Appeared (with minor editorial changes) in *Scientific Programming*, (5)4:301–317, Winter 1996.

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 that cannot fit in main memory. In such applications, data must be stored in files on disks and fetched into memory during program execution. Parallel programs with large out-of-core arrays stored in files must read/write smaller sections of the arrays from/to files. In this paper, we describe a method for accessing sections of out-of-core arrays efficiently. Our method, the *extended two-phase method*, uses collective I/O: Processors cooperate to combine several I/O requests into fewer larger granularity requests, reorder requests so that the file is accessed in proper sequence, and eliminate simultaneous I/O requests for the same data. In addition, the I/O workload is divided among processors dynamically, depending on the access requests. We present performance results obtained from two real out-of-core parallel applications—matrix multiplication and a Laplace's equation solver—and several synthetic access patterns, all on the Intel Touchstone Delta. These results indicate that the extended two-phase method significantly outperformed a direct (noncollective) method for accessing out-of-core array sections.

*This work was supported in part 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; by a National Science Foundation Young Investigator Award (CCR-9357840); and by a grant from Intel Scalable Systems Division. 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 being used increasingly to solve 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 must be stored in files on disks, as it is too large to fit in main memory [8]. The program must perform I/O to access data from disks. Examples of such applications are Hartree-Fock calculations in chemistry, very large Fast Fourier Transforms to detect faint radio pulsars, seismic data processing, weather and climate modeling, 3D turbulence simulations, scattering and radiation problems in computational electromagnetics, and several others [1].

Multidimensional arrays are widely used as data structures in scientific programs. Scientific applications with large out-of-core data sets may therefore have one or more out-of-core multidimensional arrays stored in files. At run time, the program must fetch smaller sections of these arrays from files, perform computation, and, if necessary, store the results back to files. Different processors may need different sections of the arrays depending on the data distribution, and the sections may have strides in each dimension.

In this paper, we describe a method, called the *extended two-phase method*, for parallel programs to access sections of out-of-core arrays efficiently. In this method, the requesting processors cooperate in reading or writing data—a process known as collective I/O. Specifically, processors cooperate to combine several I/O requests into fewer larger granularity requests, reorder requests so that the file is accessed in proper sequence, and eliminate simultaneous I/O requests for the same data. In addition, the extended two-phase method partitions the total I/O workload among processors dynamically, depending on the access requests. Compared to a static partitioning scheme, dynamic partitioning results in a more balanced distribution of I/O among processors and therefore performs considerably better.

We present extensive performance results comparing the extended two-phase method with a direct (non-collective) method on the Intel Touchstone Delta. For this purpose, we use two real parallel applications—out-of-core matrix multiplication and out-of-core Laplace's equation solver—as well as several synthetic access patterns. We found that the extended two-phase method performed considerably better than the direct method for a wide range of access patterns, array sizes, and number of processors.

The rest of this paper is organized as follows. In Section 2, we describe the I/O access patterns of two out-of-core parallel applications and thus motivate the need for the extended two-phase method. The method itself is explained in Section 3. In Section 4, we describe a simple 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. We draw overall conclusions in Section 6.

2 Two Out-of-Core Parallel Applications

Here we describe the I/O access patterns of two out-of-core parallel applications—matrix multiplication and a Laplace's equation solver.

2.1 Out-of-Core Matrix Multiplication

We consider an out-of-core GAXPY algorithm for matrix multiplication, described in [3]. Let A, B, and C be $n \times n$ matrices such that $C = A \times B$. The matrices can be represented in terms of their individual columns as

$$A = [a_1, \cdots, a_n],\ a_j \in \mathcal{R}^n$$
$$B = [b_1, \cdots, b_n],\ b_j \in \mathcal{R}^n$$
$$C = [c_1, \cdots, c_n],\ c_j \in \mathcal{R}^n$$

The GAXPY algorithm for computing $C = A \times B$ is

$$c_j = \textstyle\sum_{k=1}^{n} b_{kj} a_k, \qquad j = 1 : n$$

In other words, to compute the j^{th} column of C, we need the j^{th} column of B and all columns of A. An out-of-core GAXPY algorithm for matrix multiplication can be implemented as follows. In the first step, processors read two-dimensional sub-blocks of matrix A into main memory such that the sub-blocks of all processors together span entire rows (see Figure 1). The processors also read two-dimensional sub-blocks of matrix B into memory such that the sub-blocks of all processors together span entire columns. The data now present in memory is sufficient to compute the first two-dimensional sub-block of matrix C. This computation requires a global sum operation. The processors then write the newly computed sub-block of C to the file. In the following step, processors read the next set of sub-blocks of B (shown by dashed lines in Figure 1), reuse the sub-blocks of A fetched in the previous step, and calculate the second sub-block of C. This process is repeated until all the sub-blocks in the first block of rows of C are computed. The above process is then repeated with the sub-blocks from the next set of rows of A, shown by dashed lines. The entire matrix C is computed in this fashion. Note that, at any time, each processor has only one sub-block of matrices A, B, and C in memory.

