PASSION Runtime Library for Parallel I/O

Rajeev Thakur
Syracuse University, Northeast Parallel Architectures Center, and Department of Electrical and Computer Engineering, thakur@npac.syr.edu

Rajesh Bordawekar
Syracuse University

Alok Choudhary
Syracuse University, Northeast Parallel Architectures Center, and Department of Electrical and Computer Engineering

Ravi Ponnusamy
Syracuse University, Northeast Parallel Architectures Center, and Department of Electrical and Computer Engineering, ravi@npac.syr.edu

Scalable Parallel Libraries Conference, Oct. 1994

PASSION Runtime Library for Parallel I/O [*]

Rajeev Thakur *Rajesh Bordawekar* *Alok Choudhary*
Ravi Ponnusamy *Tarvinder Singh*
Dept. of Electrical and Computer Eng. and
Northeast Parallel Architectures Center
Syracuse University, Syracuse, NY 13244
thakur, rajesh, choudhar, ravi, tpsingh @npac.syr.edu

Abstract

We are developing a compiler and runtime support system called **PASSION***:* **P***arallel* **A***nd* **S***calable* **S***oftware for* **I***nput-***O***utput. PASSION provides software support for I/O intensive out-of-core loosely synchronous problems. This paper gives an overview of the PASSION Runtime Library and describes two of the optimizations incorporated in it, namely Data Prefetching and Data Sieving. Performance improvements provided by these optimizations on the Intel Touchstone Delta are discussed, together with an out-of-core Median Filtering application.*

1 Introduction

There are a number of applications which deal with very large quantities of data. These applications exist in diverse areas such as large scale scientific computations, database applications, hypertext and multimedia systems, information retrieval and many other applications of the Information Age. The number of such applications and their data requirements keep increasing day by day. Consequently, it has become apparent that I/O performance rather than CPU or communication performance may be the limiting factor in future computing systems. Recent advances in high performance computing have resulted in computers which can provide more than 100 Gflops of computing power. However, the performance of the I/O systems of these machines has lagged far behind. It is still several orders of magnitude more expensive to read data from disk than to read it from local memory. Improvements are needed both in hardware as well as software to reduce the imbalance between CPU performance and I/O performance.

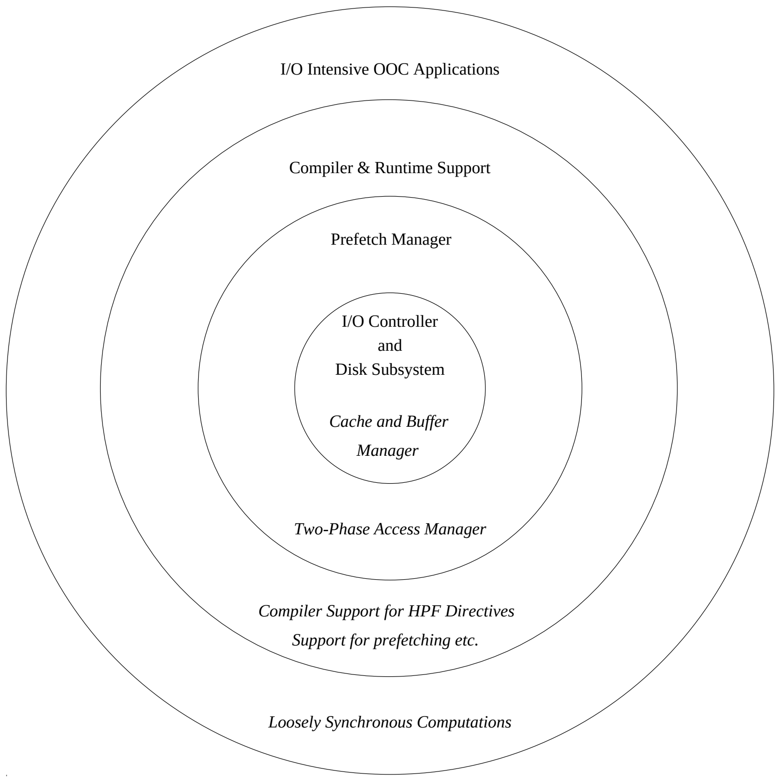
At Syracuse University, we consider the I/O problem from a language, compiler and runtime support point of view. We are developing a compiler and runtime support system called **PASSION**: **P**arallel **A**nd **S**calable **S**oftware for **I**nput-**O**utput [4]. PASSION provides support for compiling out-of-core data parallel programs [16, 1], parallel input-output of data [2], communication of out-of-core data, redistribution of data stored on disks, many optimizations including data prefetching from disks, data sieving, data reuse etc., as well as support at the operating system level. We have also developed an initial framework for runtime support for out-of-core irregular problems [4].

