CENTER FOR ADVANCED COMPUTING RESEARCH
California Institute of Technology

Scalable I/O Initiative
Technical Report

CACR–103 June 1995

An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays*

Rajeev Thakur
Math. and Computer Science Div.
Argonne National Laboratory
Argonne, IL 60439
thakur@mcs.anl.gov

Alok Choudhary
Dept. of Elect. and Comp. Eng.
Syracuse University
Syracuse, NY 13244
choudhar@cat.syr.edu

Abstract

A number of applications on parallel computers deal with very large data sets which cannot fit in main memory. In such cases, data must be stored in files on disks and fetched into main memory during program execution. In HPF programs with large out-of-core arrays stored in files, it is necessary to read/write smaller sections of the arrays from/to files. This article describes an Extended Two-Phase Method for accessing sections of out-of-core arrays in an efficient manner. This method uses collective I/O in which processors cooperate to perform I/O efficiently. The unique feature of this method is that the I/O workload is divided among processors dynamically, depending on the access requests. This results in a more balanced distribution of I/O among processors than a static assignment, and gives considerably better performance. The proposed dynamic partitioning scheme is very flexible and can be easily adapted to match the storage pattern of any parallel file system. Extensive performance studies prove the superiority of the Extended Two-Phase Method over a Direct Method for different access patterns, array sizes, and number of processors.

*This work was supported in part by a National Science Foundation Young Investigator Award (CCR-9357840); by the Scalable I/O Initiative, a multiagency project funded by the Advanced Research Projects Agency (contract number DABT63-94-C-0049), the Department of Energy, the National Aeronautics and Space Administration, and the National Science Foundation; and by a grant from Intel Scalable Systems Division. This work was performed while Rajeev Thakur was a graduate student at Syracuse University. 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 the Center for Research on Parallel Computation.

1 Introduction

Parallel computers are increasingly being used for large computationally intensive as well as data-intensive applications, such as large-scale computations in physics, chemistry, biology, engineering, medicine, and other sciences. The data required by many of these applications is too large to fit in main memory and so it must be stored in files on disks [10]. I/O needs to be performed during program execution to access data from disks. The Applications Working Group of the Scalable I/O Initiative has compiled a list of several parallel applications that require a large amount of I/O [15].

Recently, High Performance Fortran (HPF) has been defined as a standard language for writing portable parallel programs [11, 12]. It is expected that HPF will soon be the language of choice for developing large compute- and data-intensive parallel applications. The main data structures in HPF are multidimensional arrays. Hence, an HPF program for an application with a large out-of-core data set will have one or more large out-of-core arrays. During program execution, it will be necessary to fetch sections of the arrays from files, perform computations using the sections, and store the results back to files. The array sections could have strides in each dimension, for example, corresponding to the strides in some array assignment statement in the HPF program. Each processor will need different sections of the arrays depending on the data distribution. In other words, to run an out-of-core HPF program on a distributed memory machine, it has to be translated to a node program with calls to I/O routines to move array sections back and forth between main memory and disks. Examples of parallel applications that require accessing sections of out-of-core arrays are Hartree-Fock calculations in chemistry, very large Fast Fourier Transforms required to detect faint radio pulsars, seismic data processing, weather and climate modeling, 3D turbulence simulations, scattering and radiation problems in computational electromagnetics, parallel volume rendering, and several others [15].

Language constructs for accessing arrays stored in files have been proposed previously in the context of Vienna Fortran [4]. The issues involved in compiling out-of-core programs are discussed in [2]. This article describes an efficient method — called the Extended Two-Phase Method — for accessing sections of out-of-core arrays at runtime. The Extended Two-Phase Method uses collective I/O in which processors cooperate to perform I/O efficiently. The unique feature of this method is that the I/O workload is divided among processors dynamically, depending on the access requests. This results in a more balanced distribution of I/O among processors than a static assignment and gives considerably better performance. The proposed dynamic partitioning scheme is very flexible and can be easily adapted to match the storage pattern of any parallel file system.

The best way to use the Extended Two-Phase Method is to implement it as a library which can be called from a translated HPF program at runtime. We have developed a library of routines for parallel I/O, called the PASSION Runtime Library [18, 17], which includes routines for accessing sections of out-of-core arrays using the Extended Two-Phase Method.

The rest of this article is organized as follows. We first motivate the need for using collective I/O in Section 2. The Extended Two-Phase Method is explained in Section 3. In Section 4, we describe a static scheme for partitioning I/O among processors and then show how the partitioning can be improved by using a dynamic scheme. Extensive performance and scalability results are presented in Section 5. The advantages of the Extended Two-Phase Method with dynamic partitioning are discussed in Section 6, followed by Conclusions in Section 7.

2 Need for Collective I/O

HPF uses a data-parallel programming model in which 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 out-of-core data in an efficient manner, which is known as *collective I/O*. The need for collective I/O can be explained with the help of the following example.

Consider the large out-of-core array shown in Figure 1 and assume that it is stored in a file in column-major order. Four processors ($0-3$) need to access a block of rows, each as shown. Since the access requests are orthogonal to the file storage order, the data requested by each processor is located non-contiguously in the file. Also, the requests of different processors are interleaved in the file. A portion of 0's request is located at the beginnning of the file, followed by some unwanted data (gap), then a portion of 1's request followed by a gap, then a portion of 2's request followed by a gap, then a portion of 3's request followed by a gap, then again a portion of 0's request, and so on. To read the data using the interface provided by most of the existing parallel file systems, each processor must explicitly seek to the appropriate location in the file, read the small chunk of data, then seek to the next location, and so on. We call this the *Direct Method*. The Vesta file system [7] and the nCUBE file system [8] do provide support for the user to specify a logical view of the data to be read, and use a single call to read data. However, each processor's request is serviced independently and there is no collective optimization based on the requests of all processors.

