NPAC Technical Report SCCS–685, January 1995
Also available as CRPC Technical Report CRPC–TR95508–S

Accessing Sections of Out-of-Core Arrays Using an Extended Two-Phase Method *

Rajeev Thakur *Alok Choudhary*

Dept. of Electrical and Computer Eng. and
Northeast Parallel Architectures Center
Syracuse University, Syracuse NY 13244
thakur, choudhar @npac.syr.edu

Abstract

In out-of-core computations, data needs to be moved back and forth between main memory and disks during program execution. In this paper, we propose a technique called the Extended Two-Phase Method, for accessing sections of out-of-core arrays efficiently. This is an extension and generalization of the Two-Phase Method for reading in-core arrays from files, which was previously proposed in [7, 3]. The Extended Two-Phase Method uses collective I/O in which all processors cooperate to perform I/O in an efficient manner by combining several I/O requests into fewer larger requests, eliminating multiple disk accesses for the same data and reducing contention for disks. We describe the algorithms for reading as well as writing array sections. Performance results on the Intel Touchstone Delta for many different access patterns are presented and analyzed. It is observed that the Extended Two-Phase Method gives consistently good performance over a wide range of access patterns.

*This work was supported in part by NSF Young Investigator Award CCR-9357840, grants from Intel SSD and IBM Corp., in part by USRA CESDIS Contract # 5555-26 and also in part by ARPA under contract # DABT63-91-C-0028. The content of the information does not necessarily reflect the position or the policy of the US Government and no official endorsement should be inferred. Rajeev Thakur is supported by a Syracuse University Graduate Fellowship. 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.

1 Introduction

Although the CPU and communication speeds of parallel computers have improved tremendously over the years, their I/O speeds have not shown similar improvement. It is still orders of magnitude more expensive to do I/O than to do computation or communication. In order to have a balanced system, it is essential that the I/O performance is comparable to the CPU and communication performance. I/O has become particularly important in recent times because parallel computers are increasingly being used for applications with very large data sets, such as scientific computations, database applications, multimedia systems, information retrieval, visualization etc. In order to get better I/O performance, improvements are needed in I/O hardware as well as in software support for parallel I/O.

We define an *in-core program* as one in which all the data required by the program can fit in main memory at the same time. Even in the case of in-core programs, I/O may be needed to read initial data from files and to write the results back to files at the end of the computation. Also, there may be intermediate writes either for checkpointing purposes, i.e. to save the current status of the computation so that it can be restarted later, or to monitor the progress of the solution using visualization or other techniques. An *out-of-core program* is defined as one in which all the data required by the program cannot fit in main memory at the same time, and hence needs to be stored in files on disks. During program execution, data needs to be moved back and forth between main memory and disk. Hence I/O forms a critical part in out-of-core programs.

In out-of-core computations, each processor needs to fetch a section of the out-of-core array into main memory, do the computation on that section, and store the results back to disk if necessary. Depending on how the file is stored on disks, this may require accessing data elements with stride. At present, the interface provided by most parallel file systems does not support accesses with stride. Hence the data needs to be fetched using more than one read call. Since the I/O latency is very high, this may prove to be very expensive. Also, there may be some common accesses among processors. If each processor tries to read its own section independently, the I/O performance may be very low because of lower granularity of accesses and multiple accesses for the same data. Therefore, it is necessary to use a more efficient technique for doing I/O in out-of-core computations.

In the case of in-core arrays, Bordawekar, del Rosario and Choudhary [7, 3] have proposed a Two-Phase Method for reading an entire array from a file in an efficient manner, into a distributed array in main memory. This method is found to give consistently good performance for all data distributions. In this paper, we propose an Extended Two-Phase Method for accessing sections of out-of-core arrays, which is a generalization of the Two-Phase Method for in-core arrays. This Extended Two-Phase Method is used in the PASSION Runtime Library for Out-of-Core Compu-

tations [14, 4] which we are developing.

The rest of this paper is organized as follows. Section 2 gives an overview of the PASSION Runtime Library and the different data access models it supports. It also discusses the I/O issues involved in these models and motivates the need for the Extended Two-Phase Method. An overview of the Two-Phase Method for in-core arrays is given in Section 3. The Extended Two-Phase Method for out-of-core arrays is introduced in Section 4. Section 5 describes in detail the algorithm for reading sections of out-of-core arrays using the Extended Two-Phase Method, together with performance results on the Intel Touchstone Delta. The algorithm for writing sections of out-of-core arrays is explained in Section 6. Section 7 describes how the Extended Two-Phase Method relates to other work in this area, followed by Conclusions in Section 8.

2 PASSION Runtime Library for Out-of-Core Computations

Our group at Syracuse University is developing a software system called **PASSION** (**P**arallel **A**nd **S**calable **S**oftware for **I**nput-**O**utput) [4], which provides software support for high performance parallel I/O. PASSION provides support at the language [2], compiler [13], runtime [14] as well as file system level [8, 12]. The PASSION Runtime Library provides routines to efficiently perform the I/O required in loosely synchronous [9] out-of-core programs which use a Single Program Multiple Data (SPMD) model. It provides the user with a simple high-level interface, which is a level higher than any of the existing parallel file system interfaces or even the proposed MPI-IO interface [5]. 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. A number of optimizations such as Data Sieving, Data Prefetching and Data Reuse have been incorporated in the library for improved performance [14, 13]. The library can either be used directly by application programmers, or a compiler could translate out-of-core programs written in a high-level data-parallel language like HPF to node programs with calls to the runtime library for I/O.

2.1 Data Storage and Access Models

The PASSION Runtime Library supports three basic models of storing and accessing out-of-core arrays, called the Local Placement Model (LPM), the Global Placement Model (GPM) and the Partitioned In-Core Model (PIM).

2.1.1 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 disk. The local array of each processor is stored in a separate file called the **Local Array File (LAF)** of that processor, as shown in Figure 1(I). 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. 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.

2.1.2 Global Placement Model (GPM)

In this model, the global array is stored in a single file called the **Global Array File (GAF)**, as shown in Figure 1(II), and no local array files are created. The global array is only logically divided into local arrays in keeping with the SPMD programming model. The PASSION runtime system fetches the appropriate portion of each processor's local array from the global array file, as requested by the processor. The advantage of the Global Placement Model is that it does not require the initial local array file creation phase of the Local Placement Model. The disadvantage is that each processor's data may not be stored contiguously in the global array file and some optimizations, such as those suggested in this paper, are needed in order to do I/O efficiently.

