# Advanced Library Support for Irregular Data Intensive Scientific Computing on Clusters

**Peter BREZANY[1], Marian BUBAK[2,3],**
**Maciej MALAWSKI[2] and Katarzyna ZAJĄC[2]**
**[1]Institute for Software Science,**
**University of Vienna, Austria**
**[2]Institute of Computer Science, AGH, Kraków, Poland**
**[3]ACC CYFRONET-AGH, Kraków, Poland**

## Overview

- Irregular and out-of-core problems
- `lip` design goals
- Basic approach: inspector/executor, i-section
- Optimizations
- `lip` overview
- Using `lip` in Java
- Performance tests
- Summary
- Future development

# Irregular Problems

- Access to data arrays through one or more levels of *indirection arrays*
- *Indirection arrays* not known until runtime - no possibility of compile-time optimization
- Runtime optimization aimed at maximizing computation to communication ratio
- Types
  - static
  - multiphase
  - adaptive
  - ...

# Static Irregular Problems

- Indirection arrays remain unchanged during computation
- Examples
  - sparse matrix-vector multiplication
  - explicit mesh solvers

# Example of Static Irregular Problem

```
int i, j, k1, k2;
int edge1[N], edge2[N];  /* indirection arrays */
double x[M], y[M];       /* data arrays */
/*...*/
for (j = 0; j < TIME_STEPS ; j++)
  for (i = 0; i < N; i++)
  {
     k1 = edge1[i];
     k2 = edge2[i];
     y[k1] += x[k2];  /* indirect access
  }                    to 'x' and 'y' */
```

## Adaptive Irregular Problems

- Many computational phases, data dependencies are modified between consecutive phases, e.g.
  - adaptive unstructured mesh solvers
  - particle dynamics codes
  - direct Monte-Carlo simulations

# Example of Adaptive Irregular Problem

```
int i, j, k;
int edge[N];            /* indirection array */
double x[M], y[M];      /* data arrays */
/*...*/
for (j = 0; j < TIME_STEPS ; j++)
{
  for (i = 0; i < N; i++)
  {
     k = edge[i];
     y[i] += x[k];  /* indirect access to ‘x’ */
  }
  /* modification of indirection array */
  for (i = 0; i < N; i++)
    edge[i] = refine( edge[i] );
}
```

## Multiphase Irregular Problems

- multiple phases with irregular loops (each phase like a single static problem)
- indirection arrays used between phases are similar
- Examples: unstructured multigrid mesh solvers, sparse triangular solvers

# Example of Multiphase Irregular Problem

```
int i, j, k;

/* indirection arrays */
int coarse_edge[N];
int fine_edge[N];

/* data arrays */
double x[M], y[M];

/* ... */
```

```
for (j = 0 ; j < TIME_STEPS ; j++){
  /* iterations over coarse mesh */
  for (i = 0 ; i < N_COARSE ; i++){
    k = coarse_edge[i];
    /* indirect access to ‘x’ */
    y[i] += x[k];
  }
  /* iterations over fine mesh */
  for (i = 0 ; i < N_FINE ; i++) {
    k = fine_edge[i];
    /* indirect access to ‘x’ */
    y[i] += x[k];
  }
}
```

# Parallelization Techniques for Irregular Problems

1. Fetch on demand
   - data fetched when needed
   - simple parallelization steps
   - inefficient
2. Inspector/Executor
   - Inspector
     - data & work partitioning
     - analysis of indices & their translation
     - communication objects generation
   - Executor
     - communication (with use of created objects)
     - computation (almost the same as in sequential code)
   - Example tool: CHAOS library

# Out-of-Core (OOC) Problems

- Sizes of arrays are too large to fit in main memory
- Virtual Memory management provided by operating system does not solve it because
  - there are limitations on virtual memory size
  - operating system cannot optimize I/O access pattern for a particular problem
  - no collective I/O is possible (all computing nodes perform I/O operations concurrently)
- Additional tools for OOC problems are required
- MPI-IO emerges as a standard for parallel I/O after wide acceptance of MPI