The drawback of the Direct Method is that a large number of low granularity requests may arrive from different processors in any order, which usually results in poor performance since I/O latency is very high. The basic premise behind collective I/O is that, if a group of processors needs to read/write data at the same time, the processors can cooperate among themselves to perform I/O

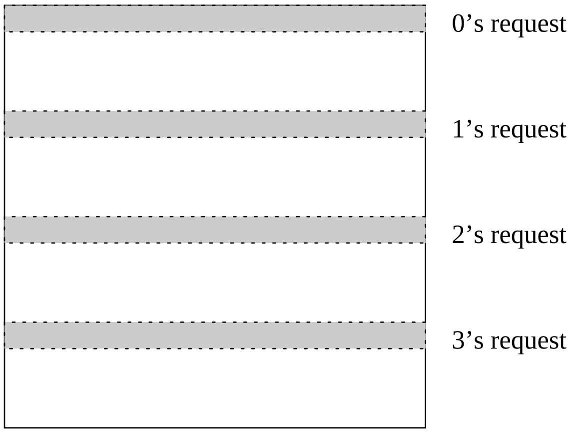

Figure 1: Accessing row blocks of a file stored in column-major order

efficiently in large chunks and in the proper order. This cooperation is usually possible in loosely synchronous SPMD programs in which all processors perform similar operations on different data sets, and so a group of processors may need to perform I/O at the same time. An appropriate interface needs to be provided for processors to specify a collective I/O request. The PASSION Runtime Library provides such an interface [5]. The MPI-IO interface [6] also provides some support for collective I/O.

The collective I/O operation can be implemented either as a library on top of the file system, or at the file system level itself. In an earlier work, our group proposed a technique, called Two-Phase I/O [9, 3], for performing collective I/O at the library level to read/write entire in-core arrays. In this method, I/O is done in two phases. In the first phase, processors cooperate to read data from files in large chunks. In the second phase, data is redistributed among processors. Disk-directed I/O [13] is a technique which aims to carry out collective I/O at the file system level. In disk-directed I/O, a collective I/O request is sent to all I/O nodes, which determine the order and timing of the flow of data.

3 Extended Two-Phase Method

The Two-Phase Method [9, 3] is a collective I/O technique for reading an entire in-core array from a file into a distributed array in main memory, and conversely, for writing a distributed in-core array to a file. I/O is done in two phases. In the first phase, processors always read data assuming a *conforming distribution*. A conforming distribution is defined as a distribution of an array among processors such that each processor's local array is stored contiguously in the file, resulting in each processor reading a single large chunk of data. For an array stored in a file in column-major order, a column-block distribution is the conforming distribution. In the second phase, data is redistributed among processors to the desired distribution. Since I/O cost is orders of magnitude more than communication cost, the cost incurred by the second phase is negligible. This Two-Phase approach is found to give consistently good performance for all distributions [9, 3].

We have extended the basic Two-Phase Method to access arbitrary sections of out-of-core arrays. This Extended Two-Phase Method performs I/O for out-of-core arrays efficiently by:

- dynamically partitioning the I/O workload among processors, depending on the access requests,
- combining several I/O requests into fewer larger granularity requests,
- reordering requests so that data is accessed in proper sequence,
- eliminating multiple disk accesses for the same data, and
- reducing contention for disks.

3.1 Reading Sections of Out-of-Core Arrays

We first describe the Extended Two-Phase Method for reading regular sections of out-of-core arrays. We define a regular section of an array as a section which can be specified by a lower-bound, upper-bound and stride in each dimension. For the purpose of explanation, we consider the case where each processor needs to read some regular section of a two-dimensional array stored in the file in *column-major order*. The Extended Two-Phase Method can actually be used for arrays with any number of dimensions, stored in any order in the file and accessed by any number of processors. Section 6 discusses how the general case can be implemented.

In the Extended Two-Phase Method, the I/O workload is divided among processors. For this, we assign *ownership* to portions of the file such that a processor can directly access only the portion of the file it *owns*. Hence, the file is logically divided into *domains*. The portion of the file which a processor can directly access is called its *File Domain (FD)*. For a file stored in column-major order, the file domain of each processor is some set of columns of the array. The selection of file domains is important because this determines how the I/O workload gets divided among processors. See Section 4 for further discussion on this topic.

Let us assume that each processor needs to read some regular section $(l_1 : u_1 : s_1, l_2 : u_2 : s_2)$ of the array in global coordinates. In the first step of the Extended Two-Phase Method, processors exchange their own access information (the indices $l_1, u_1, s_1, l_2, u_2, s_2$) with other processors, so that each processor knows the access requests of other processors. This information is stored in a data

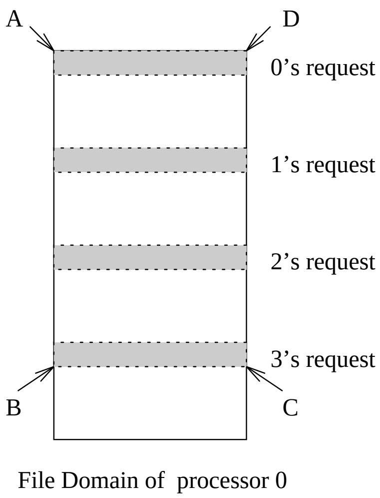

Figure 2: Processor 0 needs to read the requested data from its file domain. The section ABCD is the smallest section that includes all the data that needs to be read. This section is read using data sieving.

structure called the *File Access Descriptor (FAD)*. The FAD contains exactly the same information on all processors. This exchange phase is not required if the collective I/O interface itself provides information about other processors' access requests.

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 its own file domain. This information is stored in a data structure called the *File Domain Access Table (FDAT)*. Thus, the FDAT of a processor contains information concerning which portions of its file domain have been requested by other processors.

Each processor now reads data from its file domain, as determined by the FDAT. For example, Figure 2 shows the file domain of processor 0 and, for some access pattern, the portions of this file domain which have been requested by other processors. 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. However, as in the case of Figure 2, this may result in too many small accesses which are not in sequence. For the read to be done efficiently, it is important that the FDAT be analyzed so that the file is accessed in sequence and contiguously.

We have devised a very general method for analyzing the information in the FDAT, which ensures that the file is read in sequence and contiguously. 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 (for example, section ABCD in Figure 2). This section may also contain some data which is not required by any processor. If the processor attempts to read only the useful data, it may result in a number of small strided accesses. To avoid this, the processor uses an optimization known as *data sieving* which we proposed in an earlier work [18]. The processor reads a 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, depending on which processors need the data. The entire section is read from the file domain in this fashion. It is possible to read more than one column at a time, if there is enough temporary memory available to do sieving on the set of columns. 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 determine what data must be sent to which processor. Since each processor knows the file domains of other processors and its own access request, it can calculate how much data it needs to receive from other processors, as well as the locations in main memory where the received data needs to be placed.

The two phases of the Extended Two-Phase Method either can be done distinctly by performing all the I/O first and then the communication, or they can be overlapped (pipelined) by reading smaller portions of data and communicating it.

3.2 Writing Sections of Out-of-Core Arrays

The algorithm for writing sections is essentially the reverse of the algorithm for reading sections. From the information in the File Access Descriptor (FAD), each processor can determine the portions of its write request that lie 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, to be written to its file domain. The first phase of the Extended Two-Phase Method for writing is to perform communication to get this data from other processors.