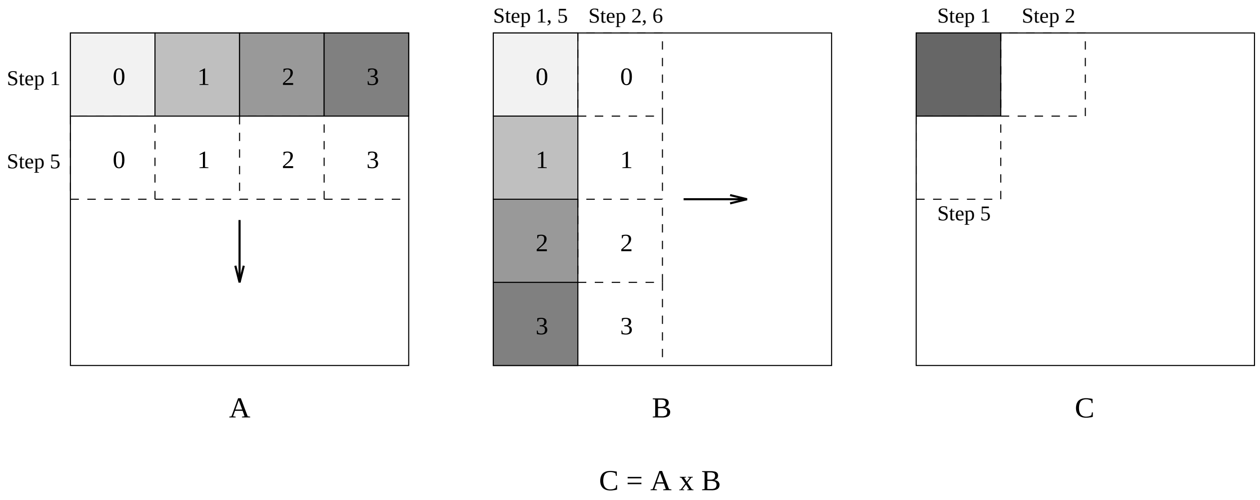

Figure 1: I/O access pattern in out-of-core matrix multiplication

2.2 Out-of-Core Laplace's Equation Solver

We consider a Laplace's equation solver that uses a Jacobi iteration method. This is a stencil computation where the value at each point is computed by using the values at its neighbors in each of the four directions.

do $k = 1, niter$
    $A(i,j) = (B(i-1,j) + B(i+1,j) + B(i,j-1) + B(i,j+1))/4, \qquad i,j = 1:n$
    Exchange A and B
end do

An out-of-core Laplace's equation solver can be implemented as follows. Divide the out-of-core array into two-dimensional sub-blocks such that two blocks (one for old values, one for new values) can fit at a time in the memory of each processor. Assign blocks to processors in a round-robin fashion as shown in Figure 2. Each processor reads one block at a time from the file containing the array. Processors can either communicate boundary rows and columns or read them directly from the file. After a processor computes new values, it writes the new block to a file containing the new array. This process is repeated on other sub-blocks of the array to complete one iteration. The algorithm is repeated for further iterations until it converges.

2.3 Accessing Out-of-Core Array Sections

In the above applications, processors access two-dimensional sub-blocks of out-of-core arrays. This type of access pattern also occurs in other applications, such as out-of-core LU solvers [10]. Since

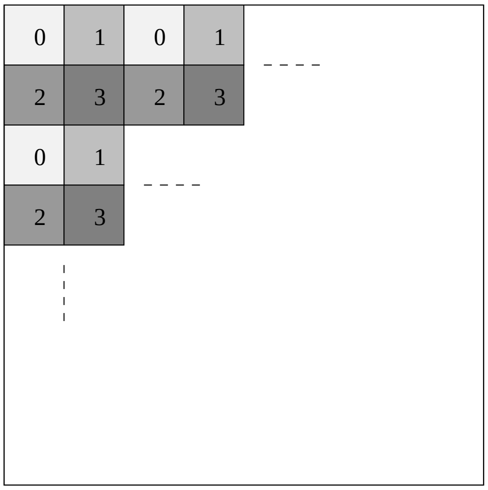

Figure 2: I/O access pattern in an out-of-core Laplace's equation solver

arrays are usually stored in a file in either column-major order (as in Fortran) or row-major order (as in C), the data required by each processor is not located contiguously in the file. In many cases, the requests of different processors are interleaved in the file. To read non-contiguous data with the interfaces currently provided by parallel file systems, each processor must explicitly seek to the appropriate location in the file, read a small chunk of data, then seek to the next location, and so on. We call this the *direct method*. The Vesta and PIOFS file systems on the IBM SP [5, 9] and the nCUBE file system [6] 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. Each processor's request, however, is serviced independently, and the file systems do not perform collective I/O.

