PASSION Runtime Library for the Intel Paragon*

Alok Choudhary *Rajesh Bordawekar* *Sachin More* *K. Sivaram*†
Dept. of Electrical and Computer Engineering
Syracuse University, Syracuse, NY 13244
`choudhar, rajesh, ssmore, sivaram @cat.syr.edu`

Rajeev Thakur
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
`thakur@mcs.anl.gov`

Abstract

We are developing a runtime library which provides a number of routines to perform the I/O required in parallel applications in an efficient and convenient manner. This is part of a project called PASSION, which aims to provide software support for high-performance parallel I/O at the compiler, runtime and file system levels. The PASSION Runtime Library uses a high-level interface which makes it easy for the user to specify the I/O required in the program. The user only needs to specify what portion of the data structure needs to read from or written to the file, and the PASSION routines will perform all the necessary I/O efficiently. This paper gives an overview of the PASSION Runtime Library and describes in detail its high-level interface.

1 Introduction

Parallel computers are becoming increasingly powerful day by day. This has made possible the solution of many problems which were previously considered intractable. These include large scale applications in physics, chemistry, biology, engineering, medicine and other sciences, as well as in other areas such as information technology. Many of these applications deal with large data sets and hence have significant I/O requirements. Improvements in the I/O performance of parallel computers have not kept pace with improvements in their computation and communication capabilities. This results in the I/O system being the bottleneck in many cases.

There are a number of reasons why I/O may be needed in a parallel program [7]. In many applications, all the data required by the program cannot fit in main memory and so has to be stored in files on disks. Such programs are called *out-of-core* programs. In out-of-core programs, I/O is needed to access the entire data set. I/O may also be required in *in-core* programs where all the data can fit in main memory. For example, it may be necessary to read input data from files at the start of the computation and write results to files at the end of the computation. During the computation, it may be necessary to periodically write data to files to monitor the progress of the solution. In applications which run for a long time, it may be necessary to checkpoint (stop) the computation at some point and restart it later. This requires saving the contents of the data structures in files. I/O may also be required for the purpose of debugging a parallel program.

We are working on a project called PASSION (Parallel and Scalable Software for Input-Output) which aims to provide software support for high-performance parallel I/O on distributed memory parallel computers [1]. PASSION provides support at the compiler, runtime and file system levels. The PASSION Runtime Library provides a number of optimized routines to perform the I/O required in parallel applications in an efficient manner. It uses a high-level interface which makes it easy for the user to specify the I/O required in the program. The interface also enables the use of collective I/O in which processors cooperate to

*This work was supported in part by a grant from Intel SSD and NSF Young Investigator Award CCR-9357840. This work was performed in part using the Intel Paragon and Touchstone Delta Systems operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by CRPC.

†Dept. of Computer and Information Science, Syracuse University

perform I/O efficiently. The user is freed from the burden of explicitly manipulating file pointers, calculating file offsets, managing buffers and other tedious tasks associated with using the low-level interface provided by parallel file systems. This paper gives an overview of the PASSION Runtime Library and describes in detail its high-level interface.

The rest of this paper is organized as follows. Section 2 gives a brief overview of the PASSION Runtime Library. The need for providing high-level interfaces for parallel I/O is explained in Section 3. Section 4 describes the various data structures used by the PASSION library. The interface used by several of the PASSION routines is described in Section 5, followed by Conclusions in Section 6.

2 Overview of the PASSION Runtime Library

The PASSION Runtime Library provides routines to efficiently perform the I/O required in parallel applications, both in-core as well as out-of-core. It supports a loosely synchronous Single Program Multiple Data (SPMD) programming model. The PASSION library uses a simple high-level interface, which is a level higher than any of the existing parallel file system interfaces, as shown in Figure 1. For example, the user only needs to specify what section of the array needs to be read in terms of its lower-bound, upper-bound and stride in each dimension, and the PASSION Runtime Library will fetch it in an efficient manner. PASSION thus provides a simple and portable level of abstraction above the native parallel file system provided on the machine. The PASSION library is designed to either be directly used by application programmers, or a compiler could translate out-of-core programs written in a high-level data-parallel language like High Performance Fortran (HPF) to node programs with calls to the library for I/O. A number of optimizations, such as two-phase I/O, data sieving, data prefetching and data reuse, have been incorporated in the library [11, 12, 10].

