'99*, San Diego, California, December 1995,
http://drl.cs.uiuc.edu/panda/

[49] Thakur, R., Bordawekar, R., Choudhary, A.: Compiler and Runtime Support for Out–of–Core HPF Programs In *Proc. of the 8th ACM International Conference of Supercomputing*, Manchester, England, July 1994

[50] Thakur, R., Choudhary, A.: An Extended Two–Phase Method for Accessing Sections of Out–of–Core Arrays in *Scientific Programming*, (5)4:301 317, Winter 1996

[51] Thakur, R., Gropp, W., and Lusk, E.: A Case for Using MPI's Derived Datatypes to Improve I/O Performance. In *Proc. of SC98: High Performance Networking and Computing* November 1998.

[52] Thakur, R., Gropp, W., and Lusk, E.: Data Sieving and Collective I/O in ROMIO. In *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, February 1999, pp. 182–189.

[53] Thakur, R., Lusk, E., and Gropp, W.:
Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. *Technical Memorandum ANL/MCS-TM-234*, Mathematics and Computer Science Division, Argonne National Laboratory, Revised July 1998; http://www.mcs.anl.gov/romio/

[54] Walshaw., C.: Parallel Jostle Library Interface: Version 1.1.6. School of Computing and Mathematical Sciences, University of Greenwich, London, SE18 6PF, UK, June 16, 1999 http://www.gre.ac.uk/jostle.

[55] Wierzbowska A.: Metody podziału danych i ich wykorzystanie w zagadnieniach nieregularnych i out–of–core. Praca Magisterska, Katedra Informatyki, AGH, Kraków, Poland

Index

Appendix A

The lip Library Specification

A.1 General Information

The full specification of the lip library is presented below. The form used in this documentation was based on the MPI standard documents [38, 39, 40]. The main feature of this form is that it is language-neutral. Furthermore, the semantics of terms and concepts used here are meant to be the same as in the MPI document. Therefore, for any details which are left unsaid, please refer to the standard documents for explanation.

The following specification presents the description of routines already implemented in the library. The routines that are not yet supported are considered in appendix B.

A.2 Irregular Partitioner

This Section describes the partitioner functions . The algorithm of data partitioning is based on Hilbert's curve and it is described in detail in the next Section. The lip library supports in-core and out-of-core versions of the partitioner.

LIP_CREATE_HILBERT_DISTRIBUTION_IN_CORE

LIP_CREATE_HILBERT_DISTRIBUTION_IN_CORE(datamap,new_datamap, coordinatesX, coordinatesY, coordinatesZ, vertex, edge, minedge, sample_count)

IN	datamap	current distribution (handle)
OUT	new_datamap	new (irregular) distribution (handle)
IN	coordinatesX	X coordinates of points in data array(double)
IN	coordinatesY	Y coordinates of points in data array (double)
IN	coordinatesZ	Z coordinates of points in data array can be null in 2D case (double)
IN	vertex	the coordinates of the vertex needed for counting Hilbert's indices (handle)
IN	edge	the length of edge of the square 2D (or cube 3D) needed for counting Hilbert indices (double)
IN	minedge	resolution of the indexing scheme (the length of the minimal distance between points that are distinguished by Hilbert's indexing scheme)
IN	sample_count	the quantity of the sample needed for counting Hilbert indices (integer)

C syntax:

```
int LIP_create_hilbert_distribution_in_core (LIP_Datamap datamap,
      LIP_Datamap *new_datamap, double *coX, double *coY, double
      *coZ, double* vertex, double edge, double minedge, int
      sample_count)
```

Java syntax:
Class Datamap:

```
public void create_hilbert_distribution_in_core (Datamap datamap,
      VDouble coX, VDouble coY, VDouble coZ, VDouble vertex, double
      edge, double minedge, int samplecount)
```

LIP_CREATE_HILBERT_DISTRIBUTION_IN_CORE creates a new_datamap object which contains a translation table that describes how a data array is partitioned among processors. Only a part of the translation table is filled - the ranks of processors that are the new owners of the data are written to the new_datamap. This is the in-core version of the algorithm that assumes that the data arrays and the translation table are in-core. To obtain the local indices of the data one should call remap function(see below).