2.1.3 Partitioned In-Core Model (PIM)

The Partitioned In-Core Model, illustrated in Figure 1(III), is a variation of the Global Placement Model. The array is stored in a single global array file as in the Global Placement Model, but there is a difference in the way data is accessed. In the Partitioned In-Core Model, the global array is logically divided into a number of *partitions*, each of which can fit in the main memory of all processors combined. Thus the computation on each partition is essentially an in-core problem and does not require any I/O. Hence the name Partitioned In-Core Model. This model is useful when the data access pattern in the program has good locality. Otherwise, creating in-core partitions itself is difficult.

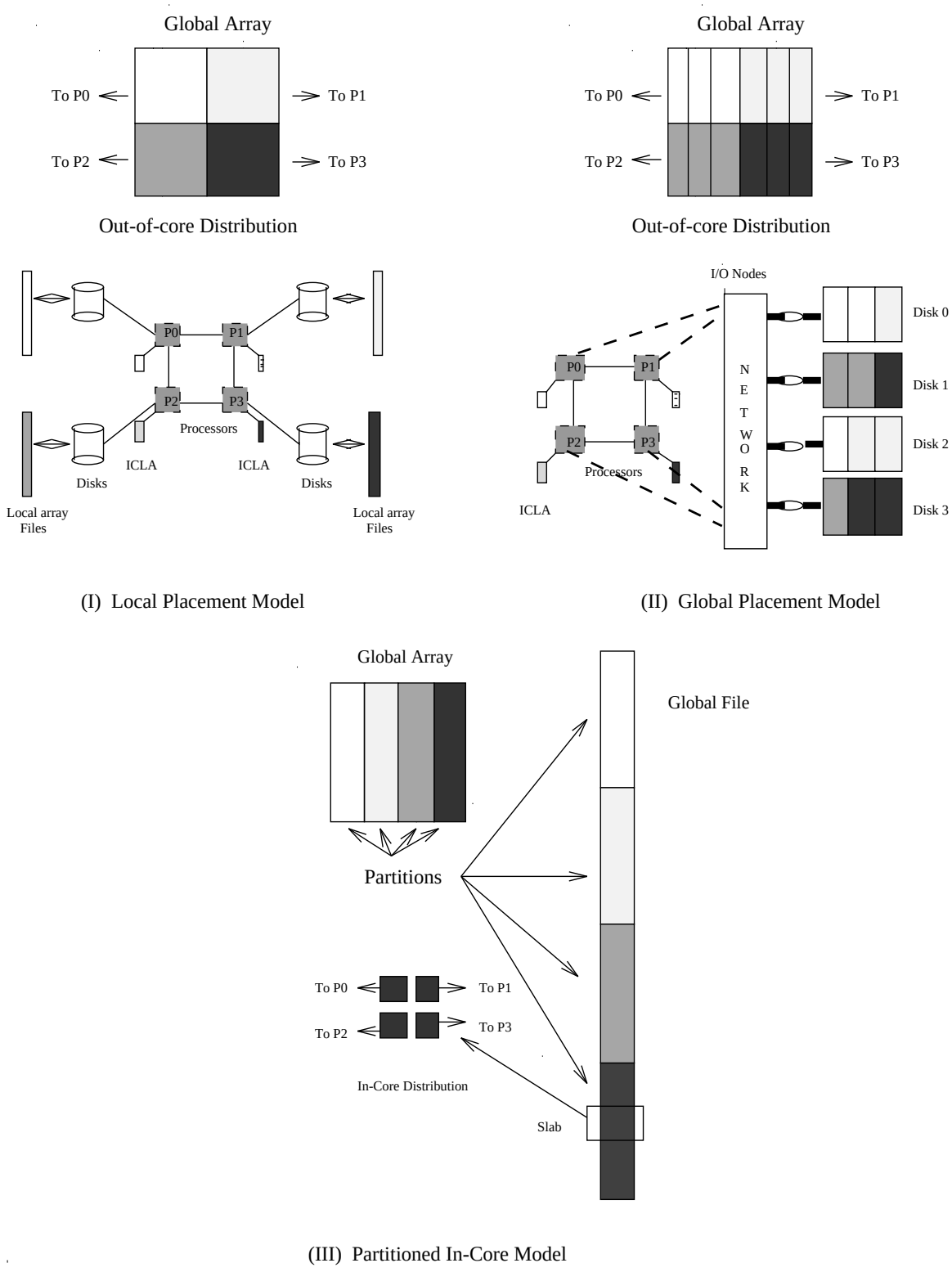

Figure 1: Data Storage and Access Models

The Extended Two-Phase Method described in this paper is used for accessing data in both the Global Placement Model and the Partitioned In-Core Model.

2.2 I/O in the Global Placement Model

In the Global Placement Model, the data required by any processor can be classified into the following types:-

1. A section of its local array which is stored in the global array file.
2. A section of the local array belonging to another processor (i.e. "off-processor data").
3. Locally computed values from other processors, which are stored in-core. For example, in order to calculate the sum of all elements in the out-of-core array, each processor computes a local sum followed by a global sum reduction. No I/O is required for this reduction operation.

In general, a processor may need to access any arbitrary portion of the global array, with or without stride. The global array may be stored in the global array file in either row-major or column-major order. As a result, the data required by each processor may not be stored contiguously in the file. Also, the requests of some processors may overlap. In the extreme case, all processors may want to access the same section of the array. Clearly, if each processor directly tries to read the data it needs, it may result in a large number of low granularity requests and multiple requests for the same data, causing contention for disks. Hence, a more improved method such as the Extended Two-Phase Method proposed in this paper needs to be used.

2.3 I/O in the Partitioned In-Core Model

In the Partitioned In-Core Model, each processor needs to access a portion of a *partition* of the global array. Depending on how the partitions are stored in the file and how a partition is distributed among processors, each processor may need to access non-contiguous data sets from the file. This results in the same kind of problems as in the Global Placement Model. Hence an improved method needs to be used for accessing data in the Partitioned In-Core Model as well.

