Akademia Górniczo-Hutnicza
w Krakowie

Wydział Elektrotechniki, Automatyki,
Informatyki i Elektroniki

Katedra Informatyki

Biblioteka wspomagająca zrównoleglanie zagadnień nieregularnych i out-of-core

Maciej Malawski, Katarzyna Zając

Praca Magisterska
kierunek studiów: Informatyka

Promotor: dr inż. Marian Bubak

Kraków, maj 2001

University of Mining and Metallurgy
in Kraków, Poland

Faculty of Electrical Engineering, Automatics,
Computer Science and Electronics

Institute of Computer Science

Advanced Library for Parallelization of Irregular and Out-of-core Problems

Maciej Malawski, Katarzyna Zając

Master of Science Thesis
Computer Science

Supervisor: Dr. Marian Bubak

Kraków, May 2001

Acknowledgements

We would like to express our thanks to our supervisor – Dr. Marian Bubak – for his invaluable help and advice on the problems encountered during preparation of this thesis. Without his great contribution this thesis would not have been possible in its current form.
We gratefully acknowledge the support of Prof. Peter Brezany from Institute for Software Science and the Parallel Systems in Vienna for his advice, remarks and support in gathering performance data.
Next, we want to thank Agnieszka Wierzbowska, Piotr Łuszczek and Dawid Kurzyniec for the valuable discussions, suggestions and help.
In addition, the help of Mr. Andrzej Oziębło from ACC CYFRONET is greatly appreciated.
Last, but not least, the thanks for valuable advice also go to Piotr Nowakowski who was helping us with English grammar and read a proof of this Thesis.

Abstract

Many large scale applications require the computation of some properties for a set of elements that are irregularly distributed in a domain. These applications result in code which implements access to data arrays through one or more levels of indirect arrays which are not defined until runtime.
In addition, many applications deal with data sets that are too large to fit into the main memory and thus they are referred to as out–of–core problems.
This Thesis presents the `lip` library – runtime support for irregular and out–of–core problems. The library is build on top of MPI–IO, and is, therefore, fully portable. The library was developed as a part of a research program within the framework of the Polish–Austrian collaboration between the Institute of Computer Science, UMM, Kraków and the Institute for Software Science, Vienna.
This thesis is organized as follows: Chapter 1 introduces the terms and concepts relevant to the area of irregular and out–of–core problems. Chapter 2 outlines the parallelizing and optimization techniques for irregular problems. Chapter 3 presents the classification of out–of–core problems and describes optimization methods used in this field. The main idea of the `lip` library is presented in Chapter 4. The detailed description of the library can be found in Chapter 5. Chapter 6 contains examples utilizing the proposed parallelization scheme. Chapter 7 outlines the Java to Native Interface Extension tool that was used to create Java bindings to the `lip` library. Chapter 8 presents the performance results from synthetic and real benchmarks performed on various parallel architectures. Finally, Chapter 9 summarizes the results of our research and outlines the proposals for future development.
The `lip` library specification is given in Appendices A and B. Appendix C contatins publications related to this thesis.

Contents

Table of contents **9**

List of figures **12**

List of tables **13**

1 Introduction **14**
1.1 Problem Outline 14
1.2 Related Work 15
1.3 History of `lip` 16
1.4 Goals of This Thesis 17

2 Issues in Irregular Computations **18**
2.1 Parallelizing Techniques for Irregular Problems 18
2.2 Optimization Methods 18
2.2.1 Data partitioning 18
2.2.2 Work distribution 19
2.2.3 Reducing communication costs 19
2.3 CHAOS Library 20
2.3.1 Data distribution 20
2.3.2 Data remapping 20
2.3.3 Loop iteration partitioning 20
2.3.4 Remapping loop iterations 20
2.3.5 Inspector 21
2.3.6 Executor 21

3 Overview of OOC Problems **22**
3.1 Classification of OOC Problems 22
3.2 Tools for OOC Computing 22
3.3 Optimization Techniques 23
3.3.1 Data reusing 23
3.3.2 Data sieving 24
3.3.3 Collective I/O 24
3.3.4 Data prefetching 25

4 The Concept of `lip` 26
4.1 Runtime Support Requirements and their Solutions . . . 26
4.1.1 Cross–platform portability . . . 26
4.1.2 Multi–language support . . . 26
4.1.3 Efficiency of performance . . . 27
4.1.4 Scalability . . . 27
4.1.5 Support for interfaces built on the top of `lip` . . . 27
4.2 The Case of In–core Data . . . 27
4.2.1 Support for irregular problems . . . 27
4.2.2 Irregular data distribution . . . 29
4.3 The Case of OOC Data . . . 29
4.3.1 Dealing with OOC index–arrays and in–core data arrays . . . 29
4.3.2 Dealing with OOC index–arrays and data arrays . . . 30
4.3.3 Irregular distribution . . . 32

5 Implementation Details 34
5.1 Data Distribution . . . 34
5.1.1 Data structures . . . 34
5.1.2 Functions . . . 34
5.2 Communication . . . 38
5.2.1 Data structures . . . 38
5.2.2 Functions . . . 38
5.3 IO Operations . . . 40
5.3.1 Data structures . . . 40
5.3.2 Functions . . . 40

6 Detailed Examples 42
6.1 Generic Irregular Computation . . . 43
6.2 Parallelization of Simple Loops . . . 43
6.3 Moving to Out–of–core Data . . . 45
6.4 Indices and Data Out–of–core . . . 47
6.5 Using Irregular Distribution . . . 49
6.6 Irregular Out-of-core Distribution . . . 50

7 Java Bindings to `lip` 51
7.1 General Concepts . . . 51
7.1.1 Introduction to *Janet* . . . 51
7.1.2 Java–Specific Features . . . 51
7.1.3 Sample function wrapper in ***Janet*** . . . 52
7.2 Class Design Details . . . 53
7.2.1 Main classes . . . 53
7.2.2 Other classes and interfaces . . . 54
7.3 Sample Java Irregular Program . . . 55
7.4 Summary . . . 56

8 Functionality and Performance Evaluation **57**
8.1 Test of the Functionality of `lip` . . . 57
8.1.1 Cross–platform portability . . . 57
8.1.2 Efficiency of performance achieved by load balancing . . . 58
8.2 Out–of–core with `lip` vs Virtual Memory . . . 58
8.3 Partitioner Testing . . . 60
8.4 Parallelization of AVL Fire Benchmark . . . 61
8.5 Java Performance . . . 62
8.6 Conclusions . . . 63

9 Concluding Remarks **65**
9.1 Summary . . . 65
9.2 Future Development . . . 65

Bibliography **67**

Index **72**

A The `lip` Library Specification **74**
A.1 General Information . . . 74
A.2 Irregular Partitioner . . . 74
A.3 Data Mapping . . . 78
A.4 Communication Schedule . . . 83
A.5 Index Translation . . . 86
A.6 Communication Functions . . . 87
A.7 Miscellanea . . . 88

B The `lip` Library Specification – Additional Functions **91**
B.1 General Information . . . 91
B.2 Communication Schedule . . . 91
B.3 Indexing Scheme . . . 93
B.4 User-defined Irregular Operation . . . 94
B.5 Miscellanea . . . 95

C Papers **98**

List of Figures

1.1 An example of *irregular* problem 14
1.2 Memory hierarchy model 15

4.1 The use of `lip` in a simple parallel C program with an in-core data array and an index array. 28
4.2 Irregular inspector in `lip` 28
4.3 Use of `lip` in a simple parallel C program with irregularly distributed in-core data array and index array. 29
4.4 Irregular inspector in `lip` for OOC index arrays and in–core data arrays 30
4.5 Use of the `lip` in a simple parallel C program with in-core data array and out-of-core index array. 30
4.6 Irregular OOC inspector in `lip` 31
4.7 Simplified C code that shows the use of parallel computation on an out-of-core data array with the use of `lip` 32
4.8 Simplified C code that shows the use of parallel computation on irregularly distributed out-of-core data array with the use of `lip` 33

5.1 Two dimensional Hilbert's curve 35
5.2 Results of partitioner functions 38
5.3 A sample schedule structure 39
5.4 An example of IOBufmap structure 40

8.1 Irregular loop used for measurements of partitioner performance . . . 58
8.2 The comparison of OOC `lip` functions and virtual memory performance 59
8.3 Linux cluster system 59
8.4 Performance of the inspector and executor phases for irregular distribution 60
8.5 Performance of communication phase 61
8.6 Irregular loop from the FIRE benchmark solver 61
8.7 The solver performance 62
8.8 Irregular loops used for measuring `lip` performance 62
8.9 The comparison of performance of C and java OOC `lip` computation 63

List of Tables

8.1 Systems used for performance tests 57

Chapter 1

Introduction

This Chapter introduces the terms and concepts relevant to the idea of irregular *and* out–of–core *problems. It briefly presents related work within this area and describes the goals of this Thesis.*

1.1 Problem Outline

The common feature of *irregular* codes is that they access data arrays through one or more levels of indirect indices that are not known until runtime so no optimization can be done by the compiler. Parallelizing such code for distributed–memory systems is a challenging problem of great importance. Application areas in which *irregular* code is found include unstructured multigrid fluid dynamic solvers [10], molecular dynamics simulations [23], Monte Carlo Simulations [41], N–body Simulations [1] and many others.