2.1 Architectural Model

The architectural model assumed by PASSION is that of any general distributed memory computer in which the processors are connected together in some fashion. The system is assumed to be provided with a set of disks and I/O nodes. The I/O nodes can either be dedicated processors or some of the compute nodes may also serve as I/O nodes. Each processor may either have its own local disk or all processors may share the set of disks. The I/O subsystem may have a separate interconnection network or it can share the same network which connects the processors together. Thus the architectural model of PASSION conforms to that of any of the commercially available parallel computers. The PASSION library was originally implemented on the Intel Paragon and Touchstone Delta systems. It is currently being ported to other machines.

2.2 Data Storage and Access Models

In out-of-core programs, all the data required by the program cannot fit in main memory, and so has to stored in files on disks in some fashion. PASSION supports two basic models for storing and accessing data, called the *Local Placement Model (LPM)* and the *Global Placement Model (GPM)*.

2.2.1 Local Placement Model (LPM)

In this model, the global array is divided into local arrays belonging to each processor. Since the local arrays are out-of-core, they have to be stored in files on disks. The local array of each processor is stored in a separate file called the *Local Array File (LAF)* of that processor. The node program explicitly reads from and writes to the file when required. The simplest way to view this model is to think of each processor as having another level of memory which is much slower than main memory. If the I/O architecture of the system is such that each processor has its own disk, the LAF of each processor will be stored on the disk attached to that processor. If there is a common set of disks for all processors, the LAF will be distributed across one or more of these disks. In other words, we assume that each processor has its own *logical disk* with the LAF stored on that disk. The mapping of the logical disk to the physical disks depends on how much control the parallel file system provides the user. At any time, only a portion of the local array is fetched and stored in main memory. The size of this portion depends on the amount of memory available. The portion of the local array which is in main memory is called the *In-Core Local Array (ICLA)*. All computations are performed on the data in the ICLA. Thus, during the course of the program, parts of the LAF are fetched into the ICLA, the new values are computed and the ICLA is stored back into appropriate locations in the LAF.

2.2.2 Global Placement Model (GPM)

In this model, the global array is stored in a single file called the *Global Array File (GAF)*, and no local array files are created. The global array is only logically divided into local arrays in keeping with the SPMD programming model. But, there is a single global array on disk. The PASSION runtime system fetches the appropriate portion of each processor's local array from the global array file, as requested by the user. The advantage of the Global Placement Model is that it saves the initial local array file creation phase in the Local Placement Model. In addition, if the distribution of the array among processors needs to be

