Towards Portable Runtime Support for Irregular and Out-of-Core Computations

Marian Bubak[1,2] and Piotr Łuszczek[1]

[1] Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland
[2] Academic Computer Centre – CYFRONET, Nawojki 11, 30-950 Kraków, Poland
email: {bubak,luszczek}@uci.agh.edu.pl
phone: (+48 12) 617 39 64, *fax:* (+48 12) 633 80 54

Abstract. In this paper, we present the `lip` – a runtime system which enables easy and portable parallelization of irregular and out-of-core computations. Functions for handling irregular data were developed using the same concept as in the CHAOS library. Out-of-core parallelization is based on the idea of *in-core section*, and functions for out-of-core data are implemented with capabilities provided by MPI–IO. The new library may be used in C, Fortran and Java programs. Results of performance tests for a generic irregular out-of-core program on HP S2000 are presented and possible further extensions are discussed.

1 Introduction

Parallelization of so called *irregular* and *out-of-core* (OOC) applications is a very important issue in development of large scale computations [1]. The characteristic feature of *irregular problems* is accessing data through one or more levels of indirection arrays. During recent decade few runtime tools have been developed to facilitate parallelization of irregular problems, with the CHAOS runtime library [2] being the most notable among them. Recently, it has been successfully ported from Intel Paragon's NX to MPI [3]. Many applications involve primary data structures which are too large to fit into the main memory; they are called *out-of-core problems*. To parallelize this kind of programs one may use explicit I/O calls, e.g., those offered by the MPI–IO. There are also dedicated tools, e.g., PIOUS [4] and Panda [5] systems. Problems which are at the same time the irregular and out-of-core ones are especially difficult to parallelize. Typical examples include particle codes, CFD programs, sparse matrix computations. They were examined by a number of research projects what resulted in the development of runtime systems like VIPIOS [6] and COMPASSION [7].

In this paper, we present the `lip` – a portable runtime support library for both in- and out-of-core irregular problems. Portability is guaranteed by the use of the MPI with its I/O features ([8]). Parallelization of irregular problems is realized in the similar way as in the CHAOS library [2]. The out-of-core functions are based on the idea of *in-core section*. The functions of the `lip` may be used in C, Fortran and Java parallel programs developed with the MPI, as well as by the parallelizing compilers.

2 MPI-Based Support for Irregular Problems

Since originally the CHAOS library was implemented on top of the NX message passing system, the techniques described in [9] were used to convert basic communication routine calls to MPI along with additional improvements to the CHAOS [3] itself (Fig.1). The version of CHAOS for MPI programs, though fully operative, lacked several important features, namely, there was no support for computations on heterogeneous systems and inherently collective operations were implemented with simple `send()` and `recv()` functions. In the second stage, we have elaborated a new `lip` library which resolves all the problems mentioned above. It uses the higher-level collective MPI routines which improve its efficiency and strictly typed communication buffers to provide support for heterogeneous environments. API of the `lip` is very similar to that of the CHAOS.

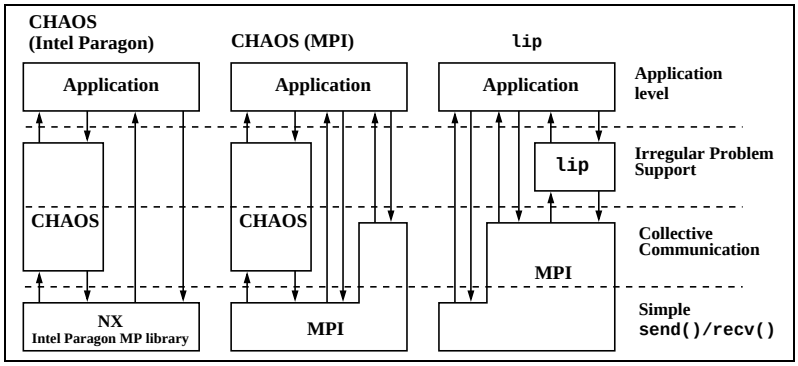


Fig. 1. Consecutive rearrangements of the CHAOS internal structure and the message passing usage.

There are at least three models of disk storage [10]: local placement model, global placement model, and partitioned in-core model. An alternative to providing I/O model is to use highly expressive I/O API calls which allow the user to fully describe the desired I/O operation. The `lip` aims at the second solution and allows the user to take advantage of the capabilities provided by the MPI–IO. The I/O operations are performed by the user whereas the `lip` provides only the data structures which might be used in these operations.

3 Parallelization of OOC Problems

Runtime support for parallel execution of irregular code is based on so called *Inspector/Executor* technique introduced in the CHAOS [2]. It splits the whole process into two independent phases: *Inspector Phase*, *Executor Phase*.