An example of an *irregular* problem is shown in Fig. 1.1.

```
int i;
int edge[N];
double x[M], y[M];

create_graph(edge1,edge2, N );

for (i = 0 ; i < N ; i++)
  y[edge2[i]] += x[ edge1[i] ];
```

Figure 1.1: An example of *irregular* problem

The index arrays `edge1` and `edge2` are created during program execution (with a call to `create_graph()`) and thus their contents are not known until runtime. They are used to access data stored in data arrays `x` and `y`. When data arrays are distributed among processors, some of the indices point to the remote memory of other nodes. This results in additional communication during the calculation loop. In order to minimize communication latency, some kind of optimization needs to be performed. This cannot be done during the compilation phase as the index arrays

are not defined at this stage. This Thesis proposes a new library that provides runtime support for *irregular* problems.

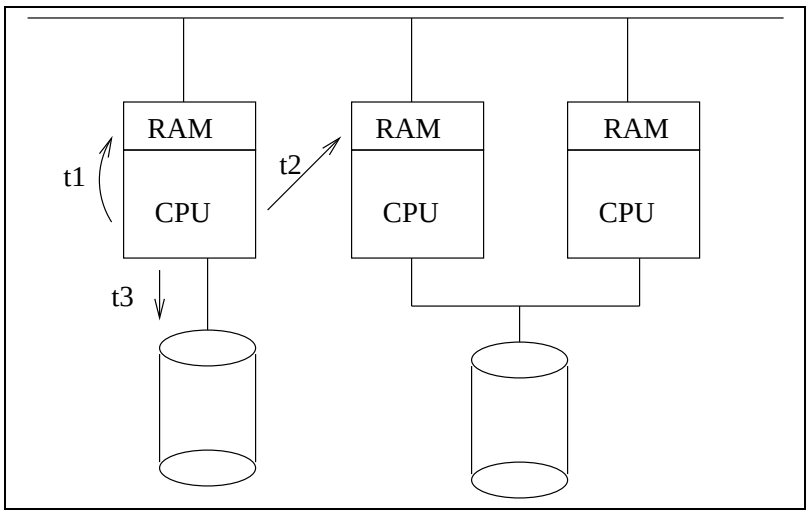

Figure 1.2: Memory hierarchy model

On the other hand, there are many large–scale parallel applications whose data sets are larger than the available primary storage and thus they are called *out–of–core* problems. Parallelizing irregular problem, in particular, often results in many additional data structures, what increases memory requirements.

The *irregular* problem presented in Fig. 1.1 becomes *out–of–core* when either array (`edge1`, `edge2`, `x` or `y`) does not fit into the main memory, so they have to be stored on disk and fetched when needed. In that case performing the calculation loop requires not only communication, but also IO operations.

From the point of view of data access costs incurred during the execution of an OOC program, a three-level hierarchy memory model can be considered (See Fig. 1.2). It is assumed that processors reference their local primary memories in $t1$ time units. Accessing data that are not in a processor's own internal memory but resides in memory of a remote processor requires communication latency of $t2$ times units. The latency caused by IO operations accessing data that reside on the local disk amounts to $t3$ time units. Accessing data stored on a disk which is attached to a remote processor requires linking the two latter cases (fetching data from disk to memory by remote processor in $t3$ time units and communication between processors in $t2$ time units).

It is assumed that $t3 >> t2 >> t1$.

This Thesis presents parallelizing techniques (based on the model described above) used to reduce communication and I/O latency by computations.

1.2 Related Work

Parallelizing *irregular* code for distributed memory systems is a challenging problem. Various methodologies for achieving load balancing as well as improvements in efficiency for *irregular* problems were designed and implemented [18, 47, 8, 3, 22]. The ideas of parallelizing *irregular* problems used in `lip` are mainly based on the

CHAOS library [47] developed at the University of Maryland for Intel Paragon machines. The CHAOS library implemented parallelization techniques such as data and work distribution based on analyzing data access patterns. It lacked, however, the cross-platform portability and support for out–of–core problems.

Other research which is worth mentioning resulted in programming libraries supporting OOC problems like Panda [48], VIPIOS [9] or Jovian library [5].

Panda is a library for I/O of multidimensional arrays on parallel and sequential platforms. Its architecture is based on server-directed collective IO which means it is distributed across computing nodes (Panda clients) and IO nodes (Panda servers) that communicate with each other during execution of the application.

VIPIOS is a parallel I/O runtime system support provided for software–generated programs produced by High Performance Fortran compilers. Its design is partly influenced by the concepts of parallel database technology.

The Jovian library was developed at the University of Maryland to optimize the I/O performance of multiprocessor architectures including multiple disk or disk arrays. The library was designed to support data mining applications as well as scientific code. Jovian uses *distributed view* which means that each process effectively requests only the part of the data structure that it requires. Using Jovian for parallelizing out–of–core codes requires also calls to communication libraries like CHAOS [47].

Several systems have demonstrated that message passing can be efficiently and portably implemented. This resulted in introducing the MPI [38, 40] standard with IO extensions called MPI–IO [40]. This standard makes use of the most attractive features of a number of existing message passing systems and thus provides portability of applications and libraries built on top of it.

To achieve I/O portability, a lot of effort was put into designing a standard for low level I/O API such as POSIX [25] and MPI–IO [40]. POSIX defines behavior of simple Unix I/O system calls, but it fails to provide optimization and portability for parallel IO systems.

Recently, issues connected with remote file access mechanisms and developing distributed file systems were considered during research carried upon High Throughput Computing (HTC) environments like Condor [4] that utilize heterogeneous resources to maximize processing capacity.

There has recently been a great deal of interest in new approaches concerning managing large data sets and data mining techniques. Among the most significant is recently approved Datagrid [19] project – funded by the European Union – to set up a computational– and data–intensive grid primarily for the analysis of data from the LHC particle collider (under construction) at CERN, the European Laboratory for Particle Physics in Geneva.

1.3 History of `lip`

The development of `lip` consists of several stages. Since CHAOS was initially based on Intel Paragon, its evolution had to take several steps including porting it to MPI [11]. This work was done by Piotr Łuszczek and Agnieszka Wierzbowska [36, 55]. The resulting library lacked efficient data distribution partitioners and support for OOC computing. Dealing with this deficiency resulted in designing of the `lip`

library. The outline of the library structure and implementation of the most basic functions of `lip` was described by Piotr Łuszczek in his M.Sc. thesis and several papers [36, 13, 12].

The review of graph partitioning methods (from approximately 400 papers) was presented by Agnieszka Wierzbowska in her M.Sc. thesis. She also proposed her own partitioner method (however, no implementation or tests were presented).

Due to the fact that Java is becoming increasingly popular in parallel and distributed computing [28], the other important approach was to provide Java bindings to the `lip` library using Java Native Interface JNI [29] as a tool for creating Java wrappers to `lip` functions.

During this work it transpired that using bare JNI is time consuming and error-prone. This resulted in creating the Java to Native Code Extension tool called ***Janet*** that was designed and implemented by Dawid Kurzyniec and presented in his M.Sc. thesis and several papers [33, 15, 16]

1.4 Goals of This Thesis

Authors of this Thesis gathered and extended the most useful ideas presented above. This resulted in a definition of goals that should be achieved to enable `lip` fully support the parallelization of the *irregular* and *out–of–core* problems. The work to be done consisted of the following stages:

- library development – the authors of this Thesis had to develop the `lip` specification to enable it fully utilize the sophisticated parallelization techniques described in next Chapters. Particularly, algorithms for OOC support and efficient data distribution needed to be designed and implemented.
- implementation – this stage required creating the full distribution of `lip` that is available on the Web [35],
- various tests using synthetic applications and real software had to be planned and performed to show that `lip` satisfies the following requirements:
  - portability guaranteed through MPI [40] low level API standard
  - multi–language support which takes into account the languages currently used in scientific computations
  - efficiency of performance achieved by load balancing and optimization techniques based on the memory hierarchy model described in Sec. 1.1
  - scalability to many computing and I/O processors

In the next Chapters the authors describe their work that was done to satisfy the requirements they defined. Finally, the evaluation of functionality and performance of the `lip` library is presented in Chapter 8.

Chapter 2

Issues in Irregular Computations

This Chapter presents various techniques for solving irregular problems. It presents and compares parallelizing techniques. Next, it outlines various optimization methods for data partitioning, work distribution and communication . In addition, the Chapter includes the description of the CHAOS [47] library which was designed to support irregular problems and serves as a basis for the solution elaborated in this Thesis.

2.1 Parallelizing Techniques for Irregular Problems

During execution of irregular loops there is a need to bring remote data into the local memory buffer. The techniques for solving the problem can be categorized as follows:

Fetch on demand – the irregularly referenced data are brought to a processor's local memory when needed. This technique is very easy to implement, but it can only be used on machines with low communication latency as it results in highly inefficient communication. Optimization of this method can be achieved by data buffering.

Inspector/Executor – this strategy involves generating two code phases for each loop specifying irregular computations. During the inspector phase indirect arrays are examined and optimized communication pattern is defined. Afterwards, the executor performs communication according to predetermined schemes and executes the actual computation.

The former method is rather easy and straightforward and it usually performs quite poorly due to the message startup cost. The latter requires a more complicated parallelizing scheme. However, in comparison to "fetch on demand" technique it produces much better results.

2.2 Optimization Methods

2.2.1 Data partitioning

Load balancing is crucial in High Performance Computing. Generally, unstructured problems solved with irregular data distributions perform more efficiently than those

using a regular data distribution such as `block` or `cyclic`. In the case of irregular distribution various partitioning algorithms were designed [43, 45, 54, 30, 7] and they can be categorized as follows:

- edge–based partitioners which use edge information to obtain data distribution patterns,
- coordinate–based partitioners assume that most of the interactions occur between vertices that are physically proximate in two or more dimensions. Usually, they provide faster but lower quality partitioning than the former methods.