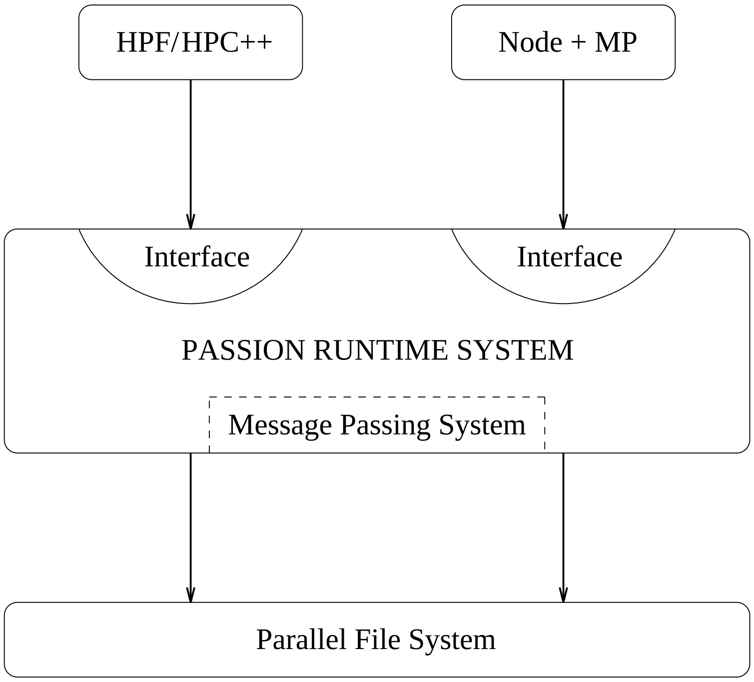

Figure 1: Software Architecture

changed during program execution, an explicit redistribution of the out-of-core data is not required. The disadvantage is that each processor's data may not be stored contiguously in the GAF, resulting in multiple read requests and higher I/O latency time. However, this drawback can be overcome to a large extent by using the Two-Phase Method for I/O [6, 12]. Also, in the Global Placement Model, explicit synchronization is required when a processor needs to access data that may have been previously modified by another processor.

2.3 Optimizations

A number of optimizations have been incorporated in the PASSION Runtime Library. We briefly describe some of them below. Further details and performance results are given in [11, 12, 10].

2.3.1 Collective I/O Using a Two-Phase Method

In data parallel programs, all processors perform similar operations but on different data sets. Hence if one processor needs to read data from disks, it is very likely that a group of processors or maybe all processors need to read data from disks at about the same time. This makes it possible for the requesting processors to cooperate in reading or writing data in an efficient manner, which is known as *collective I/O*. If processors perform I/O independently, it may result in a large number of low granularity requests which may arrive from different processors in any order. On the other hand, if processors use collective I/O, they can cooperate among themselves to perform I/O efficiently in large chunks and in the right order.

The PASSION library performs collective I/O using a Two-Phase Method [6, 12]. This can be used to read/write either entire arrays or sections of arrays with/without strides in each dimension. In the Two-Phase Method, I/O is done in two phases. In the first phase, processors cooperate to read data in large contiguous chunks. A dynamic scheme is used to partition the I/O workload among processors, depending on the access requests [12, 10]. In the second phase, data is redistributed among processors using interprocessor communication, so that each processor gets the data it requested. The main advantages of the Two-Phase Method are:-

- It results in high granularity data transfer between processors and disks.
- It makes use of the higher bandwidth of the processor interconnection network.

2.3.2 Data Sieving

All PASSION routines for reading or writing data from/to disks support the reading/writing of regular sections of arrays with strides. For example, a processor may want to read a section of an out-of-core two-dimensional array given by its lower-bound, upper-bound and stride in each dimension ($l_1 : u_1 : s_1, l_2 : u_2 : s_2$). The interfaces provided by most of the parallel file systems at present do not support strided accesses. Hence the only way of reading this array section using a direct method is to explicitly move the file pointer to each element and read it individually. This requires as many reads as the number of elements in the section. The major disadvantage of this method is the large number of I/O calls and low granularity of data transfer. Since I/O latency is very high, this method proves to be very expensive [11].

An optimization called data sieving is used in PASSION to read/write strided data efficiently. For reading a strided section, instead of reading only the requested elements, large contiguous chunks of data are read at a time into a temporary buffer in main memory. This includes unwanted data. The useful data is extracted from the buffer and passed on to the calling program. The amount of data read in each read operation depends on the amount of temporary space available. A similar method is used for writing regular sections, except that this requires an extra read before the write, to avoid overwriting any data already present in the file. The advantage of data sieving is that it results in higher granularity data transfer, though extra data is also transferred in the process. We found that data sieving provides considerable performance improvement [11, 10].

2.3.3 Data Prefetching

In both the Local and Global Placement Models, program execution proceeds by fetching data from a file, performing the computation on the data and writing the results back to a file. This is repeated on other data sets till the end of the program. Thus I/O and computation form distinct phases in the program. A processor has to wait while each data set is being read or written as there is no overlap between computation and I/O. The time taken by the program can be reduced if it is possible to overlap computation with I/O in some fashion. A simple way of achieving this is to issue an asynchronous I/O read request for the next data set immediately after the current data set has been read. This is called *data prefetching.* Since the read request is asynchronous, the reading of the next data set can be overlapped with the computation being performed on the current data set. If the computation time is comparable to the I/O time, this can result in significant performance improvement [11, 10].