LIP_CREATE_HILBERT_DISTRIBUTION_OOC

LIP_CREATE_HILBERT_DISTRIBUTION_OOC(datamap, new_datamap, coordinatesX_File, coordinatesY_File, coordinatesZ_File, count,page_size, vertex, edge, minedge, sample_count)

IN	datamap	current distribution (handle)
OUT	new_datamap	new, irregular distribution (handle)
IN	coordinatesX_File	X coordinates of points in data file(MPI_File)
IN	coordinatesY_File	Y coordinates of points in data file (MPI_File)
IN	coordinatesZ_File	Z coordinates of points in data file can be null in 2D case (MPI_File)
IN	count	size of the coordinates array (integer)
IN	page_size	size of memory buffer (integer)
IN	vertex	the coordinates of the vertex needed for counting Hilbert's indices (handle)
IN	edge	the length of edge of the square 2D (or cube 3D) needed for counting Hilbert indices (double)
IN	minedge	resolution of the indexing scheme (the length of the minimal distance between points that are distinguished by Hilbert's indexing scheme)
IN	sample_count	the quantity of the sample needed for counting Hilbert indices (integer)

C syntax:

```
int LIP_create_hilbert_distribution_ooc (LIP_Datamap datamap,
      LIP_Datamap * new_datamap, MPI_File XFILE, MPI_File YFILE,
      MPI_File ZFILE, int count, int page_size, int sample_count,
      void* vertex, double edge, double minedge)
```

Java syntax:
Class `Datmap`:

```
public void create_hilbert_distribution_ooc (Datamap datamap,
      mpi.File xfile, mpi.File yfile, mpi.File zfile, int count,
      int page size, int samplecount, VDouble vertex, double edge,
      double minedge)
```

LIP_CREATE_HILBERT_DISTRIBUTION_OOC creates object new_datamap which contains the translation table that describes how a data array is partitioned among processors. Only one part of the translation table is filled - the ranks of processors that are the new owners of the data are written to the new_datamap. The input data and this part of the translation table are stored OOC. To obtain the local indices of the data one should call remap function (see below).

LIP_REMAP_IN_CORE

LIP_REMAP_IN_CORE(old_datamap, new_datamap, schedule)

IN	datamap	current distribution (handle)
INOUT	new_datamap	new, irregular distribution (handle)
OUT	schedule	communication schedule for remapping (handle)

C syntax:

```
int LIP_remap_in_core (LIP_Datamap old_datamap, LIP_Datamap
      new_datamap, LIPSchedule schedule)
```

Java syntax:
Class Datamap:

```
public void remap_in_core(Datamap olddatamap, Schedule schedule)
```

LIP_REMAP_IN_CORE fills a part of the translation table in the new_datamap object which stores local indices of data elements on processors that are the owners of those elements. The function also creates a communication schedule needed for moving the data in order to obtain the new distribution.

LIP_REMAP_OOC

LIP_REMAP_OOC(datamap, new_datamap, data_File, data_out_File, count, page_size, datatype, flags)

IN	datamap	current distribution (handle)
INOUT	new_datamap	new,irregular distribution (handle) that contains the part of the translation table which stores information about which processor will own which element of data array after remapping
IN	data_file	file that stores data (handle)
OUT	data_out_file	file that stores data after remapping (handle)
IN	count	number of the local elements of data (integer)
IN	page_size	size of the memory buffer (integer)
IN	datatype	type of data to be remapped
IN	flags	flags (integer)

C syntax:

```
int LIP_remap_ooc (LIP_Datamap datamap, LIP_Datamap * new_datamap,
      MPI_File data_file, MPI_File data_out_file, int count, int
      page_size, MPI_Datatype datatype, int flags)
```

Java syntax:
Class Datamap:

```
public void remap_ooc (Datamap datamap, mpi.File datafile, mpi.File
      dataoutfile, int count, int pagesize, Datatype datatype, int
      flags)
```

Valid flags values are: LIP_CREATE_LOCAL and LIP_MOVE_DATA.

- When the LIP_CREATE_LOCAL flag is set, the function fills *local-indices* part of translation table in new_datamap object. This part stores local indices of data elements on processors which are the owners of those elements.
- When the LIP_MOVE_DATA flag is set, the function remaps the data from data_file. data_out_file stores data after remapping.

A.3 Data Mapping

This Section describes the functions connected with the mappings used by `lip` : the data map and the I/O buffer map.

LIP_DATAMAP_CREATE

LIP_DATAMAP_CREATE(type, global, desc, desctype, count, flags, datamap)

IN	type	type of the data map to be created (integer)
IN	global	global size of data array (integer)
IN	desc	detailed description of the data map to be created (array of integer)
IN	desctype	index type of datamap description array
IN	count	size of the description array (integer)
IN	flags	flags describing the data map to be created (integer)
OUT	datamap	new data map (handle)

C syntax:

```
int LIP_Datamap_create(int type, int global, void *desc,
      LIP_Indextype desctype, int count, int flags, LIP_Datamap
      *datamap)
```

Java syntax:
Class `Datamap`:

```
public Datamap(DataDistribution distrib, int count, int flags)
      public Datamap()
```

LIP_DATAMAP_CREATE creates a datamap object which describes how a data array is partitioned among processors. The argument pair: (type, desc) can have the following values (which correspond to the data mappings usually used in data-parallel languages):

- (LIP_BLOCK, {local data size }),
- (LIP_BLOCK_N; { local data size of node 0, ..., local data size of node N }), *not supported in* `lip` *version 1.0*
- (LIP_CYCLIC, {}) *not supported in* `lip` *version 1.0*,
- (LIP_INDIRECT, {local data size}).