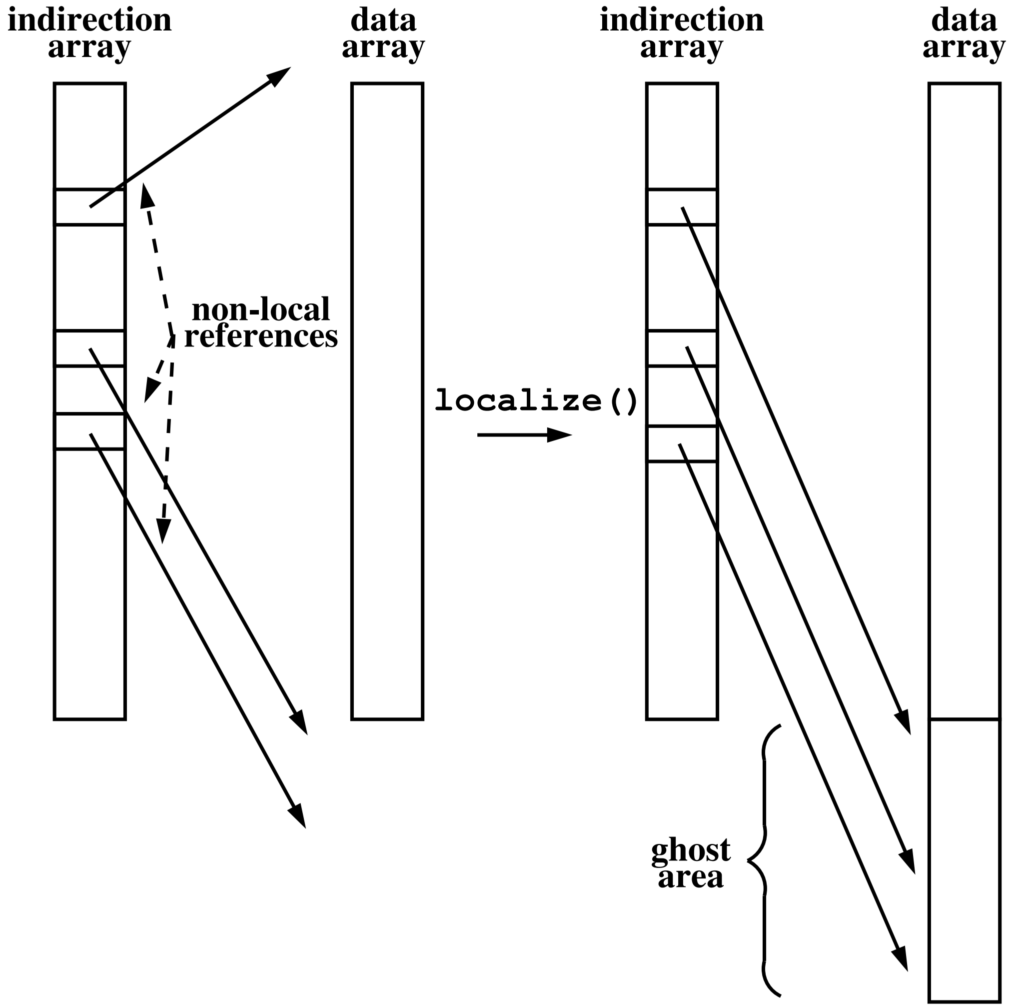
## Goals

- Requirements
  - portability
  - multilanguage support
  - scalability
  - efficiency
    - load balancing with partitioner
    - data caching
- Implementation
  - according to ANSI C standard
  - built on top of MPI and MPI–IO
- Tests
  - synthetic benchmarks and scientific applications

## Inspector/Executor technique for IC computing

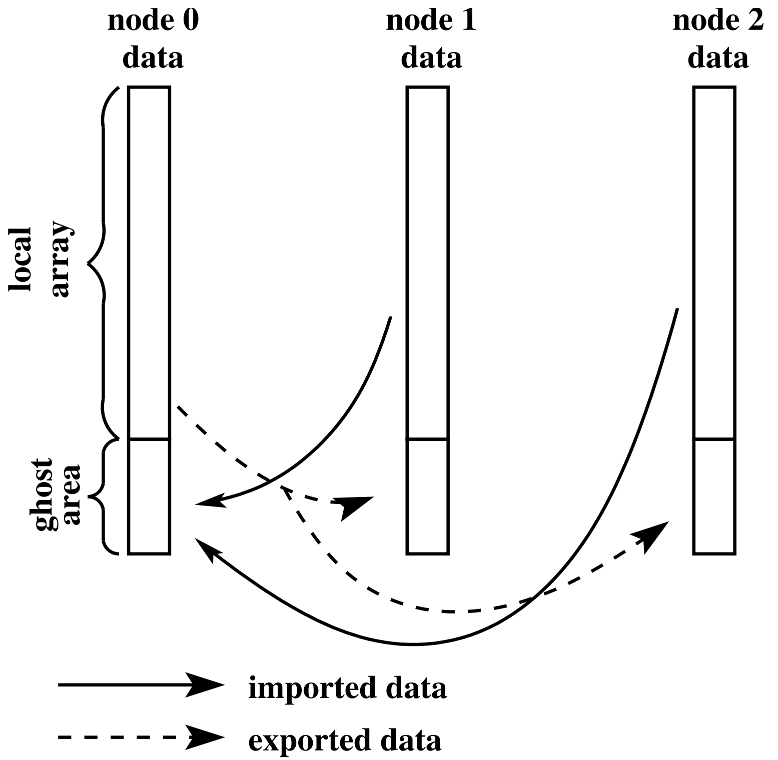
- during the *inspector* phase, data is localized (data items needed by the node are fetched)
- during the *executor* phase, actual computations are performed

# Inspector/Executor technique for IC computing – cont’d



# Definition of Schedule

- what to send (export list)
- what to receive (import list)
- communication coalescing
- global communication pattern information
- use of collective routines
- duplicate entries removed