The drawback of the direct method is that the parallel file system may receive a large number of low-granularity requests from multiple processors in any order. As I/O latency is very high, such access requests perform poorly. For many access patterns, such as in the above applications, the I/O performance can be improved by using the collective knowledge of the access requests of all processors. Processors can cooperate among themselves to perform I/O in large chunks and in the proper order, a process known as collective I/O. The extended two-phase method specifies a procedure for performing collective I/O to access out-of-core array sections. Other examples of collective I/O are disk-directed I/O [11] and server-directed collective I/O [12].

3 Extended Two-Phase Method

The two-phase method, proposed in [7, 4], 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 perform well for all array distributions [7, 4].

We have extended the basic two-phase method to access *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 the file is accessed in proper sequence, and
- eliminating simultaneous I/O requests for the same data.

3.1 Reading Sections of Out-of-Core Arrays

We first describe the extended two-phase method for reading array sections. For the purpose of explanation, we consider the case where each processor must read a section (specified in terms of a lower-bound, upper-bound, and stride in each dimension) of a two-dimensional array stored in a file in column-major order. In general, the extended two-phase method can be used for arrays with any number of dimensions, stored in any order in the file, and accessed by a subset of the total number of processors.

The extended two-phase method divides the I/O workload among processors by assigning ownership to portions of the file. A processor can directly access only the portion of the file it owns, called its *file domain*. For a file stored in column-major order, the file domain of each processor is some set of columns of the array. Section 4 describes two ways of assigning file domains to processors.

Assume that each processor must read a section $(l_1 : u_1 : s_1, l_2 : u_2 : s_2)$ of the out-of-core array, in global coordinates. The sections required by different processors may be identical, overlapping, or distinct. 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

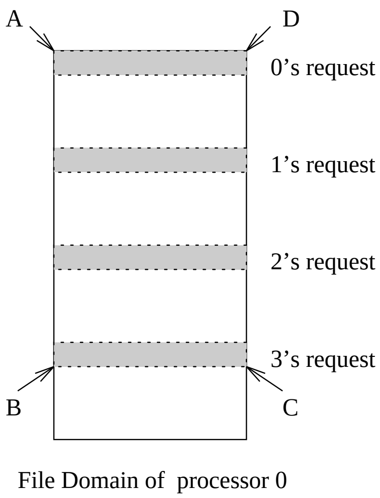

Figure 3: Processor 0 must read the requested data from its file domain. Section ABCD is the smallest section containing all the requested data. Processor 0 reads this section by using an optimization called *data sieving*.

knows the access requests of other processors. This information is stored in a data 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 the access requests of other 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 its own file domain. This information is stored in a data structure called the *file domain access table* (FDAT). The FDAT of a processor thus contains information indicating which portions of its file domain have been requested by other processors.

Each processor must now read data from its file domain as specified by the FDAT. For example, Figure 3 shows the file domain of processor 0 and, for some access pattern, the portions of this file domain that have been requested by other processors. A simple way of reading is to read all the data needed by processor 0, followed by that needed by processor 1, and so on, in order of processor number. This method, however, may result in too many small accesses that are not in sequence. For reading the data efficiently, processors must analyze the FDAT and use a read strategy that accesses the file in sequence and contiguously.

We use the following general method for this purpose. 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 containing all the data that must be read from the file domain (for example, section ABCD in Figure 3). This section may also contain some data that 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 we proposed previously, called *data sieving* [14, 13]. The processor reads a column (for column-major order) 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. The processor may read more than one column at a time, if sufficient memory is 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. From the information in the FDAT, each processor determines what data must be sent to which processor. In addition, since each processor knows the file domains of other processors and its own access request, it can calculate how much data to receive from other processors and where to store it in memory.

The two phases of the extended two-phase method either can be done distinctly by performing all I/O first and then 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 FAD, each processor determines what portions of its write request are located in the file domains of other processors; those portions must be sent to the respective processors. From the FDAT, each processor determines what portions of the write requests of other processors are located in its own file domain; those portions must be received from the respective processors. This communication forms the first phase of the extended two-phase method for writing sections.