This paper gives an overview of PASSION and describes some of the main features of the PASSION Runtime Library. We explain the basic model of computation and I/O used by the runtime library. The runtime routines supported by PASSION are discussed. A number of optimizations have been incorporated in the runtime library to reduce the I/O cost. We describe in detail two of these optimizations, namely Data Prefetching and Data Sieving. Performance improvements provided by these optimizations on the Intel Touchstone Delta are discussed, together with an out-of-core Median Filtering application.

2 PASSION Overview

PASSION provides software support for I/O intensive loosely synchronous problems. It has a layered approach and provides support at the compiler, runtime and operating systems level as shown in Figure 1. The PASSION compiler translates out-of-core HPF programs to message passing node programs with explicit parallel I/O. It extracts information from user directives about the data distribution, which is required by the PASSION runtime system. It restructures loops having out-of-core arrays and also decides the transformations on out-of-core data to map the distribution on disks with the usage in the loops. The PASSION compiler uses well known techniques such as loop stripmining, iteration blocking etc. to generate

[*]This work was supported in part by NSF Young Investigator Award CCR-9357840, grants from Intel SSD and IBM Corp., and in part by USRA CESDIS Contract # 5555-26. This work was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by CRPC.



I/O Intensive OOC Applications
Compiler & Runtime Support
Prefetch Manager
I/O Controller
and
Disk Subsystem
Cache and Buffer
Manager
Two-Phase Access Manager
Compiler Support for HPF Directives
Support for prefetching etc.
Loosely Synchronous Computations

Figure 1: PASSION Rings

efficient code for I/O intensive applications. It also embeds calls to appropriate PASSION runtime routines which carry out I/O efficiently. The Compiler and Runtime Layers pass data distribution and access pattern information to the Two-Phase Access Manager and the Prefetch Manager. They optimize I/O using buffering, redistribution and prefetching strategies. At the operating system level, PASSION provides support to handle prefetching and buffering.

The PASSION runtime support system makes I/O optimizations transparent to users. The runtime procedures can either be used together with a compiler to translate out-of-core data parallel programs, or used directly by application programmers. The runtime library performs the following functions:-

- hides disk data distribution from the user.
- provides consistent I/O performance independent of data distribution.
- reorders I/O requests to minimize seek time.
- eliminates duplicate I/O requests to reduce I/O cost.
- prefetches disk data to hide I/O latency.

Writing message passing parallel programs with efficient parallel I/O is a tedious process. Instead, a program written in a high-level data parallel language like HPF can be translated into efficient code using the PASSION compiler and runtime system. A detailed description of all the features of PASSION is given in [4].

2.1 Model for Computation and I/O

In the SPMD (Single Program Multiple Data) programming model, each processor has a local array associated with it. In an in-core program, the local array resides in the local memory of the processor. For large data sets, however, local arrays cannot entirely fit in main memory. In such cases, parts of the local array have to be stored on disk. We refer to such a local array as an **Out-of-core Local Array (OCLA)**. Parts of the OCLA need to be swapped between main memory and disk during the course of the computation.

The basic model for computation and I/O used by PASSION is shown in Figure 2. 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. Since the local arrays are out-of-core, they have to be stored in files on disk. 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 into the file when required. If the I/O architecture of the system is such that each processor has its own disk,