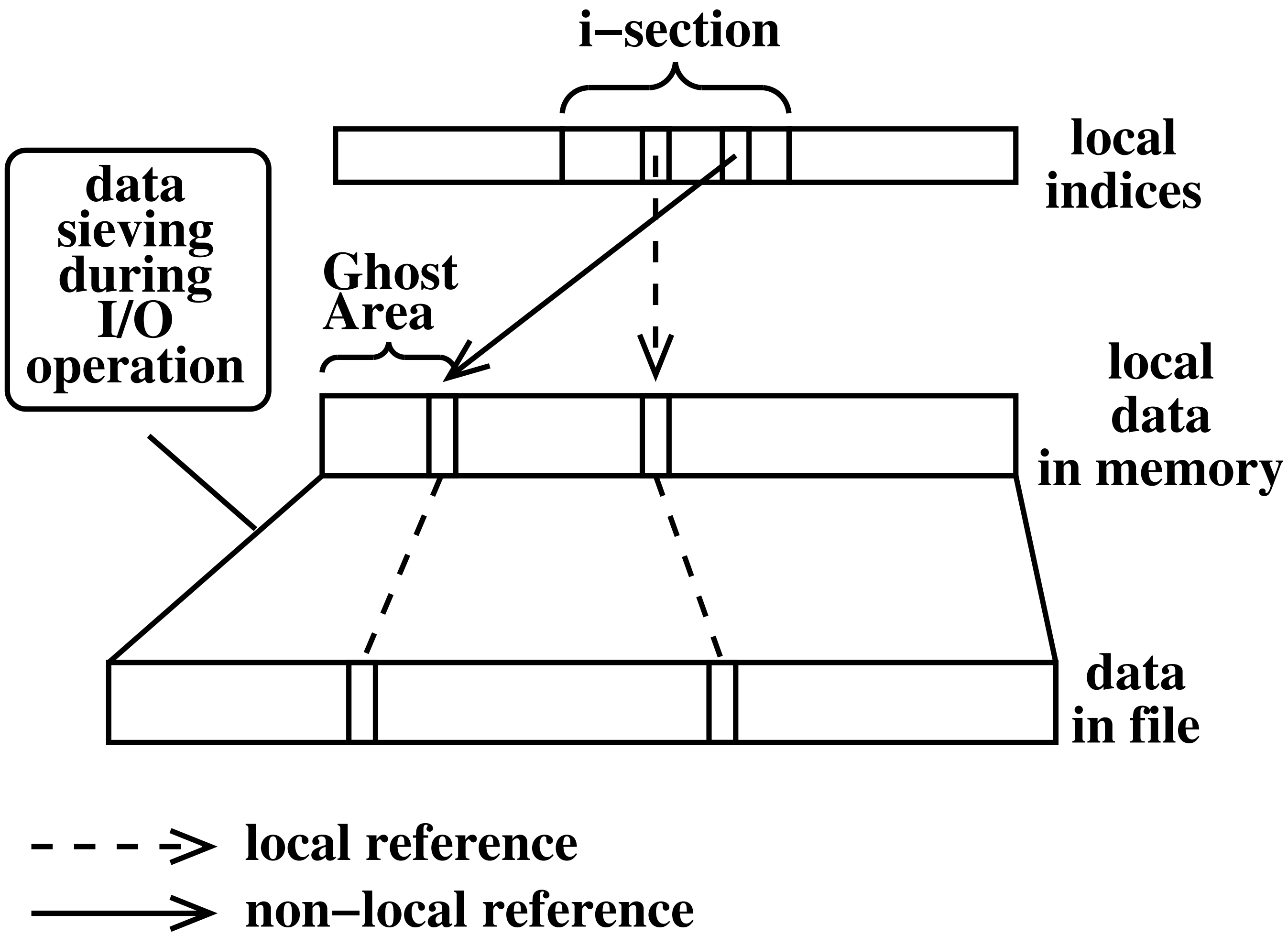


## OOC Support in `lip`

- Parallelization is based on the concept of *i-sections*
- Support for cases when indices array, data array or both are out-of-core
- multiple passes over parts of index array
- the same functions as in in-core version
- user defined *i-section* size

# OOC Support in `lip` - cont'd

## OOC local (node) view



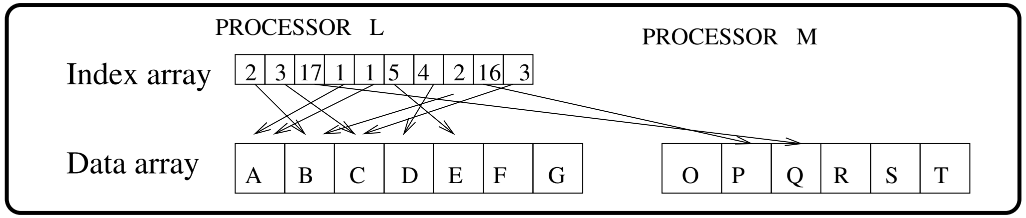
# The `lip` Library Optimizations

- Data partitioning and remapping for better load balance.
  - global method
  - uses any number of processors
  - in–core and out–of–core version
  - coordinate-based – Hilbert’s curve
  - uses bucket sort algorithm – complexity O(n)
- Data caching
  - special structure *IOBufmap* to memorize data mapping between memory and disk
  - exchange only for the elements not residing in a memory buffer

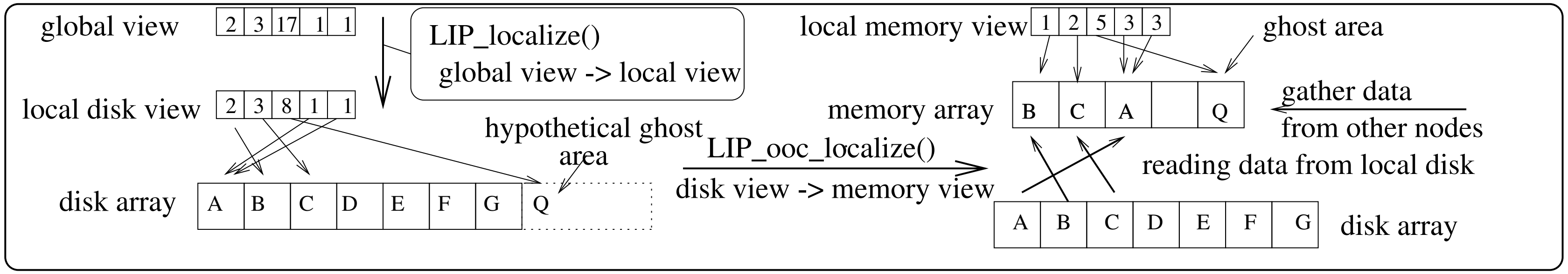
## The `lip` Library Optimizations – cont’d