The partitioning methods are usually very complex as they also take under consideration generalizations such as different weights for vertices or edges, arbitrary numbers of final partitions, specific load balance criteria or focus on even more complex cost functions. The performance of partitioning methods is highly implementation-dependent and it might be very time consuming for the user to optimize the partitioning code.

2.2.2 Work distribution

Once the data distribution is obtained, the loop iterations that access the distributed arrays must also be partitioned among processors to balance computational load and maintain locality so that communication costs are reduced. There is a large number of possible schemes for work distribution [44].

The easiest strategy is to allocate a block of contiguous iterations to processors. The main advantage of this solution is its simplicity. However, due to poor performance, this method is usually not suitable for irregular distributions of data.

Parallelizing irregular problems usually requires more sophisticated methods. The most widely used are *owner computes* or *almost owner computes* strategies which are also present in the CHAOS library [47]. Using the *owner computes* method results in assigning of the loop iteration to the processor that owns the distributed array element on the left hand side of statement. In the *almost owner computes* technique a loop iteration is executed on a processor that owns the maximum number of distributed array elements referenced in that iteration.

2.2.3 Reducing communication costs

Typically, on distributed memory machines the communication latency and message startup costs have to be taken into account. The most common techniques used to reduce the impact of communication on performance are:

- software caching that involves storing non–local data in a local memory buffer instead of performing communication each time data are needed.
- communication coalescing that aggregates all the needed data and sends them together in one message.

2.3 CHAOS Library

Solving irregular problems on distributed memory machines that was introduced in CHAOS [47] runtime support usually involves six major phases. The first four phases concern mapping data and computations onto processors. The next two steps concern analyzing data access patterns in a loop and generating optimized communication calls. A brief description of these phases follows:

2.3.1 Data distribution

Phase 2.3.1 calculates how data arrays are to be partitioned by making use of partitioners provided by CHAOS or by the user. CHAOS supports a number of parallel partitioners that return an irregular assignment of array elements to processors, which is stored as a CHAOS construct called `the translation table`. A `translation table` is a globally accessible data structure which lists the home processor and offset address of each data array element.

During investigation on various partitioning methods it was discovered that partitioners provided by CHAOS are not fully parallel and thus not very efficient. Therefore, it was crucial to design algorithms that not only satisfy mentioned requirements, but also support partitioning of out–of–core data. One of the major goals of the authors of `lip` was to design and implement data distribution functions and augment them with an inspector phase. The idea of `translation table` taken from CHAOS served as a basis for that purpose.

2.3.2 Data remapping

Phase 2.3.2 remaps data arrays from the current distribution to the newly calculated irregular distribution. The CHAOS procedure `remap` is used to generate an optimized communication schedule for moving data array elements from their original distribution to the new distribution. Other CHAOS procedures, `gather`, `scatter`, and `scatter_append`, use the communication schedule to perform data movement.

This idea was extended by the authors of this Thesis to support out–of–core remapping.

2.3.3 Loop iteration partitioning

Phase 2.3.3 determines how loop iterations should be partitioned across processors. CHAOS uses the almost-owner-computes rule to assign loop iterations to processors and also supports the *owner-computes* rule.

2.3.4 Remapping loop iterations

Phase 2.3.4 is similar to phase 2.3.2. Elements of indirect arrays are remapped to conform with the loop iteration partitioning.

2.3.5 Inspector

Phase 2.3.5 carries out the preprocessing needed for communication optimizations and index translation. The general concept that was retained in the `lip` library was the idea of a communication schedule that serves as a description of the communication pattern which should take place between computing nodes .

2.3.6 Executor

Phase 2.3.6 uses information from the earlier phases to carry out the computation and communication. Communication is carried out by CHAOS data transportation primitives which use communication schedules constructed in Phase 2.3.5.

Chapter 3

Overview of OOC Problems

As it was mentioned before, the basic characteristics of OOC applications are that the problem size is too large to fit in the main memory. This chapter presents the classification and optimization techniques for OOC problems.

3.1 Classification of OOC Problems

The models for OOC computation provide abstractions with which a runtime system or a compiler does not have to deal with machine specific features and low–level APIs. The three most common execution models [21] are described below.

- In the *global placement* model a file is stored on a single virtual disk and is shared between all processors.
- In the *local placement* model, the out–of–core local array of each processor is stored in separate logical file. A file owed by some other processor cannot be explicitly accessed and requires communication with the owner that performs IO operations.
- in the *partitioned in–core* model the global data array is logically divided into a number of *partitions* each of which can fit in the combined main memory of the processors. Thus computation on each partition is primarily an in–core problem and no I/O is required during the computation of the partition.

Additionally, irregular OOC problems can be divided with respect to the type of arrays which are OOC [8].

- *out–of–core index and in–core data arrays*; usually the size of indirectional arrays is greater than that of corresponding data arrays . In some cases, data arrays are small enough to be kept in the main memory.
- *out–of–core index and data arrays*; In the general version of OOC problems there is an assumption that both types of arrays are out–of–core.

3.2 Tools for OOC Computing

The need for high performance I/O resulted in designing various tools on hardware and software levels. Their main groups can be categorized as follows:

- *Hardware support* – addresses small scale parallelism and does not directly address the important software issues. An example of such technology is RAID (Redundant Arrays of Inexpensive Disks) system [20] .

- *Parallel file systems* – provide interface to high performance secondary storage and mechanisms of remote file access. They are a significant part of High Throughput Computing (HTC) environments [4].

- *Virtual memory* [2, 17] (also referred to as paging mechanism) allows the user to use a larger memory system than what is physically installed in the system. This is achieved by storing parts of memory contents (memory pages) on the disk and fetching them as needed. The swapping heuristics tends to be as general as possible to handle a wide range of applications. This technique is very convenient as it is transparent to the user. However, it does not take into account special application requirements. In addition, many OOC applications require much more space than is guaranteed by the paging mechanism.

- *Low level APIs* – (Application Programming Interface) are sets of low–level routines usually provided by the system. They allow the user to explicitly handle the OOC data and result in more efficient, but also less portable code. To achieve portability various API standards were designed. A well–established standard in the scientific community is MPI–IO [40] – an IO extension to the MPI standard which is particularly important in this thesis as the `lip` library heavily depends on it.

- *Programming libraries* deal with system–specific details and therefore enable a user to use the available I/O resources efficiently. Examples of such solutions include aforementioned Panda [48], Jostle [54] and ViPIOS [9].

3.3 Optimization Techniques

OOC problems depend heavily on external means of manipulating great amounts of data, namely, secondary (for example hard drivers) and tertiary (for example tape libraries) storage devices. These applications require a highly efficient I/O system. To achieve this goals various optimization techniques were devised [52, 50]. The most significant ones are described in this Section.

3.3.1 Data reusing

This technique is basic for IO access optimization and is used in every operating system (disk caching).

Instead of accessing data on a disk whenever needed, a copy of the data is stored in a memory buffer. The efficiency of this method relies strongly upon the strategy of buffer allotment. Various algorithms implemented in the Unix kernel were presented in [2].

This method avoids performing an unnecessary number of disk accesses. The idea described above was also utilized in the `lip` library and will be described in the following Chapters.

3.3.2 Data sieving

Numerous studies of the I/O characteristics of parallel applications have shown that many applications need to access a large number of small, noncontiguous pieces of data from a file [52]. For good I/O performance, however, the size of an I/O request must be large (in the order of megabytes). The I/O performance suffers considerably if applications access data by making many small I/O requests. What is more, to reduce the effect of high I/O latency, it is critical to make as few requests to the file system as possible. Due to the reasons mentioned above, an optimization technique called *data sieving* was introduced.

Instead of accessing each contiguous portion of the data separately, a single contiguous chunk of data starting from the first requested byte up to the last requested byte is read into a temporary buffer in memory. Additionally, the requested portions of data are extracted and placed in the user's buffer.

A potential problem with this simple algorithm is its memory requirement. The temporary buffer into which data is first read must be as large as the total number of bytes between the first and last byte requested by the user. To avoid this problem, the technique described above is performed in parts, reading only as much data at a time as defined by a special parameter.

The advantage of data sieving is that data is always accessed in large chunks, although at the cost of reading more data than needed. For many common access patterns, the holes between useful data are not unduly large, and the advantage of accessing large chunks far outweighs the cost of reading extra data. In some access patterns, however, the holes could be so large that the cost of reading the extra data outweighs the cost of handling such cases as well. The algorithm can analyze the user's request and decide whether to perform data sieving or access each contiguous data segment separately. Data sieving can similarly be used for writing data.

The `lip` library is based on the MPI–IO standard and tested on its portable implementation called ROMIO [53] which implements the technique described above by means of MPI derived data types [51].

3.3.3 Collective I/O

In many parallel applications, despite the fact that each process may need to access several noncontiguous portions of a file, the requests of different processes are often interleaved and may together span large contiguous portions of the file. Thus, I/O performance can be improved significantly by merging the requests of different processes and servicing the merged request. Such optimization is broadly referred to as collective I/O and can be performed at the disk level (disk-directed I/O [32]), at the server level (server-directed I/O [48]), or at the client level (two-phase I/O [50]).

Two phase I/O is also used in the ROMIO implementation [52]. If the entire I/O access pattern of all processes is known, the data can be accessed efficiently by splitting the access into two phases. In the first phase, processes access data assuming a distribution in memory that results in each process making a single, large, contiguous access. In the second phase, processes redistribute data among themselves to the desired distribution.

The advantage of this method is that by making all file accesses large and contiguous, the I/O time is reduced significantly. The added cost of interprocess com-

munication for redistribution is small compared with the savings in I/O time.

The algorithm for collective writes is similar to that for collective reads, except that the first phase of the two-phase operation is communication and the second phase is I/O.