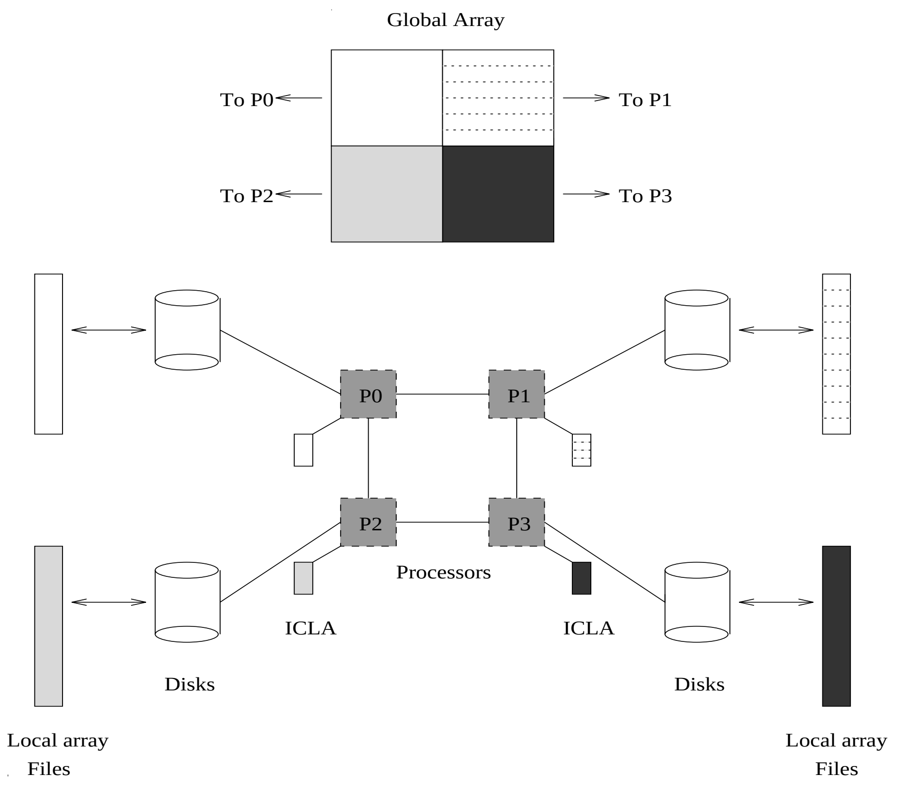


Figure 2: Model for Computation and I/O

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 is system dependent. 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.

3 Runtime Support in PASSION

During program execution, data needs to be moved back and forth between the LAF and the ICLA. Also, since the global array is distributed, a processor may need data from the local array of another processor. This requires data to be communicated between processors. Thus, runtime support is needed to perform I/O as well as communication. The PASSION Runtime Library consists of a set of high level specialized routines for parallel I/O and collective communication. These routines are built using the native communication and I/O primitives of the system and provide a high level abstraction which avoids the inconvenience of working directly with the lower layers. For example, the routines hide details such as buffering, mapping of files on disks, location of data in files, synchronization, optimum message size for communication, best communication algorithms, communication scheduling, I/O scheduling etc.

3.1 PASSION Runtime Library

The PASSION routines can be divided into four main categories based on their functionality — Array Management/Access Routines, Communication Routines, Mapping Routines and Generic Routines. Some of the basic routines and their functions are listed in Table 1.

3.1.1 Array Management/Access Routines

These routines handle the movement of data between the LAF and the ICLA. Any arbitrary regular section of the OCLA can be read for an array stored in either row-major or column-major order. The information about the array such as its shape, size, distribution, storage format etc. is passed to the routines using a

	Array Management Routines	
	PASSION Routine	Function
1	**PASSION_read_section**	Read a regular section from LAF to ICLA
2	**PASSION_write_section**	Write a regular section from ICLA to LAF
3	**PASSION_read_with_reuse**	read_section with data reuse [16]
4	**PASSION_prefetch_read**	Asynchronous (non-blocking) read of a regular section
5	**PASSION_prefetch_wait**	Wait for a prefetch to complete
	Array Communication Routines	
	PASSION Routine	Function
6	**PASSION_oc_shift**	Shift type collective communication on out-of-core data
7	**PASSION_oc_multicast**	Multicast communication on out-of-core data
	Mapping Routines	
	PASSION Routine	Function
8	**PASSION_oc_disk_map**	Map disks to processors
9	**PASSION_oc_file_map**	Generate local files from global files
	Generic Routines	
	PASSION Routine	Function
10	**PASSION_oc_transpose**	Transpose an out-of-core array
11	**PASSION_oc_matmul**	Perform out-of-core matrix multiplication

Table 1: Some of the PASSION Runtime Routines

data structure called the Out-of-Core Array Descriptor (OCAD) [16]. The Data Sieving Method described in Section 5 is used for improved performance.

3.1.2 Communication Routines

The Communication Routines perform collective communication of data in the OCLA. We use the Explicit Communication Method described in [16]. The communication is done for the entire OCLA, i.e. all the off-processor data needed by the OCLA is fetched during the communication. This requires inter-processor communication as well as disk accesses.

3.1.3 Mapping Routines

The Mapping Routines perform data and processor/disk mappings. Data mapping routines include routines to generate local array files from a global file. Disk mapping routines map physical disks onto logical disks.

3.1.4 Generic Routines

The Generic Routines perform computations on out-of-core arrays. Examples of these routines are out-of-core transpose and out-of-core matrix multiplication.

3.2 Two-Phase Approach

The performance of parallel file systems depends to a large extent on the way data is distributed on disks and processors. The performance is best when the data distribution on disks conforms to the data distribution on processors. Other distributions give much lower performance. To alleviate this problem, the Two Phase Access Strategy has been proposed in [8, 2]. In the Two Phase Approach, data is first read in a manner conforming to the distribution on disks and then redistributed among the processors. This is found to give consistently good performance for all distributions [8, 2]. The PASSION runtime library uses this Two Phase Approach for parallel I/O. In the first phase, data is accessed using the data distribution, stripe size, and set of reading nodes (possibly a subset of the computational array) which conforms with the distribution of data over the disks. In the second phase, the data is redistributed at run-time to match the application's desired data distribution.

The Two-Phase Approach provides the following advantages over the conventional Direct Access Method:-

1. The distribution of data on disks is effectively hidden from the user.

2. It uses the higher bandwidth of the interconnection network.

3. It uses collective communication and collective I/O operations.

4. It provides software caching of the out-of-core data in main memory to exploit temporal and spatial locality.

5. It aggregates I/O requests of compute nodes so that only one copy of each data item is transferred between disk and main memory.