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 calculates the minimum and maximum of all indices in its FDAT, which determines the smallest section containing all the data to be written to the file domain. The processor uses data sieving [14, 13] to write the useful data in this section. Note that, since there may be "holes" between the useful data to be written, an extra read operation is required before writing. This extra read is not required if the useful data is located contiguously in the file.

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 the particular implementation of the extended two-phase method. 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 the I/O Workload

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, I/O can be partitioned either statically or dynamically. Note that we are referring to a logical partitioning of the file among processors; the file is not physically repartitioned into separate files.

4.1 Static Partitioning

One way of partitioning I/O (for an array stored in column-major order) 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. The file domain of each processor is therefore a block of columns of the array, stored contiguously in the file. The size of each file 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 scheme. Figure 4(A) shows the file domains of four processors, with static partitioning of I/O.

4.2 Dynamic Partitioning

The main drawback of static partitioning is that the partitioning is independent of the access requests. For many access patterns, static partitioning may result in an imbalance of I/O among processors; some processors may perform more I/O than others, some may not perform any I/O at all. For example, consider the access pattern in Figure 4. With static partitioning, the access requests span the file domains of only two processors (1 and 2); therefore, only two processors perform all the I/O. In addition, if we increase the size of the out-of-core array, keeping the number of processors fixed, the size of each file domain also increases, and the access requests span the file domains of fewer processors, resulting in greater I/O imbalance.

A dynamic partitioning scheme, based on access requests, can divide the I/O workload more evenly and therefore improve I/O throughput. Figure 4(B) illustrates such a partitioning scheme. For a file stored in column-major order, each processor calculates the first and last 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 is located contiguously in the file. Figure 4(B) shows

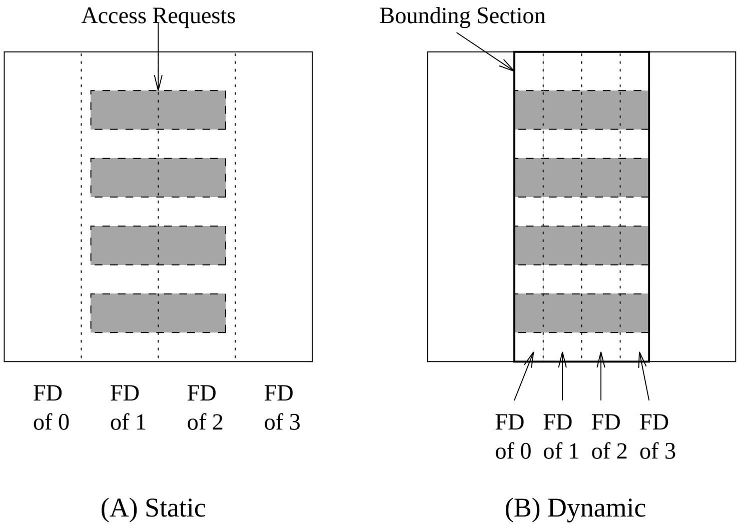

Figure 4: Static versus dynamic partitioning; FD = file domain

the bounding section for the given access requests. File domains are determined by dividing the bounding section among processors in a column-block fashion. The file domain of each processor is thus a contiguous chunk of the bounding section.

If the requested sections span all the columns of the out-of-core array, the dynamically selected file domains are identical to those determined statically. If the requested sections span only a few columns, however, dynamic partitioning provides a much better balance of I/O among processors (as Figure 4 shows). It also reduces the memory requirements of the extended two-phase method, because the file domain of each processor is smaller. With static partitioning, if all requested sections are located in a single processor's file domain, all the requested data may not fit in the memory of that processor. Consequently, I/O and communication may need to be done in stages, several times. This situation is less likely to occur with dynamic partitioning, because the requested data is more evenly divided among processors.

For an array stored in row-major order, file domains are determined as follows. Each processor calculates the first and last among the rows of the sections requested by all processors. The bounding section is the section formed by these rows and all the columns of the out-of-core array. File domains are determined by dividing the bounding section among processors in a row-block fashion.

Figure 5 summarizes the extended two-phase method for reading sections of out-of-core arrays, with dynamic partitioning of I/O.

1. Exchange access information with other processors and fill in the *file access descriptor* (FAD).

2. Calculate the smallest section, called the *bounding section*, that includes the sections requested by all processors.

3. Determine the *file domain* of each processor by dividing this bounding section among processors in a column-block manner for arrays stored in column-major order or row-block manner for arrays stored in row-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 by using *data sieving*, and communicate the data to the requesting processors.