In the *Inspector Phase* phase, data references (from indirection array) are examined and translated either into references to node's local data array (global

to local index translation) or into references to node's temporary buffer (referred to as a *ghost area*). The latter case is tightly connected with generation of *schedule* which is a data structure describing how communication is to be performed efficiently to bring the desired data into *ghost area*. The *Inspector Phase* is depicted in Fig. 2 as a transition from global to local view. The last part of the Fig. 2 shows OOC local view which pertains to the case when index array (which is examined during *Inspector phase*) is too large to fit into memory.

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

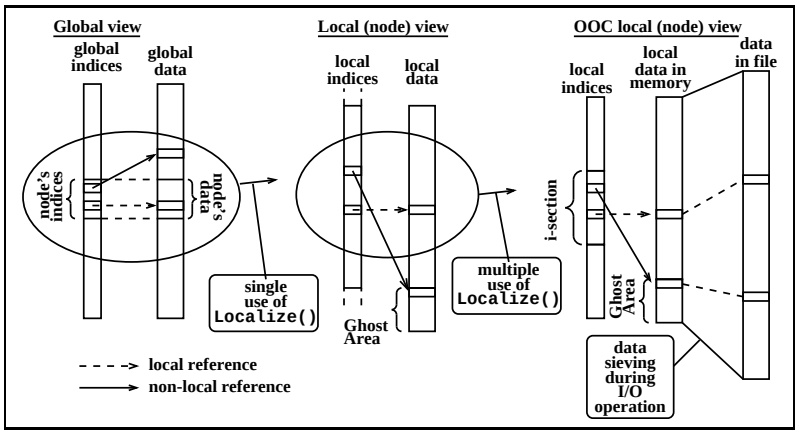
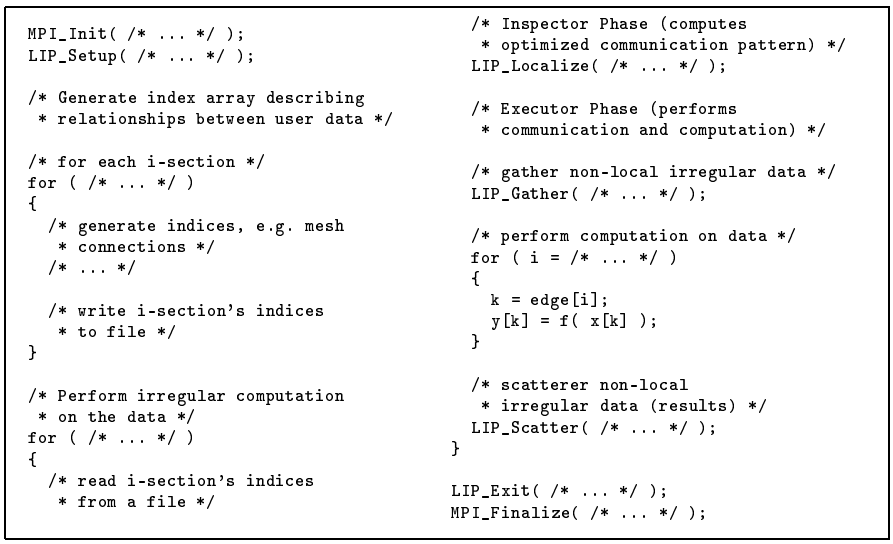


Fig. 2. Restructuring of arrays for OOC computing.

The *Inspector/Executor* approach remains almost unaltered due to introduction of the concept of the *in-core section* (tile or slab) [11]. The whole index array is stored in a file and only a small chunk of it (*i-section*) is loaded into the main memory to enable the optimization described previously. In this way, the technique successfully used for the irregular in-core problems have been generalized for the irregular OOC problems with *in-core* data arrays and OOC index arrays. A generic program which uses *i-section* is given in Fig. 3.

However, this concept has to be further modified when not only the index arrays but also the 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. The detailed analysis leads to a conclusion that three types of non-local accesses may be encountered in the index array: access



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

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

/* for each i-section */
for ( /* ... */ )
{
  /* generate indices, e.g. mesh
   * connections */
  /* ... */

  /* write i-section's indices
   * to file */
}

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

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

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

  /* 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( /* ... */ );
}

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

Fig. 3. Sample C code which shows OOC computing with the `lip`.

to data of other node (the data are present in the node's main memory), access to data of other node (the data are *not* present in the node's main memory), access to local node data which are not present in memory (they are stored on a disk).

