Irregular and Out-of-Core Parallel 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,
Liechtensteinstrasse 22, A-1090 Vienna, Austria
[2]Institute of Computer Science, AGH, al. Mickiewicza 30,
30-059 Kraków, Poland
[3]Academic Computer Centre – CYFRONET, Nawojki 11,
30-950 Kraków, Poland
`brezany@par.univie.ac.at, bubak@uci.agh.edu.pl,`
`malawski@uci.agh.edu.pl, kzajac@uci.agh.edu.pl`
phone: (+48 12) 617 39 64, *fax:* (+48 12) 633 80 54

Abstract. Clusters of workstations are a popular alternative to integrated parallel systems designed and built by a vendor. Besides their huge cumulative processing power, they also provide a large data storage capacity which allows efficient implementations of large-scale applications which are I/O intensive. This paper proposes a parallelization strategy and runtime library solution to the problem of both irregular and out-of-core computing. Although the library is implemented in the C language, it can be easily ported into Fortran and Java environments. The paper presents and discusses experimental performance results using the implementation of the developed software on a PC cluster.
Keywords: parallelization, irregular problems, out-of-core computing, run time library, clusters, Java.

1 Introduction

In the past few years, clusters of workstations have emerged as an important and rapidly expanding approach to parallel computing [12]. These systems often employ inexpensive commodity processors, open source operating systems and communication libraries and commodity networking hardware to deliver supercomputer performance at the lowest possible price. Clusters, besides their large cumulative processing power, also provide a large data storage capacity which allows efficient implementations of large-scale applications which are input and output (I/O) intensive. In order to effectively exploit the power of clusters, appropriate programming environments and development tools are a must.

In this context, a very important research issue is the development of a parallel I/O software support for I/O intensive cluster applications. So far, the major software research efforts were led along two lines, i.e., parallel file system research (e.g., PIOFS [2], PPFS [15], and Galley [22]) and parallel I/O library research (e.g., PASSION [14], DRA [21], and Panda [26]). An excellent description of the recent development in this field can be found in [19]. As an extension to the MPI standard [20], the MPI-IO interface was proposed and the portability of its functions is provided with a software system called ROMIO [28]. Carns et al. [13] developed a parallel file system for Linux clusters, called the Parallel Virtual File System (PVFS) and implemented MPI-IO on top of it.

Although current cluster systems involve large main memories, many of advanced applications require significantly more data than can be held in main memory. Hence, data needs to be stored on disks. Such programs are called out-of-core (OOC) programs; on the other hand, for in-core (IC) programs it is possible to store all their data in the main memory. OOC computations require the program (or the operating system) to move portions of the data set from secondary to primary storage and back as data is needed. A poor I/O capability can severely degrade the performance of the entire program.

Parallelizing irregular OOC programs is a relatively new topic and there has been little research in this area. The report on the PASSION project [14] very briefly outlines the functionality of the runtime support provided for OOC irregular problems. Brezany et al. [6, 7] proposed OOC extensions to the CHAOS library and experimentally implemented several software modules on the Intel Paragon system. So far, more research work has addressed parallelizing regular OOC applications. Nieplocha and Foster [21] developed a parallel library optimizing accesses to two-dimensional out-of-core arrays. Bordawekar et al. [27] worked on compiler methods for out-of-core HPF regular programs.

This paper presents a method for parallelization of OOC irregular problems that is based on a new runtime library system called `lip`. The preliminary versions of the `lip` were presented in [8, 9]. The advanced version of `lip` discussed in this paper provides support for non–trivial load balancing strategies and optimization techniques for I/O operations like buffering and caching of data array accessed irregularly from disks. Its primitives can either be used together with a compiler to translate irregular data parallel OOC programs written in a language like HPF, or used directly by application programmers. The parallelization method and performance are illustrated on synthetic benchmarks and a template taken from the real irregular application.

The paper is organized as follows. Section 2 outlines the strategy used for the parallelization of irregular and out-of-core problems. Section 3 introduces the execution model for the parallelized programs. The kernel part is Section 4 which describes several optimizations techniques we have developed. The library implementation is discussed in Section 5 and performance issues are addressed in Section 6; the benchmarks we have used are implemented in both: *C* and *Java*. We briefly conclude in Section 7.