Data is written to the file in the second phase. 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 includes all the data that needs to be written to the file domain. It may also include 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 [18]. However, for writing using data sieving, we cannot directly use the reverse of the method used for reading. If the useful data is placed at appropriate locations (possibly with stride) in a temporary buffer, and the buffer is written to the file, the contents of the buffer between the useful data elements will overwrite the data in the file. To maintain data consistency, it is first necessary to 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 first must be read into memory. It is possible to avoid this extra reading in cases where the entire column contains useful data to be written. This requires each processor to do a more extensive analysis of the FDAT, to make certain that there are no "holes" between the data sets being written by different processors in the same column, and that there are no strides in the section being written by each processor. As in the case of reading sections, it is possible to do sieving with more than one column at a time, if there is enough temporary memory available.

If the sections requested to be written by different processors have some elements in common, there is a data consistency problem. The result depends on how the Extended Two-Phase Method has been implemented. In our implementation, if there are write requests from multiple processors to the same location, the data from the highest numbered processor is written to the file.

4 Partitioning I/O Among Processors

In the Extended Two-Phase Method, processors cooperate to perform I/O. The exact partitioning of the I/O workload among processors depends on how file domains are defined. In general, the partitioning of I/O can be done either statically or dynamically.

4.1 Static Partitioning

One way of partitioning I/O is to assign a block of columns of the entire out-of-core array to each processor, as if the array were distributed among processors in a column block fashion. Thus the file domain of each processor is a block of columns of the array, which is stored contiguously in the file. The file domains are predefined and fixed. The size of each domain can be determined from the size of the array and the number of processors, and is independent of the access requests. This is called a static partitioning of I/O among processors. Note that this is just a logical partitioning of the file among processors, so that each processor directly accesses only a particular portion of the file (its file domain). Figure 3(A) shows the file domains with static partitioning when there are four processors.

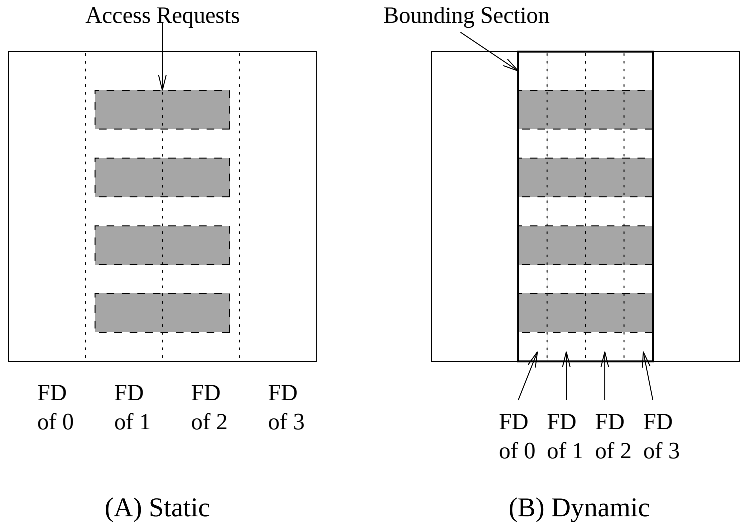

Figure 3: Static versus Dynamic Partitioning, FD = File Domain

4.2 Dynamic Partitioning

The main drawback of static partitioning is that the partitioning is independent of the type of access requests. For many types of access patterns, this may result in an imbalance of I/O among processors. Some processors may perform more I/O than others and some may not perform any I/O at all. For example, consider the access pattern in Figure 3. If the partitioning is done statically, the access requests span the file domains of only two processors; therefore, only two processors (1 and 2) perform all the I/O. The remaining two processors (0 and 3) wait to receive data sent by 1 and 2. Another drawback of static partitioning is that, as the size of the out-of-core array is increased, keeping the number of processors fixed, the size of each file domain also increases. Hence, for the same access pattern, as the size of the out-of-core array is increased, the access requests span the file domains of fewer processors resulting in greater imbalance of I/O and lower performance.

The I/O throughput can potentially be improved by partitioning the I/O workload among processors dynamically, depending on the array sections requested. This partitioning is illustrated in Figure 3(B). For a file stored in column-major order, each processor calculates the lowest and highest among the columns of the sections requested by all processors. The section formed by these columns and all the rows of the out-of-core array is called the *bounding section.* The bounding section includes the sections requested by all processors and it lies contiguously in the file. Figure 3(B) shows the bounding section for the given access requests. File domains are determined by dividing the bounding section among the processors in a column-block fashion. Therefore, the file domain of each processor is a contiguous chunk of the bounding section.

1. Exchange access information with other processors and fill in the File Access Descriptor (FAD).
2. Determine the smallest section (called the bounding section) which includes the sections requested by all processors.
3. The file domain of each processor is determined by dividing this bounding section among the processors in a column-block manner for arrays stored in column-major order.
4. Compute the intersection of the FAD and this processor's file domain, and fill in the File Domain Access Table (FDAT).
5. 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.
6. Read this section using data sieving and communicate the data to the requesting processors.

Figure 4: Extended Two-Phase Method for reading sections of out-of-core arrays with dynamic partitioning of I/O

If the requested sections span all the columns of the out-of-core array, the dynamically selected file domains are exactly the same as those determined statically. However, if the sections span only a few columns, as in Figure 3, dynamic partitioning provides a much better balance of I/O among processors. The memory requirements of the Extended Two-Phase Method are also reduced, because the file domain of each processor is smaller. In the static case, if all requested sections lie in a single processor's file domain, all the requested data may not fit in the memory of that processor. Hence, I/O and communication may need to be done in stages several times. This is less likely to occur in the case of dynamic partitioning, because the requested data is more evenly divided among processors.

The algorithm for reading sections of out-of-core arrays using the Extended Two-Phase Method with dynamic partitioning is given in Figure 4. 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 along rows.

5 Performance

We used the Intel Touchstone Delta System for an experimental study of the performance of the Extended Two-Phase Method. The Touchstone Delta has 512 compute nodes (each an Intel i860/XR microprocessor) and 32 I/O nodes (each an Intel 80386 microprocessor). Each I/O node is connected to two disks, resulting in a total of 64 disks. A Concurrent File System [16] is provided for parallel access to files. By default, a file is striped across all 64 disks in a round-robin fashion in blocks of size 4 Kbytes. The performance of the Concurrent File System on the Touchstone Delta is discussed in detail in [1].