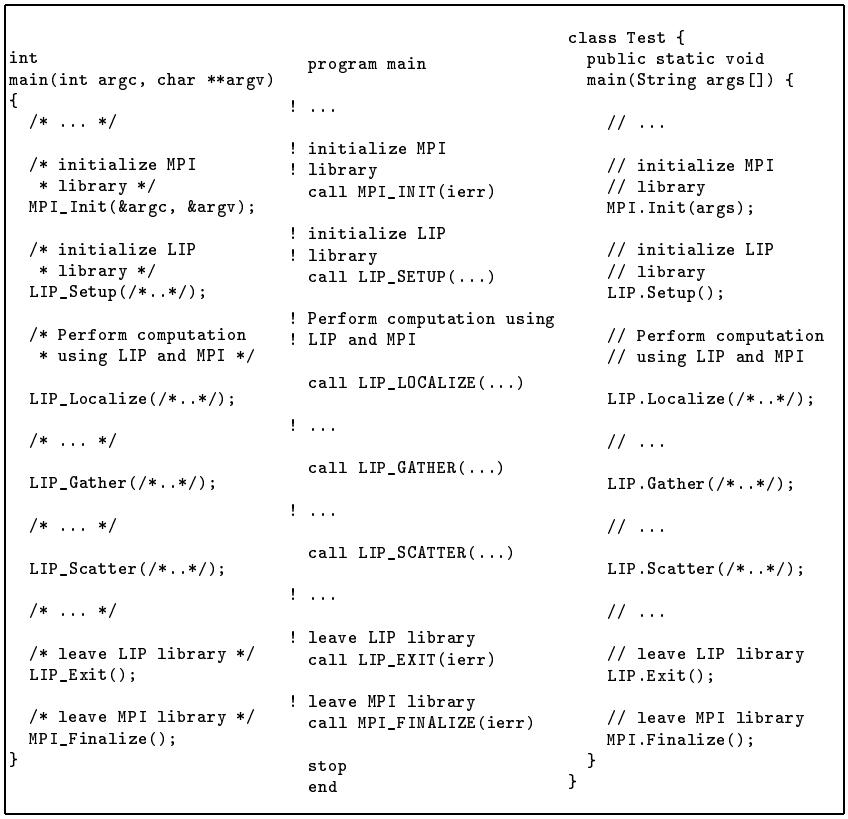
It would be heavily inefficient to read each element from a disk just as it becomes needed. Therefore, at the *Inspector Phase*, the optimization has to be aimed at efficient communication and also the I/O accesses associated with the *not-in-memory* data elements have to be minimized. Furthermore, the data transfer should be performed using the available optimization techniques such as: data sieving and collective I/O with MPI derived data types [13, 12].

We have used ROMIO 1.0.1 [14] (the portable implementation of the MPI–IO) which is included in the HP MPI 1.4 – MPI implementation from HP (built on the `mpich` [15]), which, in turn, was used in the tests we have performed (see section 5). ROMIO lacks some routines defined in the MPI–IO standard, namely the split collective routines that may be used to employ the performance-increasing technique of overlapping I/O operations with communication and computation.

4 Bindings to Fortran and Java

Since text `lip` is supposed to serve as a portable runtime support library not only across different platforms but also across different host languages, we have developed bindings of the library to Fortran and Java.

The Fortran wrappers could have been created using techniques presented already in [15, 16] and this could be done in a virtually automatic manner. How-



```
int
main(int argc, char **argv)
{
  /* ... */

  /* initialize MPI
   * library */
  MPI_Init(&argc, &argv);

  /* initialize LIP
   * library */
  LIP_Setup(/*..*/);

  /* Perform computation
   * using LIP and MPI */

  LIP_Localize(/*..*/);

  /* ... */

  LIP_Gather(/*..*/);

  /* ... */

  LIP_Scatter(/*..*/);

  /* ... */

  /* leave LIP library */
  LIP_Exit();

  /* leave MPI library */
  MPI_Finalize();
}
```

```
      program main

! ...

! initialize MPI
! library
      call MPI_INIT(ierr)

! initialize LIP
! library
      call LIP_SETUP(...)

! Perform computation using
! LIP and MPI

      call LIP_LOCALIZE(...)

! ...

      call LIP_GATHER(...)

! ...

      call LIP_SCATTER(...)

! ...

! leave LIP library
      call LIP_EXIT(ierr)

! leave MPI library
      call MPI_FINALIZE(ierr)

      stop
      end
```

```
class Test {
  public static void
  main(String args[]) {

    // ...

    // initialize MPI
    // library
    MPI.Init(args);

    // initialize LIP
    // library
    LIP.Setup();

    // Perform computation
    // using LIP and MPI

    LIP.Localize(/*..*/);

    // ...

    LIP.Gather(/*..*/);

    // ...

    LIP.Scatter(/*..*/);

    // ...

    // leave LIP library
    LIP.Exit();

    // leave MPI library
    MPI.Finalize();
  }
}
```