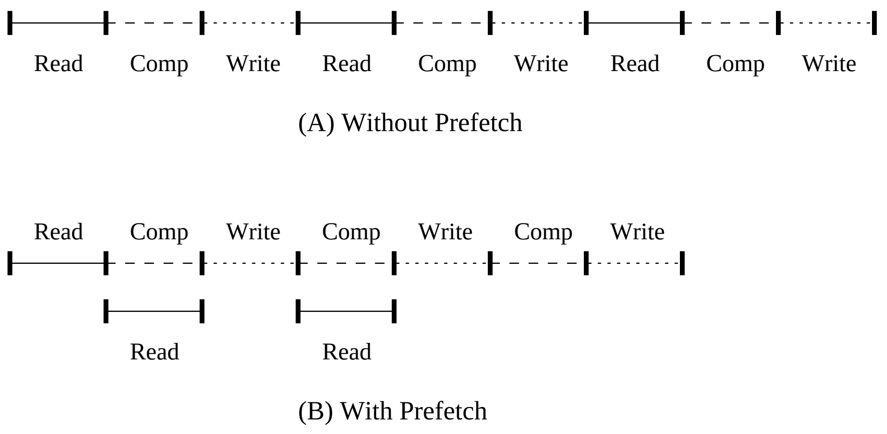


Figure 3: Data Prefetching

3.3 Optimizations

A number of optimizations have been incorporated in the PASSION runtime library to reduce the I/O cost. One optimization called *Data Reuse* [16] reduces the amount of I/O by reusing data already fetched into main memory instead of reading it again from disk. Two other optimizations, *Data Prefetching* and *Data Sieving*, are described in Sections 4 and 5 respectively. In addition, some other optimizations such as *Software Caching* to reduce the number of I/O requests and *Access Reordering* to reduce I/O latency time, have been incorporated.

4 Data Prefetching

In the model of computation and I/O described earlier, the OCLA is divided into a number of *slabs*, each of which can fit in the ICLA. Program execution proceeds as follows:- a slab of data is fetched from the LAF to the ICLA; the computation is performed on this slab and the slab is written back to the LAF. This is repeated on other slabs till the end of the program. Thus I/O and computation form distinct phases in the program. A processor has to wait while each slab is being read or written as there is no overlap between computation and I/O. This is illustrated in Figure 3(A) which shows the time taken for computation and I/O on 3 slabs. For simplicity, reading, writing and computation are shown to take the same amount of time, which may not be true in certain cases.

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 slab immediately after the current slab has been read. This is called *Data Prefetching*. Since the read request is asynchronous, the reading of the next slab can be overlapped with the computation being performed on the current slab. If the computation time is comparable to the I/O time, this can result in significant performance improvement. Figure 3(B) shows how prefetching can reduce the time taken for the example in Figure 3(A). Since the computation time is assumed to be the same as the read time, all reads other than the first one get overlapped with computation. The total reduction in program time is equal to the time for reading two slabs, as only two of the three reads can be overlapped in this example.

Prefetching can be done using the routine `PASSION_prefetch_read()` and the routine `PASSION_prefetch_wait()` can be used to wait for the prefetch to complete.

4.1 Performance

We use an out-of-core Median Filtering program to illustrate the performance of Data Prefetching. Median Filtering is frequently used in computer vision and image processing applications to smooth the input image. Each pixel is assigned the median of the values of its neighbors within a window of a particular size, say 3×3 or 5×5 or larger. We have implemented a parallel out-of-core Median Filtering program using PASSION runtime routines for I/O and communication. The image is distributed among processors in one dimension along columns and stored in local array files. Depending on the window size, each processor needs a few columns from its right and left neighbors. This requires a shift type communication which is implemented using the routine `PASSION_oc_shift()`.

Tables 2 and 3 show the performance of Median Filtering on the Intel Touchstone Delta for windows of size 3×3 and 5×5 respectively. The image is of size 2K $\times$ 2K pixels. We assume this to be out-of-core for the purpose of experimentation. The number of processors is varied from 4 to 64 and the size of the ICLA is varied in each case in such a way that the number of slabs varies from 4 to 16. Since the Touchstone Delta has 64 disks, each processor's LAF can be stored on a separate disk.

The following observations can be made from these

Table 2: Performance of Median Filtering using 3×3 window (time in sec.)

	4 slabs		8 slabs		16 slabs	
Procs	Prefetch	No Prefetch	Prefetch	No Prefetch	Prefetch	No Prefetch
4	36.37	46.56	33.63	46.75	30.65	47.21
8	18.32	23.37	16.72	24.41	16.36	24.86
16	9.180	12.33	8.730	12.60	8.580	13.35
32	5.340	6.830	5.260	7.000	5.080	7.160
64	5.650	5.850	4.970	5.970	5.410	6.230

Table 3: Performance of Median Filtering using 5×5 window (time in sec.)