We have studied the performance of the Extended Two-Phase Method versus the Direct Method extensively for many different access patterns, using both static and dynamic partitioning. The access patterns can be classified into three basic types:-

1. *Common Sections*: All processors need to access 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.

5.1 Reading Common Sections

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 5 shows approximately where each of these sections is located in the array. The performance of the Extended Two-Phase Method was measured using both static and dynamic partitioning. We observe that in all cases, the Extended Two-Phase Method performs considerably better than the Direct Method. 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 read the same section from the file, resulting in extra I/O overhead.

In all cases, the Extended Two-Phase Method takes a lot less time with dynamic partitioning than with static partitioning. In the case of static partitioning, for a 4K $\times$ 4K array with 16 processors, each processor's file domain is of size 4K $\times$ 256. Thus, all sections except V are located in the file domains of only a few processors. On the other hand, with dynamic partitioning, each section is evenly divided among all available processors, resulting in higher I/O throughput. Since

Table 1: Comparison of Direct Method and Extended Two-Phase Method (static and dynamic partitioning) for reading common sections. Array size 4K $\times$ 4K real numbers (single precision), 16 processors, time in seconds.

No.	Array Section	Direct Read	Extended Two-Phase Static	Extended Two-Phase Dynamic
I	(1:100:1, 1:100:1)	1.632	1.027	0.431
II	(200:300:1, 200:300:1)	1.867	0.883	0.363
III	(400:800:1, 400:800:1)	6.265	3.692	1.056
IV	(32:64:1, 128:1024:1)	9.995	2.780	1.318
V	(1:16:1, 1:4096:1)	52.06	3.241	3.241
VI	(1:4096:1, 1:16:1)	1.216	2.024	0.420

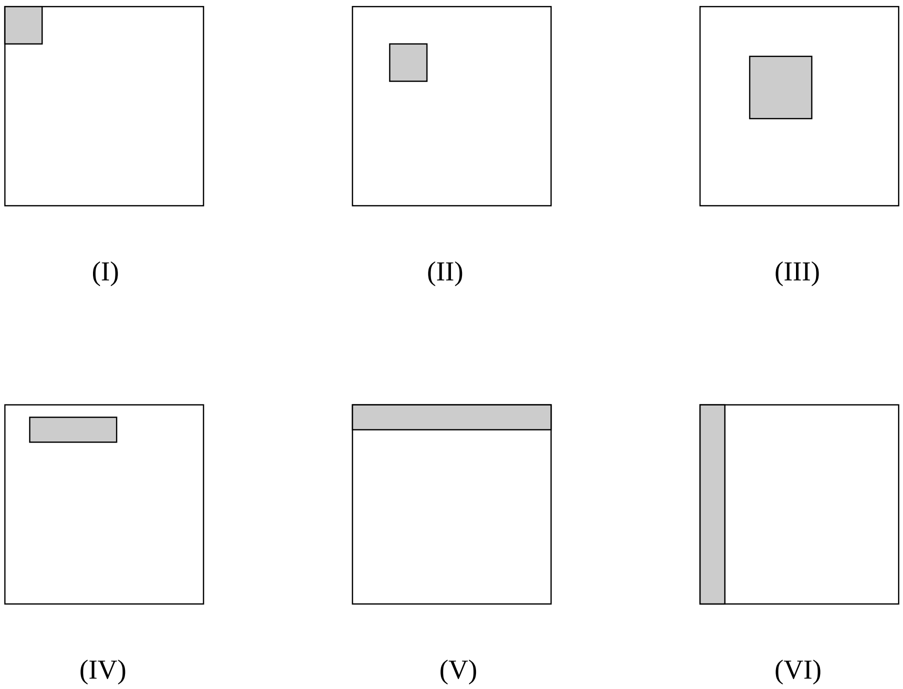

Figure 5: The common sections listed in Table 1 (not to scale)

Section V spans all 4096 columns, the statically and dynamically selected file domains are identical, resulting in identical performance.

In cases where the sections span a large number of columns (e.g., Section V), the Extended Two-Phase Method provides a significant improvement over the Direct Method. This improvement occurs because for such cases, the Direct Method results in a large number of small requests spread across the entire file. Conversely, in the Extended Two-Phase Method, the I/O gets evenly divided among all processors, the requests are reordered, and data is read in large chunks.

5.2 Reading Overlapping Sections

Table 2 compares the performance of the Extended Two-Phase Method and the Direct Method for reading overlapping sections. Figure 6 shows approximately where these sections are located in the array. 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 cases I — IV overlap along columns, the sections in cases V — VII overlap along rows, and the sections in case VIII overlap in both dimensions.

We observe that the Extended Two-Phase Method with dynamic partitioning performs the best in all cases. The sections in cases I and II are of the same size, but they differ in the amount of overlap. The sections in case I have more overlap than those in case II. Since the total number of columns of the out-of-core array spanned by the sections in case I is less than that by the sections in case II, it takes less time to read the sections in case I. Sections in cases IV, V and VI span only a few columns. For these cases, the Direct Method performs better than the Extended Two-Phase Method with static partitioning. This is because static partitioning results in only a few processors performing I/O. However, the Extended Two-Phase Method with dynamic partitioning performs better than the Direct Method, since there is a better distribution of I/O among processors. The sections in case VII span all columns of the array, which is the worst case for the Direct Method. Finally, in case VIII, the sections have overlap in both dimensions and again the Extended Two-Phase Method with dynamic partitioning takes the least time.

Table 2: Comparison of Direct Method and Extended Two-Phase Method (static and dynamic partitioning) for reading overlapping sections. Array size 4K $\times$ 4K real numbers (single precision), 16 processors, time in seconds.