3 Two-Phase Method for In-Core Arrays

Bordawekar, del Rosario and Choudhary [7, 3] have proposed a Two-Phase Method for reading/writing in-core arrays from/to disks. This method can be used to efficiently read an entire in-core array from a single file into a distributed array in main memory, and conversely to efficiently write an entire distributed in-core array, to a single file.

The basic principle behind this Two-Phase Method is as follows. In a distributed memory computer with a parallel file system, data is distributed among processors in some fashion and stored in files on disks in some fashion. It is a well known fact that I/O performance is better when processors make a small number of high granularity requests, instead of a large number of low granularity requests. When data is distributed among processors in such a way that it *conforms* to the way it is distributed on disks, each processor can directly read its portion of the in-core array in a single request. This is called the *conforming distribution*. If an array is stored in a file in column-major order, a column-block distribution among processors is the conforming distribution. In this case, each processor can directly read its local array from the file in a single operation. For any other distribution, a processor's local array will be stored in a non-contiguous manner in the file, and if each processor directly tries to read its local array (called the Direct Method), it will result in a large number of low-granularity requests. Hence, in the Direct Method, the I/O performance is best for the conforming distribution, but it degrades drastically for any other distribution.

The Two-Phase Method [7, 3] proposes to read the entire in-core array into a distributed array in main memory in two phases. In the first phase, the processors always read data assuming the conforming distribution. In the second phase, data is redistributed [16] among processors, using interprocessor communication, to whatever is the actual desired distribution. This two phase approach is found to give consistently good performance for all distributions [7, 3]. 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.

Figure 2 shows the performance improvement provided by the Two-Phase Method over the Direct Method. The timings shown are for reading an array of size 10K $\times$ 10K on the Intel Touchstone Delta using 64 processors, for different distributions. Since the column-block distribution is the conforming distribution, the Two-Phase and Direct Access Methods take the same time. For any other distribution, the Two-Phase Method performs considerably better. Among the distributions shown, the row-cyclic distribution results in the largest number of non-contiguous accesses. Hence the Direct Method takes the longest time for this case. However, the Two-Phase Method takes nearly the same time for all distributions. This is because the difference in time for the different distributions is only due to the difference in time for the redistribution phase, and the time required for redistribution is orders of magnitude lower than that required for I/O.

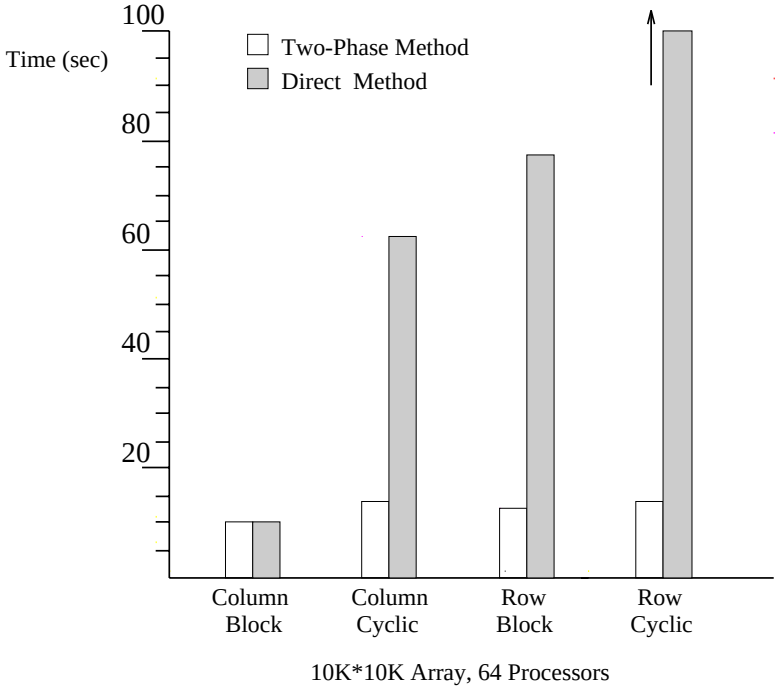

Figure 2: Performance of the Two Phase Method for In-Core Arrays

4 Extended Two-Phase Method for Out-of-Core Arrays

We have extended the basic Two-Phase Method for in-core arrays proposed by Bordawekar, del Rosario and Choudhary [7, 3], to access arbitrary sections of out-of-core arrays. This method performs I/O for out-of-core arrays efficiently by combining several I/O requests into fewer larger requests, eliminating multiple disk accesses for the same data and reducing contention for disks.

Consider the large out-of-core array shown in Figure 3 which is stored in a file in column-major order. Assume that there are four processors, each requesting a block of rows as shown. Because of the column-major ordering, each processor's request lies in a non-contiguous fashion in the file and the requests of different processors are interleaved. One way to perform the I/O in this case is for each processor to directly fetch the data it needs, oblivious of the requests of other processors (the Direct Method). The obvious disadvantage of this method is that there are too many small I/O requests which are not properly ordered. The Extended Two-Phase Method does not have the drawbacks of the Direct Method and hence provides better performance.

The Extended Two-Phase Method assumes a collective I/O interface, i.e. all processors must call the Extended Two-Phase read/write routine. Each processor may request a different amount of data. Even if a processor does not need any data, it must still call the routine with a request for 0 bytes, and participate in the two-phase process. This is a reasonable assumption given that

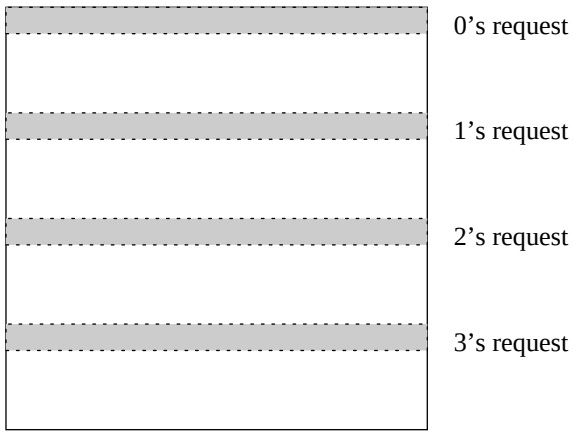

Figure 3: Accessing Global Array Sections

our intention is to support a *loosely synchronous* model of parallel computation [9]. An advantage of using collective I/O is that since all processors are participating, they can cooperate to perform certain optimizations. Another advantage is that if any processor needs to access some data which was previously modified by some other processor, it can be done using just a read call without any additional synchronization. The idea of collective I/O has also been used in other schemes such as in [1, 11, 10].