Valid flags values are: NULL or LIP_OOC.

- When LIP_OOC flag is set, the translation table is stored on disk.

LIP_DATAMAP_IC_TTABLE_COPY

LIP_DATAMAP_IC_TTABLE_COPY(datamap, procs, procstype, local_adr, local_adr_type, ttable_size, flag)

INOUT	datamap	data map to be changed (handle)
IN	procs	array that contains the part of the translation table which stores information about which processor will own which element of data array after remapping
IN	procstype	index type of the procs array (handle)
IN	local_adr	stores local indices of data elements on processors that are owners of those elements.
IN	local_adrtype	index type of the local_adr array (handle)
IN	ttable_size	size of procs and local_adr arrays
IN	flags	flags describing if the procs and local_adr arrays should be copied or attached

C syntax:

```
int LIP_Datamap_ic_ttable_copy(LIP_Datamap datamap, void* procs,
      LIP_Indextype procstype, void* local_adr, LIP_Indextype
      localtype, int ttable_size, int flag)
```

LIP_DATAMAP_IC_TTABLE_COPY copies (if LIP_COPY flag was specified) or attaches (if LIP_ATTACH flag was specified) the procs and local_adr arrays to the datamap structure.

LIP_DATAMAP_IC_INIT_TABLE_WITH_PROC

LIP_DATAMAP_IC_INIT_TABLE_WITH_PROC(datamap, procs, procstype, ttable_size, flag)

INOUT	datamap	data map to be changed (handle)
IN	procs	array that contains the part of the translation table which stores information about which processor will own which element of data array after remapping
IN	procstype	index type of the procs array (handle)
IN	ttable_size	size of procs array
IN	flags	flags describing if the procs array should be copied or attached

C syntax:

```
int LIP_Datamap_ic_init_table_with_proc(LIP_Datamap datamap, void*
      proc, LIP_Indextype procstype, int ttable_size, int flags )
```

LIP_DATAMAP_IC_INIT_TABLE_WITH_PROC copies (if LIP_COPY flag was specified) or attaches (if LIP_ATTACH flag was specified) the procs array to the datamap structure.

LIP_DATAMAP_IC_TTABLE_DETACH

LIP_DATAMAP_IC_TTABLE_DETACH(datamap, procs, local_adr, ttable_size)

INOUT	datamap	data map to be changed (handle)
OUT	procs	array that contains the part of the translation table which stores information about which processor will own which element of data array after remapping
OUT	local_adr	that stores local indices of data elements on processors that are owners of those elements.
OUT	ttable_size	size of procs and local_adr arrays

C syntax:

```
int LIP_Datamap_ic_ttable_detach(LIP_Datamap datamap, int** procs,
       int** local_adr, int* ttable_size)
```

LIP_DATAMAP_IC_TTABLE_DETACH gets the procs and local_adr arrays and sets the apropriate fields in the datamap structure to NULL.

LIP_DATAMAP_IC_TTABLE_FREE

LIP_DATAMAP_IC_TTABLE_FREE(datamap)

INOUT	datamap	data map to be changed (handle)

LIP_DATAMAP_IC_TTABLE_FREE releases the memory space occupied by arrays of translation table stored in datamap.

LIP_DATAMAP_OOC_TTABLE_ATTACH

LIP_DATAMAP_OOC_TTABLE_ATTACH(datamap, procs_fname, local_adr_fname, ttable_size, ttable_membuf_size)

INOUT	datamap	data map to be changed (handle)
IN	procs_fname	name of the file that contains the part of the translation table which stores information about which processor will own which element of data array after remapping
IN	local_adr_fname	name of the file that stores local indices of data elements on processors that are owners of those elements.
IN	ttable_size	size of procs and local_adr files
IN	ttable_size	size of memory buffer for translation table

C syntax:

```
int LIP_Datamap_ooc_ttable_attach( LIP_Datamap datamap, char*
       proc_fname, char* local_adr_fname, int ttable_size, int
       ttable_membuf_size)
```

LIP_DATAMAP_OOC_TTABLE_ATTACH attaches the files named procs_fname and local_adr_fname to the datamap structure.