No.	Array Section (p = processor number)	Direct Read	Extended Two-Phase Static	Extended Two-Phase Dynamic
I	(1:100:1, 1+10p:100+10p:1)	2.000	1.830	0.693
II	(1:100:1, 1+50p:100+50p:1)	4.627	1.859	0.875
III	(400:800:1, 400+100p:800+100p:1)	8.097	3.348	2.477
IV	(1:4096:1, 1+8p:16+8p:1)	1.152	3.374	0.826
V	(1+50p:100+50p:1, 1:100:1)	1.579	1.994	0.524
VI	(400+100p:800+100p:1, 400:800:1)	7.442	11.84	1.361
VII	(1+8p:16+8p:1, 1:4096:1)	50.32	2.992	2.992
VIII	(200+100p:400+100p:1, 200+100p:400+100p:1)	3.104	2.986	1.739

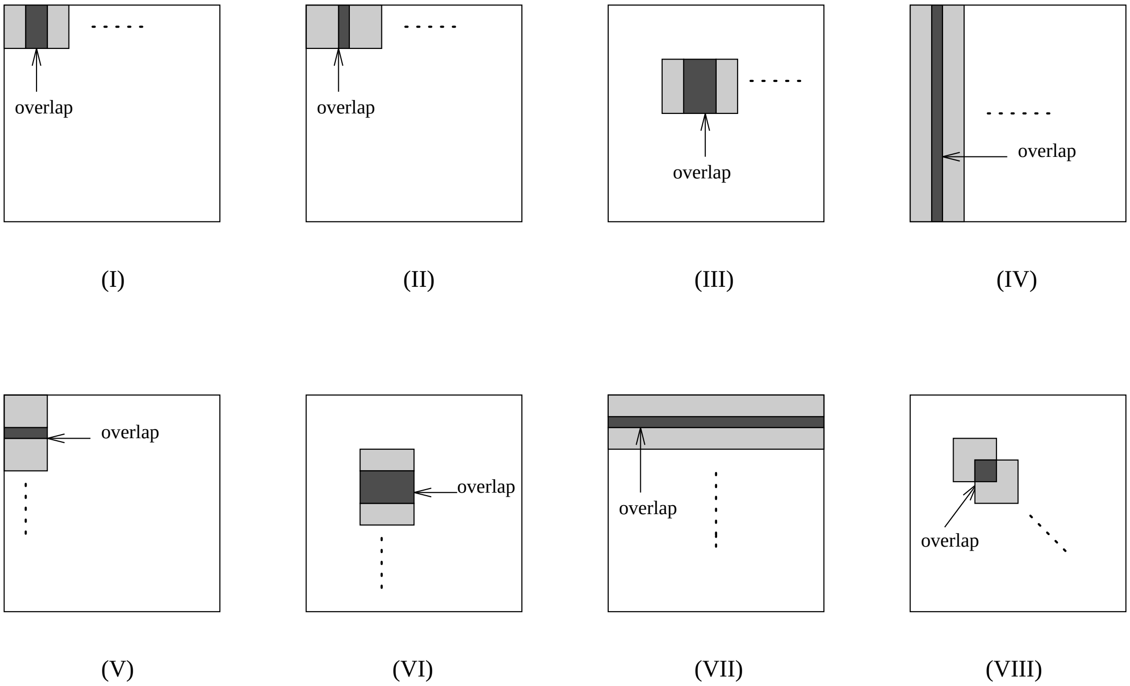

Figure 6: The overlapping sections listed in Table 2 (not to scale)

5.3 Reading Distinct Sections

Table 3 compares the performance of the Extended Two-Phase Method and the Direct Method for reading distinct sections. Figure 7 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.

In all cases, the Extended Two-Phase Method with dynamic partitioning performs the best. The relative performance of the three methods is similar to that for overlapping sections in Table 2. The sections in case I are located along rows; thus, the requests of different processors are situated in separate locations in the file. As a result, the Extended Two-Phase Method performs only slightly better. The sections in cases II — IV are located along columns, and thus the requests of different processors lie interleaved in the file. Hence, the Extended Two-Phase Method performs considerably better. Static partitioning does not give good performance for the sections in case II because they span only a few columns. The best case for the Extended Two-Phase Method is case IV since the sections span all the columns. The sections in cases V and VI lie partly interleaved in the file. Even for these cases, the Extended Two-Phase Method performs the best.

5.4 Writing Distinct Sections

We consider only 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 (Figure 7).

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

5.5 Accessing Sections with Non-Unit Strides

We have also tested the performance for accessing sections with non-unit strides. When an array section has a non-unit stride, each element of the array lies strided in the file. The only way of reading such array sections using a direct method is to seek explicitly to each individual element

Table 3: Comparison of Direct Method and Extended Two-Phase Method (static and dynamic partitioning) for reading distinct sections. Array size 4K $\times$ 4K real numbers (single precision), 16 processors, time in seconds.

No.	Array Section (p = processor number)	Direct Read	Extended Two-Phase Static	Extended Two-Phase Dynamic
I	(1:100:1, 1+100p:100+100p:1)	1.976	1.954	1.304
II	(1+100p:100+100p:1, 1:100:1)	1.633	2.182	0.548
III	(200+200p:400+200p:1, 512:1024:1)	8.016	5.680	1.725
IV	(1+32p:16+32p:1, 1:4096:1)	51.63	4.823	4.823
V	(200+200p:400+200p:1, 1+200p:512+200p:1)	5.466	4.524	3.912
VI	(1+32p:32+32p:1, 1+100p:1024+100p:1)	12.02	2.991	2.371

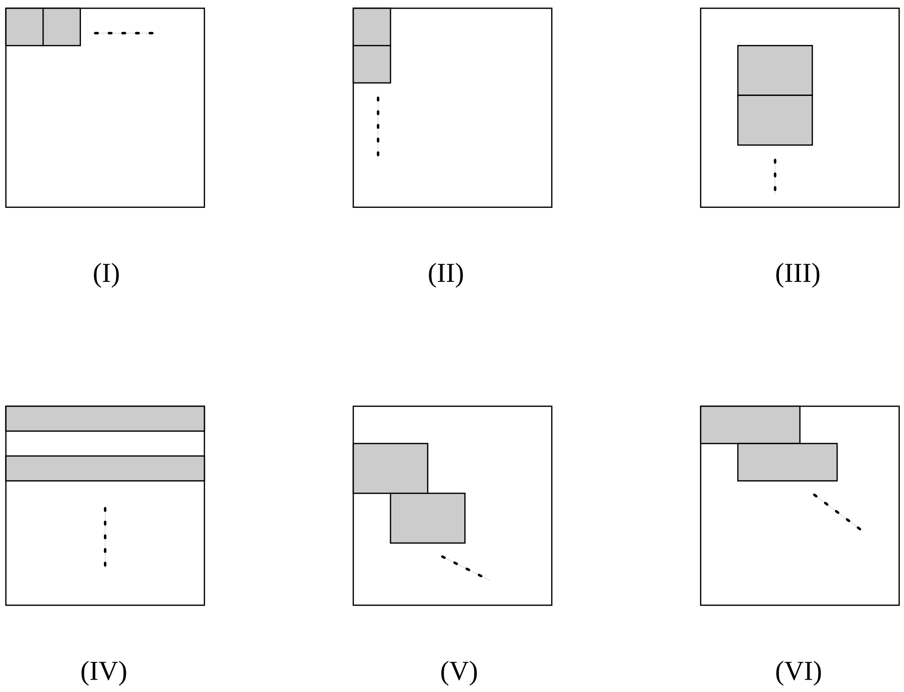

Figure 7: The distinct sections listed in Table 3 (not to scale)

Table 4: Comparison of Direct Method and Extended Two-Phase Method (static and dynamic partitioning) for writing distinct sections. Array size 4K $\times$ 4K real numbers (single precision), 16 processors, time in seconds.