In the next section, we describe the Extended Two-Phase Method for reading sections of out-of-core arrays. The method for writing sections is analogous and is discussed in Section 6.

5 Reading Sections of Out-of-Core Arrays

Let us assume that each processor needs to read some regular section of the global array given by its lower-bound, upper-bound and stride in each dimension $(l_1 : u_1 : s_1, l_2 : u_2 : s_2)$ in global coordinates. For the purpose of explanation, let us assume that the array is stored in the file in column-major order.

The Extended Two-Phase Method for out-of-core arrays assigns *ownership* to portions of the file such that a processor can directly access only the portion of the file it *owns*. The file is effectively divided into *domains*. The portion of the file which a processor can directly access is called its **File Domain (FD)**. For arrays stored in column-major order, we assume that each processor owns a block of columns of the array, as if the array were distributed in a column-block fashion among the processors. Thus the File Domain of each processor is a block of columns of the array, which is stored contiguously in the file. Figure 4 shows the File Domain of each processor when there are four processors. This concept of File Domains is analogous to the concept of *conforming distribution* in the Two-Phase Method for in-core arrays. Given the size of the array and the number of processors,

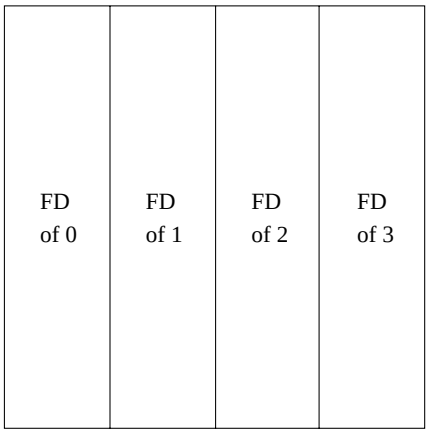

Figure 4: File Domains (FDs)

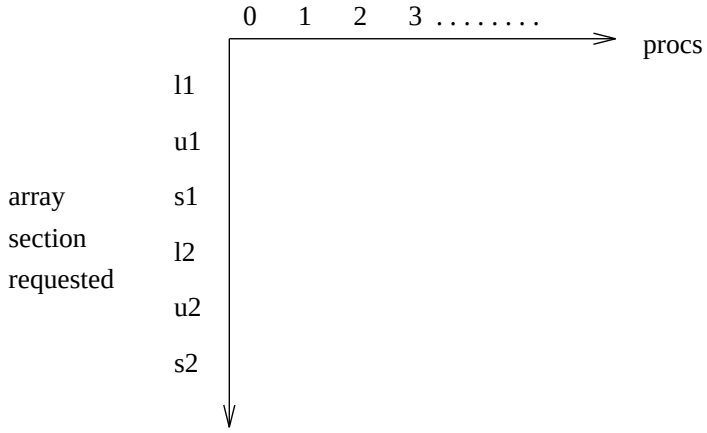

Figure 5: File Access Descriptor (FAD)

each processor can determine its own File Domain and also the File Domains of other processors.

In the first step of the Extended Two-Phase Method, all processors exchange their own access information (the indices $l_1, u_1, s_1, l_2, u_2, s_2$) with all other processors, which requires a complete exchange or all-to-all type communication [15, 17]. Thus each processor knows the access requests of all other processors. This information is stored in a data structure called the **File Access Descriptor (FAD)**, shown in Figure 5. The FAD contains exactly the same information on all processors.

Since each processor knows its own File Domain and the access requests of other processors, it can determine what portion of the data in its File Domain is needed by other processors. This is done by computing the intersection of the requests of other processors from the FAD and the indices of its own File Domain. This information is stored in a data structure called the **File Domain Access Table (FDAT)**, which is shown in Figure 6. Thus the FDAT of a processor contains

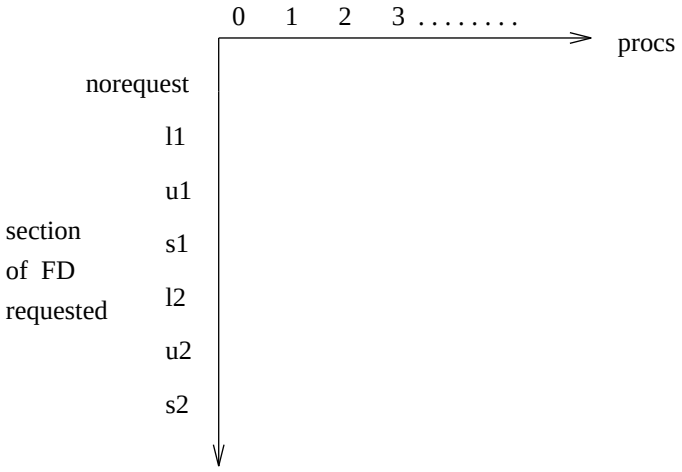

Figure 6: File Domain Access Table (FDAT)

information about which sections of its File Domain have been requested by other processors. If no data requested by a particular processor lies in this processor's File Domain, the corresponding "norequest" field in the FDAT of this processor is set to –1. Clearly, the FDAT on different processors contains different information.

Each processor now has to read data from its File Domain, as determined by the FDAT. A simple way of reading would be to read all the data needed by processor 0, followed by that needed by processor 1 and so on in order of processor number. But in many cases, this may result in too many small requests which are not in sequence. For example, consider Figure 7 which shows two different access patterns. In Figure 7(A), each processor needs to access a block of rows, and because of the column-major ordering, the requests of all processors are interleaved. In Figure 7(B), each processor needs to access a block of sub-columns, so the requests of processors 0 and 1 are interleaved and also the requests of processors 2 and 3.