Fig. 4. Calling the `lip` from C, Fortran and Java.

ever, we had to cope with the previously unresolved technical issue concerning translation of array indices (which are ubiquitously used in irregular applications) between C and Fortran array-indexing schemes. A much greater effort has to be invested to produce Java wrappers since MPI Forum does not legitimize any standard MPI to Java binding. This leads to the necessity of a much more careful choice of implementation details that would successfully resolve the platform-specific problems with *Java Native Interface*, which are mostly related to the handling of the shared libraries on different operating systems and seamless cooperation of these libraries not only with Java Virtual Machine but also with each other. Moreover, there is an issue related to the currently being investigated possibility of various extensions to Java language and its API for more efficient scientific computing (see e.g. [17]). Nevertheless, we have implemented an experimental Java binding for the `lip` with `mpich` [15] since the LAM implementation of MPI [18] encountered problems with Java Virtual Machine that `fork()`'ed. The work on other implementations of wrappers is in progress. Fig. 4 presents generic programs in C, Fortran and Java with calls to the `lip` functions.

5 Sample Results

To evaluate performance of the `lip` we have used generic irregular loop which operates on large data and index sets (see Fig. 3). First, data are generated along with indices (which could describe the mesh or graph connections), and next, they are used to index the data array. The index array is OOC, so it is divided into *i-sections* and stored on disk(s). During the computation phase, the *i-sections* are loaded and used.

#processors	1	2	4	6	8	10	12	14
Elapsed time (in seconds) for local I/O.								
NOOC small	204.8	111.2	60.3	42.6	34.2	29.4	26.5	25.1
OOC small	204.3	110.7	59.6	43.3	34.7	30.2	27.8	25.8
OOC big	273.2	166.0	100.4	78.8	66.6	63.1	53.6	56.4
Elapsed time (in seconds) for global I/O.								
NOOC small	204.8	111.2	60.3	42.6	34.2	29.4	26.5	25.1
OOC small	204.0	111.3	61.2	44.2	35.9	32.1	29.6	27.9
OOC big	266.5	168.0	116.1	90.7	82.9	77.4	73.6	73.6
Elapsed time (in seconds) for constant-time problem.								
Local I/O model	188.1	205.1	215.2	233.0	244.3	257.2	269.8	289.7
Global I/O model	188.5	204.4	223.5	236.7	256.1	276.2	297.9	401.2

Table 1. The timings for execution of a simple irregular loop.

Three test cases were run. The first one (*NOOC small*) creates data (of global size 23063040 C language `double` values) and index arrays (of global size 5765760 C language `int` values) which are large enough to fill up all the memory that a user process can allocate. In the second case (*OOC small*), the sizes of the arrays are the same as previously but the `lip` handles I/O accesses (instead of the Exemplar's virtual memory system) through the MPI–IO (*i-section* size is 360360). In the third case (*OOC big*), the arrays are so large (data: 2882880; indices: 57657600; *i-section*: 360360) that they can only be handled through the `lip` (process cannot allocate sufficient memory to fit them into the main memory even with the virtual memory system support).

The tests were run on the HP S2000, on a dedicated 14-processor subcomplex. The computer itself has one 16-processor hypernode with 4 GB memory, 6 physical disks (of total size 49 GB) connected through 3 PCI interfaces. One of the disks (4 GB alone on one PCI interface) serves as a swap space for the virtual memory management system whereas the remaining 5 disks (9 GB each) are for the user (they are connected through 2 remaining PCI interfaces - 2 disks on the first one and another 3 on the second one).

The timings for different runs are shown in Table 1. The first two *small* cases show virtually the same performance of the `lip` and the virtual memory system. The *OOC big* case with its speedup below 5 for 14 MPI processes reveals the weakness of the tested system, which stems from the fact that it has only

five disk drives. Overlapping of I/O and computation would hide this flaw to some extent, unfortunately, the MPI–IO implementation we have used, have not appropriate capabilities (i.e. the full support for all kinds of non-blocking I/O routines). Table 1 shows also the elapsed time of application that had a constant data size for a single process. In this case, the performance of local an global I/O can be interpreted in favor of the former as it was also true for the previously described results. A possible explanation is an inefficient implementation of collective routines (again, ROMIO 1.0.1 lacks some of them), since we did not cope with the Exemplar-specific system calls which may perform better.