No.	Array Section (p = processor number)	Direct Write	Extended Two-Phase Static	Extended Two-Phase Dynamic
I	(1:100:1, 1+100p:100+100p:1)	1.944	3.250	2.801
II	(1+100p:100+100p:1, 1:100:1)	1.182	2.901	0.854
III	(200+200p:400+200p:1, 512:1024:1)	4.202	8.715	3.569
IV	(1+32p:16+32p:1, 1:4096:1)	24.85	10.25	10.25
V	(200+200p:400+200p:1, 1+200p:512+200p:1)	5.155	5.461	4.401
VI	(1+32p:32+32p:1, 1+100p:1024+100p:1)	8.233	4.994	4.274

Table 5: Comparison of Direct Method and Extended Two-Phase Method (static and dynamic partitioning) for reading sections with non-unit strides. Array size 4K $\times$ 4K real numbers (single precision), 16 processors, time in seconds.

No.	Array Section (p = processor number)	Direct Read	Extended Two-Phase Static	Extended Two-Phase Dynamic
I	(p+1:4096:$nprocs$, p+1:4096:$nprocs$)	210.8	9.330	9.330
II	(1+250p:250+250p:2, 1+250p:250+250p:2)	53.13	3.610	2.842
III	(1+200p:500+200p:3, 1+200p:500+200p:3)	87.19	4.394	4.387
IV	(1+64p:64+64p:2, 500:2500:3)	96.20	4.759	3.848
V	(500:2500:3, 1+64p:64+64p:2)	130.7	4.574	2.340

and read only that element. This results in very low granularity of data transfer, which is very expensive. The Extended Two-Phase Method overcomes this drawback of the Direct Method by reordering requests and using data sieving for larger granularity reads.

Table 5 shows the performance of the Extended Two-Phase Method for reading sections with non-unit strides. The sections in case I span almost the entire array, with stride equal to the number of processors. As a result, static and dynamic partitioning take exactly the same time. The sections in cases II and III are located diagonally across the out-of-core array. The sections in case IV are located along columns and the sections in case V are located along rows. In all cases, the Extended Two-Phase Method is more than 20 times faster than the Direct Method. Table 6 shows the performance of the Extended Two-Phase Method for writing sections with non-unit strides. The sections chosen are the same as in Table 5. Even for writing sections, the Extended Two-Phase Method provides considerable performance improvement.

Table 6: Comparison of Direct Method and Extended Two-Phase Method (static and dynamic partitioning) for writing sections with non-unit strides. Array size 4K $\times$ 4K real numbers (single precision), 16 processors, time in seconds.

No.	Array Section (p = processor number)	Direct Write	Extended Two-Phase Static	Extended Two-Phase Dynamic
I	(p+1:4096:$nprocs$, p+1:4096:$nprocs$)	53.28	22.77	22.77
II	(1+250p:250+250p:2, 1+250p:250+250p:2)	25.22	6.438	3.775
III	(1+200p:500+200p:3, 1+200p:500+200p:3)	44.64	8.696	7.516
IV	(1+64p:64+64p:2, 500:2500:3)	71.35	8.858	7.279
V	(500:2500:3, 1+64p:64+64p:2)	79.24	7.724	4.405

5.6 Scalability

We have also studied the scalability of the Extended Two-Phase Method for a large number of processors, large array sections, and large out-of-core arrays. Since dynamic partitioning always performs better than, or at least as well as static partitioning, we consider only dynamic partitioning for the scalability experiments. Table 7 shows the timings obtained by varying the number of processors requesting array sections from 4 to 128, for both reading and writing. We have selected a few sections in each category, i.e., common, overlapping, distinct, and also sections with non-unit strides. Note that each processor is requesting a section; thus, as the number of processors increases, the amount of I/O required also increases.

The main observation is that the Extended Two-Phase Method scales well with the number of processors. In many cases, the time taken increases only slightly as the number of processors is increased, which indicates that we are able to get higher I/O throughput by increasing the number of processors. For example, for the sections in case I, the time taken increases from 1.282 sec. to only 2.130 sec. when the number of processors is increased from 4 to 128. In some cases, such as case II, the time taken even decreases. The Direct Method performs quite poorly as the number of processors is increased, especially for sections in cases II, IV and VIII. The Extended Two-Phase Method also scales well for writing sections. For small number of processors, the Direct Method takes less time for writing than the Extended Two-Phase Method. This is because of the extra read required for every write in the Extended Two-Phase Method. However, for large number of processors ($\geq$ 16), the Extended Two-Phase Method performs better. For sections with non-unit strides, the Extended Two-Phase Method performs considerably better than the Direct Method. For the sections in case VIII, the stride depends on the number of processors, and so the total amount of I/O decreases as the number of processors is increased.

Table 7: Scalability of the Extended Two-Phase Method. The number of processors accessing sections is varied from 4 to 128. Array size 4K $\times$ 4K real numbers (single precision), time in seconds. DR = Direct Read, ETP = Extended Two-Phase Method with dynamic partitioning, DW = Direct Write.

I = (400:800:1, 400:800:1), Figure 5(III)

II = (1:16:1, 1:4096:1), Figure 5(V)

III = (400:800:1, 400+25p:800+25p:1), Figure 6(III)

IV = (1+8p:16+8p:1, 1:4096:1), Figure 6(VII)

V = (1+25p:16+25p:1, 1:4096:1), Figure 7(IV)

VI = (1+32p:32+32p:1, 1+24p:1024+24p:1), Figure 7(VI)

VII = (p+1:4096:$nprocs$, p+1:4096:$nprocs$)

VIII = (500:2500:3, 1+32p:32+32p:2)