	4 slabs		8 slabs		16 slabs	
Procs	Prefetch	No Prefetch	Prefetch	No Prefetch	Prefetch	No Prefetch
4	81.47	94.09	79.25	95.63	78.58	96.88
8	41.81	47.76	41.35	49.32	41.01	50.59
16	21.57	25.41	21.40	27.28	21.74	27.81
32	11.36	12.83	11.40	13.64	11.43	14.81
64	7.110	9.010	6.810	9.110	8.090	9.197

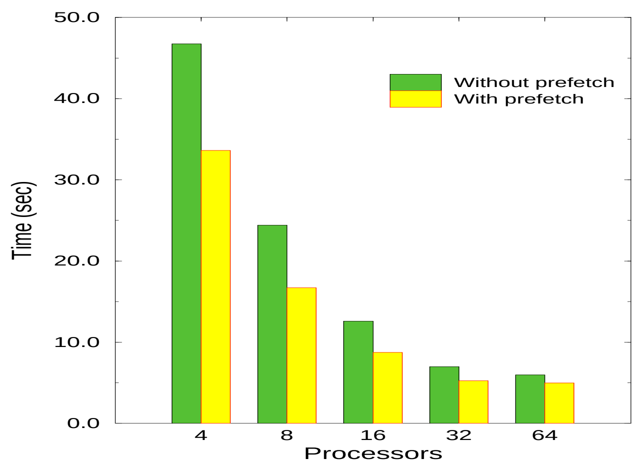


Figure 4: Median Filtering using 3×3 window

tables:-

1. In all cases, prefetching improves performance considerably. In some cases, the improvement is close to 50%. Figures 4 and 5 show the relative performance with and without prefetching when the number of slabs is 8.

2. Without prefetching, as the number of slabs is increased, the time taken increases. This is because more number of slabs means a smaller slab size which results in more number of I/O requests.

3. With prefetching, as the number of slabs in increased, the time taken decreases in most cases. Since the first slab can never be prefetched, all processors have to wait for the first slab to be read. As the slab size is reduced, the wait time for the first slab is also reduced and there is more overlap of computation and I/O. However, the number of I/O requests increases. When the slab size is large, a reduction in the slab size by half improves performance because the saving in the wait time for the first slab is higher than the increase in time due to the larger number of I/O requests. But when the slab size is small (64 processor case with 8 or 16 slabs), the higher number of I/O requests costs more than the decrease in wait time for the first slab. Hence the performance actually degrades in this case.

5 Data Sieving

All the PASSION runtime routines for reading or writing data from/to disks support the reading/writing of regular sections of arrays. We define a *regular section* of an array as any portion of an array which can be specified in terms of its lower bound, upper bound and stride in each dimension. The need for reading array sections from disks may arise due to a number of reasons, for example FORALL or array assignment statements involving sections of out-of-core arrays.

Consider the array of size (11,11) shown in Figure 6, which is stored on disk. Suppose it is required to read the section (2:10:2,3:9:2) of this array. The elements to be read are circled in the figure. Since these