2 Parallelization of Irregular and Out–of–core Problems

Parallelizing in-core irregular codes for distributed-memory systems is a challenging problem. In such codes, access patterns to major data arrays are only known at runtime, which requires preprocessing in order to determine the data access patterns and consequently, to find what data must be communicated and where it is located. One strategy used for parallelization irregular codes transforms each irregular data-parallel loop into three phases, called *data distributor*, *inspector*, and *executor* [25]. The data distributor determines how to spread the data among the available processors to minimize the communication overhead. The inspector analyzes the communication patterns of a loop, computes the description of communication, and derives the translation functions between global and local accesses, while the executor performs the actual communication and executes the loop iterations. All the phases are supported by an appropriate runtime library. For example, the CHAOS [23] library is a well-known system which supports the handling of irregular computations on massively parallel systems.

Large scale irregular applications involve large arrays and other data structures. The runtime preprocessing provided for these applications results in construction of additional large data structures which substantially increase the memory usage of the program . Consequently, a parallel program may quickly run out of memory. Therefore, some data structures must be stored on disks. Such a program is called *out-of-core program.* The performance of an OOC program strongly depends on how fast the processors the program runs on can access data from disks. It is difficult for the programmer to manage the memory-disk interface explicitly, and moreover, programmer-inserted I/O significantly reduces portability. Therefore, the development of appropriate parallelization methods for OOC irregular programs that are based on portable libraries is an important research issue.

3 Execution Model for Irregular OOC Programs

The *Inspector/Executor* model presented in the previous section can be extended and applied to irregular OOC computing. This can be done by dividing index array into chunks small enough to fit into main memory. The portion of the array which is in processor's memory is called the *i–section.* The main idea is to perform *Inspector* and *Executor* phases for all i–sections as shown in the Fig. 1

The OOC *Inspector* phase is spread into two subphases. First, 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 *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 the desired data into *ghost area.* This part of the *Inspector* phase is activated using `LIP_localize()` call. Next, the references to data residing on a disk are examined and translated into the references to memory buffer. This is done by using `LIP_ooc_localize()` routine after `LIP_localize()` was called. The translation process changes indices in the indirection array and in *communication schedule* so they point into the local memory buffer. This process gives rise to in a data structure called *I/O buffer mapping.* It is updated upon the examination of subsequent *i-sections* to record information what data has to be moved between disk and the main memory

```
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 (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];
  }

  /* 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( /* ... */ );
```

Fig. 1. C program with AVL FIRE kernel that shows parallel computations with out–of–core data array using of `lip`.

as the data that are in the memory are no longer needed for the processing of the next i-section. Once the translation process is done, an I/O buffer mapping object may be used to create MPI derived data types to perform the actual movement of data during the *Executor* phase. Obtaining of the MPI datatypes is done via the `LIP_IObufmap_get_datatype()` call.

The OOC *Executor phase* consists of actual I/O operations, communication and calculation phases. In the I/O phase data that are not needed by the current i–section are written to a disk and replaced by new data. This is done via MPI-IO routines (like `MPI_File_read` and `MPI_File_write()`). Next, communication *schedule* structure is used to perform communication (that brings non-local data into *ghost area* and is done through a call to `LIP_Gather()` routine) and then the computation is performed which uses translated indices and local data array with additional buffer space (*ghost area*). Next, another communication has to be performed to scatter back the results of the computation. The communication *schedule* is used again through a call to `LIP_Scatter()` function. Once the last i–section is processed the rest of the data is written to the disk.

4 Optimizations of Communication and I/O Operations

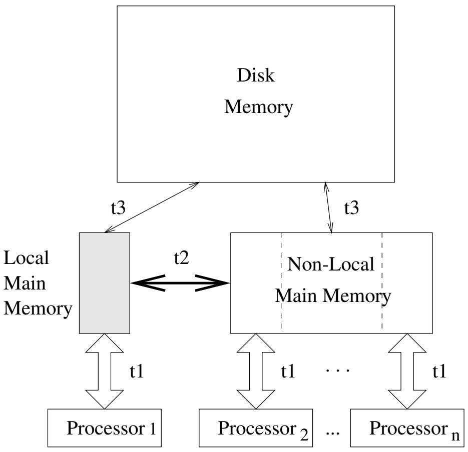

Fig. 2. A simplified scheme of memory hierarchy.