LIP_DATAMAP_OOC_TTABLE_GET_FILES

LIP_DATAMAP_OOC_TTABLE_GET_FILES(datamap, procs_fname, local_adr_fname)

INOUT	datamap	data map (handle)
OUT	procs_fname	pointer to a string that will store the name of the procs part of translation table (pointer)
OUT	local_addr_fname	pointer to a string that will store the name of the local_addr part of translation table (pointer)

C syntax:

```
int LIP_Datamap_ooc_ttable_get_files( LIP_Datamap datamap, char*
      procs_fname, char* local_adr_fname)
```

LIP_DATAMAP_OOC_TTABLE_GET_FILES returns the names of procs and local_adr files from the datamap structure.

LIP_DATAMAP_FREE

LIP_DATAMAP_FREE(datamap, flags)

INOUT	datamap	data map to be deleted (handle)
IN	flags	flags (integer)

C syntax:

```
int LIP_Datamap_free(LIP_Datamap * datamap, int flags)
```

Java syntax:
Class Datamap:

```
public void free(int flags)
```

LIP_DATAMAP_FREE releases the memory space occupied by datamap.
Valid flags values are:

- LIP_DATAMAP_STORE – (in OOC case) translation table files are not deleted
- NULL – the translation table files are deleted from disk

LIP_IOBUFMAP_CREATE

LIP_IOBUFMAP_CREATE(size, bufmap)

IN	size	size of the data buffer (integer)
OUT	bufmap	new data buffer map (handle)

C syntax:

```
int LIP_IOBufmap_create(int size, LIP_IOBufmap * bmap)
```

Java syntax:
Class `IOBufmap`:

```
public IOBufmap(int size)
```

LIP_IOBUFMAP_CREATE creates the I/O buffer map object which will store information about the mapping of the memory buffer of size size onto a file.

LIP_IOBUFMAP_FREE

LIP_IOBUFMAP_FREE(bufmap)

INOUT	bufmap	data buffer map to delete (handle)

C syntax:

```
int LIP_IOBufmap_free(LIP_IOBufmap * bmap)
```

Java syntax:
Class `IOBufmap`:

```
public void free()
```

LIP_IOBUFMAP_FREE releases the memory space occupied by bufmap.

LIP_IOBUFMAP_GET_DATATYPE

LIP_IOBUFMAP_GET_DATATYPE(bufmap, datatype, memtype, filetype,flags)

IN	bufmap data buffer map (handle)
IN	datatype type of stored data
OUT	memtype new MPI data type (handle) that describes indices of data in memory (used in MPI_File_write_* and MPI_File_read_* functions)
OUT	filetype MPI_Datatype that describes indices of data on disc (used in MPI_File_set_view)
IN	flags flags describing MPI data type (integer)

C syntax:

```
int LIP_IOBufmap_get_datatype(LIP_IOBufmap bmap, MPI_Datatype
      datatype, MPI_Datatype *memtype, MPI_Datatype *filetype, int
      flags)
```

Java syntax:
Class `IOBufmap`:

```
public IndicesDatatypes get_datatype(mpi.Datatype datatype, int
      flags)
```

LIP_IOBUFMAP_GET_DATATYPE creates a new MPI data type which may be used in MPI–IO routines for reading or writing data.

The flags argument can combine the following constants:

- LIP_DATA_OLD – then the returned MPI data type will refer to the old data elements (which should be written out to disk),
- LIP_DATA_NEW – then the returned MPI data type will refer to the new data elements (which should be read in from disk).
- LIP_DATA_ALL-then the returned MPI data type will refer to all data elements in the memory buffer - useful when the user wants to write all the data after performing the last calculation.

A.4 Communication Schedule

This Section describes functions connected with the *communication schedule* object.

LIP_SCHEDULE_CREATE

LIP_SCHEDULE_CREATE(size, schedule)

IN	size	size of the local data array served by a schedule (integer)
OUT	schedule	new communication schedule (handle)

C syntax:

```
int LIP_Schedule_create(int size, LIP_Schedule *schedule)
```

Java syntax:
Class `Schedule`:

```
public Schedule(int size)
```

LIP_SCHEDULE_CREATE creates a communication schedule for a local data array of size size.

LIP_SCHEDULE_COMMIT

LIP_SCHEDULE_COMMIT(schedule)
INOUT schedule communication schedule to be committed (handle)

C syntax:

```
int LIP_Schedule_commit(LIP_Schedule *schedule)
```

Java syntax:
Class `Schedule`:

```
public void commit()
```

LIP_SCHEDULE_COMMIT makes the schedule ready for communication after it is updated by the index translation functions of the `lip` library.

LIP_SCHEDULE_FREE

LIP_SCHEDULE_FREE(schedule)
INOUT schedule communication schedule (handle)

C syntax:

```
int LIP_Schedule_free(LIP_Schedule *schedule)
```

Java syntax:
Class `Schedule`:

```
public void free()
```

LIP_SCHEDULE_FREE releases the memory space occupied by schedule.

