Irregular and Out-of-Core Computing on SGI Origin 2000

Marian Bubak[1,2], Piotr Łuszczek[3], Maciej Malawski[1], and Katarzyna Zając[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
[3]Dept. of Computer Science, Univ. of Tennessee, Knoxville, TN 37996-1301, USA
`bubak@uci.agh.edu.pl, luszczek@cs.utk.edu`

Abstract

This paper presents the `lip` runtime system which was elaborated to support parallelization of irregular and out-of-core computations. Functions for handling irregular data were developed using the similar approach as in the CHAOS library while out-of-core parallelization is based on the idea of *in-core section.* Functions for out-of-core data are implemented with capabilities provided by MPI–IO. Preliminary results of performance tests for a generic irregular out-of-core program on SGI Origin 2000 are presented.
Keywords: irregular problems, out-of-core computing, I/O operations, runtime support.

1 Introduction

Irregular and out-of-core (OOC) applications are examples of interesting and important problems in the field of large scale computing [1]. In irregular problems the access to data is through one or more levels of indirection arrays. A few runtime tools have been developed to facilitate parallelization of irregular problems, with CHAOS runtime library [2] being the most notable among them. Many applications involve primary data structures which are too large to fit into main memory; they are called *out-of-core* problems. To parallelize this kind of programs one may use explicit I/O calls, e.g., these offered by MPI–IO. There are also some dedicated tools, e.g., PIOUS [3] and Panda [4] systems.

As one may expect, especially difficult to parallelize are problems which are at the same time irregular and out-of-core like large particle codes, CFD programs, and sparse matrix computations. Their parallelization is facilitated by runtime systems like VIPIOS which is dedicated to be used together with Vienna HPF [5] and COMPASSION [6].

This paper presents the `lip` – a runtime support library for both in- and out-of-core irregular problems. Portability of the library is guaranteed by the use of MPI with I/O extensions ([7]). Parallelization of irregular problems is similar to that of CHAOS [2], while out-of-core functions are based on the idea of *in-core section.* The `lip` may be used in C, Fortran and Java parallel programs developed with MPI, as well as by parallelizing compilers. Previously, we have tested the `lip` on HP S2000 [8] and on a clusters of PCs [9]; here, we present preliminary results of performance tests for a generic irregular out-of-core program on SGI Origin 2000.

2 Parallelization of Irregular Problems

Runtime support for parallel execution of irregular code is based on so called *Inspector/Executor* technique introduced in CHAOS [2]. Parallelization is split into two independent phases: *Inspector Phase* and *Executor Phase*.

In the *Inspector 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 *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*.

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 the call to `LIP_Gather()` routine) and then 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 results of the computation. Communication *schedule* is used again through call to `LIP_Scatter()` function.

3 Idea of Parallelization of OOC Problems

The *Inspector Phase* is presented in Fig. 1 as transition from global to local view. The last part of the Fig. 1 shows OOC local view which pertains to the case when index array (which is examined during *Inspector phase*) is to large to fit into memory. The *Inspector/Executor* approach remains almost unaltered thanks to the introduction of the concept of *in-core section* which is sometimes referred to as tile or slab [10]. The whole index array is stored in a file and only small part of it – *i-section* – is loaded into main memory to enable the previously described optimisation. 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. Generic program which uses *i-section* is given in Fig. 2.

The concept presented above has to be further modified when not only index arrays but also data arrays become *out-of-core*. 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 conclusion that three types of non-local accesses may be observed in the index array:

- access to data which is stored in another's node memory,

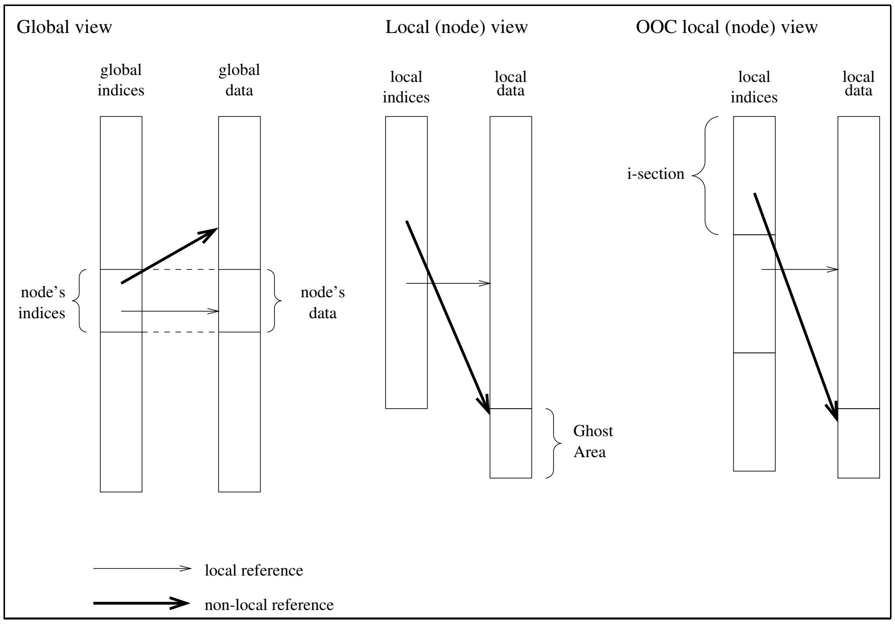

Fig. 1: Translation of index arrays for irregular and out-of-core computing.