From the point of view of data access the costs incurred during the execution of an OOC program, a *three level hierarchy memory model* can be considered which is depicted in the Fig. 2. It is assumed that processors reference their *local main memories* in unit time, $t_1 = 1$. The model further stipulates that a processor's reference to data that is not within its own internal memory requires a latency of t_2 time units fulfilled if the data is currently in memory of a remote processor and t_3 time units if the data is in disk memory. It is assumed that $t_3 >> t_2 >> t_1$. These memory hierarchy features force programmers and compiler developers to attend to the matter of: (1) locality and data reuse, i.e. how to arrange computations so references to data in local internal memory are maximized, and the time-consuming references to data on disks are minimized, and (2) hiding communication and I/O latency by computations.

Basing on the information about data correlation, `lip` supports the data partitioner that assigns data to the processors so that the related data elements reside on the local disk of the same node. The distribution obtained by the partitioner minimizes the communication overhead.

Irregular accesses to arrays may exhibit some sort of temporal locality. If a data element is referenced in a one i–section, it will tend to be referenced again soon. It is possible to make use of this feature, when optimizing the OOC execution model. To achieve this goal, the contents of the memory buffer are memorized in a special structure, *IOBufmap*. Based on the information stored in this structure, the exchange between memory and disk is done only for the elements that are not currently residing in the memory.

Of course, fetching from the remote memory or reading every element from a disk just as it becomes needed would be very inefficient, so, at the *Inspector Phase*, the optimization has to be aimed at efficient communication and also to minimize I/O accesses associated with *not-in-memory* data elements. After having examined and translated the indices the communication scheme and *IOBufmap* structure are created. The I/O and communication phases are performed only once before computation. In that way the startup message cost and the I/O routines cost are minimized. It is assumed that the optimization of reading or writing data of an irregular access pattern from a disk is supported by MPI–IO implementation and it is usually solved by implementing the data sieving techniques [29].

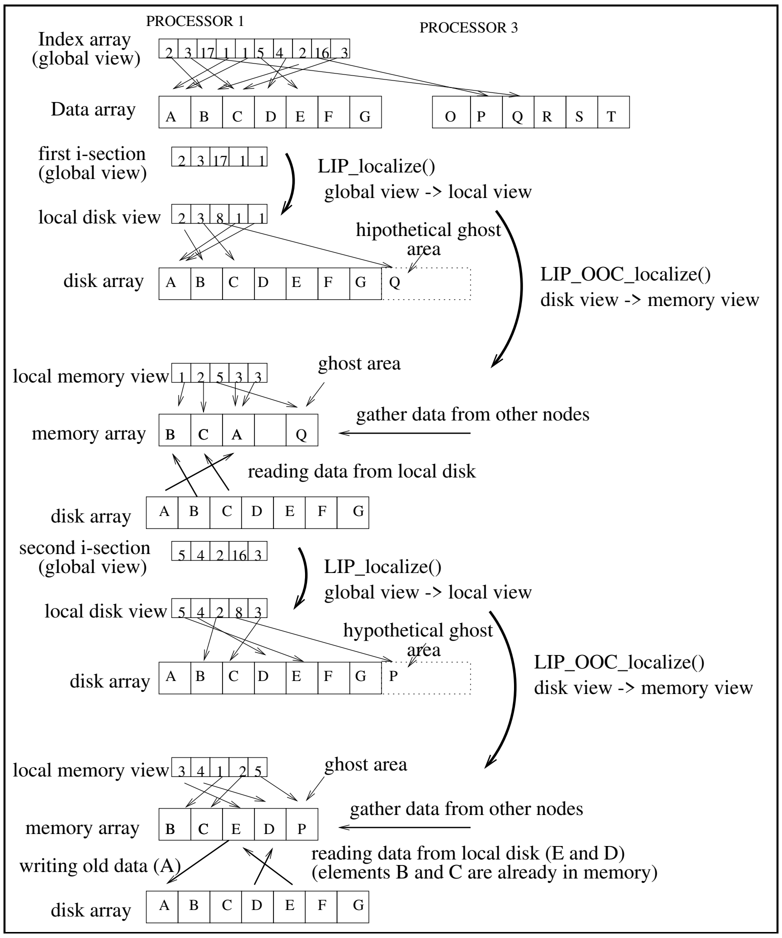

Fig. 3. Explanation of I/O caching and operations coalescing.