LIP_OOC_SCHEDULE_CREATE

LIP_OOC_SCHEDULE_CREATE(schedule_ooc)
OUT schedule_ooc new ooc communication schedule (handle)

C syntax:

```
int LIP_OOC_Schedule_create (LIP_Schedule_ooc * so)
```

LIP_OOC_SCHEDULE_CREATE creates a file on a disk to store the communication schedule structures. The file is created in the directory defined by the environmental variable LIP_TEMP

LIP_OOC_SCHEDULE_WRITE

LIP_OOC_SCHEDULE_WRITE(schedule, schedule_ooc)

IN	schedule	communication schedule (handle) to store in the file defined by schedule_ooc
INOUT	schedule_ooc	ooc communication schedule (handle)

C syntax:

```
int LIP_OOC_Schedule_write (LIP_Schedule s, LIP_Schedule_ooc * so)
```

LIP_OOC_SCHEDULE_WRITE appends a schedule structure to the file defined by the schedule_ooc structure. The same file can store more then one schedule. They can be read in the order they were written in.

LIP_OOC_SCHEDULE_READ

LIP_OOC_SCHEDULE_READ(schedule, schedule_ooc)

OUT	schedule	communication schedule (handle) to be read from the file defined by schedule_ooc
IN	schedule_ooc	ooc communication schedule (handle)

C syntax:

```
int LIP_OOC_Schedule_read (LIP_Schedule s, LIP_Schedule_ooc * so)
```

LIP_OOC_SCHEDULE_READ reads a schedule from the file defined by the schedule_ooc structure. The pointer in the file is moved to the next schedule stored in this file or to the end of the file.

LIP_OOC_SCHEDULE_RESET

LIP_OOC_SCHEDULE_RESET(schedule_ooc)

INOUT	schedule_ooc	ooc communication schedule(handle)

C syntax:

```
int LIP_OOC_Schedule_reset (LIP_Schedule_ooc * s)
```

LIP_SCHEDULE_RESET moves the schedule file pointer to the beginning of the file.

LIP_OOC_SCHEDULE_FREE

LIP_OOC_SCHEDULE_FREE(schedule_ooc)

INOUT	schedule_ooc	ooc communication schedule (handle)

C syntax:

```
int LIP_OOC_Schedule_free (LIP_Schedule_ooc * so)
```

LIP_OOC_SCHEDULE_FREE deletes the file storing communication schedule structures and frees the memory occupied by the schedule_ooc structure.

A.5 Index Translation

This Section describes the functions that deal with the index translation issues.

LIP_LOCALIZE

LIP_LOCALIZE(datamap, iglobal, igtype, ilocal, iltype, count, flags, schedule, info)

IN	datamap	data map (handle)
INOUT	iglobal	globally numbered index array (choice)
IN	igtype	index type of the globally numbered index array (handle)
OUT	ilocal	locally numbered index array (choice)
IN	iltype	index type of the locally numbered index array (handle)
IN	count	number of indices in the index arrays (integer)
IN	flags	flags providing additional information (integer)
INOUT	schedule	communication schedule (handle)
INOUT	info	index information (handle)

C syntax:

```
int LIP_Localize(LIP_Datamap datamap, void *iglobal, LIP_Indextype
      igtype, void *ilocal, LIP_Indextype iltype, int count, int
      flags, LIP_Schedule *schedule, LIP_Indexinfo *info)
```

Java syntax:
Class LIP:

```
public static void Localize(Datamap datamap, Indexer iglobal,
      Indexer ilocal , int flags, Schedule schedule)
```

LIP_LOCALIZE translates the globally numbered indices from iglobal (with layout igtype) into locally numbered indices ilocal (with layout iltype). The number of indices is supplied in count. After the translation the schedule communication schedule is created which could be used to transmit non-local data. The info object stores information about indices which might help in the translation process.

LIP_OOC_LOCALIZE

LIP_OOC_LOCALIZE(ilocal, iltype, itlocal, ittype, count, flags, buf, datatype, bmap, schedule)

INOUT	ilocal	locally numbered index array (choice)
IN	iltype	index type of the locally numbered index array (handle)
OUT	itlocal	transformed index array (choice)
IN	ittype	index type of the transformed index array (handle)
IN	count	number of indices in index arrays (integer)
IN	flags	flags providing additional information (integer)
INOUT	buf	memory data buffer (handle)
IN	datatype	data type of the buffer's elements (handle)
INOUT	bmap	I/O buffer map (handle)
INOUT	schedule	communication schedule (handle)

C syntax:

```
int LIP_OOC_localize(void *ilocal, LIP_Indextype iltype, void
      *itlocal, LIP_Indextype ittype, int count, int flags, void
      *buf, MPI_Datatype datatype, LIP_IOBufmap *bmap, LIP_Schedule
      *schedule)
```

Java syntax:
Class LIP:

```
public static void OOC_Localize(Indexer ilocal, Indexer itlocal,
      int flags, VDouble buf, mpi.Datatype datatype, IOBufmap bmap,
      Schedule schedule)
```

LIP_OOC_LOCALIZE transforms indices provided in ilocal (with layout iltype) into itlocal (with iltype layout). The schedule communication schedule is also translated. The translation process is done so as the indices point to a contiguous in-core data buffer afterwards (beforehand they use an out-of-core data buffer). The number of indices is provided in count. The in-core buffer is provided by buf and the type of the single buffer element – by datatype. The result of the translation is stored in bmap.