6 Conclusions and Future Work

An important feature of the `lip` is its support for efficient handling of irregular access patterns (as they are encountered in irregular problems) through the high-level data structures called *schedules*. This is done in a uniform manner for the I/O access routines and for communication. With these data structures, information may be transferred from the phase of analysis of the user data (*Inspector Phase*) to the communication and I/O phase (*Executor Phase*). Then it is possible to take full advantage of the MPI–IO routines expressiveness.

Our goal was to provide the user with higher level routines than those available in the MPI–IO but still retain the flexibility of the MPI library so the user could further tune his program through direct calls to the MPI–IO. On the other hand, the `lip` routines perform as much optimization as possible (such as the data sieving described in [12, 13]) with the information gained while examining the user data. This may be useful when the user does not want to cope with MPI–IO routines directly. Besides the classical large scale computing the `lip` may also be applied to develop the interfaces for the high performance storage and retrieval of data [19] providing support for parallelism and out-of-core operations, on the other hand, owing to Java bindings, it may be a very important extension to HP Java [20].

The future work includes implementation of efficient data partitioning and remapping routines which should result in a better load-balance and improved efficiency. In addition to this, we are currently working on improvement of the Java binding to the `lip` in order to develop portable and robust wrappers.

Acknowledgments We would like to thank Prof. Peter Brezany for introducing us into the field of out-of-core problems and for many valuable discussions. We are also grateful to Mr Dawid Kurzyniec for his collaboration and to Ms Zofia Mosurska and Dr. Piotr Wyrostek from the ACC CYFRONET for their help. This research was done in the framework of the Polish-Austrian collaboration and it was supported in part by the KBN grant 8 T11C 006 15.

References

1. Brezany, P.: Input/Output Intensively Parallel Computing, LNCS **1220**, Springer, Berlin Heildelberg New York (1997).

2. Saltz, J., et al.: A Manual for the CHAOS Runtime Library, UMIACS Technical Reports CS-TR-3437 and UMIACS-TR-95-34, University of Maryland; March 1995; `ftp://hpsl.cs.umd.edu/pub/chaos_distribution`
3. 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, LNCS **1497**, Springer, pp. 131-136.
4. Moyer, S. and Sunderam, V.S.: Parallel I/O as a Parallel Application. *International Journal of Supercomputer Applications*, **9** (1995) 95-107.
5. 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/`
6. 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, LNCS **1184**, Springer, pp. 85-96.
7. Carretro, J., No, J., Park, S., Choudhary, A., and Chen, P.: COMPASSION: A Parallel I/O Runtime System Including Chunking and Compression for Irregular Applications. In: Sloot, P., Bubak, M., Hertzberger, B., (eds.): *Proc. Int. Conf. High Performance Computing and Networking*, Amsterdam, April 21-23, 1998, LNCS **1401**, Springer, 1998, pp. 668-677.
8. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface, July 18, 1997; `http://www.mpi-forum.org/docs/mpi-20.ps`
9. Saphir, W.: Porting Applications from NX to MPI. Talk presented at NAS on July 26, 1994, `http://parallel.nas.nasa.gov/Parallel/Talks/nx2mpi/nx2mpi.html`
10. Bordawekar, R., Choudhary, A.: Issues in Compiling I/O Intensive Problems. In Jain, R., Werth, J., and Browne, J.C. (Eds.) *Input/Output in Parallel and Distributed Computer Systems*, chapter 3, pp. 69-96, Kluwer, 1996.
11. Brezany, P., Choudhary, A., and Dang, M.: Parallelization of Irregular Codes Including Out-of-Core Data and Index Arrays. In *Proceedings of Parallel Computing 1997 - PARCO'97*, Bonn, Germany, September 1997, Elsevier, pp. 132-140.
12. 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.
13. 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.
14. 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/`
15. MPICH – A Portable Implementation of MPI, `http://www.mcs.anl.gov/mpi/mpich/`
16. Geist, A., et al.: PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, Massachusetts (1994)
17. Java Grande Forum, `http://www.javagrande.org/`
18. LAM/MPI Parallel Computing: `http://www.mpi.nd.edu/lam/`
19. Brezany, P., Winslett, M.: Advanced Data Repository Support for Java Scientific Programming In: Sloot, P., Bubak, M., Hoekstra, A., Hertzberger, B., (eds.): *Proc. Int. Conf. High Performance Computing and Networking*, Amsterdam, April 12-14, 1999, LNCS **1593**, Springer, 1999, pp. 1127-1136.
20. HP Java Project, `http://www.npac.syr.edu/projects/pcrc/HPJava/`