In order that the read is done efficiently, it is important that the FDAT is analyzed so that the file is accessed in sequence and contiguously, as far as possible. We have devised a very general method of analyzing the information in the FDAT, which ensures that the file is read contiguously and in sequence. Each processor calculates the minimum of the lower-bounds and the maximum of the upper-bounds of all sections in its FDAT. This effectively determines the smallest section which contains all the data that needs to be read from the File Domain. It may also contain some data which is not required by any processor. If the processor tries to read only the useful data, it may result in a number of small strided accesses. In order to avoid this, it uses an optimization known as Data Sieving which is described in [14]. The processor reads an entire column of the section at a time in a single operation into a temporary buffer. This may include some unwanted data. The useful data is extracted from the temporary buffer and placed in communication buffers

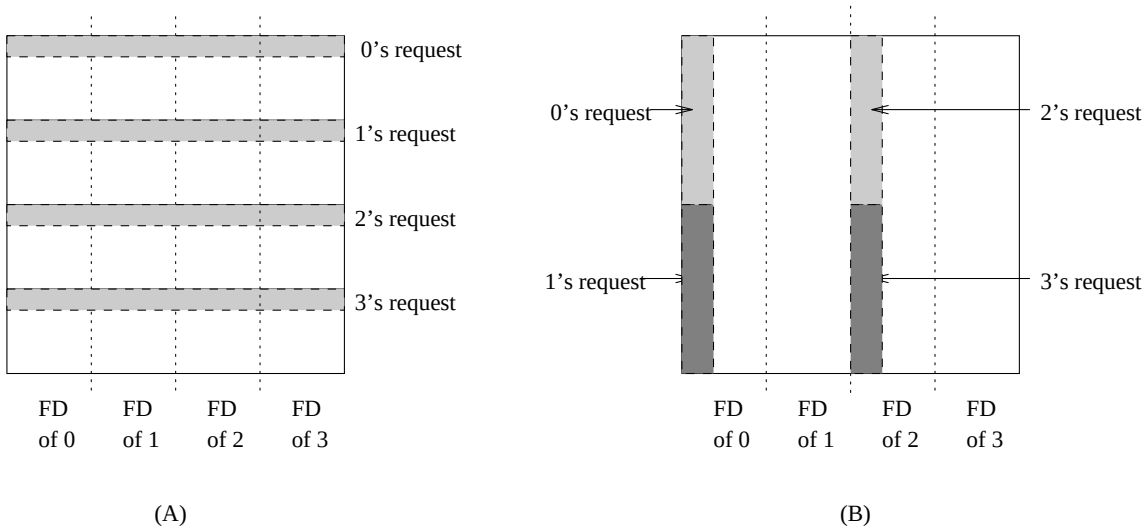

Figure 7: Examples of Accesses within File Domains

depending on which processors need the data. For example, for the requests in Figure 7(A), each column read contains data needed by all processors plus some unwanted data. For the requests in Figure 7(B), each column read contains data needed by two processors and no unwanted data. The entire section is read from the File Domain one column at a time using Data Sieving. This forms the first phase of the Extended Two-Phase Method.

The second phase of the Extended Two-Phase Method consists of communicating the data read in the first phase to the respective processors. The information in the FDAT is sufficient for each processor to know what data has to be sent to which processor. Since each processor knows the File Domains of all processors and its own access request, it can calculate how much data it needs to receive from other processors, as well as the locations where the received data needs to be placed.

An observation from Figure 7(B) is that for this particular access pattern, processors 1 and 3 do not do any I/O. This is because of the fact that File Domains have been defined as column blocks. None of the access requests of any processor lie in the File Domains of processors 1 and 3. One way to reduce this imbalance of I/O between the processors is to modify the definition of File Domains as follows. Instead of each processor owning one large block of columns, a smaller block of columns could be assigned to each processor in a cyclic fashion (block-cyclic assignment). This is analogous to the way files are striped across disks in a parallel file system. However, a preliminary study to determine a good way of defining File Domains has shown that the simple column-block distribution performs well in most cases. File Domains defined using a block-cyclic distribution of columns result in extra complications in the calculation of the FDAT, and in determining the source and destination processor and index sets during the communication phase, and do not improve the

1. Exchange access information with other processors and fill in the File Access Descriptor (FAD).
2. Compute intersection of FAD and this processor's File Domain (FD), and fill in the File Domain Access Table (FDAT).
3. Calculate the minimum of the lower-bounds and the maximum of the upper-bounds of all sections in the FDAT to determine the smallest section containing all the data needed from the File Domain.
4. Read this section using Data Sieving.
5. Communicate the data to the requesting processors.

Figure 8: Extended Two-Phase Algorithm for Reading Sections of Out-of-Core Arrays

I/O performance in many cases.

If the array is stored in the file in row-major order instead of column-major order, the only difference would be that the File Domains would be defined in terms of row-blocks and Data Sieving would be done one row at a time. The algorithm for reading sections of out-of-core arrays using the Extended Two-Phase Method is given in Figure 8.

5.1 Advantages

The Extended Two-Phase Method provides a very general way of accessing arbitrary sections of out-of-core arrays in an efficient manner. The first phase performs I/O optimizations at the cost of inter-processor communication in the second phase. Since communication cost is orders of magnitude lower than I/O cost, the overhead of communication is negligible. This method combines many small requests into single larger requests, thus providing larger granularity of data transfer and lower latency time. Another advantage is that multiple accesses to the same data in the file are eliminated. For example, if all processors need to read exactly the same section of the array, it will be read only once from the file and then broadcast to other processors over the interconnection network. Similarly, if the requests of two or more processors are overlapping, the overlapping portion will only be read once from the file.

5.2 Performance

We have tested the performance of the Extended Two-Phase Method versus the Direct Method on the Intel Touchstone Delta for many different access patterns. These access patterns can be

classified into three types:-

1. *Common Sections*: All processors need to read exactly the same section of the array.
2. *Overlapping Sections*: Parts of the section requested by a processor may overlap with parts of the sections requested by other processors.
3. *Distinct Sections*: The section requested by each processor does not have any data in common with the section requested by any other processor.

Table 1 compares the performance of the Extended Two-Phase Method and the Direct Method for reading common sections. The array size is 4K $\times$ 4K and the number of processors is 16. Figure 9 shows approximately where each of these sections is located in the array. We observe that the Extended Two-Phase Method performs considerably better than the Direct Method in all cases. This is primarily because, in the Extended Two-Phase Method, the common section is read only once and then broadcast to other processors, whereas in the Direct Method, all processors simultaneously try to access the same portion of the file resulting in extra I/O and contention for disks. For a 4K $\times$ 4K array with 16 processors, each processor's File Domain is of size 4K $\times$ 256. Thus, section I in Table 1 lies entirely in processor 0's File Domain. Section II is of the same size as Section I, but it lies partly in the File Domains of processors 0 and 1. Since the I/O is distributed among 2 processors, we observe that the Extended Two-Phase Method takes less time to read section II than section I. Section III is larger than sections I and II and lies in the File Domains of processors 1, 2 and 3. Section IV has a small number of rows but a large number of columns. The Extended Two-Phase Method performs particularly well for this case, because the I/O is distributed among many processors. Section V is an extreme case of this, as it contains only 16 rows but all 4096 columns. The Extended Two-Phase Method provides very significant improvement over the Direct Method in this case. Section VI is the transpose of V and actually resides contiguously in the file. Hence it can be read in a single operation. Even in this case, the Extended Two-Phase Method gives better performance, because the section is read only once.