A.6 Communication Functions

This Section provides the description of the communication functions.

LIP_GATHER

LIP_GATHER(sendbuf, recvbuf, datatype, schedule)

IN	sendbuf	send buffer (choice)
OUT	recvbuf	receive buffer (choice)
IN	datatype	MPI data type of buffer elements (handle)
IN	schedule	communication schedule (handle)

C syntax:

```
int LIP_Gather(void *sendbuf, void *recvbuf, MPI_Datatype datatype,
      LIP_Schedule schedule)
```

Java syntax:
Class LIP:

```
public static void Gather(VDouble sendbuf, VDouble recvbuf, Datatype
      datatype, Schedule schedule)
```

LIP_GATHER is a collective function which gathers from other nodes the data on which the computations are to be performed locally. The data are taken from sendbuf and are received in recvbuf (which is usually the *ghost area*). Each buffer element has the type datatype. The communication pattern is described by schedule.

LIP_SCATTER

LIP_SCATTER(sendbuf, recvbuf, datatype, schedule, op)

IN	sendbuf	send buffer (choice)
OUT	recvbuf	receive buffer (choice)
IN	datatype	MPI data type of the buffer elements (handle)
IN	schedule	communication schedule (handle)
IN	op	irregular operation to perform on communicated data (handle)

C syntax:

```
int LIP_Scatter(void *sendbuf, void *recvbuf, MPI_Datatype datatype,
      LIP_Schedule schedule, LIP_Op op)
```

Java syntax:
Class LIP:

```
public static void Scatter(VDouble sendbuf, VDouble recvbuf, Data
      type datatype, Schedule schedule, int op)
```

LIP_SCATTER is a collective function which scatters to other nodes the data that was computed locally. The data is taken from sendbuf (which is usually the *ghost area*) and is received in recvbuf. Each buffer element has the type datatype. The communication pattern is described by schedule.

A.7 Miscellanea

In this Section, all the auxiliary functions are described.

LIP_SETUP

LIP_SETUP(nodes, count)

IN	nodes	array of node numbers (array of integer)
IN	count	size of array of nodes (integer)

C syntax:

```
int LIP_Setup(int *nodes, int count)
```

Java syntax:
Class LIP:

```
public static void Setup(int[] nodes, int count)
```

LIP_SETUP initializes the lip library. The nodes that will be included in a lip communication context are provided in the nodes array which has a size count. Only these nodes will perform lip–related computations and communication. The ranks of the nodes, which were put into the lip library, correspond to the order in which they were supplied in the nodes array.

LIP_EXIT

LIP_EXIT()

C syntax:

```
int LIP_Exit(void)
```

Java syntax:
Class LIP:

```
public static void Exit()
```

LIP_EXIT finalizes the operation of the lip library. All the data structures are deleted and the memory space occupied by them is released to the system.

LIP_RANK

LIP_RANK(node)
OUT node node number (integer)

C syntax:

```
int LIP_Rank(int *node)
```

Java syntax:
Class LIP:

```
public static native "C" int Rank()
```

LIP_RANK returns the rank of the node within the lip communication context.

LIP_SIZE

LIP_SIZE(nodes)
OUT node number of all nodes (integer)

C syntax:

```
int LIP_Size(int *nodes)
```

Java syntax:
Class LIP:

```
public static int Size()
```

LIP_SIZE returns the number of nodes within the lip communication context.

Appendix B

The lip Library Specification – Additional Functions

B.1 General Information

The following specification presents the routines of the `lip` library that are not supported in `lip` version 1.0. Their implementation should be considered in the future.

B.2 Communication Schedule

This section describes additional functions connected with the *communication schedule* object.

LIP_SCHEDULE_BUFFER_ATTACH

LIP_SCHEDULE_BUFFER_ATTACH(schedule, buf, size)

IN	schedule	communication schedule (handle)
IN	buf	address of the buffer to be attached (choice)
IN	size	size (in bytes) of buffer (integer)

C syntax:

```
int LIP_Schedule_buffer_attach(LIP_Schedule schedule, void *buf, int
       size)
```

LIP_SCHEDULE_BUFFER_ATTACH attaches a memory buffer of size (in bytes) size to the schedule object which is provided by schedule. There is only one memory buffer which can be attached to a schedule. For information of how to obtain the optimal buffer size, see the description of LIP_SCHEDULE_GET_INFO.