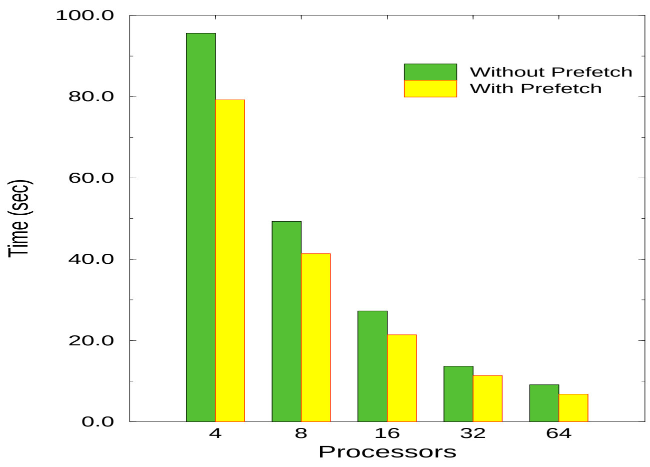


Figure 5: Median Filtering using 5×5 window

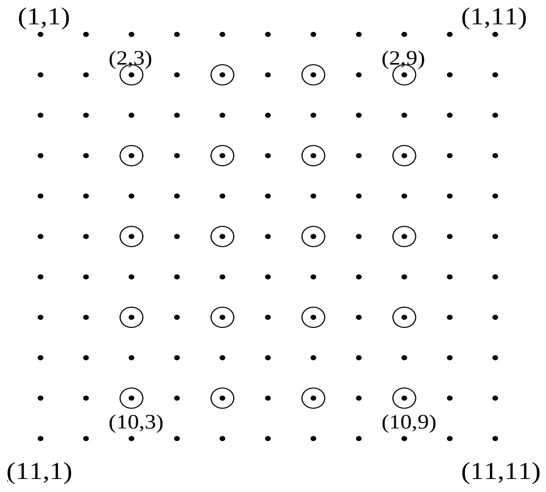


Figure 6: Accessing out-of-core array sections

elements are stored with a stride on disk, it is not possible to read them using one read call. A simple way of reading this array section is to explicitly move the file pointer to each element and read it individually, which requires as many reads as the number of elements. We call this the *Direct Read Method.* A major disadvantage of this method is the large number of I/O calls and low granularity of data transfer. Since the I/O latency is very high, this method proves to be very expensive. For example, on the Intel Touchstone Delta using 1 processor and 1 disk, it takes 16.06 ms. to read 1024 integers as one block, whereas it takes 1948 ms. to read all of them individually.

Suppose it required to read a section of a two-dimensional array specified by $(l_1 : u_1 : s_1, l_2 : u_2 : s_2)$. The number of array elements in this section is $(\lfloor (u_1 - l_1)/s_1 \rfloor + 1) \times (\lfloor (u_2 - l_2)/s_2 \rfloor + 1)$. Therefore, in the Direct Read Method,
No. of I/O requests $= (\lfloor (u_1-l_1)/s_1 \rfloor+1) \times (\lfloor (u_2-l_2)/s_2 \rfloor+1)$
No. of array elements read per access $= 1$
Thus in this method, the number of I/O requests is very high and the number of elements accessed per request is very low, which is undesirable.

We propose a much more efficient method called *Data Sieving* to read or write out-of-core array sections having strides in one or more dimensions. Data Sieving can be explained with the help of Figure 7. As explained earlier, each processor has an out-of-core local array (OCLA) associated with it. The OCLA is (logically) divided into slabs, each of which can fit in main memory (ICLA). The OCLA shown in the figure has four slabs. Let us assume that it is necessary to read the array section shown in Figure 7, specified by $(l_1 : u_1 : s_1, l_2 : u_2 : s_2)$, into the ICLA. Although this section spans three slabs of the OCLA, because of the stride all the data elements can fit in the ICLA.

In the Data Sieving Method, the entire block of data from column l_2 to u_2 if the storage is column major, or the entire block from row l_1 to u_1 if the storage is row major, is read into a temporary buffer in main memory using one read call. The required data is then extracted from this buffer and placed in the ICLA. Hence the name Data Sieving. A major advantage of this method is that it requires only one I/O call and the rest is data transfer within main memory. The main disadvantage is the high memory requirement. Another disadvantage is the extra amount of data that is read from disk. However, we have found that the savings in the number of I/O calls increases performance considerably. For this method, assuming column major storage,
No. of I/O requests $= 1$
No. of array elements read per access $=$
$$(u_2 - l_2 + 1) \times nrows$$

Data Sieving is a way of combining multiple I/O requests into one request so as to reduce the effect of high I/O latency time. A similar method called *message coalescing* is used in interprocessor communication, where small messages are combined into a single large message in order to reduce the effect of communication latency. However, Data Sieving is different because instead of coalescing the required data elements together, it actually reads even unwanted data elements so that large contiguous blocks are read. The useful data is then filtered out by the runtime system in an intermediate step and passed on to the program. The unwanted data read into main memory is dynamically discarded.

5.1 Reducing the Memory Requirement

If the stride in the array section is large, the amount of memory required to read the entire block from col-