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

- 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).

Of course, 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 not only efficient communication but also to minimize 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 memory buffer. The translation process changes indices in indirection array and in *communication schedule*. This process results in a data structure called *I/O buffer mapping*. It is updated upon examination of subsequent *i-sections* to record information what data has to be moved between disk and main memory as data that are in the memory are no longer needed for the during processing of the next i-section. After translation process is done, I/O buffer mapping object may be used to create MPI derived data types to perform actual movement of data using MPI-IO routines. In this way the irregular problems for which all the arrays are OOC may also be parallelized. Generic program which uses this technique is shown in Fig. 3.

```c
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
   * and disk-memory mapping pattern)*/
  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) */

  /* Read data from disk to memory
   * in order to store in the memory
   * the data that will be needed
   * in this i-section */
  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 derived 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( /* ... */ );
```

Fig. 3: Simplified C code that shows the use of parallel computation on out-of-core data array with the use of the `lip`.

4 Short Overview of the lip Library Specification

The functions of the `lip` library are divided into the following groups:

- data mapping,
- communication schedule generation and transformation,
- index translation,
- communication functions,
- miscellaneous.

The data mapping group consists of functions to create and manipulate `datamap` objects which describe how data array is partitioned among processors and I/O buffer mapping objects which store information about the mapping of the memory buffer onto a file. There are also functions to obtain the MPI derived data types according to the information stored in the I/O buffer mapping object in order which, in turn, may be used to perform data movement with MPI-IO functions.

The next group of functions is used to create a communication schedule for local data array of a given size, make a schedule object ready for communication after it was updated by the index-translation functions of the `lip`, and to release the memory space occupied by a schedule.

Two functions: `LIP_Localize()` and `LIP_OOC_localize()` are used for index translation. The former translates globally numbered indices into locally numbered counterparts. It also updates communication schedule so it could be used to transmit non-local data. The latter function transforms indices which point to data residing on a disk into indices to memory data buffer. During this process the communication schedule is altered 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.

The `lip` library includes two general communication functions: `LIP_Gather()` and `LIP_Scatter()`. They perform collective communication between nodes before and after computational phase. They support numerous communication patterns and data copy/update schemes. They use schedule object to obtain information about source and destination of data.

5 Implementation

One of our goals in developing the `lip` was to use MPI–IO standard API to achieve portability and efficiency. Thus, we have used ROMIO [11] (portable implementation of MPI–IO) version 1.0.1 which is included in native MPI implementation from SGI MPT 1.4, which, in turn, was used in the tests (see Section 6).

In order to take advantage of efficiency, flexibility and expressivenes of native MPI implementation we have used derived data types with MPI–IO functions to access data that are out-of-core. However, in the course of our experiments we have encountered several problems, all of which pertained to the derived data types that are crucial in our case. The SGI's bug 777712 was discovered when we have attempted to use index array with nonincreasing set of indices. This was solved by a software upgrade to SGI's MPT 1.4.0.1. But even the upgraded library didn't support the `MPI_Create_indexed_block()` function from MPI 2.0 standard (section 4.4), so we have implemented this function using `MPI_Type_indexed()` (MPI 1.2 standard). This, however, incurred excessive memory use as the data and index arrays grew bigger. This resulted in submission of SGI's bug 796623.

6 Sample Results

To evaluate performance of the `lip` we have used generic irregular loop which operates on large data and indices sets (see Fig. 2). First, data are generated along with indices (which could describe mesh or graph connections), and next they are used to index the data array. The indiex array is OOC so it is divided into *i-sections* and is stored on the disk(s). During computation phase *i-sections* are loaded and used in irregular computation.

Tests were run on SGI Origin 2000 with IRIX 6.5 operating system, on the subcomplex of 8 processors and with total limit of 2 GB memory. For I/O operations global I/O model was used, with one file shared between all processors. Native MPI implementation from SGI MPT 1.4 was used to perform MPI-IO operations. Test program consisted of generic irregular loop computing a sum with index array of size 453600 and data array 907200 8-byte double values per processor.