Table 2 compares the performance of the Extended Two-Phase Method and the Direct Method for reading overlapping sections. Figure 10 shows approximately where these sections are located in the array. In order to represent these overlapping sections for all processors concisely, we use the following notation. Each processor's request is denoted by $(l_1 + ov1 \times p : u_1 + ov1 \times p : s_1, l_2 + ov2 \times p : u_2 + ov2 \times p : s_2)$, where p is the processor number and $ov1$, $ov2$ are some constants. The amount of overlap can be changed by varying $ov1$ and $ov2$. For example, the notation (1:100:1, $1+10p$:$100+10p$:1) in row I of Table 2 represents a group of overlapping sections with processor 0 requesting section (1:100:1, 1:100:1), processor 1 requesting section (1:100:1, 11:110:1), processor 2 requesting section (1:100:1, 21:120:1) and so on. The sections in rows I — IV overlap along

Table 1: Time for Reading Common Sections

4K × 4K array, 16 processors

No.	Array Section	Time (sec.)	
		Direct Read	Extended Two-Phase
I	(1:100:1, 1:100:1)	3.842	1.027
II	(200:300:1, 200:300:1)	4.145	0.883
III	(400:800:1, 400:800:1)	32.57	3.692
IV	(32:64:1, 128:1024:1)	29.09	2.780
V	(1:16:1, 1:4096:1)	118.2	3.241
VI	(1:4096:1, 1:16:1)	3.251	2.024

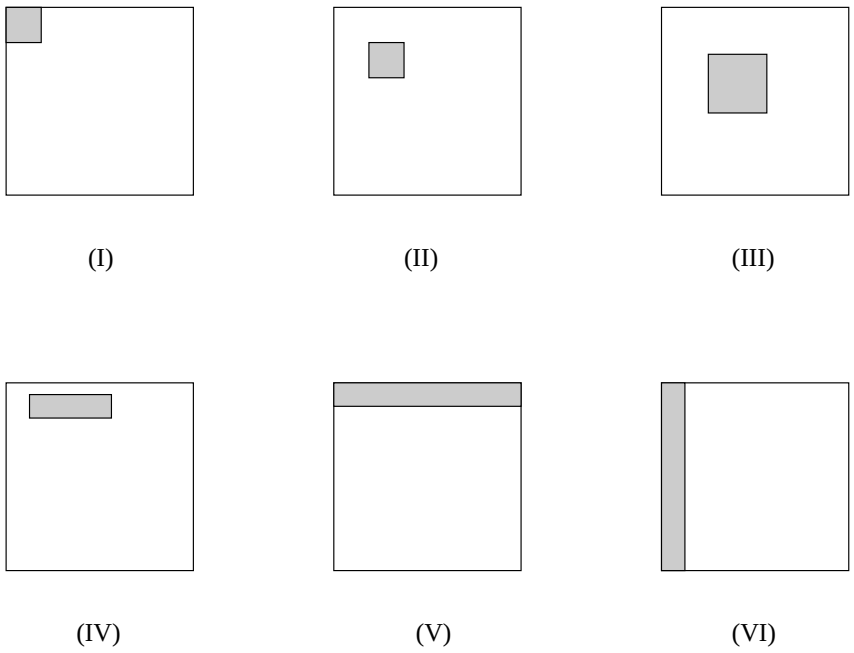

Figure 9: The Common Sections listed in Table 1 (not to scale)

Table 2: Time for Reading Overlapping Sections

4K × 4K array, 16 processors

No.	Array Section (p = processor number)	Time (sec.)	
		Direct Read	Extended Two-Phase
I	(1:100:1, 1+10p:100+10p:1)	1.522	1.830
II	(1:100:1, 1+50p:100+50p:1)	1.163	1.859
III	(400:800:1, 400+100p:800+100p:1)	16.51	3.348
IV	(1:4096:1, 1+8p:16+8p:1)	3.951	3.374
V	(1+50p:100+50p:1, 1:100:1)	3.763	1.994
VI	(400+100p:800+100p:1, 400:800:1)	21.79	11.84
VII	(1+8p:16+8p:1, 1:4096:1)	111.6	2.992
VIII	(200+100p:400+100p:1, 200+100p:400+100p:1)	3.184	2.986