- Communications and I/O operations coalescing
    - a single routine for reading all needed elements
    - a single routine for writing all old elements
    - a single routine for communication

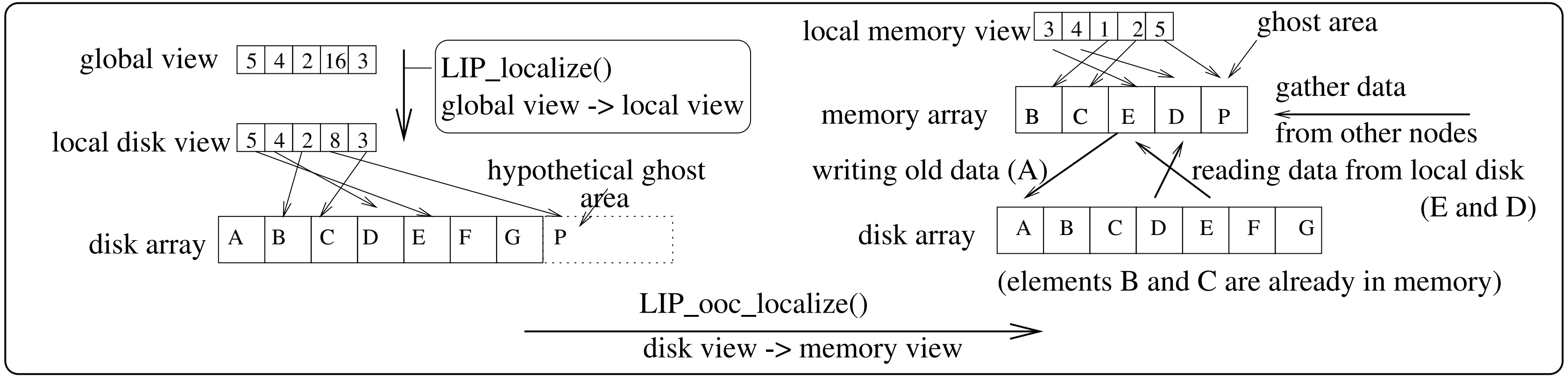
# I/O Caching and Operations Coalescing



**First i-section of processor L**



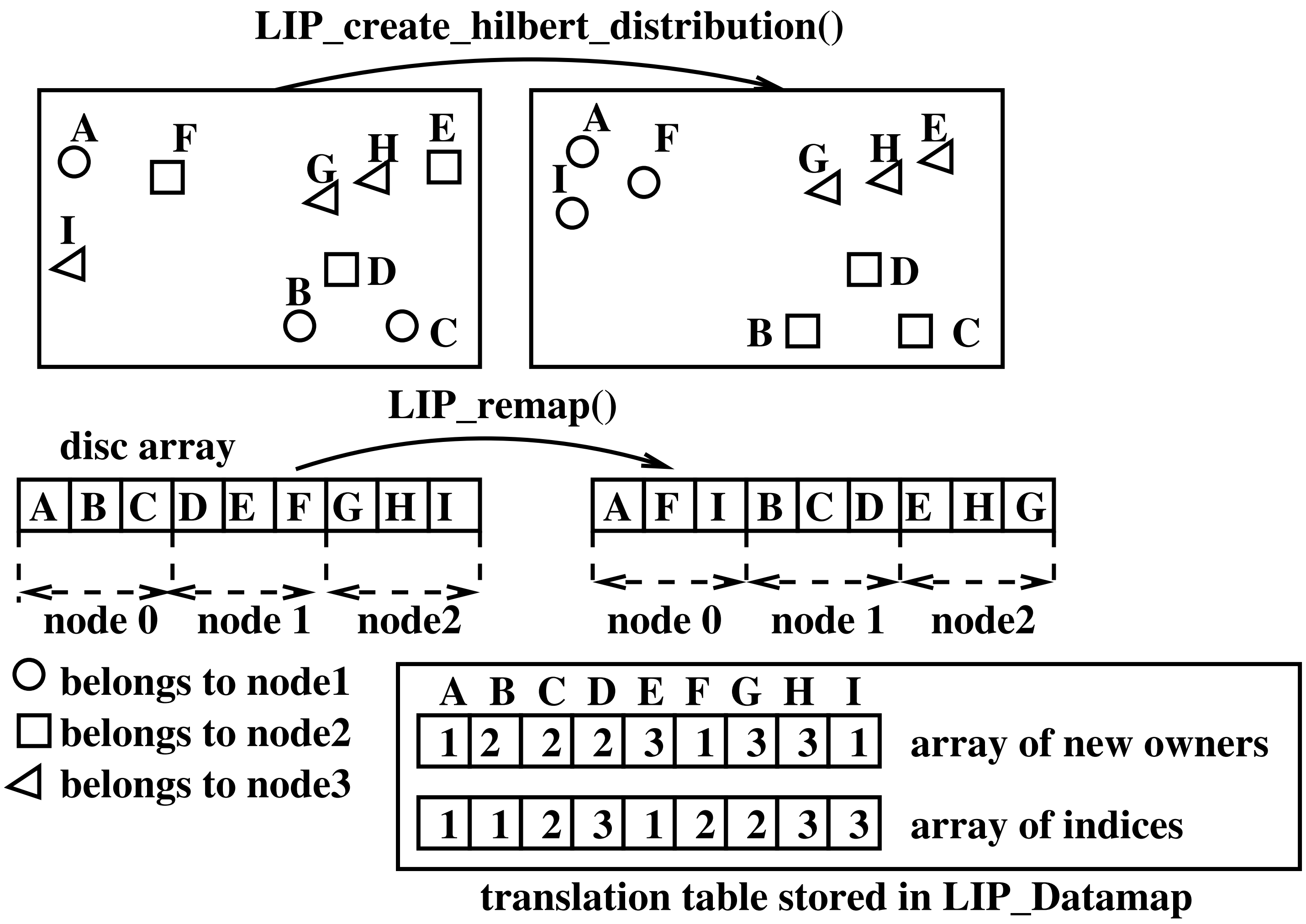
**Second i-section of processor L**



## Datamaps

- mapping of user data onto nodes
- major mappings supported (from data-parallel languages)
  - `BLOCK`
  - `CYCLIC`
  - `INDIRECT`
  - other
- distributed/local storage

# Irregular Distribution Creation



**LIP_create_hilbert_distribution()**

**LIP_remap()**

**disc array**

○ **belongs to node1**

□ **belongs to node2**

◁ **belongs to node3**

| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| **array of new owners** | 1 | 2 | 2 | 2 | 3 | 1 | 3 | 3 | 1 |
| **array of indices** | 1 | 1 | 2 | 3 | 1 | 2 | 2 | 3 | 3 |

**translation table stored in LIP_Datamap**

# LIP Library Overview

Six main groups of LIP library functions :

- Functions to create and manipulate data mapping objects which describe how data array is partitioned among processors - involved partintioner functions :
  - `LIP_create_hilbert_distribution_in_core()` create irregular distribution for in–core data
  - `LIP_create_hilbert_distribution_ooc()` create irregular distribution for out–of–core data

  and remapping functions.
- Functions that create objects for mapping the memory buffer onto a file
- Functions for communication schedule generation and transformation

## LIP Library Overview – cont'd

- Two functions for index translation -
  - `LIP_Localize()` translates globally numbered indices into locally numbered counterparts
  - `LIP_OOC_localize()` transforms indices which point to data residing on a disk into indices to a memory data buffer
- Communication functions that perform collective communication between nodes
- Miscellaneous - setup etc.

## In-core Inspector

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

| | | |
|---|---|---|
| IN | datamap | data mapping handle (handle) |
| IN | flags | combines flags specified by user (integer) |
| IN | iglobal | index array with global indices (choice) |
| IN | igtype | layout description of global indices (handle) |
| OUT | ilocal | index array for local indices (choice) |
| IN | iltype | layout description for local indices (handle) |
| IN | count | number of indices (integer) |
| INOUT | schedule | communication schedule (handle) |
| INOUT | info | additional information about indices (handle) |

## In-core Inspector – cont'd

- count indices from indirection array iglobal are transformed into ilocal array