3.3.4 Data prefetching

While dealing with OOC problems, data is divided into chunks small enough to fit into memory. After reading desired amount of data, the required calculations are performed and the results are written back to the disk. Then, operations of reading, calculating and writing are performed on another chunk of data.

The optimization strategy is based on the idea of non-blocking accesses to a disk. In this solution, the IO operations can be performed concurrently with calculations, so the next chunk of data can be read while performing the calculations on the previous one [49].

Chapter 4

The Concept of `lip`

The ideas described in the previous Chapters served as a basis for the design of the `lip` *library. This Chapter explains solutions used to satisfy the requirements described in Section 1.4 and gives the main idea of support for both in– and out–of–core irregular problems.*

4.1 Runtime Support Requirements and their Solutions

4.1.1 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.

4.1.2 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 7.

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.

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 described in Section 3.3.2.

4.1.4 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.

4.1.5 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 which is described in the Chapter 9.

4.2 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.

4.2.1 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] and described in Chapter 2. A piece of pseudocode using this technique is shown in Fig. 4.1

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

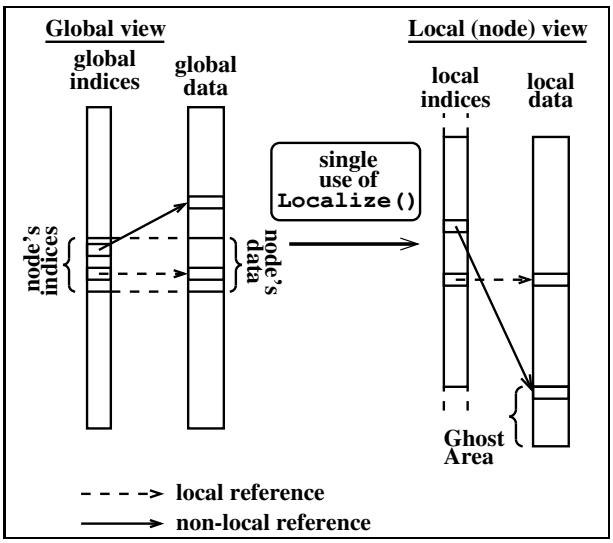

Figure 4.2: Irregular inspector in `lip`

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 desired data into *ghost area*. The *Inspector* is activated using LIP_localize() call and its idea is shown in Fig. 4.2.

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.

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

4.2.2 Irregular data distribution

In order to achieve better load balancing and to minimize the communication phase, various techniques for creating data distribution were presented as described in Chapter 3.

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. 4.3 shows pseudocode that uses the partitioner.

The most significant difference between Fig. 4.1 and Fig. 4.3 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.

4.3 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. As described in Section 3.1 out–of–core problems can be divided taking into account the types of OOC arrays.

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

The pseudocode in Fig. 4.5 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

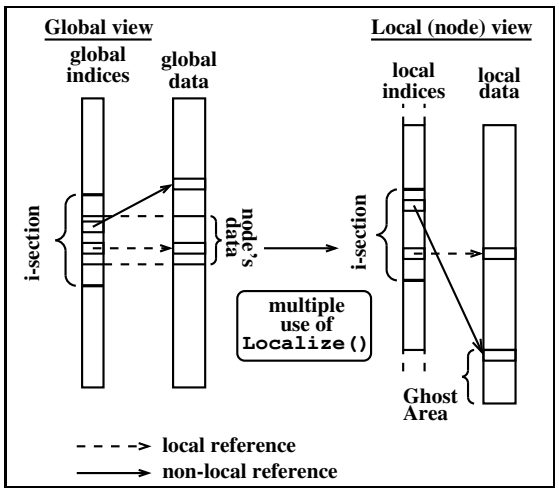

Figure 4.4: Irregular inspector in `lip` for OOC index arrays and in–core data arrays

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

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. 4.4. 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.

4.3.2 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:

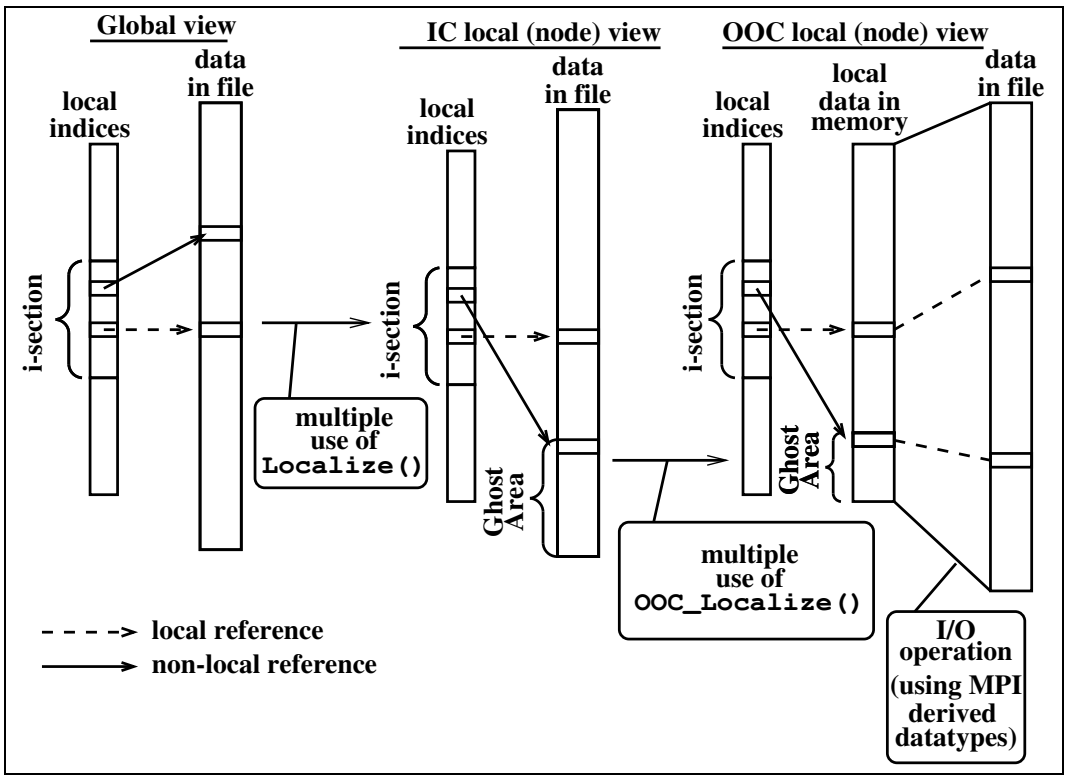

Figure 4.6: Irregular OOC inspector in `lip`

- 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. 4.6. The respective pseudocode is shown in Fig. 4.7. 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. 4.6. 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 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.

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

4.3.3 Irregular distribution

As in the case of in–core data (Subsection 4.2.2), 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. 4.8. 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. 4.7.

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

Chapter 5

Implementation Details

The previous Chapter presented the `lip` *library from the user's point of view and this Chapter outlines a more detailed description of the algorithms and data structures provided by* `lip` *for data distribution, communication and IO operations phases used to perform the computational phase similarly to its sequential version without losing efficiency. The* `lip` *library specification is presented in Appendix A*

5.1 Data Distribution

As it was mentioned before, the distribution of data among processors is a significant issue while developing parallel applications. The `lip` library uses a data structure called *datamap* in order to store the information about data mapping. It also provides a set of functions for creating and manipulating the *datamap* objects for both regular and irregular distributions.

5.1.1 Data structures

The *datamap* objects are used to determine the actual position of data from the given global index. This issue is not very complicated for regular distributions (such as block and cyclic) as only the size of the data array and the number of processors is needed for performing the desired calculation. However, the case of irregular distribution requires explicit addresses of data elements. The addresses are stored in a special object called *translation table* which is a part of *datamap* structure. For each data element, *translation table* stores the number of the processor that owns the element and the local index on that processor.

5.1.2 Functions

Datamap manipulation

In the case of regular distribution `lip` provides sets of functions to create and manipulate *datamap* objects described above. These functions do not require complex algorithms. A more detailed description can be found in appendix A.

Data partitioners

Dealing with irregular distributions requires more sophisticated algorithms for data partitioning. The `lip` library supports in–core and out–of–core versions of the partitioner, but both of them are based on the same idea taken from [55] and described in this section. The language–independent description as well as C and Java syntax of the functions can be found in appendix A.

The algorithm assumes that most interactions occur between vertices that are physically proximate in two or more dimensions. The main idea is to use the continuous space–filing curve which passes through every point in a space or cube. One of them, proposed by David Hilbert, was used in the described partitioner to index pixels in a two– or three–dimensional grid and assign the sorted data to the processors in the order determined by that curve.

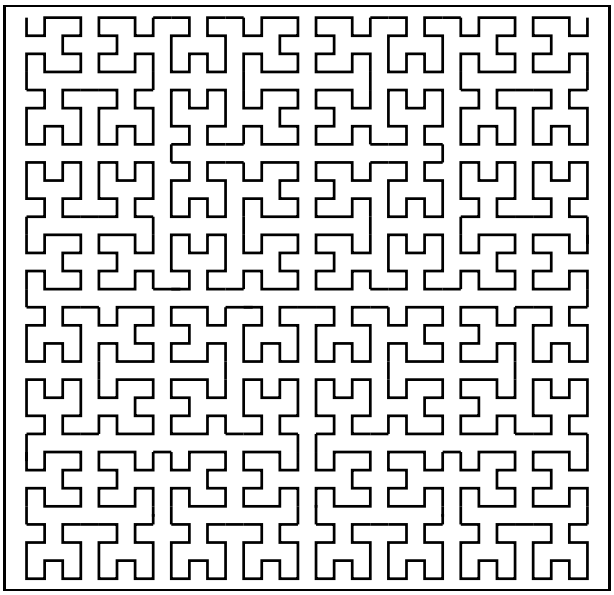

Figure 5.1: Two dimensional Hilbert's curve

The two–dimensional Hilbert's curve is shown in the Fig. 5.1. The detailed description of the curve itself and the indexing scheme can be found in [55].