2.3.4 Data Reuse

In many applications, a portion of the current data set fetched from the file is also needed for computation on the next data set. To reduce the amount of I/O, the data already fetched into main memory can be reused

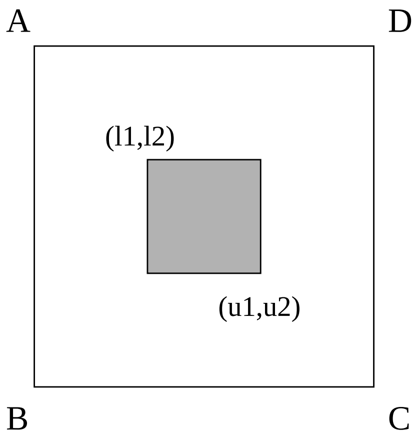

Figure 2: Processor 0 needs to access section ($l_1 : u_1, l_2 : u_2$) of the out-of-core array ABCD, stored in a file in column-major order.

instead of reading it again from disk. The amount of data reuse depends on the intersection of the sets of data needed for computation on the portion of data currently fetched into memory and the portion that will be fetched next.

3 High-Level Interfaces

Most parallel file systems provide a one-dimensional view of data, i.e. the file is viewed as a linear sequence of records. The user needs to know how the data structure in the program is mapped to this one-dimensional sequence of records. For example, a two-dimensional array may be stored in the file in row-major or column-major order. To read/write a portion of the data, the user has to explicitly calculate where the data is located in the file, move the file pointer to that location and then read/write data. Also, the interface provided by most parallel file systems does not support strided accesses. If the required data lies strided in the file, the user has to explicitly seek to each contiguous portion and read/write that contiguous portion. We call such an interface a low-level interface.

For example, consider Figure 2. ABCD is a large out-of-core array stored in a file in column-major order. Processor 0 needs to read a section of this array given by the indices ($l_1 : u_1, l_2 : u_2$). This section does not lie contiguously in the file. Each column of the section is located contiguously, but the individual columns are separated by some other data. The only way to read this section using the traditional low-level interface provided by a parallel file system is to explicitly seek to the first element of each column, read all elements in the column, then seek to the first element of the next column and so on. There are several drawbacks to directly using the low-level interface:-

- Calculating offsets and manipulating file pointers is tedious to the user.

- Since the I/O latency is very high, the larger the number of requests required to access data, lower is the performance.

- The file system cannot perform optimizations based on the access requests of all processors, since in general, there is no support for processors to make collective requests.

We believe that high-level interfaces that facilitate the use of semantic knowledge about the accesses from parallel application programs are necessary for simple, portable and efficient programming. For example, in the case of Figure 2, the user should be able to specify in a simple way and in a single call, that the section $(l_1 : u_1, l_2 : u_2)$ of the array needs to be read. A library of optimized routines can be developed to read the necessary data using the low-level interface provided by the file system. PASSION provides such a high-level interface for the convenience of the user, and a library of routines which support this interface efficiently.

Recently, some file systems have been developed, such as the Vesta file system [4] and the nCUBE file system [5], which provide some limited support for the user to specify a logical view of the data to be accessed. There have also been some proposals for file system interfaces which allow the user to specify strided requests in a single read/write call [3, 8]. Specialized interfaces are also provided by other runtime libraries such as [7, 9]. The PASSION Runtime Library provides a very general high-level interface. For example, the user can access arbitrary array sections with strides in each dimension. The array elements can be of any type, even user-defined records. The array can be stored in the file in any storage order and the file can have a header containing some additional information. PASSION also supports a collective interface, so that optimizations can be performed based on the knowledge of the access requests of all processors. Sections 4 and 5 describe the PASSION interface in detail.

4 PASSION Data Structures

The PASSION library provides support for reading/writing entire arrays as well as sections of arrays stored in files. It uses the following data structures for this purpose.