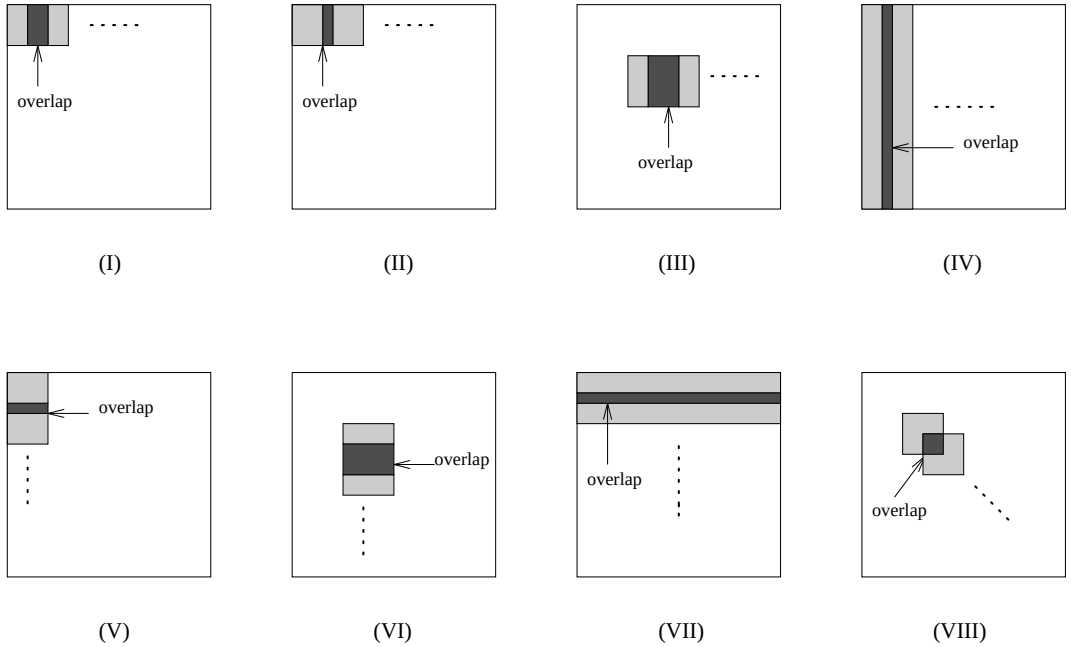

Figure 10: The Overlapping Sections listed in Table 2 (not to scale)

Table 3: Time for Reading Distinct Sections

4K × 4K array, 16 processors

No.	Array Section (p = processor number)	Time (sec.)	
		Direct Read	Extended Two-Phase
I	(1:100:1, 1+100p:100+100p:1)	1.747	1.954
II	(1+100p:100+100p:1, 1:100:1)	2.676	2.182
III	(200+200p:400+200p:1, 1:512:1)	9.246	5.680
IV	(1+32p:16+32p:1, 1:4096:1)	112.2	4.823
V	(200+200p:400+200p:1, 1+200p:512+200p:1)	10.52	4.524
VI	(1+32p:32+32p:1, 1+100p:1024+100p:1)	12.95	2.991

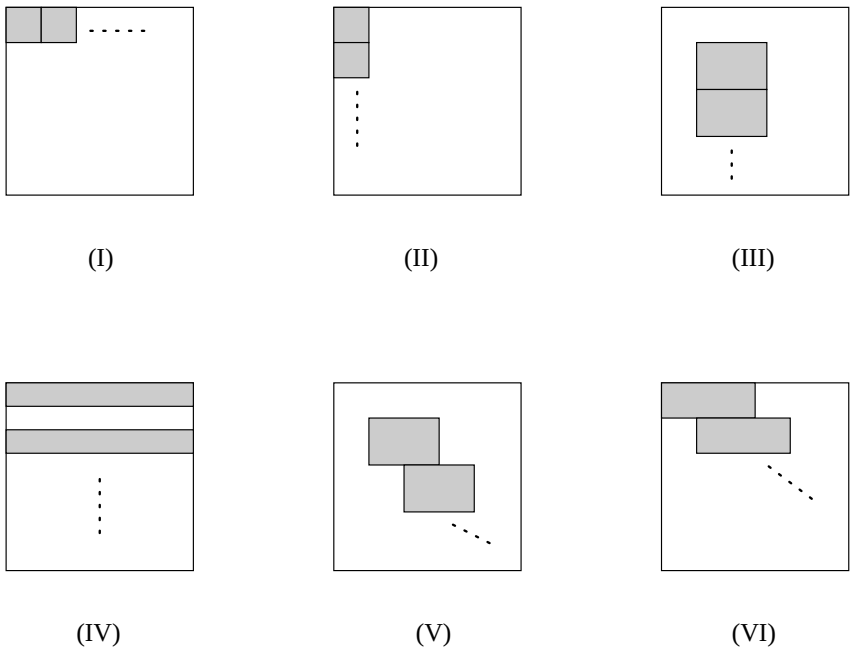

Figure 11: The Distinct Sections listed in Table 3 (not to scale)

columns, the sections in rows V — VII overlap along rows and the sections in row VIII overlap in both dimensions.

The sections in rows I and II are of the same size, but they differ in the amount of overlap. The sections in row I have more overlap than those in row II. For these two cases, we find that the Direct Method itself performs well because the sections are small, there are few columns and only parts of the sections overlap. The performance of the Direct Method is better for the sections in row II because there is less overlap among sections. The extra processing involved in the Extended Two-Phase Method is not actually required for these two cases, hence it takes more time than the Direct Method.

The sections in row III are much larger than in rows I and II, and we find that the Extended Two-Phase Method performs better. The sections in row IV contain all 4096 rows and only 16 columns and they lie contiguously in the file. Hence the Extended Two-Phase Method performs only slightly better than the Direct Method. The sections in rows V — VII are overlapped along rows, so they lie interleaved in the file. For these cases, the Extended Two-Phase Method performs considerably better because it reorders I/O requests so that data is read contiguously. Finally, in row VIII, the sections have overlap in both dimensions and the Extended Two-Phase Method again performs better.

Table 3 compares the performance of the Extended Two-Phase Method and the Direct Method for reading distinct sections. Figure 11 shows approximately where these sections are located in the array. We use the same notation as above, $(l_1+ov1\times p : u_1+ov1\times p : s_1, l_2+ov2\times p : u_2+ov2\times p : s_2)$, for representing distinct sections. The overlap factors $ov1$ and $ov2$ need to be large enough to ensure that the sections are distinct.

The sections in row I are located along rows and so the requests of different processors lie in separate locations in the file. Also, they are small and have few columns. For this case, the Direct Method itself performs well. The sections in rows II — IV are located along columns and so the requests of different processors lie interleaved in the file. Hence the Extended Two-Phase Method performs considerably better. The performance improvement is significant for the sections in row IV which contain all 4096 columns but only 16 rows. The sections in rows V and VI lie partly interleaved in the file. Even for these cases, the Extended Two-Phase Method performs the best.

6 Writing Sections of Out-of-Core Arrays

The Extended Two-Phase Method can also be used for writing sections of out-of-core arrays. The algorithm for this is essentially the reverse of the algorithm for reading sections. The write routine also assumes a collective I/O interface. All processors exchange access information and fill up the File Access Descriptor (FAD). Since each processor knows its own File Domain as well as the File

Domains of other processors, it can determine what portion of the section it needs to write lies in the File Domains of other processors. Each processor also computes its own File Domain Access Table (FDAT), which indicates how much data it needs to receive from other processors. The first phase of the Extended Two-Phase Method is to perform communication, so that all processors receive all the data that needs to be written to their File Domain.