LIP_SCHEDULE_BUFFER_DETACH

LIP_SCHEDULE_BUFFER_DETACH(schedule, buf, size)

IN	schedule	communication schedule (handle)
OUT	buf	detached buffer (choice)
OUT	size	size (in bytes) of buffer (integer)

C syntax:

```
int LIP_Schedule_buffer_detach(LIP_Schedule schedule, void *buf, int
        *size)
```

LIP_SCHEDULE_BUFFER_DETACH detaches the memory buffer attached to the schedule object provided by schedule and returns the address of the buffer in buf. The size (in bytes) of the of the detached buffer is returned in size.

LIP_SCHEDULE_GET_INFO

LIP_SCHEDULE_GET_INFO(schedule, info, count)

IN	schedule	communication schedule (handle)
OUT	info	array which will be filled with information (array of integer)
IN	count	size of the information array (integer)

C syntax:

```
int LIP_Schedule_get_info(LIP_Schedule schedule, int *info, int
        count)
```

LIP_SCHEDULE_GET_INFO returns information about the schedule schedule in the info array. The info array is similar to Status object from MPI standard. The size of info array should be at least as large as the value of the LIP_SCHEDULE_INFO_SIZE constant. The positions of the info array contain information about the schedule schedule. These position are defined by the following constants:

- LIP_EXPORT_SIZE – the number of the data elements which are exported to other nodes,
- LIP_IMPORT_SIZE – the number of the data elements which are imported from other nodes,
- LIP_DATA_COUNT – the size of the data array for which the schedule was created,
- LIP_COMM_BUF_SIZE – the number of the elements (size in bytes depends on the type of a single data element) which will be stored in a communication buffer (this could be used before the call to LIP_SCHEDULE_BUFFER_DETACH).

B.3 Indexing Scheme

This section describes functions which could be used for defining the layout of indices within the index arrays, called *index type*.

LIP_INDEXTYPE_BLOCK

LIP_INDEXTYPE_BLOCK(offset, blksize, blkdispls, displcount, flags, itype)

IN	offset	displacement (in bytes) of the first block (integer)
IN	blksize	size (in bytes) of each block (integer)
IN	blkdispls	array of displacements within a block (array of integer)
IN	displcount	size of the displacement array (integer)
IN	flags	flags that contain additional information (integer)
OUT	itype	new index type (handle)

C syntax:

```
int LIP_Indextype_block(int offset, int blksize, int *blkdispl, int
      displcount, int flags, LIP_Indextype *itype)
```

LIP_INDEXTYPE_BLOCK creates the object itype which describes a layout of indices in the memory. Indices could be stored in groups (represented by contiguous blocks of memory); each group has the same layout, although the layout can be arbitrary within the group. The size of each block is specified by blksize. Within a block the consecutive indices are given by their displacements which are provided in the blkdispls array. The first block is at the offset displacement from the beginning of the index array. The flags argument can combine the following constants:

- LIP_INDEXTYPE_FORTRAN – the data arrays are numbered starting from one
- LIP_INDEXTYPE_C – the data arrays are numbered starting from zero

LIP_INDEXTYPE_ADDRESSES

LIP_INDEXTYPE_ADDRESSES(addrs, itypes, count, mlevel, itype)

IN	addrs	array of the byte displacements of index arrays (array of integer)
IN	itypes	array of indextypes (array of handles to indextype objects)
IN	count	number of index arrays (integer)
IN	mlevel	true if multi-level indexing; false otherwise;
OUT	itype	new index type (handle)

C syntax:

```
int LIP_Indextype_multiple(MPI_Aint *displ, LIP_Indextype *itypes,
      int count, int mlevel, LIP_Indextype *itype)
```

LIP_INDEXTYPE_ADDRESSES creates a new object itype which describes an indexing scheme defined by the count indexing arrays provided (as their absolute byte displacements) in addrs. Each of the arrays has the index layout defined by the corresponding position in the itypes array. The mlevel argument, if true, specifies that the index arrays form a single multi-level indexing scheme.

LIP_INDEXTYPE_FREE

LIP_INDEXTYPE_FREE(itype)

INOUT	itype	index type to be committed (handle)

C syntax:

```
int LIP_Indextype_free(LIP_Indextype *itype);
```

LIP_INDEXTYPE_FREE releases the memory space occupied by itype.

B.4 User-defined Irregular Operation

This section describes the functions connected with *user-defined irregular operation*. It is very similar to the operation created via MPI_OP_CREATE, however, it adds indirect indexing to it.