4.1 Out-of-Core Array Descriptor (OCAD)

Each out-of-core array has a descriptor associated with it called the *Out-of-Core Array Descriptor* (OCAD). The OCAD contains the following information about the array

- Number of dimensions

- Size of the global array

- Size of each element of the array in bytes : Each element of the array could potentially be a *structure* or *record*. This enables the PASSION library to support arrays of any data type.

- Number of processors in each dimension

- Distribution of the array in each dimension

- Size of the In-Core Local Array (ICLA)

- Size of the overlap area

- Size of the Out-of-Core Local Array (OCLA)

4.2 Parallel File Pointer (PFILE)

The parallel file pointer is the parallel equivalent of the file pointer associated with a sequential file. It is allocated by the `PASSION_open` routine. It needs to be passed as a parameter to all PASSION routines that access files. The parallel file pointer contains the following information about the parallel file :

- System file descriptor

- Header size

4.3 Prefetch Descriptor

The prefetch descriptor is used to store information about prefetch read operations in progress. It is allocated by the routine `PASSION_prefetch_read`. It is used by the `PASSION_prefetch_wait` routine which waits for a previously initiated prefetch operation to complete.

4.4 Reuse Descriptor

This data structure is used to implement the data reuse operation. It is allocated by the `PASSION_reuse_init` routine, which initiates a reuse operation. It is updated on the subsequent calls to the `PASSION_read_reuse` routine which actually does the reuse.

4.5 Access Descriptor

This data structure is used to specify which section of the array needs to be read or written. It is a two dimensional array; row i specifies the lower bound, upper bound and stride in dimension i of the section to be accessed.

5 PASSION Interface

We describe the interface used by several of the PASSION routines. Further details can be found in the PASSION User's Guide [2].

5.1 Setting up the OCAD

All PASSION routines which access arrays require a pointer to the OCAD. The OCAD can be created and initialized as follows :

- The OCAD has to first be allocated using the routine `PASSION_malloc_OCAD`.

  ```
  OCAD *PASSION_malloc_OCAD(int dimensions);
  ```

  The parameter to this routine is the number of dimensions of the out-of-core array.

- After the OCAD has been allocated, it can be initialized using the routine `PASSION_fill_OCAD`.

  ```
  int PASSION_fill_OCAD(OCAD* OCADptr,
  int *size, int distribution[][2],
  int *nprocs, int *ocla_size,
  int icla_size[][2], int overlap[][2],
  int elemsize, int storage);
  ```

  The parameters to this routine are a pointer to the OCAD, size of the array, distribution of the array, number of processors, size of the OCLA, size of the ICLA, overlap information, size of each element of the array, and the storage order of the array in the file (`ROW_MAJOR` or `COLUMN_MAJOR`).

Once the OCAD is initialized, it can be used to access the out-of-core array. After all the accesses have been performed, the OCAD is no longer necessary and should be deallocated. This can be done using the routine `PASSION_free_OCAD`.

```
void PASSION_free_OCAD(OCAD *OCADptr);
```

5.2 Opening and Closing Files

Files should only be opened and closed with the routines `PASSION_open` and `PASSION_close`.

```
PFILE *PASSION_open(char *FileName,
unsigned int HeaderSize);
int PASSION_close(PFILE *PFilePtr);
```

The parameters to `PASSION_open` are the name of the file and size of the header at the start of the file. It returns a parallel file pointer. Note that in the Local Placement Model, each processor opens its own separate local array file, whereas in the Global Placement Model, all processors open a common file.

5.3 Accessing the File Header

PASSION provides support for files containing some other information, in addition to the array, in the form of a header at the start of the file. The header can be read using the routine `PASSION_read_header`.

```
int PASSION_read_header(PFILE *PFilePtr,
char *HBuf);
```

The parameters are a parallel file pointer and a pointer to a buffer in memory to store the header. This routine can be called immediately after the file is opened, even before calling `PASSION_fill_OCAD`. This allows the application program to store information about the array in the file header and use that information to fill in the OCAD.