In Fig. 3, we show a case study of I/O optimization techniques used with the `lip` routines for a simple example. In this figure, processor 1 needs the data elements marked with A, B, C, D, E from the local disk and the data elements marked with P and Q that reside on a remote node's disk. As the data cannot fit in the memory, the index array is divided into two i–sections. When processing the first i–section, the processor needs to fetch the data element marked with Q from the remote node and read the data elements marked with A, B and C from the local disk. When translating the indices from the global to the local view, the communication structure is created to be used to fetch the non–local elements to the hypothetical ghost area on the local disk. At this stage all the indices point to the local disk space. The next step transforms them in such a way that they point into the local memory buffer. After the translation process the data element marked Q is fetched to the ghost area. Also the local elements A, B, C are read from the disk to the memory. The reading is performed using a single routine call for all the elements.

When processing the second i–section, the processor needs the B, C, E and D data elements. To be able to accommodate them in the memory buffer, it has to write one element, A, to the disk. B and C remain in the buffer, because they are still needed. The new elements, D and E, are loaded to the memory using a single reading routine call. The P data element is fetched from the remote node to the ghost area.

5 Implementation

The functions of the `lip` library are divided into the following groups: partitioning, data redistribution, index translation, IODatamap manipulation, Schedule manipulation and communication.

Partitioning At present, `lip` supports coordinate-based partitioner using Hilbert's curve [24, 30] to optimize the data partitioning. The library supports both the in-core and OOC version of the partitioner. A partitioner is activated using the `LIP_create_hilbert_distribution_incore()` or `LIP_create_hilbert_distribution_ooc()` call.

Data Redistribution The procedures `LIP_remap_incore()` and `LIP_remap_ooc()` are provided which allow to dynamically redistribute the data. They include descriptors of the source and target distributions as argument.

Index Translation Function `LIP_Localize()` translates the globally numbered indices into the locally numbered counterparts. It also updates communication schedule so it could be used to transmit the non-local data. With a call to the `LIP_ooc_localize()` function the indices which pointed to data residing on disk are translated into references to the memory data buffer. During this process the communication schedule is updated and I/O buffer mapping structure is filled with information which may be used to obtain the MPI derived data types for moving the data between disk and memory.

IODatamap Manipulation `lip` includes I/O buffer mapping objects which store information about the mapping of the memory buffer onto a file. The library also contains the functions to operate on these objects and to get the information used to perform data movement with MPI-IO functions.

Schedule Manipulation and Communication The `lip` library includes group of functions used to create and manipulate on a communication schedule. The library includes two general communication functions: `LIP_Gather()` and `LIP_Scatter()`. They perform collective communication between nodes prior to and after computational phase, support numerous communication patterns and data copy/update schemes, and they use schedule object to obtain information about the source and destination of data.

6 Performance Results

6.1 `lip` library performance in comparison to virtual memory

To demonstrate the advantages of the out–of–core support provided by `lip`, we compared performance of codes using `lip` with their performance achieved when using virtual memory support. The target system used was a cluster of 8 Linux Workstations with processor Pentium III 600 MHz with 128 KB cache and 64 MB RAM. Each node was equipped with a local disk that was used by the OOC computation or as a system swap area. The test computation consisted of two generic loops that access the `double` array `x` in indirect way for reading and for writing (see Fig. 4). The size of the problem was scaling with the number of processors so that the arrays on each node consisted of 3,024,000 elements (`y` and `perm`) and 6,048,000 for `x`. The tests were carried out on a cluster system where each node was equipped with 64MB of RAM. There was not enough main memory space for allocation of data and index arrays and all auxiliary data structures (e.g. communication schedules), so the Linux system was forced to use its virtual memory mechanisms to enable allocation of the needed space in the disk swap area.

On the other hand, when the `lip` out–of–core functions were applied (in the way shown on the Fig. 1) the arrays stored on the local disk of each node could be divided into 10 i–sections, each of them small enough to fit into memory so that no swapping was required. The results shown in Fig. 5 demonstrate a good scalability of the computation using `lip` out–of–core functions. Except the trivial one–node case, when no communication and, consequently, no additional memory–consuming data structures are needed, the virtual memory mechanisms perform with much lower efficiency than the `lip` library.

```
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++)
{
      x[perm[i]]+=y[i];
}
```

Fig. 4. Irregular loop used for OOC performance evaluation.