The second phase is to write the data to the file in sequence and contiguously as far as possible. The FDAT is analyzed in the same way as in the read algorithm. Each processor determines the minimum and maximum of all indices in its FDAT. This effectively determines the smallest section which contains all the data that needs to be written to the File Domain. It may also contain some data which is not being written by any processor. The processor writes the useful data in this section one column at a time using Data Sieving [14]. However, for writing using Data Sieving, we cannot directly use the reverse of the method used for reading in Section 5. If the useful data is placed at appropriate locations (possibly with stride) in a temporary buffer and the temporary buffer is written to the file, the contents of the buffer between the useful data elements will overwrite the data in the file. In order to maintain data consistency, it is necessary to first read the entire column from the file into the temporary buffer. Then, the data elements to be written in that column can be stored at appropriate locations in the buffer and the entire column can be written back to disk. Thus, writing sections requires twice the amount of I/O compared to reading sections, because to write each column, the corresponding column has to first be read into memory. It may be possible to avoid this extra reading in cases where the entire column contains useful data to be written. However, detecting this fact requires each processor to do a more extensive analysis of the FDAT, to make sure that there are no "holes" between the data sets being written by different processors.

6.1 Performance

We only consider the case where each processor writes a distinct section to the file, because it is unlikely that processors will want to write overlapping or common sections. Table 4 compares the performance of the Extended Two-Phase Method and the Direct Method for writing distinct sections. The sections chosen are the same as those for reading in Table 3, and are shown in Figure 11.

We use the most general algorithm for writing in the Extended Two-Phase Method, which requires an extra read for each write. Hence for the sections in row I, the Direct Method performs better because it does not require the extra read and also these sections are small with few columns. The sections in rows II – IV lie interleaved in the file, so the Extended Two-Phase Method performs much better than the Direct Method. The sections in rows V and VI lie partly interleaved in the file and even for these cases, the Extended Two-Phase Method performs considerably better.

Table 4: Time for Writing Distinct Sections

4K $\times$ 4K array, 16 processors

No.	Array Section	Time (sec.)	
		Direct Write	Extended Two-Phase
I	(1:100:1, 1+100p:100+100p:1)	1.839	3.250
II	(1+100p:100+100p:1, 1:100:1)	2.678	2.501
III	(200+200p:400+200p:1, 1:512:1)	11.64	8.715
IV	(1+32p:16+32p:1, 1:4096:1)	98.96	10.25
V	(200+200p:400+200p:1, 1+200p:512+200p:1)	11.33	6.461
VI	(1+32p:32+32p:1, 1+100p:1024+100p:1)	13.75	4.994

7 Related Work

The Jovian library [1] also provides support for accessing sections of out-of-core arrays. This library uses separate processes called *coalescing processes* to perform I/O optimizations. All application processes send I/O requests to predetermined coalescing processes. Each coalescing process is responsible for accessing a particular logical I/O device. The coalescing processes communicate with each other to exchange access information, so that each coalescing process knows what requests are directed at its logical I/O device. The coalescing processes perform the necessary I/O and forward data directly to the original requesting application processes. The Extended Two-Phase Method proposed in this paper provides similar functionality using only a single routine for reading sections and a single routine for writing sections, which can be implemented very easily. It does not incur the overhead of managing many processes and the associated context switching.

Disk-directed I/O [11] is another technique which uses collective I/O. In disk-directed I/O, compute processors collectively send a single request to all I/O processors, which then perform I/O efficiently, and send the data to the compute processors. In other words, it is the I/O processors that decide the best way of performing I/O, and not the compute processors. Disk directed I/O needs to be implemented at the operating system or file system level which requires considerable effort and the implementation is not portable. There is no working implementation of disk-directed I/O on any parallel machine at present. On the other hand, we have implemented the Extended Two-Phase Method on the Intel Touchstone Delta and Paragon, and we plan to port it to the IBM SP-1/SP-2 using the Vesta/PIOFS file system [6]. It can also be implemented on top of any new portable standard interfaces such as the proposed MPI-IO interface [5], resulting in portable implementations.

8 Conclusions

We have proposed a technique, called the Extended Two-Phase Method, for accessing sections of out-of-core arrays in an efficient manner. This method performs I/O efficiently by combining several I/O requests into fewer larger requests, eliminating multiple disk accesses for the same data and reducing contention for disks. We have tested the performance of this method for a wide range of access patterns. Except for a few simple cases where the Direct Method is itself good enough, the Extended Two-Phase Method is found to provide considerable performance improvement over the Direct Method for both reading and writing data. Another advantage of the Extended Two-Phase Method is that it is simple and can be easily implemented.

References

[1] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz. Jovian: A Framework for Optimizing Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, October 1994.

[2] R. Bordawekar and A. Choudhary. Language and Compiler Support for Parallel I/O. In *Proceedings of IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*, April 1994.

[3] 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.

[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. Also available as CRPC Technical Report CRPC–TR94483–S.

[5] 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 0.2. Technical Report IBM Research Report RC 19841(87784), IBM T. J. Watson Research Center, November 1994.

[6] 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.

[7] 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*, April 1993.

[8] J. del Rosario, M. Harry, and A. Choudhary. The Design of VIP-FS: A Virtual Parallel File System for High Performance Parallel and Distributed Computing. Technical Report SCCS-628, NPAC, Syracuse University, May 1994.

[9] G. Fox, R. Williams, and P. Messina. *Parallel Computing Works*. Morgan Kaufmann Publishers Inc., 1994.

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

[11] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994.

[12] T. Singh and A. Choudhary. ADOPT: A Dynamic Scheme for Optimal Prefetching in Parallel File Systems. Technical Report SCCS–627, NPAC, Syracuse University, 1994.

[13] 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.

[14] 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*, October 1994.

[15] R. Thakur and A. Choudhary. All-to-All Communication on Meshes with Wormhole Routing. In *Proceedings of the* 8^{th} *International Parallel Processing Symposium*, pages 561–565, April 1994.

[16] R. Thakur, A. Choudhary, and G. Fox. Runtime Array Redistribution in HPF Programs. In *Proceedings of the Scalable High Performance Computing Conference*, pages 309–316, May 1994.

[17] R. Thakur, R. Ponnusamy, A. Choudhary, and G. Fox. Complete Exchange on the CM-5 and Touchstone Delta. to appear in *The Journal of Supercomputing*.