Information can be written to the file header using the routine `PASSION_write_header`.

```
int PASSION_write_header(PFILE *PFilePtr,
char *HBuf);
```

5.4 Reading the Array

A number of routines are provided to read the array from the file. If each processor's local array can fit in its main memory, then the entire local array can be read using the routine `PASSION_read`.

```
int PASSION_read(PFILE *PFilePtr,
OCAD *OCADptr, char *Array);
```

The parameters are a parallel file pointer, pointer to the OCAD, and a pointer to a buffer in main memory to store the array. This routine is only for the Local Placement Model. In the Global Placement Model, even if the entire local array fits in memory, it has to be read by specifying its lower bound, upper bound and stride in the global array.

5.4.1 Reading Array Sections

If the array cannot fit in memory, sections of the array need to be read at a time. PASSION provides routines to read sections of the array with strides in each dimension. Separate routines are provided for reading array sections in the Local and Global Placement Models.

1. **Local Placement Model**: The routine `PASSION_read_section` is used to read array sections in the Local Placement Model.

   ```
   int PASSION_read_section(PFILE *PFilePtr,
   OCAD *OCADptr, char *Array, int *Index,
   int AccessArray[][3]);
   ```

   The parameters are a parallel file pointer, pointer to the OCAD, buffer in memory to store the section, coordinates of the location in the buffer from where the section is to be stored, and the section to be read specified by an access descriptor (see Section 4.5). Data sieving is used to read strided sections [11, 10]. This routine reads the array section from the local array file to the specified location in memory. The shape of the section is retained. To save memory, the section is stored without stride in memory, even if there was a stride in the OCLA.

2. **Global Placement Model**: The routine `PASSION_global_read` can be used to read array sections in the Global Placement Model. Each processor can access any arbitrary section of the array. The sections requested by different processors could be distinct, overlapping or even identical.

   ```
   int PASSION_global_read(PFILE *PFilePtr,
   OCAD *OCADptr, char *Array, int *Index,
   int AccessArray[][3], int nprocs);
   ```

   The parameters are the same as for `PASSION_read_section` with the addition of the number of processors since this is a collective read operation. This routine uses the Extended Two-Phase Method described in [12, 10].

5.4.2 Data Prefetching

The PASSION library provides routines for prefetching data before it is needed. Prefetching is basically a non-blocking read operation. This can be used to overlap computation with I/O and thus reduce the time spent in waiting for I/O.

```
PREFETCH *PASSION_read_prefetch(PFILE *PFilePtr,
OCAD *OCADptr, char *Array, int *Index,
int AccessArray[][3]);
```

This routine is used to start a prefetch operation. The parameters are the same as for `PASSION_read_section`. It returns a pointer to a prefetch descriptor (see Section 4.3).

The routine `PASSION_prefetch_wait` can be used to wait for a previously initiated prefetch operation to complete.

```
int PASSION_prefetch_wait(PREFETCH *PREFETCHptr);
```

5.4.3 Data Reuse

Data reuse can be performed using the routines `PASSION_reuse_init` and `PASSION_read_reuse`.

```
REUSE *PASSION_read_reuse(PFILE *PFilePtr,
OCAD *OCADptr, int start);
```

`PASSION_reuse_init` initializes the reuse descriptor (see Section 4.4). The parameters are a parallel file pointer, pointer to the OCAD and the position in the OCLA from where the read operation is to start. It returns a pointer to the reuse descriptor.

`PASSION_read_reuse` is used to read data with reuse.

```
int PASSION_read_reuse(REUSE *REUSEptr,
char *Array);
```

The parameters are a pointer to the reuse descriptor and a pointer to a buffer in memory to store data. The return value indicates when end of file is reached. Figure 3 illustrates how reuse works.

5.5 Writing the Array

A number of routines are provided to write arrays to files. If each processor's local array can fit in its main memory, then the entire local array can be written using the routine `PASSION_write`.

```
int PASSION_write(PFILE *PFilePtr,
OCAD *OCADptr, char *Array);
```

The parameters are a parallel file pointer, pointer to the OCAD, and a pointer to a buffer in main memory containing the array. This routine is only for the Local Placement Model. In the Global Placement Model, even if the entire local array fits in memory, it has to be written by specifying its lower bound, upper bound and stride in the global array.