READING COMMON SECTIONS												
Section	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP
I	2.620	1.282	3.184	1.040	4.421	1.056	8.734	1.169	16.28	1.436	32.64	2.130
II	12.16	4.315	13.95	3.099	19.65	3.241	32.96	2.647	60.11	3.432	116.7	3.219
READING OVERLAPPING SECTIONS												
Section	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP
III	3.079	1.748	5.208	1.699	6.850	1.991	13.61	2.798	24.98	3.801	47.95	4.602
IV	13.75	4.450	13.77	3.391	19.63	2.992	32.70	3.696	60.58	4.791	115.9	7.401
READING DISTINCT SECTIONS												
Section	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP
V	12.37	4.791	13.57	3.929	19.76	4.149	32.38	6.109	46.12	7.276	54.82	8.161
VI	3.704	1.893	2.396	1.585	4.125	1.638	7.806	2.418	19.77	2.970	26.23	4.110
WRITING DISTINCT SECTIONS												
Section	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
	DW	ETP	DW	ETP	DW	ETP	DW	ETP	DW	ETP	DW	ETP
V	3.129	7.900	6.971	6.861	12.45	8.554	27.52	12.74	37.70	18.52	52.41	24.74
VI	0.982	1.937	1.803	2.218	3.954	3.058	6.436	5.028	7.139	6.234	21.20	9.403
READING SECTIONS WITH NON-UNIT STRIDES												
Section	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP
VII	799.2	22.82	216.6	15.83	210.8	9.331	103.1	10.89	54.94	8.307	50.60	9.657
VIII	56.44	1.342	77.78	1.440	83.87	1.870	163.1	3.123	331.5	5.062	867.4	7.711
WRITING SECTIONS WITH NON-UNIT STRIDES												
Section	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
	DW	ETP	DW	ETP	DW	ETP	DW	ETP	DW	ETP	DW	ETP
VII	668.7	42.75	147.3	39.11	84.54	31.40	64.53	26.42	35.35	28.40	51.38	31.16
VIII	9.041	1.612	18.83	1.603	35.17	2.972	75.95	4.812	163.6	7.915	341.8	21.75

Table 8 shows the performance for accessing large sections of a large out-of-core array of size 16K $\times$ 16K single precision real numbers (file size 1Gbyte). Figure 8 shows approximately where these sections are located in the array. We consider common, overlapping and distinct sections for reading, and distinct sections for writing. The trend in the results is the same as in Table 7 for a 4K $\times$ 4K array, which confirms the scalability of the Extended Two-Phase Method and its superiority over the Direct Method. We observe that the Direct Method performs much worse for accessing large sections than for small sections, whereas the Extended Two-Phase Method performs consistently well for sections of any size. The relative performance of the two methods for reading and writing the sections in case VI of Table 8 is illustrated in Figures 9 and 10, respectively.

6 Advantages

The Extended Two-Phase Method with dynamic partitioning 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 interprocessor communication in the second phase. Since communication cost is orders of magnitude lower than I/O cost, the overhead of communication is negligible. In all our experiments, we found the time spent on communication to be less than 5% of the total time. The Extended Two-Phase Method combines many small requests of different processors into single larger requests, thus providing larger granularity of data transfer and lower latency time. Another advantage is that multiple accesses by different processors 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.

The Extended Two-Phase Method also provides a great deal of flexibility in dividing I/O among processors. This can be done by defining the file domains appropriately. We have described one way of doing this dynamically as a function of the access requests. This performs better than a static assignment, but it may be possible to do even better. For example, instead of dividing the bounding section in a column-block manner among the processors, it could be divided in a block-cyclic fashion, so that if the bounding section includes some unwanted columns, they are evenly distributed. Another approach is to divide I/O among processors so that the I/O requests of different processors go to different disks or I/O nodes. For this, we need to know on which disks the requested sections are located. This information is not provided by any of the parallel file systems at present, but we expect it to be available in future versions of the file systems. Furthermore, if the ratio of processors to disks on the machine is very high, it is possible to have

Table 8: Scalability of the Extended Two-Phase Method for large requests. Array size 16K $\times$ 16K real numbers (single precision), 1 Gbyte file. The number of processors accessing sections is varied from 4 to 128. DR = Direct Read, ETP = Extended Two-Phase Method with dynamic partitioning, DW = Direct Write. Time in seconds.

I = (5000:6000:1,5000:6000:1)

II = (1+100p:300+100p:1,4000:8000:1)

III = (1+100p:400+100p:1,2000+20p:2800+20p:1)

IV = (4000:8000:1,1+4p:8+4p:1)

V = (1+100p:100+100p:1, 1+100p:1024+100p:1)

VI = (1+20p:16+20p:1,4000:12000:1)

READING SECTIONS												
Sec-	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
tion	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP
I	23.65	7.880	43.43	7.795	78.99	7.935	151.3	9.085	302.7	9.368	605.1	11.86
II	53.30	26.51	103.3	28.10	132.3	28.50	157.6	32.49	162.3	40.03	182.4	52.08
III	13.31	5.061	24.11	6.489	31.49	7.400	39.81	9.253	41.28	10.12	44.29	13.23
IV	0.683	0.699	0.841	0.939	1.343	1.173	2.189	1.663	4.149	2.850	8.486	4.994
V	10.97	5.380	19.31	8.475	26.52	10.58	35.06	12.69	52.81	14.10	124.2	22.06
VI	57.29	21.94	74.05	23.05	127.3	32.88	240.8	51.26	500.2	112.2	799.7	98.68
WRITING SECTIONS												
Sec-	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
tion	DW	ETP	DW	ETP	DW	ETP	DW	ETP	DW	ETP	DW	ETP
V	7.108	12.01	15.21	18.98	32.20	23.37	35.99	30.17	53.10	35.76	98.90	32.54
VI	48.35	44.18	71.85	52.07	151.4	73.34	272.8	122.3	548.1	174.1	746.6	164.2

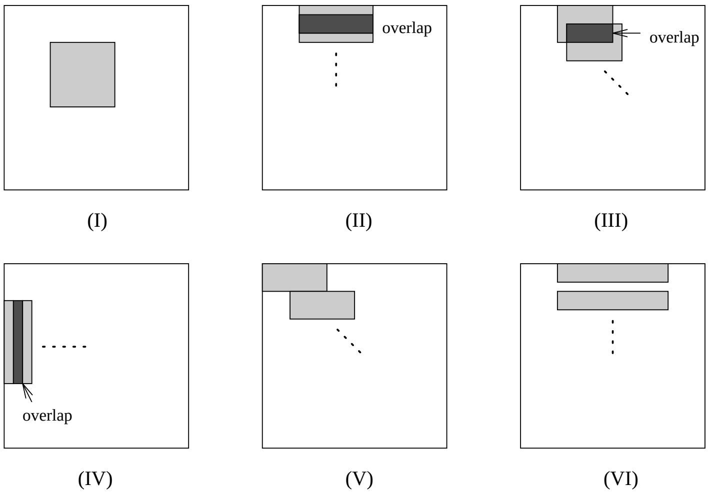

Figure 8: The sections listed in Table 8 (not to scale)