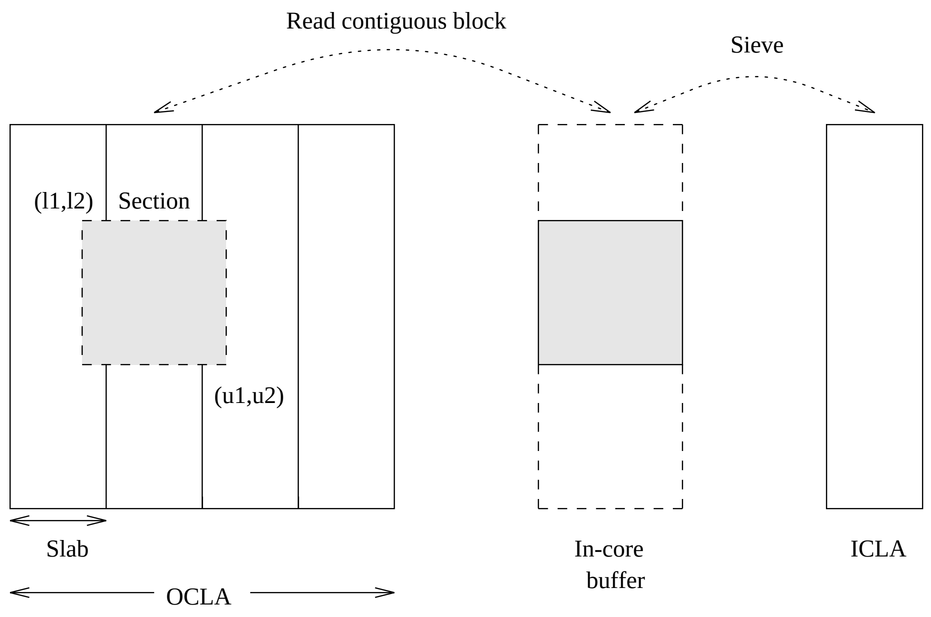


Figure 7: Data Sieving

umn l_2 to u_2 will be quite large. There may not be enough main memory available to store this entire block. Since the amount of memory available to create a temporary buffer is not known, we make the assumption that there is always enough memory to create a buffer of size equal to that of the ICLA. The Data Sieving Method described above is modified as follows to take this fact into account. Instead of reading the entire block of data from column l_2 to u_2, we read only as many columns (or rows) at a time as can fit in a buffer of the same size as the ICLA. For each set of columns read, the data is sieved and passed on to the program. This reduces the memory requirements of the program considerably and increases the number of I/O requests only slightly. Let us assume that the array is stored in column major order on disk and n columns of the OCLA can fit in the ICLA. Then for this case
No. of I/O requests $= \lceil (u_2 - l_2 + 1)/n \rceil$
No. of array elements read per access $= n \times nrows$

5.2 Writing Array Sections

Suppose it is required to *write* an array section $(l_1 : u_1 : s_1, l_2 : u_2 : s_2)$ from the ICLA to the LAF. The issues involved here are similar to those described above for reading array sections. A *Direct Write Method* can be used to write each element individually, but it suffers from the same problems of large number of I/O requests and low granularity of data transfer. In order to reduce the number of I/O requests, a method similar to the Data Sieving Method described above needs to be used. If we directly use Data Sieving in the reverse direction, ie. elements from the ICLA are placed at appropriate locations in a temporary buffer with stride, and the buffer is written to disk, the data in the buffer between the strided elements will overwrite the corresponding data elements on disk. In order to maintain data consistency, it is necessary to first read the entire block from the LAF into the temporary buffer. Then, data elements from the ICLA can be stored at appropriate locations in the buffer and the entire buffer can be written back to disk.

This is similar to what happens in cache memories when there is a write miss. In that case, a whole line or block of data is fetched from main memory into the cache and then the processor writes data into the cache. This is done in hardware in the case of caches. PASSION does this in software when writing array sections using Data Sieving. Thus, writing sections requires twice the amount of I/O compared to reading sections, because for each write to disk the corresponding block has to first be fetched into memory. Therefore, for writing array sections
No. of I/O requests $= 2\lceil (u_2 - l_2 + 1)/n \rceil$
No. of array elements transferred per access =
$n \times nrows$

5.3 Performance

Table 4 gives the performance of Data Sieving versus the Direct Method for reading and writing array sections. An array of size 2K $\times$ 2K is distributed among 64 processors in one dimension along columns. We measured the time taken by the `PASSION_read_section()` and `PASSION_write_section()` routines for reading and

Table 4: Performance of Direct Read/Write versus Data Sieving (time in sec.)

2K × 2K global array on 64 procs. (local array size 2K × 32), slab size = 16 columns

	`PASSION_read_section`		`PASSION_write_section`	
Array Section	Direct Read	Sieving	Direct Write	Sieving
(1:2048:2, 1:32:2)	52.95	1.970	49.96	5.114
(1:2048:4, 1:32:4)	14.03	1.925	13.71	5.033
(10:1024:3, 3:22:3)	8.070	1.352	7.551	4.825
(100:2048:6, 5:32:4)	7.881	1.606	7.293	4.756
(1024:2048:2, 1:32:3)	18.43	1.745	17.98	5.290

Table 5: I/O requirements of Direct Read and Data Sieving Methods

2K × 2K global array on 64 procs. (local array size 2K × 32), slab size = 16 columns

	No. of I/O requests		No. of array elements read	
Array Section	Direct Read	Sieving	Direct Read	Sieving
(1:2048:2, 1:32:2)	16384	2	16384	65536
(1:2048:4, 1:32:4)	4096	2	4096	65536
(10:1024:3, 3:22:3)	2373	2	2373	40960
(100:2048:6, 5:32:4)	2275	2	2275	57344
(1024:2048:2, 1:32:3)	5643	2	5643	65536

writing sections of the out-of-core local array on each processor. We observe that Data Sieving provides tremendous improvement over the Direct Method in all cases. The reason for this is large number of I/O requests in the Direct Method, even though the total amount of data accessed is higher in Data Sieving. Table 5 gives the number of I/O requests and the total amount of data transferred for each of the array sections considered in Table 4. We observe that in the Data Sieving Method, the number of data elements transferred is more or less the same for all cases. This is because the total amount of data transferred depends only on the lower and upper bounds of the section and is independent of the stride. Hence the time taken using Data Sieving does not vary much for all the sections we have considered. However, there is a wide variation in time for the Direct Method, because only those elements belonging to the section are read. The time is lower for small sections and higher for large sections.