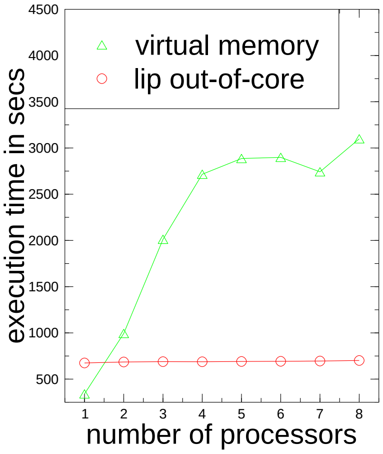

Fig. 5. Advantage of `lip` library over virtual memory.

6.2 Performance of AVL FIRE kernel

As a test application we used a benchmark including a loop whose computation structure is shown in Fig. 1. This benchmark is derived from the FIRE benchmark solver `GCCG` [1, 5]

The performance of the OOC GCCG solver on our Linux cluster for different number of processors is given in Fig. 6. The tests were performed on a dataset consisting of 47312 cells (size of `direc2` array). The data was block-wise distributed. In the first case all arrays were located *in-core*, while in the second one the index `lcc` array was distributed on local disks (*out-of-core*). As can be seen, the problem scales with a number of processors for both the in-core and OOC version.

6.3 Performance of irregular application in Java

To create the Java bindings to the `lip` library we used the tool called ***Janet*** [10, 11, 16] . ***Janet*** is an extension to Java language that allows inserting native (i.e. C) language statements directly into Java source code files. The files with so extended Java syntax are the input for ***Janet*** translator that creates automatically Java and native source files with all required calls to JNI [17] interface. It enables clear and flexible creation of Java interfaces to the native library.

To test the performance of irregular computation in Java we used the OOC version of test loops presented in Fig. 4, and data consisted of 133,400 cells associated with `double` values. The results presented in Fig. 7 were obtained on our Linux cluster with Java virtual machine from Sun JDK 1.3.

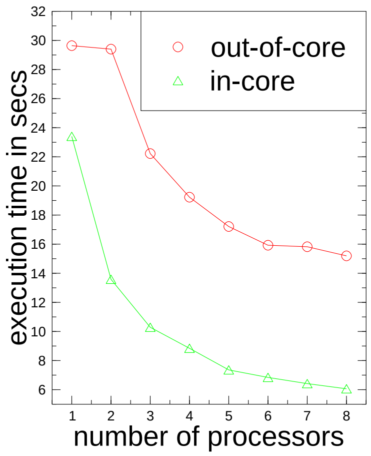

Fig. 6. Comparison between in–core and OOC solver performance.

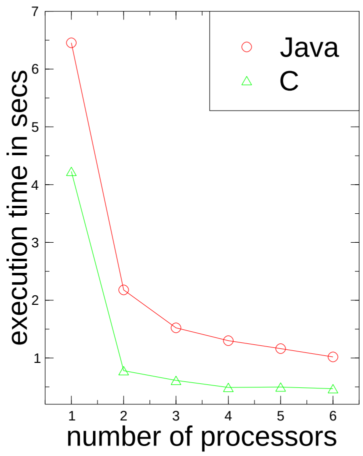

Fig. 7. Comparison between C and Java performance.

7 Summary

Based on several studies of parallelization of in-core and out-of-core irregular applications for distributed-memory systems, we have designed a new parallel library that is intended to provide high performance to a wide variety of applications. lip is based on the inspector and executor approach and on using MPI–I/O functions. The library is available on the web [18].

This paper presents performance results for the kernel of the AVL FIRE code on a cluster of PCs. Work is going on improving the stability of the system in general. Future work will focus mainly on performance optimization.

Acknowledgements We are grateful to Agnieszka Wierzbowska, Piotr Łuszczek, Dawid Kurzyniec for their contribution and Dr Włodzimierz Funika for his remarks. 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. G. Bachler, R. Greimel. Parallel CFD in the Industrial Environment. Unicom Seminars, London, 1994.
2. F. Bassow. IBM AIX Parallel I/O File System: Installation, Administration, and Use. *IBM, May 1995. Document Number SH34-6065-00.*
3. R. Bordawekar, A. Choudhary. Issues in Compiling I/O Intensive Problems. In: R. Jain et al. (eds.) Input/Output in Parallel and Distributed Computer Systems, Kluwer Academic Publ., 1996, pp. 69–96.