To sum up, data partitioning can be described as follows:

1. Each processor randomly picks a sample of elements from a set of data and determines its Hilbert's index.

2. A parallel merging sort is performed to sort the sampling elements.

3. The sorted sampling list is divided into p equal parts, when p is number of processors, and the maximum elements in each partial list are recorded to form boundaries for all processors.

4. The bucket sorting algorithm is used to sort the whole set of data. Each processor calculates the new owner of each data element it currently owns with respect to the boundaries determined in the previous step and puts the element into the appropriate bucket assigned to that owner. It also fills the *translation table* structure with that information.

5. Each processor sends its buckets to their owners and receives its own buckets.

6. Each processor calculates the number of elements in its bucket, compares it to the number required by the appropriate load balancing strategy and sends this information to the other processors.

7. The processors determine how many data elements should change owners in order to make a correction. Appropriate indices are chosen and the translation table is updated.

All steps are performed in linear time and space, that is, with O(n) time and space complexity. The main advantage of this algorithm is its scalability. It also allows the user to control the size of temporary data structures created during execution. However, the presented method requires special knowledge about the physical distribution of data and therefore cannot be used by some irregular applications. This issue should be considered in the future.

Partitioners are activated using `LIP_create_hilbert_distribution_xxx(...)` calls, where the suffix "`xxx`" can be either "`in_core`" or "`ooc`" depending on whether data can fit into memory or not.

Both functions take as input current distribution described by *datamap* object, the coordinates of the data and the parameters describing a two– or three–dimensional area within which data are physically distributed (determined by the bottom-left *vertex* and the *edge*). A *minedge* parameter determines the resolution for the indexing scheme that means that if the distance between data elements is shorter than *minedge* they will be assigned the same Hilbert's index. The *sample count* parameter determines the size of the sample. For the OOC partitioner the page size must also be specified. The functions result in an updated *datamap* object that describes the new distribution. The *translation table* stored in the object contains information about the new owners of the data elements.

Data remapping functions

The full description of the new distribution also requires the local addresses of data elements. What is more, the data should be physically moved. The `lip` library supports the remapping functions designed for that purpose.

In the case of in–core problems the described routine does not move data. Instead, it calculates the *communication schedule* (section 5.2) and lets the user use the appropriate communication functions. In the case of OOC problems it was decided to leave the decision whether to move data to the user via a specific flag (appendix A).

Both routines take as an input the current distribution described by *datamap* object and the desired new distribution described in the partially filled *translation table* stored in the other *datamap* object. The *translation table* must contain information about new owners of data elements and is usually obtained from one of the partitioner functions described above.

The information about which data should be exported could be obtained from the translation table without any communication. However, to obtain the full communication pattern and new local addresses of data elements this information must be exchanged between nodes.

The algorithm of the in–core version can be summarized as follows:

1. Each processor calculates the number of elements that should be exported to the other nodes.

2. The information obtained in the previous step is exchanged so that each processor can determine new local addresses of each data element and the information about imported data.

3. the translation table is updated and the communication schedule used for data remapping is created as well

The algorithm described above can also be implemented in the OOC version. Therefore the proposed steps would have to be repeated for all pages of the translation table. This would result in many schedule structures (one for each page) that should be stored on a disk. Instead, we have decided to use MPI–IO functions to perform the actual data movement. The decision weather to move data, calculate local indices or perform both was left to the user via a specific flag.

The full OOC algorithm consists of the following steps :

For each page of the translation table :

1. read how many data elements should be owned by other processors

2. exchange this information with other nodes

3. if the flag for calculating local indices was specified, calculate the new local addresses of data elements and write them in the translation table on a disk

4. if the flag for data movement was specified the following steps are performed:

   - read data from the file containing the old distribution
   - calculate the new global addresses of data elements and write data in the file containing the new distribution

The functions are activated via `LIP_remap_xxx(...)` calls, where the suffix "`xxx`" can be either "`in_core`" or "`ooc`" depending on the type of the problem. The results of data distribution functions are shown in the Fig. 5.2. In this figure, data elements A, B and C were initially assigned to processor 1, while D, E and F to processor 2 and G, H and I to processor 3. Each data element has its own Cartesian coordinates that determine its position within a rectangle. The `LIP_create_hilbert_distribution` routine changed this assignment, so each processor owns elements that are close to each other. The new assignment is stored in the *translation table*, a part of the *IOBufmap* structure as shown in the Fig. 5.2.

After the new assignment is created, one can call remapping functions in order to actually redistribute data. Fig. 5.2 shows the data array before and after redistribution.

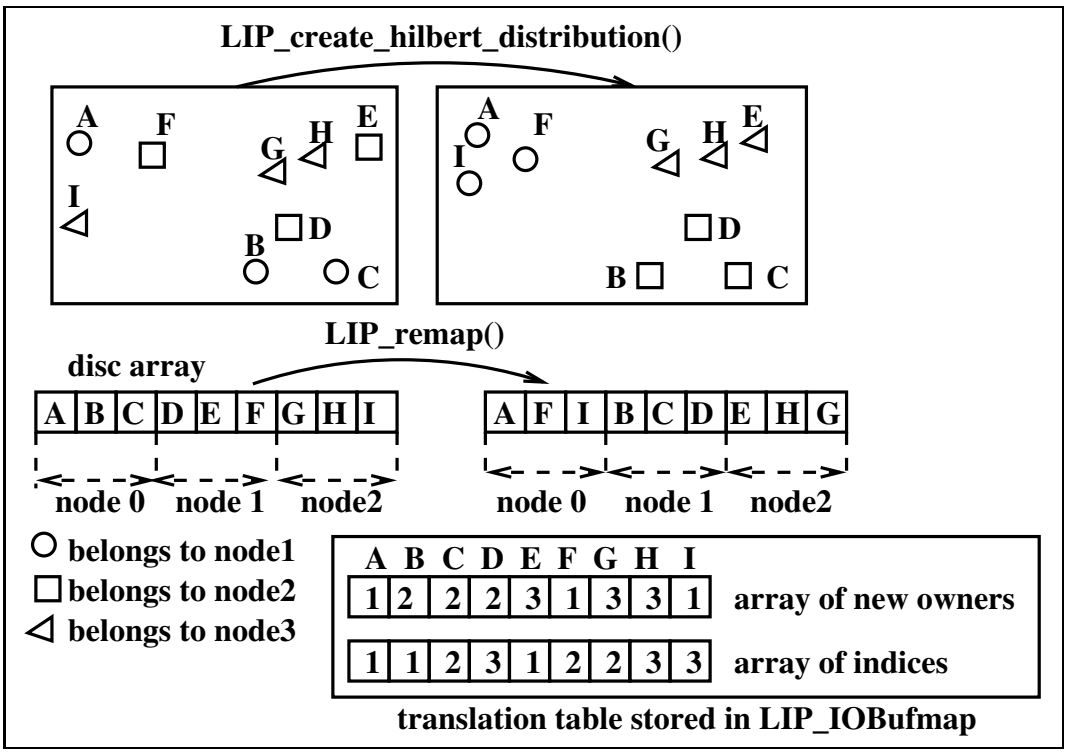

Figure 5.2: Results of partitioner functions

Implementation remarks

The OOC functions described in this section were implemented using the MPI–IO routines and assume the global placement model (Chapter 3) as they are using collective IO and shared file pointers [40, 53]. However, there is also a need for an implementation design for the local placement model (i.e. in case of local disks that are only accessible by one node). This issue should be considered in the future.

5.2 Communication

5.2.1 Data structures

One of the main goals of the inspector phase is creating the communication pattern stored in the *schedule* object. The idea of the *schedule* object is shown in Fig. 5.3. To minimize communication overhead, the memory data buffer is extended in order to store non–local data imported from other nodes. This part of the data buffer is called the *ghost area*. The schedule object consists of the information which elements of data should be exported to other nodes and how many elements should be imported to the ghost area from other nodes. The structure has to be committed before being used.

5.2.2 Functions

`lip` provides set of functions for creating and managing *schedule* objects. Their syntax is described in Appendix A and Appendix B. The main two groups of

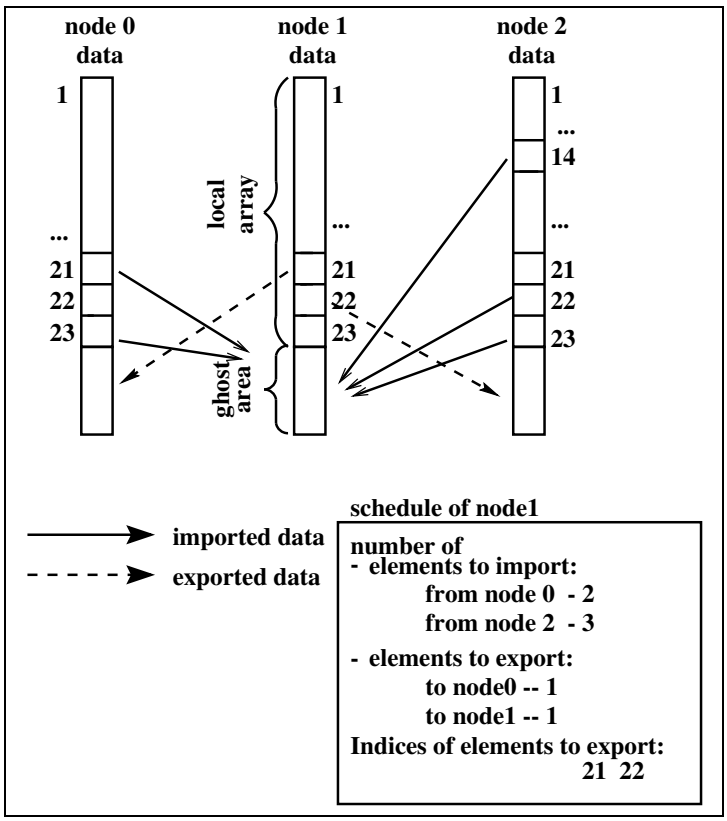

Figure 5.3: A sample schedule structure

functions are presented below.