LIP_OP_CREATE

LIP_OP_CREATE(function, op)

IN	function	function which will an perform operation (function)
OUT	op	new operation (handle)

C syntax:

```
int LIP_Op_create(LIP_User_function *function, LIP_Op *op)
```

LIP_OP_CREATE creates a new operation object in op which stores the function function which, in turn, could be applied to the data.

LIP_USER_FUNCTION

LIP_USER_FUNCTION(iin, tin, invec, iinout, tinout, inoutvec, len, datatype)

IN	iin	input vector index array (choice)
IN	tin	input vector index type (handle)
IN	invec	input vector (choice)
IN	iinout	input/output vector index array (choice)
IN	tinout	input/output vector index type (handle)
INOUT	inoutvec	input/output vector (choice)
IN	len	number of elements in both vectors (integer)
IN	datatype	MPI data type of elements in both vectors (handle)

C syntax:

```
typedef void (*LIP_User_function) (void *iin, LIP_Indextype tin,
      void *invec, void *iinout, LIP_Indextype tinout, void
      *inoutvec, int *count, MPI_Datatype *datatype)
```

LIP_USER_FUNCTION is a prototype of a user-supplied function which could be used in a call to LIP_OP_CREATE. Its semantics with respect to the invec, inoutvec, len and datatype arguments are the same as in the user function required for MPI_OP_CREATE. However, since the data arrays are indexed indirectly, the indirect arrays are provided in iin and tin for invec and in iinout and tinout for inoutvec. It is guaranteed that the indexing types passed to the user function will already be known by the user.

LIP_OP_FREE

LIP_OP_FREE(op)

INOUT	op	operation (handle)

C syntax:

```
int LIP_Op_free(LIP_Op *op)
```

LIP_OP_FREE releases the memory space occupied by op.

B.5 Miscellanea

In this section, all the auxiliary functions are described.

LIP_INDEXINFO_ANALYZE

LIP_INDEXINFO_ANALYZE(dmap, indarr, itype, count, flags, info)

IN	dmap	data map (handle)
IN	indarr	index array (choice)
IN	itype	indexing type for the index array (handle)
IN	count	number of indices (integer)
IN	flags	flags that contain additional information (integer)
OUT	info	information about indices (handle)

C syntax:

```
int LIP_Indexinfo_analyze(LIP_Datamap dmap, void *indarr,
      LIP_Indextype itype, int count, int flags, LIP_Indexinfo
      *info)
```

LIP_INDEXINFO_ANALYZE analyzes the count indices in indarr (with layout itype) and returns information about them in an info object. The location of indices is taken from dmap. flags controls which information is obtained about the indices. The flags argument can combine the following constants:

- LIP_INDEXINFO_UNIQUE – the number of globally unique indices is found,
- LIP_INDEXINFO_NODE_COUNT – the number of indices on each node is found,
- LIP_INDEXINFO_NODE_UNIQUE – the number of unique indices on each node is found.

LIP_INDEXINFO_SET

LIP_INDEXINFO_SET(count, ucount, inode, iunique, info)

IN	count	number of indices (integer)
IN	ucount	number of unique indices (integer)
IN	inode	number of indices from each node (array of integer)
IN	iunique	number of unique indices from each node (array of integer)
OUT	info	information about indices (handle)

C syntax:

```
int LIP_Indexinfo_set(int count, int ucount, int *inode, int
      *iunique, LIP_Indexinfo *info)
```

LIP_INDEXINFO_SET stores the information about a index array in the info object. The number of all indices is specified in count, while the number of unique indices – in ucount. The array inode specifies, for each node, the number of indices referencing the data from this node, whereas the array iunique provides the information on unique indices for each node.

LIP_INDEXINFO_GET

LIP_INDEXINFO_GET(info, count, ucount, inode, iunique)

IN	info	information about indices (handle)
OUT	count	number of indices (integer)
OUT	ucount	number of unique indices (integer)
OUT	inode	number of indices from each node (array of integer)
OUT	iunique	number of unique indices from each node (array of integer)

C syntax:

```
int LIP_Indexinfo_get(LIP_Indexinfo info, int *count, int *ucount,
      int *inode, int *iunique)
```

LIP_INDEXINFO_GET obtains the information about an index array from the info object. The number of all indices is stored in count, while the number of unique indices – in ucount. The array inode will be filled with the number of indices referencing the data on each node, whereas the array iunique will store the information on the unique indices for each node.

LIP_INDEXINFO_FREE

LIP_INDEXINFO_FREE(info)
INOUT info information about indices (handle)

C syntax:

```
int LIP_Indexinfo_free(LIP_Indexinfo *info)
```

LIP_INDEXINFO_FREE releases the memory space occupied by info.