4. R.R. Bordawekar et al., *A Model and Compilation Strategy for Out-of-Core Data Parallel Programs.*
5. P. Brezany et al., *Automatic Parallelization of the AVL FIRE Benchmark for a Distributed-Memory System*, PARA95.
6. P. Brezany, A. Choudhary, and M. Dang, Parallelization of Irregular Out-of-Core Applications for Distributed-Memory Systems. Proceedings of HCPN 97, Vienna, April 1997, Springer-Verlag, LNCS 1225.
7. P. Brezany, A. Choudhary, and M. Dang, Parallelization of Irregular Codes Including Out-of-Core Data and Index Arrays in "Parallel Computing 1997 - PARCO'97", Bonn, Germany, September 1997.
8. M. Bubak, P. Łuszczek, Towards Portable Runtime Support for Irregular and Out-of-Core Computations. In: J. Dongarra et al. (Eds.), *Proc. of 6th European PVM/MPI Users' Conf.*, Barcelona, Spain, Sept. 1999, LNCS **1697**, Springer-Verlag, pp. 59-66.
9. M. Bubak, P. Łuszczek, M. Malawski, K. Zając, Irregular and Out-of-Core Computing on SGI Origin 2000 in: Proceedings of 1st SGI Users Conference, Cracow, Poland, October 2000, pp. 185-193.
10. 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.
11. 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.
12. R. Buyya, ed., High Performance Cluster Computing, Vol. 1 and 2. *Prentice Hall*, New Jersey, 2000.
13. P.H. Carns er al.. PVFS: A Parallel File System for Linux Clusters. In *Proc. of the Externe Linux Track: 4th Annual Linux Showcase and Conference*, October 2000.
14. A. Choudhary et al., PASSION: parallel and scalable software for input-output. *Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University*, September 1994.
15. J. Huber et al., PPFS: A High-Performance Portable Parallel File System. *Proc. of the 9th ICS Conference, Barcelona*, July 1995, pp. 385-394.
16. Janet homepage: `http://www.icsr.agh.edu.pl/janet`
17. JavaSoft: Java Native Interface.
`http://java.sun.com/products/jdk/1.2/docs/guide/jni/`
18. `lip` library `http://www.icsr.agh.edu.pl/ malawski/lip`
19. J.M. May. Parallel I/O for High Performance Computing. *Morgan Kaufmann Publ.*, 2000.
20. MPI Forum, MPI-2. July 1997.
21. J. Nieplocha and I. Foster, DRA: An Array-Oriented I/O Library for Out-Of-Core Computations. In *Proc. of the 6th Symposium on the Frontiers of MP Computation*, IEEE Press, October 1996, pp. 196–204.
22. N. Nieuwejaar and D. Kotz, The Galley parallel file system. In *Proc. of the 10th ACM International Conference on Supercomputing*, May 1996.
23. R. Ponnusamy et al., CHAOS runtime library. Techn. report, University of Maryland, May 1994.
24. R. Preis, R. Diekmann, The PARTY Partitioning-Library, User Guide - Version 1.1. Technical Report, University of Paderborn, Sep. 1996.

25. J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, *Run-time scheduling and execution of loops on message passing machines*. Journal of Parallel and Distributed Computing, 8(2):303–312, 1990.
26. K.E. Seamons and M. Winslett, Multidimensional Array I/O in Panda 1.0. In *Journal of Supercomputing*, Vol. 10, No. 2, pages 191–211.
27. R. Thakur, R. Bordawekar, and A. Choudhary, Compiler and Runtime Support for Out-of-Core HPF Programs. In *Proceedings of the 1994 ACM ICS Conf.*, pages 382–391, Manchester, July 1994.
28. R. Thakur, W. Gropp, and E. Lusk, On Implementing MPI-IO portably and with high performance. In *Proc. of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23-32. ACM Press, May 1999.
29. 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.
30. C. Walshaw, Parallel Jostle Library Interface: Version 1.1.6. University of Greenwitch, London, June 16, 1999, (http://www.gre.ac.uk/jostle).