5.5.1 Writing Array Sections

If the array cannot fit in memory, sections of the array need to be written at a time. PASSION provides routines to write sections of the array with strides in each dimension. Separate routines are provided for writing array sections in the Local and Global Placement Models.

1. **Local Placement Model**: The routine `PASSION_write_section` is used to write array sections in the Local Placement Model.

   ```
   int PASSION_write_section(PFILE *PFilePtr,
   OCAD *OCADptr, char *Array, int *Index,
   int AccessArray[][3]);
   ```

   The parameters are a parallel file pointer, pointer to the OCAD, buffer in memory containing the section, coordinates of the starting location of the section in the buffer, and the section to be written specified by an access descriptor (see Section 4.5). Data sieving is used to write strided sections [11, 10]. This routine writes the array section from the specified location in the buffer

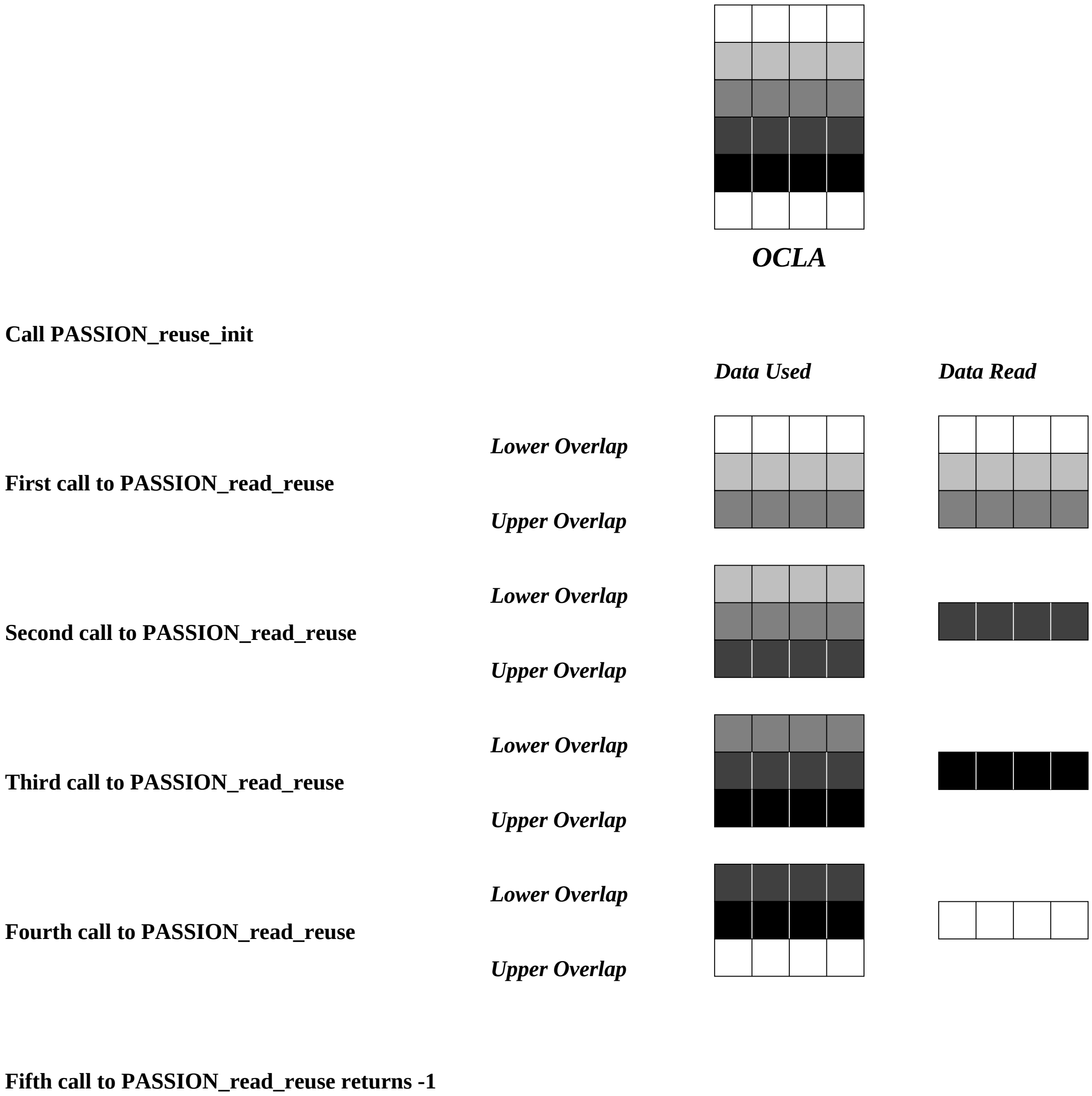

Figure 3: Data Reuse

to the local array file. The shape of the section is retained. The section is assumed to be stored with unit stride in memory, but is written to the file with the specified stride.

2. **Global Placement Model**: The routine `PASSION_global_write` can be used to write array sections in the Global Placement Model. If the sections requested to be written by different processors have some elements in common, there is a potential data consistency problem. `PASSION_global_write` has been implemented such that if there are write requests from multiple processors to the same location, the data from the highest numbered processor is written to the file.

```
int PASSION_global_write(PFILE *PFilePtr,
OCAD *OCADptr, char *Array, int *Index,
int AccessArray[][3], int nprocs);
```

The parameters are the same as for `PASSION_write_section` with the addition of the number of processors since this is a collective write operation. The Extended Two-Phase Method is used for writing sections [12, 10].

6 Conclusions

Portable high-level interfaces, such as the PASSION interface, make it easier for the user to specify the I/O required in parallel applications. There is no standard high-level I/O interface at present, but we believe that the ideas used in PASSION and the experience gained in its development would help in the definition of such a standard.