Index translation

Two routines: `LIP_localize()` and `LIP_OOC_localize()` actually fill the *schedule* with appropriate information. Both functions are used for index translation. The former analyzes their globally numbered indices and transforms them into locally numbered counterparts. It also updates the communication schedule so it can be used to transmit non-local data. A more detailed description of the algorithm can be found in [36].

The latter function transforms indices which point to data residing on a disk into indices pointing to the memory data buffer. During this process the communication schedule is altered. This function is fundamental for optimizing the IO phase and thus is described in the next Section.

Communication routines

The `lip` library also includes two general communication functions: `LIP_Gather()` and `LIP_Scatter()`. They perform collective communication between nodes before and after the computational phase by means of the *schedule* object.

5.3 IO Operations

5.3.1 Data structures

The last structure described in this chapter is the I/O buffer mapping object called *IOBufmap*. Fig. 5.4 shows the disk and memory data arrays. Elements A, C, E

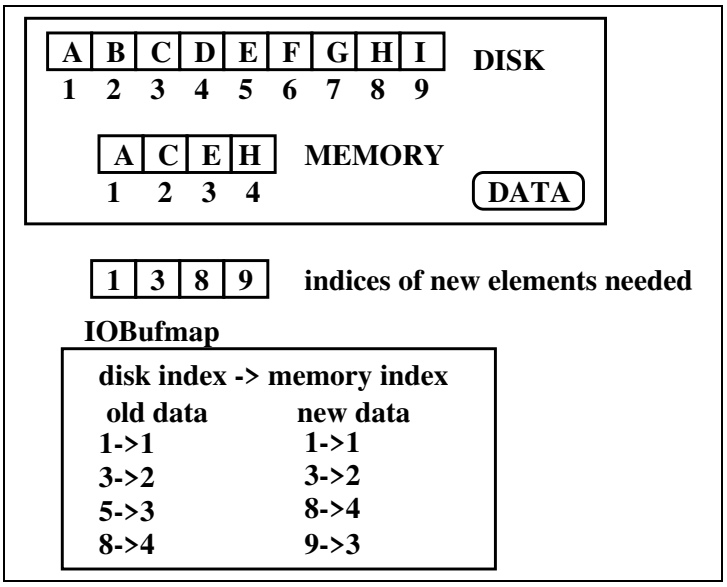

Figure 5.4: An example of IOBufmap structure

and H reside in memory. The new elements needed for the next i–section are also shown. These are A, C, H and I. The *IOBufmap* structure stores the assignment of positions (indices) in the memory to the positions on the disk as illustrated in the described figure. This information may be used to obtain the MPI derived data types for moving data between disk and memory. The structure of *IOBufmap* object is based on hash tables [26], that is, data structures consisting of (key,value) pairs. The position in a table is determined by a key, so the searching algorithm does not depend on the size of the table.

IOBufmap contains two hash tables storing information about memory contents before and after the communication phase. The keys for *IOBufmap* hash entries are defined by global indices of data elements and the appropriate values determine the memory data buffer positions of these elements.

5.3.2 Functions

As in the case of other structures `lip` contains set of functions to manipulate the *IOBufmap* object. All of the routines are described in appendix A. The function that actually fills the described data structure is `LIP_ooc_localize()` which translates the indices so they point to a contiguous in-core data buffer afterwards (beforehand they use an out-of-core data buffer). The algorithm is based on hash tables described in the previous section and works as follows:

1. For each index element reserve the entry in the hash table responsible for new memory contents and, if it has entry in a hash table responsible for old memory contents copy that entry to the new hash table.

2. For each new element needed find a free place in the memory buffer (step 1 determines which elements should stay in the buffer and which can be replaced) and complete the appropriate entry in the new hash table

Those steps are performed not only for indices stored in the current i–section of the indirect array (Chapter 4), but also for indices of data elements that are needed by other nodes and thus have to be read from disk and exported (the appropriate information is stored and updated in the *schedule* object).

The routine `LIP_IOBufmap_get_datatype()` is used to obtain the MPI derived data types for moving data between disk and memory. The algorithm is based on the comparison of the two hash tables stored in the *IOBufmap* structure .

Chapter 6

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.*

6.1 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.

6.2 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. 4.2 in Chapter 4. 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 );
```

6.3 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_1, LIP_INDEXTYPE_INT, 1, 0,
  &datamap );
/* create IO bufmap which maps local data buffer on the data array
 * on disk */
LIP_IOBufmap_create(n_1, &bufm);

LIP_Schedule_create( l_1, &schedule );

LIP_Localize( datamap, perm, LIP_INDEXTYPE_INT, perm_1, LIP_INDEXTYPE_INT,
  n_1, 0, &schedule, LIP_INDEXINFO_NULL );

LIP_OOC_localize(perm_1, LIP_INDEXTYPE_INT, tperm_1, LIP_INDEXTYPE_INT,
                n_1, 0, x_1, 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_1*rank*8,MPI_DOUBLE,filetype,"native",info);
MPI_File_read_at(file,0, x_1, 1, memtype, &status);

LIP_Gather( x_1, x_1 + n_1, MPI_DOUBLE, schedule );
```

The irregular loops are computed using the `tperm_1` array.

```
for (i = 0 ; i < n_1 ; i++)
  y[i] = -x_1[ tperm_1[i] ];
for (i = 0 ; i < n_1 ; i++)
  x_1[ i + n_1 ] = 0.0;
for (i = 0 ; i < n_1 ; i++)
   x_1[tperm_1[i] ] += y[i];

LIP_Scatter( x_1 + n_1, x_1, 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_1*rank*8,MPI_DOUBLE,filetype,"native",info);
MPI_File_write_at(file,0, x_1, 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.

6.4 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 4.6 in the Chapter 4.

```
  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`.

6.5 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, 1, &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.

6.6 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_1, LIP_INDEXTYPE_INT, 1, 0,
    &datamap );

LIP_create_hilbert_distribution_ooc(datamap, &datamap1, fx, fy, fz,
l_1, ps, sample, vertex,edge,minedge);

LIP_remap_ooc(datamap,&datamap1,file1_reg,file1,l_1, isec_l_1,MPI_DOUBLE,
              LIP_CREATE_LOCAL | LIP_MOVE_DATA);

LIP_remap_ooc(datamap,&datamap1,file1_reg,file2,l_1,isec_l_1,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 7

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.*

7.1 General Concepts

7.1.1 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.

7.1.2 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.

7.1.3 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 “ ‘ ”.

7.2 Class Design Details

According to Java–specific features described in Sec. 7.1 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.

7.2.1 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;

7.2.2 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.

7.3 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 6.6. 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(1, 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 re-

places 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;
```

7.4 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.

Chapter 8

Functionality and Performance Evaluation

This Chapter presents tests of the `lip` *functionality. It shows that the presented library satisfies the requirements specified in the Chapter 1.*

8.1 Test of the Functionality of `lip`

8.1.1 Cross–platform portability

The `lip` performance was measured on various parallel architectures. To test portability of the library, `lip` was ported to following systems:

- cluster of PC workstations running Linux
- cluster of SUN SPARC workstations running Solaris
- SGI Origin 2000
- HP Exemplar S2000

The clusters used to test library's performance are specified in Tab. 8.1.

System	CPU	Memory	No. of nodes
Linux	Intel Celeron 600MHz	64 MB	8
SUN	Micro SPARC II 110MHz	160 MB	6
SUN	Ultra SPARC-IIi 440MHz	256 MB	6

Table 8.1: Systems used for performance tests

`lip` requires MPI with MPI-IO extensions. A free MPI-IO implementation called ROMIO can be downloaded from the Web [53]. To take full advantages of the `lip` library, ROMIO ver. 1.0.2 or greater is required since it implements all the needed MPI-IO standard functions (including shared file pointers).

Cross–language portability

The library was written in C programming language according to ISO C standard specification [24]. However, it is also possible to use it from Java. This was achieved using wrappers generated by ***Janet*** tool described in the Chapter 7. The performance tests of `lip` routines called from Java are presented below.

8.1.2 Efficiency of performance achieved by load balancing

`lip` supports advanced load balancing strategy. The tests of the partitioner algorithm showed the usefulness of non-trivial data distribution that reduces communication phase of the application and are described in the Section 8.3.

Scalability

An important aspect of performance analysis is the study of how algorithm performance varies with parameters such as problem size and processor count. Large parallel computers are frequently used not only to solve fixed-size problems faster, but also to solve larger problems. This Chapter presents benchmarks including fixed problem size and scaled problem size to show `lip` performance in both cases.

8.2 Out–of–core with `lip` vs Virtual Memory

To test `lip` performance we considered the following problem. The data structure was a directed graph with vertices geometrically distributed on a plane square. The loop goes through all the edges so it accesses the data located in the vertices in indirect way. The code structure of the test program is similar to that in Section 6.6 and the kernel loop of this program (computational phase) is shown in Fig. 8.1.