- communication schedule is created for data (which is distributed according to datamap) exchange between nodes
- additional info is passed via flags and info

# Out-of-core Inspector

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

| | | |
|---|---|---|
| IN | ilocal | index array with local indices (choice) |
| IN | iltype | layout description for local indices (handle) |
| OUT | itlocal | index array for translated local indices (choice) |
| IN | ittype | layout description of translated indices (handle) |
| IN | count | number of indices (integer) |
| IN | flags | combines flags specified by user (integer) |
| INOUT | buf | memory data buffer (choice) |
| INOUT | bmap | mapping of memory buffer elements onto file (handle) |
| INOUT | schedule | communication schedule (handle) |

## Out-of-core Inspector – cont’d

- Count indices from indirection array ilocal are transformed into itlocal array
- memory-disk mapping bmap is created for data exchange between in-memory data buffer buf and disk via MPI–IO routines
- communication schedule is modified to reflect new mapping
- additional info is passed via flags.

## Sample Code Using `lip`

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

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

/* Generate irregular distribution*/
LIP_create_hilbert_distribution(/* */);

/* Data remapping */
LIP_remap_ooc(/*...*/);

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

  LIP_Localize( /* ... */ );
  LIP_OOC_Localize(/* ... */);

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

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

  /* Exchange between disk and memory  */

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

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

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

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

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

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

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

## ***Janet* tool**

- extension of the Java language
- allows inserting native (i.e. C) language statements directly into Java source code files
- *Janet* translator creates automatically Java and native source files with all the required JNI calls
- frees the programmer from calling low level JNI API
- makes interface creation more flexible and clearer

## Java Bindings to `lip`

- Using ***Janet*** tool to create Java bingings
- Implementation of Java-specific features
  - Class-based library design instead of flat function set
  - Objects instead of structures and handles
  - Exceptions instead of error return values
- Performance evaluation