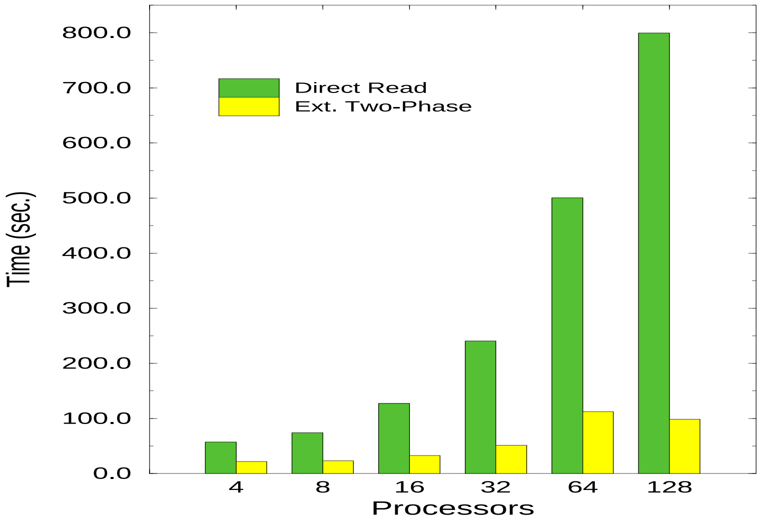

Figure 9: Scalability results, 16K × 16K array, time for reading sections in case VI of Table 8

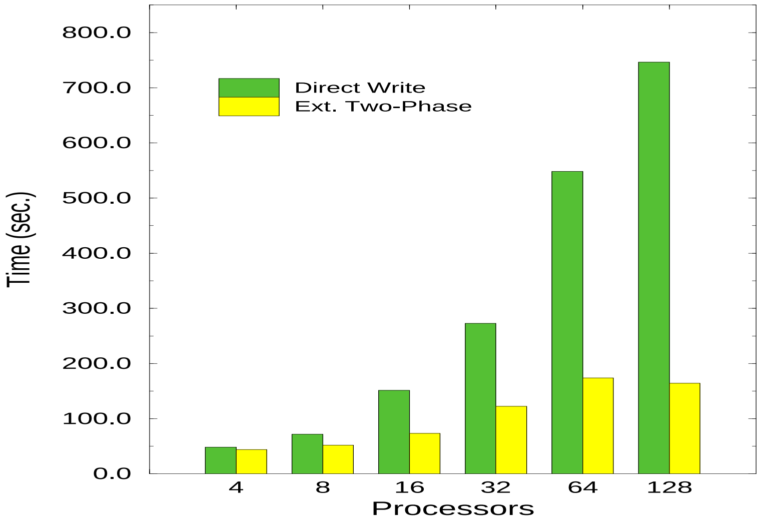

Figure 10: Scalability results, 16K × 16K array, time for writing sections in case VI of Table 8

only a few processors perform I/O, therefore reducing contention for disks.

The Extended Two-Phase Method can be used for accessing arrays with any number of dimensions and any storage order. For the dynamic partitioning scheme we have proposed, the file domains for an n-dimensional array can be obtained by first calculating the n-dimensional bounding section of all requests. This section is divided among processors such that the file domain of each processor is located contiguously in the file.

Array sections other than those which can be represented by a lower-bound, upper bound and stride in each dimension, for example sections with non-uniform strides, can also be accessed using the Extended Two-Phase Method. This requires a more general notation for representing such sections. The data structures such as the File Access Descriptor and the File Domain Access Table need to be modified to handle this, but the basic idea remains the same.

It is not necessary that all processors must call the Extended Two-Phase read/write routine. It is possible to define a group of processors involving only those processors that need to access data from the file. This is similar to process groups in MPI [14]. Only the processors in the group need to call the Extended Two-Phase routine and participate in the two-phase process. The I/O workload can be divided among the processors in this group.

The Extended Two-Phase Method is not specific to any particular machine, file system, or architecture; it can be easily implemented on top of any of the existing file system interfaces or portable interfaces such as the proposed MPI-IO interface [6], resulting in portable implementations. It can also be easily modified and tuned for any particular system, which only requires defining the file domains appropriately, and possibly using a different algorithm for interprocessor communication.

7 Conclusions

The Extended Two-Phase Method clearly has several advantages over a direct method for accessing sections of out-of-core arrays. These advantages are confirmed by its superior performance for different access patterns, array sizes, and number of processors. We believe that with better support from parallel file systems, it is possible to improve the performance of the Extended Two-Phase Method even further. We are currently investigating several alternate schemes for partitioning I/O dynamically. The Extended Two-Phase Method has been implemented in the PASSION Runtime Library, which can be obtained from the URL `http://www.cat.syr.edu/passion.html`.

References

[1] R. Bordawekar, A. Choudhary, and J. del Rosario. An Experimental Performance Evaluation of Touchstone Delta Concurrent File System. In *Proceedings of the* 7^{th} *ACM International Conference on Supercomputing*, July 1993.

[2] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A Model and Compilation Strategy for Out-of-Core Data Parallel Programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, July 1995. To appear.

[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] P. Brezany, M. Gerndt, P. Mehrotra, and H. Zima. Concurrent File Operations in a High Performance Fortran. In *Proceedings of Supercomputing '92*, pages 230–238, November 1992.

[5] 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.

[6] 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.

[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] 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.

[9] 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.

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

[11] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Version 1.0, May 1993.

[12] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *High Performance Fortran Handbook.* The MIT Press, 1994.

[13] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994. Revised as Technical Report PCS–TR94–226, Dept. of Computer Science, Dartmouth College.

[14] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* Version 1.0, May 1994.

[15] Applications Working Group of the Scalable I/O Initiative. Preliminary Survey of I/O Intensive Applications. Scalable I/O Initiative Working Paper Number 1. On World-Wide Web at http://www.ccsf.caltech.edu/SIO/SIO.html, 1994.

[16] 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.

[17] 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.

[18] 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.

Availability

This technical report (CACR–103) is available on the World-Wide Web at `http://www.cacr.caltech.edu/techpubs/`. Titles and abstracts of all CACR and CCSF technical reports distributed by Caltech's Center for Advanced Computing Research are available at the above URL. For more information on the Scalable I/O Initiative and other CACR high-performance computing activities, please contact CACR Techpubs, Caltech, Mail Code 158-79, Pasadena, CA 91125, (818) 395-4116, or send email to: `techpubs@cacr.caltech.edu`.