```
for(i=0;i<edge_counter;i++)
{
    x2[edge2[i]]+=x1[edge1[i]]/n_succ;
    x2[edge1[i]]-=x1[edge1[i]];
}
```

Figure 8.1: Irregular loop used for measurements of partitioner performance

The experiments were carried out on a cluster of 8 LINUX workstations connected by 100Mb Fast Ethernet with Intel Celeron 600 MHz processors, 64 MB RAM and 256 MB swap size. For out–of–core data exchange each node was using a local disk (see Fig. 8.3) instead of one filesystem globally available via NFS, which resulted in good scalability.

The ratio of the data size to virtual and real memory sizes was constant (1,300,000 vertices per processor), so the number of all data elements was proportional to the number of nodes. In addition, the size of the application grew with the

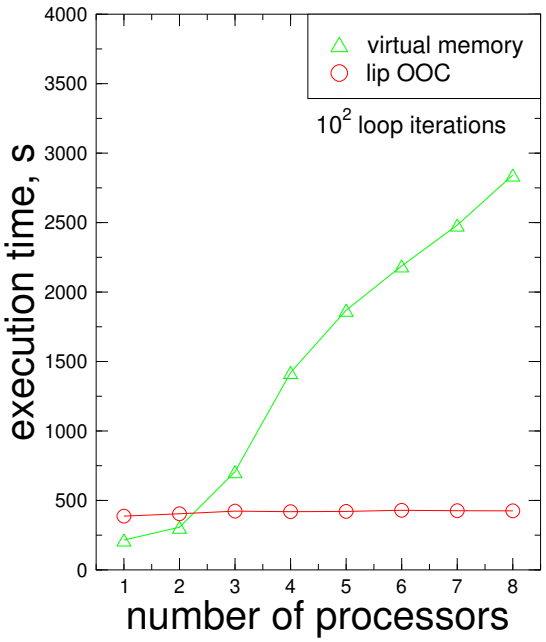

Figure 8.2: The comparison of OOC `lip` functions and virtual memory performance

size of the schedule structure and consequently the number of nodes. This increase did not effect performance of ooc lip functions as the data were divided into chunks (16 in our example) small enough to fit in main memory. As can be seen in Fig 8.2, the execution time does not increase with the number of data elements.

However, the tests that use virtual memory show significant decrease of performance with the increasing size of the application. The reason for it are frequent IO operations during the calculation phase due to more general algorithms used for the implementation of disk data accesses.

To make the computational phase more time consuming, the loop depicted in Fig. 8.1 was executed 100 times.

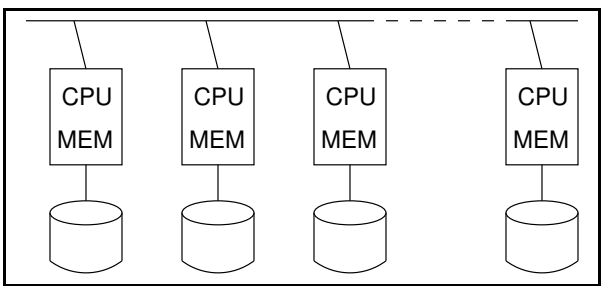

Figure 8.3: Linux cluster system

8.3 Partitioner Testing

To test the coordinate-based partitioner, we assigned the Cartesian coordinates to the vertices of the graph used in the previous example, so the distance between the adjacent vertices was not greater then five percent of the distance between the farthest vertices.

The experiments were carried out on a cluster of 6 workstations connected by 100 Mb Fast Ethernet network. Each node was a SUN SPARCstation 4 with a 110MHz processor and 160MB RAM running Solaris 8.

Applying the partitioner should result in such a distribution that most references to neighboring vertices are local and do not require communication.

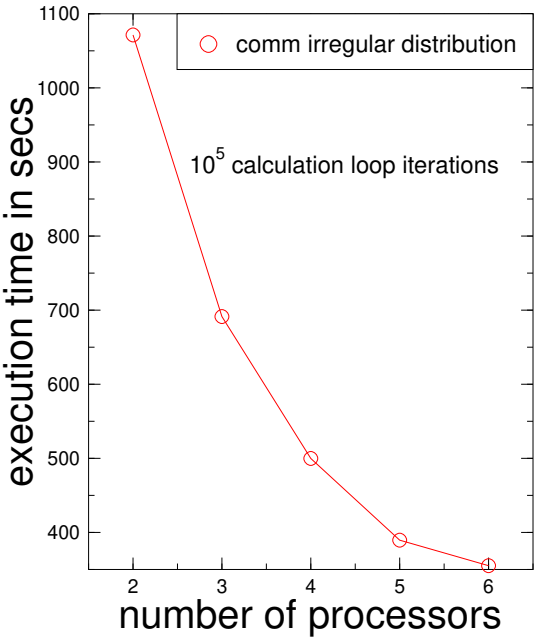

Figure 8.4: Performance of the inspector and executor phases for irregular distribution

Fig. 8.4 shows the execution time of the inspector and executor phases of the program that uses irregular distribution for a graph with 12,000 vertices.

Fig. 8.5 shows the difference of communication phase performance between when irregular and block distributions.

When the irregular distribution is used, more of the graph's adjacent vertices reside on the same processor so less communication is performed during the executor phase. Due to side–effect communication the irregular inspector is more time consuming. However, if the application requires performing the I/O phase and the communication phase many times using data corresponding to indices localized once during the inspector phase, the irregular distribution performs better than the block one. The results in the Fig. 8.5 show the communication phase duration for 10^4 executor loops.

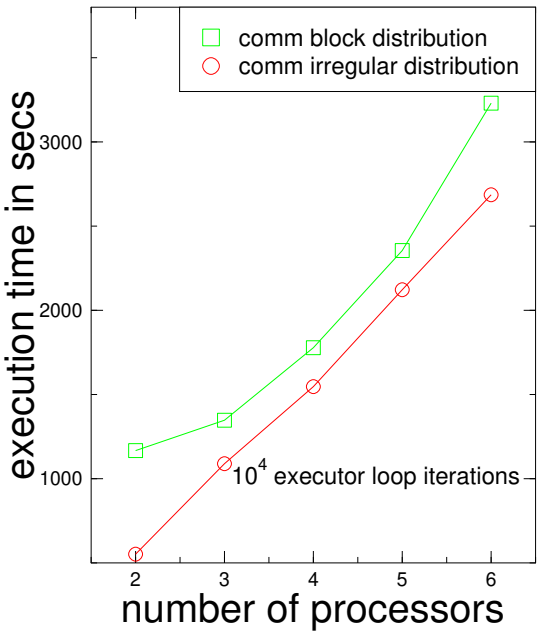

Figure 8.5: Performance of communication phase

8.4 Parallelization of AVL Fire Benchmark

FIRE [10] is a general purpose computational fluid dynamics (CFD) software developed at the AVL company in Graz, Austria. It was developed for computing turbulent fluid flows encountered in engineering. As a result of discretization a set of nonlinear equations is obtained. The sparse matrices that have to be solved are stored in a linear array and are accessed by indirect addressing. The irregular loop used in the computation kernel is shown in Fig. 8.6.

```
for(nc=0;nc<nintcf;nc++)
{
 direc2[nc]=bp[nc] * direc1[nc]
   -bs[nc] * direc1[lcc[6*nc]-1]
   -bw[nc] * direc1[lcc[6*nc+3]-1]
   -bl[nc] * direc1[lcc[6*nc+4]-1]
   -bn[nc] * direc1[lcc[6*nc+2]-1]
   -be[nc] * direc1[lcc[6*nc+1]-1]
   -bh[nc] * direc1[lcc[6*nc+5]-1];
}
```

Figure 8.6: Irregular loop from the FIRE benchmark solver

We have parallelized the GCCG benchmark solver from FIRE using the `lip` library. The parallelization scheme from Fig. 4.5 was used with the irregular code.

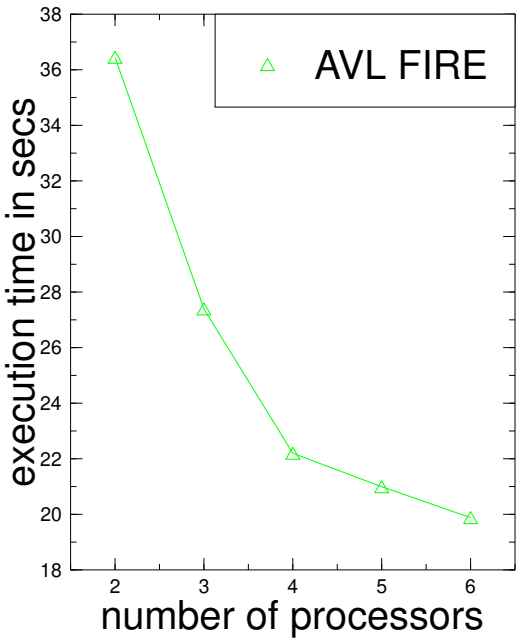

Figure 8.7: The solver performance

The tests were performed on a dataset consisting of 13845 cells (size of `direc2` array). The data was distributed block-wise and all arrays were located *in-core*. The tests were performed on a cluster of 6 workstations connected by 100 Mb Fast Ethernet network. Each node was a SUN Ultra 10 with an UltraSPARC-IIi 440MHz processor and 256MB RAM running Solaris 8. The results of the tests are shown in Fig. 8.7. They show the scalability of the program taken from a real–life computation problem.

8.5 Java Performance