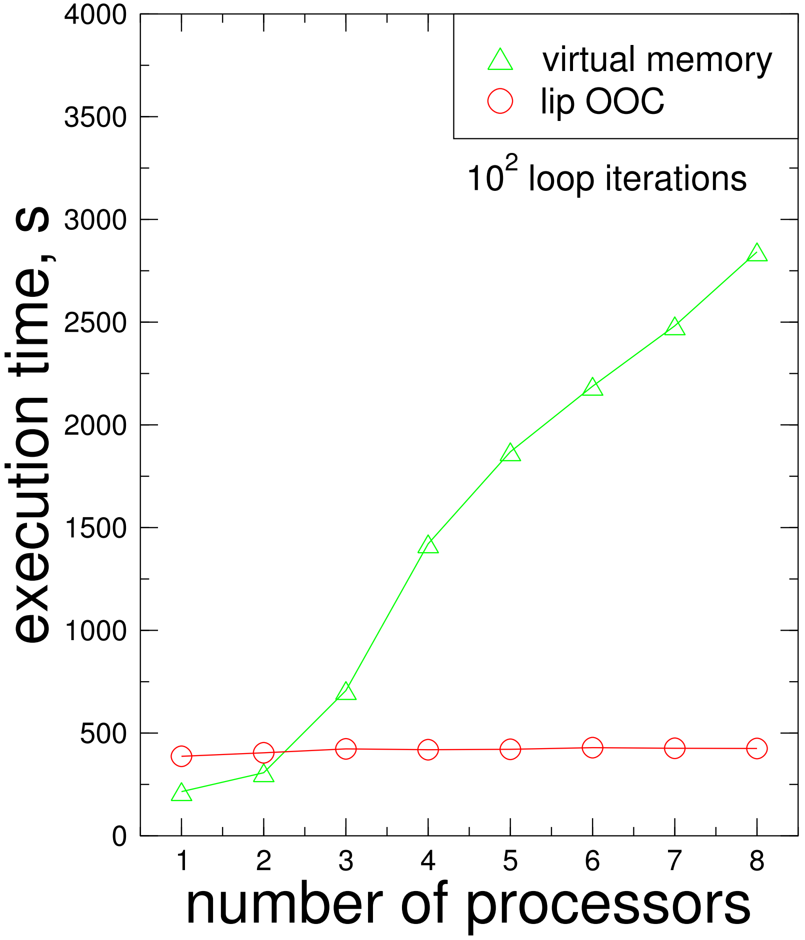
# Sample Java OOC code with `lip`

```
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;
```