#loop iterations	#nodes	2	3	4	5	6	7	8
1	time	1.37	1.85	2.09	2.31	2.65	3.41	3.61
	δ	0.12	0.16	0.36	0.16	0.33	0.22	0.25
100	time	45.6	47.8	36.0	49.4	38.2	44.6	70.8
	δ	7.64	7.46	1.63	8.61	3.77	0.89	5.39

Tab. 1: Elapsed time (s), all data in-core

#loop iterations	#nodes	2	3	4	5	6	7	8
1	time	13.4	15.6	16.6	17.4	18.7	19.7	22.0
	δ	0.31	0.66	0.54	2.53	1.23	3.04	4.44
100	time	33.8	36.4	39.8	46.2	48.0	43.3	58.6
	δ	3.14	2.59	4.34	3.56	6.20	2.33	11.65

Tab. 2: Elapsed time (s), data out-of-core.

Two test cases were run: one in which all index and data arrays were stored in memory (rendering the problem completely *in-core*) – see Table 1, and the second where the data array was OOC and was operated on using calls to the `lip` – see Table 2. For each case there were two subcases: in the first one computations in irregular loop were very light weight in terms of time, while in the second one they were much more intesive thus dominating communication. The timings were performed under usual workload of other users who were sharing disk and communication interconnect – only processors and memory was dedicated to subcomplex, so measurements were repeated 10 times and the averaged results with their standard error are shown in the tables.

7 Conclusions and Future Work

The `lip` library may be used for parallelization of irregular and/or out-of-core applications on MPPs, clusters and on heterogeneous networks of workstations. Binding to Fortran and Java enhance considerably its potential usage. Important feature of the `lip` is its support for efficient handling of irregular access patterns (as they are encountered in irregular problems) through high-level data structures called *schedules*. This is done in uniform manner for I/O access routines and for communication. With these data structures, information may be transferred from the phase of analysis of user data (*Inspector Phase*) to communication and I/O phase (*Executor Phase*). Then it is possible to take full advantage of MPI–IO routines expressiveness.

Our goal was to provide user with higher level routines than those available in MPI–IO but still retain flexibility of the MPI library so the user could further tune his program through direct calls to MPI–IO. On the other hand, routines of the `lip` perform as much optimization as possible (such as data sieving described in [12, 13]) with the information gained while examining user's data. This may be useful when the user does not want to cope with MPI–IO routines directly.

Besides the classical large computing the `lip` library may be also applied to develop interfaces for high performance storage and retrieval of data [14] providing support for parallelism and out-of-core operations, on the other hand, thanks to Java bindings, it may be a very important extension to HP Java [15].

At present we are working on implementation of efficient data partitioning and remapping routines which should result in better load-balance and increase of efficiency.

Acknowledgements
We are grateful to Ms. Zofia Mosurska, Mr. Andrzej Ozięblo, and Mr. Stefan Świąć from the ACC CYFRONET, as well to Dr. Wolfgang Mertz – HPC Consultant, CSBU Europe, SGI, Germany, and Dr. Timothy Sirianni, SGI 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, Lecture Notes in Computer Science, **1220**, Springer, Berlin Heildelberg New York (1997).
2. Saltz, J., Ponnusammy, 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; `ftp://ftp.cs.umd.edu/pub/hpsl/chaos_distribution/`.
3. Moyer, S. and Sunderam, V.S.: Parallel I/O as a Parallel Application. *International Journal of Supercomputer Applications*, **9** (1995) 95-107.
4. 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/`
5. Brezany, P., Mueck, T.A., and Schikuta, E.: A Software Architecture for Massivelly 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.
6. 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, Lecture Notes in Computer Science **1401**, Springer, 1998, pp. 668-677.
7. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface, July 18, 1997; `http://www.mpi-forum.org/docs/mpi-20.ps`
8. Bubak, M., Luszczek, P., "Out-of-core Computations on HP S2000", in: *Proc. Int. Conf. High Performance Computing on Hewlett-Packard Systems*, Tromsoe, Norway, June 27-30, 1999, Book 2, pp. 419-424, Universitet Tromso, 1999.
9. Bubak, M., Luszczek, 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, Lecture Notes in Computer Science **1697**, Springer-Verlag Berlin-Heidelberg, 1999, pp. 59-66.
10. 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, North-Holland, pp. 132-140.
11. 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/`
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. 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, Lecture Notes in Computer Science **1593**, Springer, 1999, pp. 1127-1136.

15. Bubak, M., Kurzyniec, D., Łuszczek, P., "A Versatile Support for Binding Native Code to Java", in: Bubak, M., Afsarmanesh, H., Williams, R., Hertzberger, B., (Eds.), Proc. Int. Conf. High Performance Computing and Networking, Amsterdam, May 8-10, 2000, Lecture Notes in Computer Science **1823**, pp. 373-384, Springer, 2000.