```
for(i=0;i<edge_counter;i++)
{
    y[i] = -x[ perm[i] ];
}
for(i=0;i<edge_counter;i++)
{
    x[ perm[i] ] += y[i];
}
```

Figure 8.8: Irregular loops used for measuring `lip` performance

To test the performance of Java bindings to `lip` library we used a generic irregular loop with data located out–of–core. The code is shown on Fig. 8.8. The data array

consisted of 113,400 cells with double precision values. The experiments were carried out on our Linux cluster (see Tab. 8.1). For out–of–core data exchange each node was using its local disk drive.

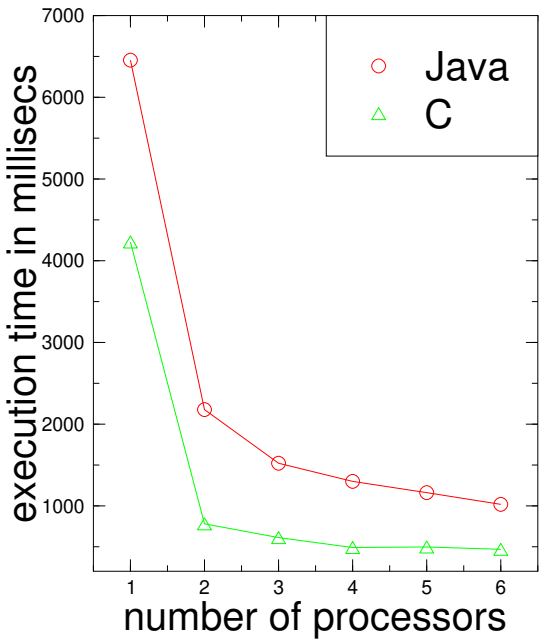

Figure 8.9: The comparison of performance of C and java OOC `lip` computation

The results presented on Fig. 8.9 show the scalability for both C and Java versions of the program.

8.6 Conclusions

The results obtained by the authors show the following features of `lip` library:

- **Portability** As the library was ported to many different systems available for the authors, it seems to fulfill the portability design goal. Based on the experience with installing the library on these systems, no crucial problems should be expected in moving to other environments which support MPI with I/O extensions.
- **Scalability** The tests performed on different kinds of irregular problems show the possibility of successful parallelization of such codes. They also show the `lip` scalability for fixed and scaled problem size. The results were achieved not only for artificial and simplified benchmarks, but for real scientifical codes as well.
- **Advantages of data partitioning** In more complicated cases the possible advantages of data partitioning and irregular distribution have been shown.

- **Usefulness in ooc case** The advantages of `lip` over virtual memory–based approach to dealing with big datasets were proved in the tests.
- **Java bindings** The possibility to create Java bindings that achieve reasonable computation time results was examined and confirmed.

Chapter 9

Concluding Remarks

This Chapter summarizes the ideas presented in this Thesis. It also describes issues that should be considered in the future.

9.1 Summary

The design of the `lip` library presented in this Thesis was based on several studies of parallelization of in-core and out-of-core irregular applications for distributed-memory systems. The library is intended to provide high performance for a wide variety of applications. According to the classification of irregular problems [44] `lip` can be easily used for parallelization of static, multiphase and adaptive in– and out–of–core irregular problems.

The most important features of the `lip` library are its portability, scalability and flexibility for interfaces built on top of it in the future. It provides an easy to use and clear set of API functions which are similar to other solutions found in the field. The cross–platform portability is guaranteed by the MPI standard which is the underlying layer of `lip` library. It is also easy to use `lip` in other languages such as Fortran and Java by means of wrapper functions. The Java bindings were obtained using the Java to Native Code Extensions translator, which was another project of the Institute of Computer Science and is currently developed in cooperation with Dawid Kurzyniec at the Department of Math and Computer Science of the Emory University in Atlanta, Georgia, USA, under supervision of Prof. Vaidy Sunderam.

The library was tested on various parallel architectures with synthetic and real applications which includes computational fluid dynamics (CFD) software devaloped at AVL company in Graz, Austria.

9.2 Future Development

Appendix A describes the set of functions of `lip` library that was presented in this Thesis. However there are some additional routines of which implementation should be considered in the future to enhance `lip` functionality and which include a support for the general indexing scheme, user defined operations and some additional routines used to retrieve information stored in the *schedule* structure.

A general indexing scheme was introduced to make `lip` more flexible not only for the message passing code written directly by the user, but also for an automatically

generated code resulting from a data-parallel language compiler. This feature makes it possible to work with index arrays that have a more complex memory layout than simple integer arrays. This is particularly important for the data–parallel language compilers as they may produce intermediate code with non–contiguous indirect indices.

Additionally, the possible user-defined irregular operations should be implemented to allow adapting `lip` to particular problem.

Last, but not least, additional functions used to obtain information stored in the *schedule* structure. Implementation of these routines would be helpful to the user in making full use of the library.

These routines are described in the other part of `lip` specification presented in Appendix B.

Further research work is also still possible in the field of data and work partitioning which comes as the most common solution for load balancing. Additionally, other data distributions like CYCLIC or BLOCK_N (see AppendixA) should be considered as well. Currently, `lip` supports a coordinate-based partitioner which has several limitations. Particularly, it requires a strict knowledge of the physical distribution of the data which makes it useful only for some irregular problems.

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

A

AVL FIRE benchmark, 61

B

bucket sort, 35

C

CHAOS, programming library, 16, 18–21, 27
collective IO, 16, 24, 38
  disk directed, 24
  two phase, 24
communication coalescing, 19
communication schedule object, 20, 21, 27–29, 31, 36–39, 41, 44, 45, 50, 59, 66

D

data mining, 16
data prefetching, 25
data reusing, 23
data sieving, 24, 27
datamap object, 29, 34, 36, 44, 50, 52–54
disk arrays, 16, 48, 49
distributed–memory systems, 14

E

exception handling, 51–53

F

fetch on demand technique, 18
fluid dynamics, 61, 65

G

general indexing scheme, 65
ghost area in data buffer, 44, 45
grid, 14, 16, 35

H

hash tables, 40, 41
High Performance Computing (HPC), 18
High Throughput Computing (HTC), 16, 23
Hilbert's curve, 29, 35, 36, 50, 55

I

i-section, 30, 31, 47, 48
Inspector/Executor technique, 18, 21, 27–30
IO API, 16
IO model, 22, 38
  global placement, 22, 38
  local placement, 22, 38
  partitioned in–core, 22
IObufmap object, 31, 37, 40, 41, 45, 46, 48, 52, 54
irregular distribution, 19, 20, 27, 34, 35, 49, 50, 60
irregular problem, 14, 15, 17–20, 22, 26, 27, 29–36, 42, 55, 61, 65, 66
  adaptive, 65
  multiphase, 65
  static„ 65

J

Java Native Interface (JNI), 17, 51
Java to Native Code Interfaces (janet), 17, 26, 51, 52
Java, programming language, 17, 26, 51–55, 65
jostle, programming library, 23

L

latency, 14, 15, 18, 19, 24, 27

lip, programming library, 15–17, 20, 21, 23, 24, 26–36, 38–40, 42, 45, 49, 51–55, 57–59, 61, 65, 66
load balance, 15, 17–19, 29, 36
local memory, 18, 19, 31, 45

M

memory hierarchy, 17
message startup, 19
molecular dynamics, 14
MPI derived datatypes, 22, 24, 27, 29–31, 40, 41, 46, 48, 49, 54
MPI–IO, standard, 16, 23, 24, 27, 37, 38, 57

O

object oriented, 51
out–of–core (OOC) problem, 14–17, 20, 22, 23, 25–27, 29–32, 35–38, 42, 55, 58, 59, 65

P

Panda, programming library, 16, 23
parallelizing compilers, 27
partitioner, 16–20, 27, 29, 32, 35, 36, 38, 44, 49, 55, 58, 60, 66
  coordinate–based, 19, 29, 49
  edge–based, 19
POSIX, standard, 16

R

RAID, 23
regular distribution, 19, 20, 27, 32, 34, 35, 49, 50, 60
remapping, 20, 32, 36, 37, 49, 50
remote file access, 16, 23
ROMIO, portable MPI–IO implementation, 24, 57

S

scalability, 17, 27, 36, 58, 65
shared file pointers, 38, 57
software caching, 19

T

translation table, 20, 34–37

V

ViPIOS, software architecture, 23
virtual array, 55
virtual memory, 58, 59

W

work distribution, 19
  almost owner computes, 19
  owner computes, 19
wrapper, 17, 26, 51, 52

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.

Appendix C

Papers

Below are the papers which have describe results obtained when preparing this Thesis. These papers already been published or have been accepted and will be published soon. The authors of this thesis are also co–authors of each of the presented papers. The papers describe different stages of the development of the `lip` library.

The papers are:

1. Bubak, M., Łuszczek, P., Malawski, M., Zając, K.: Irregular and Out–of–core Computing on SGI Origin 2000 in: M. Bubak, J. Mościński, M. Noga (eds.) *Proceedings of 1st SGI Users Conference*, Cracow, Poland, October 2000, ACC Cyfronet UMM, 2000, pp. 185-193. `http://www.cyf-kr.edu.pl/sgi2k/index.html`

2. Brezany, P., Bubak, M., Malawski, M., Zając, K.: Advanced Library Support for Irregular and Out-of-Core Parallel Computing to appear in: L.O. Hertzberger, R. Williams, A. Hoekstra (eds.) *Proceedings of Intl. Conference High Performance Computing and Networking*, Amsterdam, June 25–27, 2001, Springer Verlag, Lecture Notes in Computer Science. To be printed. `http://www.science.uva.nl/events/HPCN2001/`

3. Brezany, P., Bubak, M., Malawski, M., Zając, K.: Irregular and Out-of-Core Computing on Clusters submitted to Parallel Computing Conference, Italy, 4–7 September 2001, `http://139.174.100.167/`