## Kernel loop

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

# Advantage of `lip` Library over Virtual Memory

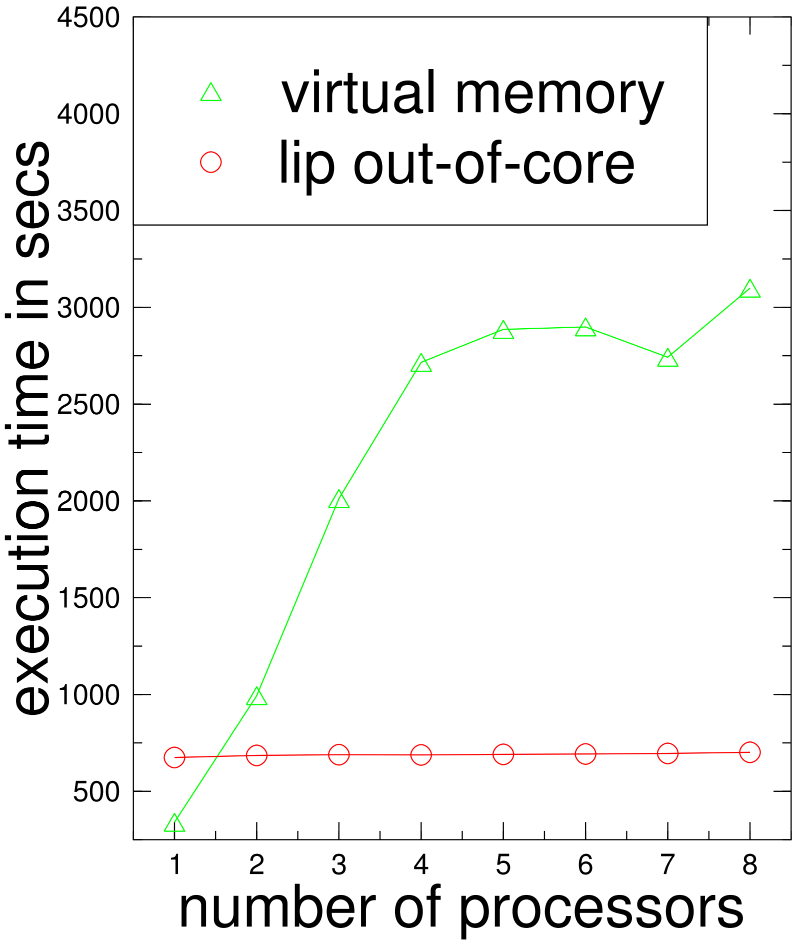


## Kernel loop

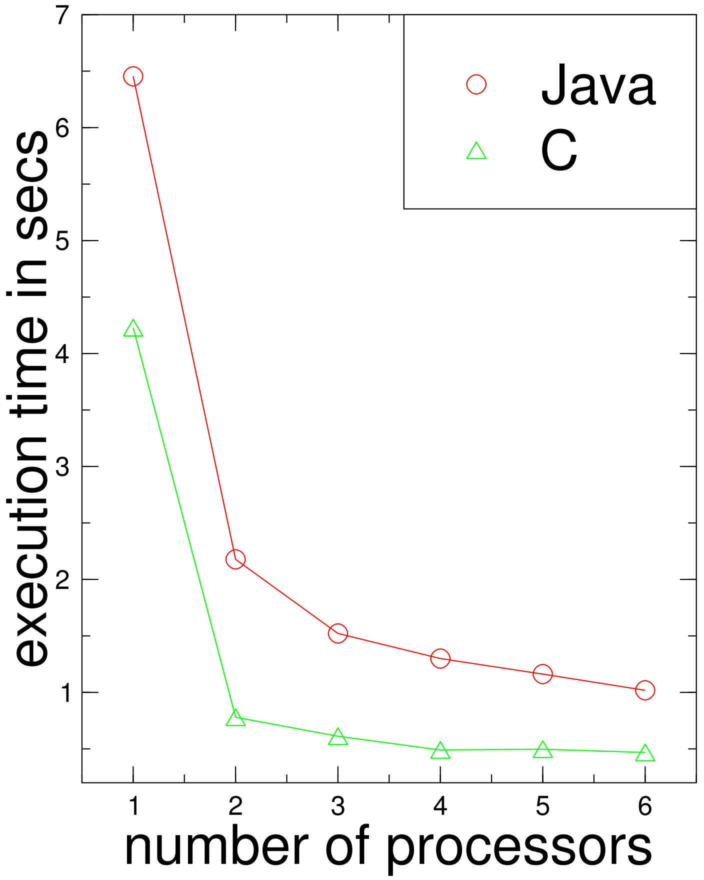
```
int perm[n];
double x[2*n];
double y[n];

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

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

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

# **Advantage of `lip` Library over Virtual Memory**



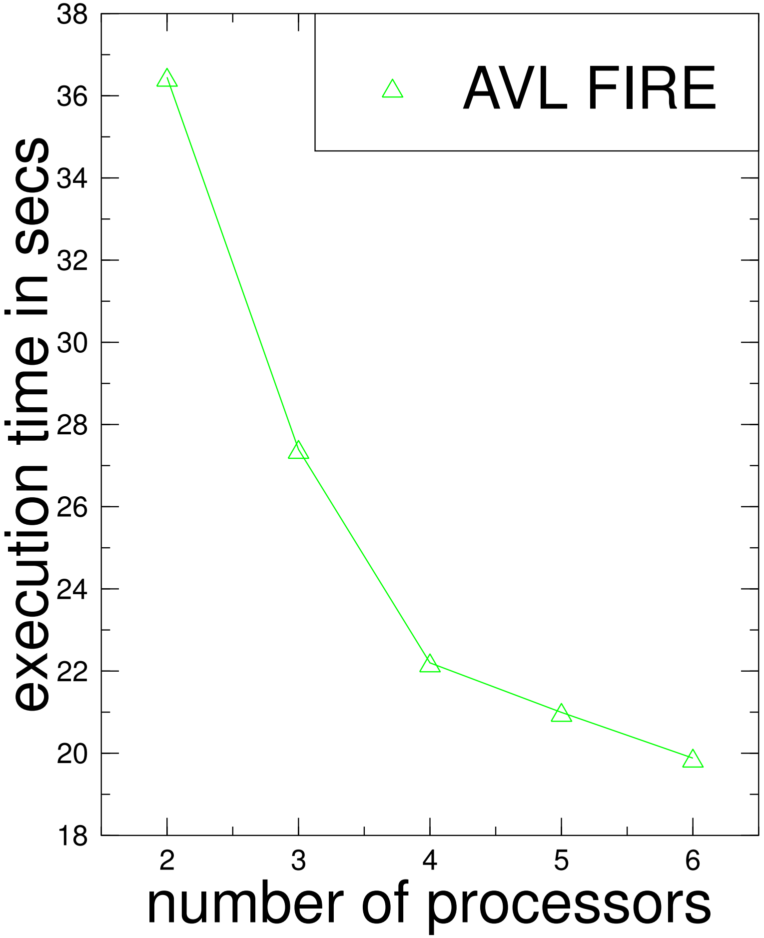
# Comparison between C and Java Performance



# Kernel loop of the GCCG solver