The development of the PASSION library is an ongoing process. Version 1.0 has been available since February 1995 and Version 1.1 will be released soon. We are also in the process of using the PASSION library for I/O in several real parallel applications and studying the performance benefits. Further information about PASSION, including the code, can be obtained from the URL `http://www.cat.syr.edu/passion.html`.

References

[1] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: Parallel and Scalable Software for Input-Output. Technical Report SCCS–636, NPAC, Syracuse University, September 1994. Also available as CRPC Technical Report CRPC–TR94483–S.

[2] A. Choudhary, R. Bordawekar, S. More, K. Sivaram, and R. Thakur. A User's Guide for the PASSION Runtime Library Version 1.0. Technical Report SCCS–702, NPAC, Syracuse University, February 1995.

[3] P. Corbett, D. Feitelson, Y. Hsu, J. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong. MPI-IO: A Parallel I/O Interface for MPI, Version 0.3. Technical Report NAS-95-002, NASA Ames Research Center, January 1995.

[4] P. Corbett, D. Feitelson, J. Prost, and S. Baylor. Parallel Access to Files in the Vesta File System. In *Proceedings of Supercomputing '93*, pages 472–481, November 1993.

[5] E. DeBenedictis and J. del Rosario. nCUBE Parallel I/O Software. In *Proceedings of* 11^{th} *International Phoenix Conference on Computers and Communications*, pages 117–124, April 1992.

[6] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Runtime Access Strategy. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS '93*, pages 56–70, April 1993.

[7] N. Galbreath, W. Gropp, and D. Levine. Applications-Driven Parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471, November 1993.

[8] N. Nieuwejaar and D. Kotz. Low-level Interfaces for High-level Parallel I/O. In *Proceedings of the Third Annual Workshop on I/O in Parallel and Distributed Systems*, pages 47–62, April 1995.

[9] K. Seamons and M. Winslett. An Efficient Abstract Interface for Multidimensional Array I/O. In *Proceedings of Supercomputing '94*, pages 650–659, November 1994.

[10] R. Thakur. *Runtime Support for In-Core and Out-of-Core Data-Parallel Programs.* PhD thesis, Dept. of Electrical and Computer Engineering, Syracuse University, May 1995.

[11] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. PASSION Runtime Library for Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.

[12] R. Thakur and A. Choudhary. Collective I/O Using an Extended Two-Phase Method with Dynamic Partitioning. Technical Report SCCS–704, NPAC, Syracuse University, March 1995.