We observe that even for writing array sections, Data Sieving performs better than Direct Write even though it requires reading the section before writing. As expected, `PASSION_write_section()` takes about twice the time as `PASSION_read_section()` when using Data Sieving. Comparing the Direct Write and Direct Read Methods, we find that writing takes slightly less time than reading data. This is due to the way I/O is done in the Intel Touchstone Delta. The *cwrite* call returns after data is written to the cache in the I/O node, without waiting for the data to be written to disk.

All PASSION routines involving array sections use Data Sieving for greater efficiency.

6 Related Work

There has been some related research in software support for high performance parallel I/O. The Two-phase I/O read/write strategy was first proposed by Bordawekar et al [8, 2]. The effects of prefetching blocks of a file in a multiprocessor file system are studied in [11]. Prefetching for in-core problems is discussed in [13, 3]. Vesta is a parallel file system designed and developed at IBM T. J. Watson Research Center [7, 5, 6] which supports logical partitioning of files. File declustering, where different blocks of a file are stored on distinct disks is suggested in [12]. This is used in the Bridge File System [10], in Intel's Concurrent File System (CFS) [15] and in various RAID schemes [14]. An overview of the various issues involved in high performance I/O is given in [9].

7 Conclusions

The PASSION Runtime Library provides high-level runtime support for loosely synchronous out-of-core computations on distributed memory parallel computers. The routines perform efficient parallel I/O as well as interprocessor communication. The PASSION runtime procedures can either be used together with a compiler to translate out-of-core data parallel programs, or used directly by application programmers.

A number of optimizations have been incorporated in the runtime library for greater efficiency. The two optimizations described in this paper, namely Data Prefetching and Data Sieving, provide considerable performance improvement. Data Prefetching overlaps computation with I/O, while Data Sieving improves the granularity of I/O accesses for reading or writing array sections.

The PASSION Runtime Library is currently available on the Intel Paragon, Touchstone Delta and iPSC/860 using Intel's Concurrent File System. Efforts are underway to port it to the IBM SP-1 and SP-2 using the Vesta Parallel File System. Additional information about PASSION is available on the World Wide Web at `http://www.cat.syr.edu/passion.html`. PASSION related papers can also be obtained from the anonymous ftp site `erc.cat.syr.edu`.

Acknowledgments

We thank Geoffrey Fox, Ken Kennedy, Chuck Koelbel, Paul Messina and Joel Saltz for many fruitful discussions and helpful comments.

References

[1] R. Bordawekar, A. Choudhary, and R. Thakur. Data Access Reorganizations in Compiling Out-of-core Data Parallel Programs on Distributed Memory Machines. Technical Report SCCS–622, NPAC, Syracuse University, September 1994.

[2] R. Bordawekar, J. del Rosario, and A. Choudhary. Design and Evaluation of Primitives for Parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, November 1993.

[3] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proceedings of ASPLOS 91*, pages 40–52, 1991.

[4] 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.

[5] P. Corbett, S. Baylor, and D. Feitelson. Overview of the Vesta Parallel File System. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS '93*, pages 1–16, April 1993.

[6] P. Corbett and D. Feitelson. Overview of the Vesta Parallel File System. In *Proceedings of the Scalable High Performance Computing Conference*, pages 63–70, May 1994.

[7] 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.

[8] J. del Rosario, R. Bordawekar, and A. Choudhary. A Two-Phase Strategy for Achieving High-Performance Parallel I/O. Technical Report SCCS-408, NPAC, Syracuse University, October 1992.

[9] J. del Rosario and A. Choudhary. High Performance I/O for Parallel Computers: Problems and Prospects. *IEEE Computer*, pages 59–68, March 1994.

[10] P. Dibble, M. Scott, and C. Ellis. Bridge: A High Performance File System for Parallel Processors. In *Proceedings of the 8^{th} International Conference on Distributed Computing Systems*, pages 154–161, June 1988.

[11] D. Kotz and C. Ellis. Prefetching in File Systems for MIMD Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, pages 218–230, April 1990.

[12] M. Livny, S. Khoshafian, and H. Boral. Multi-Disk Management Algorithms. In *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 69–77, May 1987.

[13] T. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of ASPLOS 92*, pages 62–73, October 1992.

[14] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, June 1988.

[15] P. Pierce. A Concurrent File System for a Highly Parallel Mass Storage Subsystem. In *Proceedings of 4^{th} Conference on Hypercubes, Concurrent Computers and Applications*, pages 155–160, March 1989.

[16] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and Runtime Support for Out-of-Core HPF Programs. In *Proceedings of the 8^{th} ACM International Conference on Supercomputing*, pages 382–391, July 1994.