```
double direc1[NNCELL];
double direc2[NNCELL];
int lcc[6*NNCELL];
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];
}
```

# Performance of AVL FIRE benchmark solver

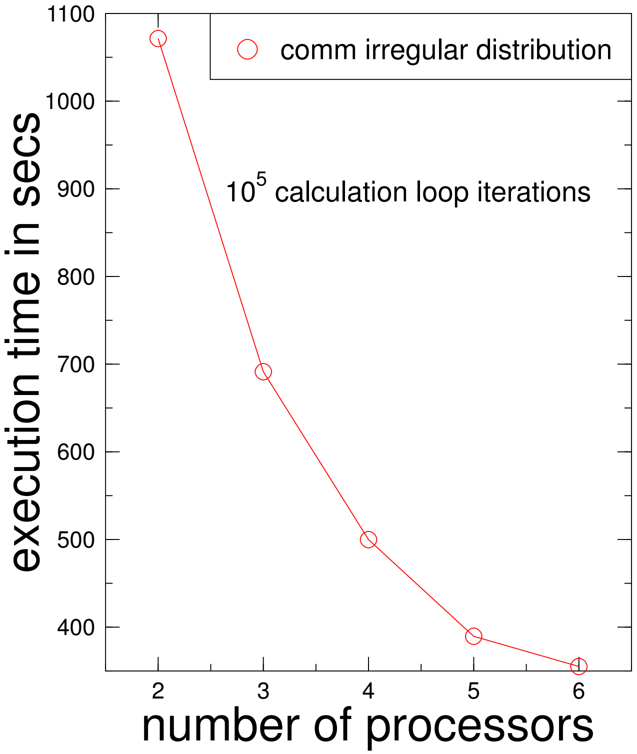


# Kernel loop of partitioner test

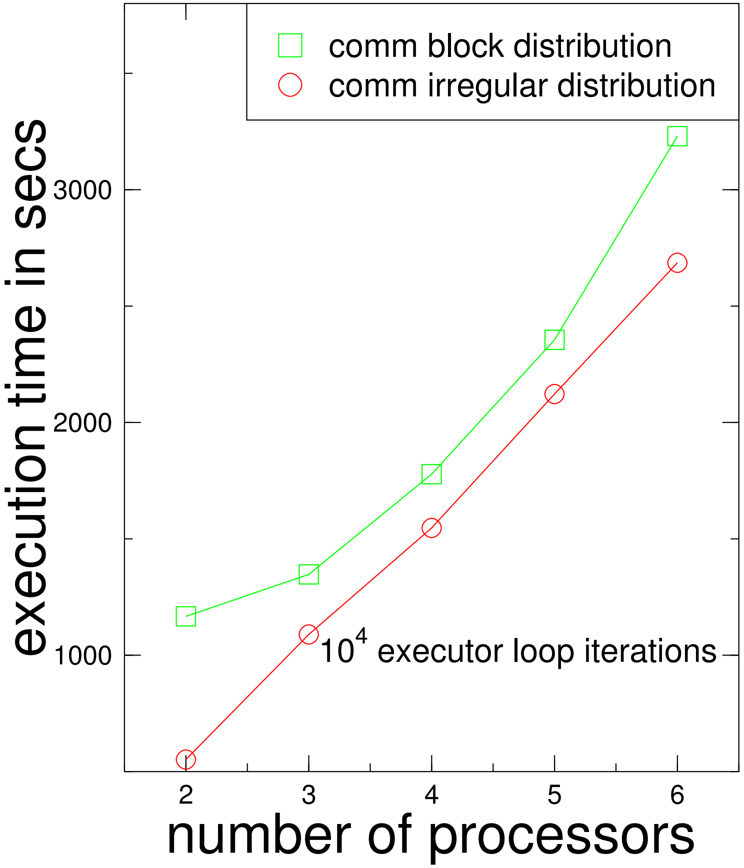
```
for(i=0;i<edge_counter;i++)
{
  x2[edge2[i]]+=x1[edge1[i]]/n_succ;
  x2[edge1[i]]-=x1[edge1[i]];
}
```

- vertices of a graph distributed randomly on a plain square
- distance between adiacent vertices cannot be greater then fixed value

# Performance of inspector and executor phase for irregular distribution

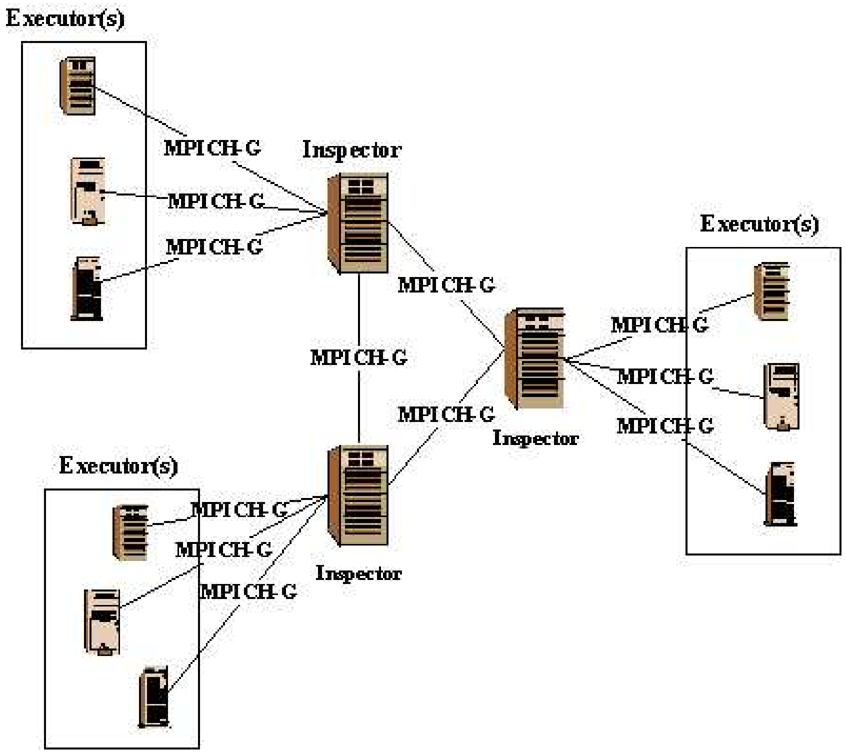


# Performance of communication phase



# Master-Worker scheme for Grid LIP (G-LIP)

- useful in Grid environment
- divides work between masters (inspectors) and wokers (executors)



## Summary

- Portability based on MPI and MPI-IO
- Scalability
- Advantages of data partitioning
- Multi-language support (C and Java)
- Available on:
  `http://galaxy.uci.agh.edu.pl/~kzajac/`
- Janet tool:
  `http://www.icsr.agh.edu.pl/janet/`
- first attempt for porting to a Grid

## Future Development

- User defined operations
  - used to customize `lip` to a particular problem
- More partitioners
  - use edge information to obtain data distribution patterns
- Extend lip to the OGSA architecture