Figure 5: Extended two-phase method for reading sections of out-of-core arrays with dynamic partitioning of I/O

5 Performance

We used the Intel Touchstone Delta 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. Intel's Concurrent File System (CFS) provides parallel access to files. By default, CFS stripes files across all 64 disks in 4-Kbyte blocks. See [2] for a detailed discussion of the performance of CFS.

We studied the performance of the extended two-phase method versus the direct method extensively for several synthetic access patterns as well as for two real out-of-core parallel applications—matrix multiplication and a Laplace's equation solver. We report the results of these experiments below.

5.1 Synthetic Access Patterns

We used three basic types of synthetic access patterns:

1. *Common sections*: All processors access 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.

5.1.1 Reading Common Sections

Table 1 shows the performance of the direct and extended two-phase methods for reading common sections (4K $\times$ 4K array, 16 processors). Figure 6 illustrates the approximate location of each of these sections in the array. We measured the performance of the extended two-phase method with both static and dynamic partitioning. In all cases, the extended two-phase method performed considerably better than the direct method, because it read the common section only once and broadcast it to other processors. In the direct method, on the other hand, all processors read the same section from the file simultaneously, resulting in extra I/O overhead.

In all cases, the extended two-phase method took much less time with dynamic partitioning. With static partitioning, each processor's file domain was of size 4K $\times$ 256. Therefore, all sections, except those in case V, were located in the file domains of only a few processors. With dynamic partitioning, on the other hand, the I/O requests were evenly divided among all available processors, resulting in higher I/O throughput. Since the section in case V spanned all 4096 columns, the statically and dynamically selected file domains were identical, and so was the performance. For case V, the extended two-phase method performed considerably better than the direct method, because the direct method resulted in a large number of small requests spread across the entire file.

5.1.2 Reading Overlapping Sections

Table 2 shows the time taken for reading various overlapping sections. Figure 7 illustrates the approximate location of each of these sections 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 case I of Table 2 represents a group of

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

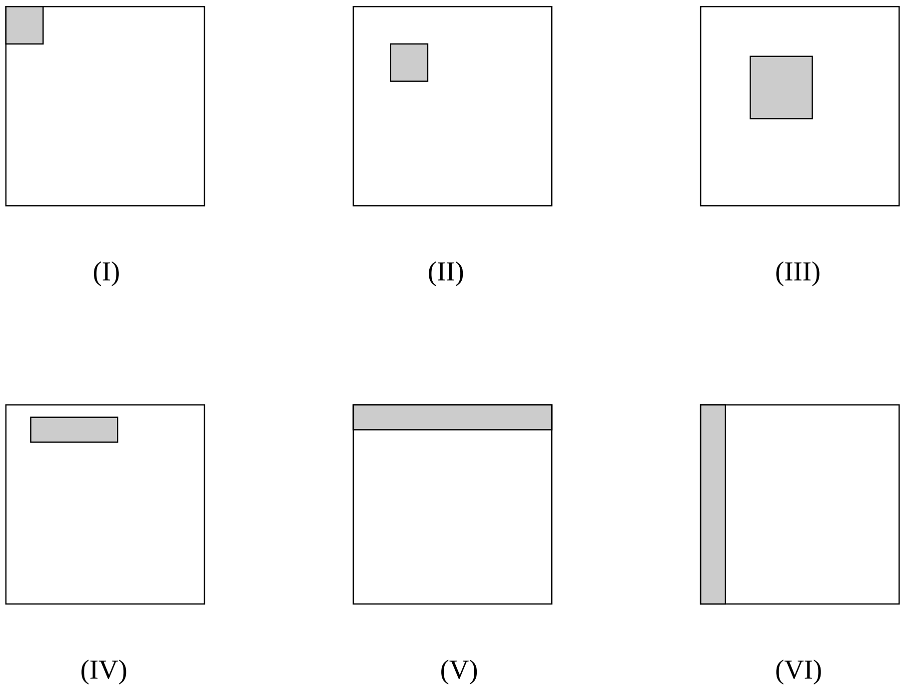

Figure 6: The common sections listed in Table 1 (not to scale)

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 extended two-phase method with dynamic partitioning performed the best in all cases. The sections in cases I and II were of the same size, but they differed in the amount of overlap; the sections in case I had 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 was less than that by the sections in case II, it took less time to read the sections in case I. The sections in cases IV, V, and VI spanned only a few columns. For these cases, the direct method performed better than the extended two-phase method with static partitioning, because static partitioning resulted in only a few processors performing I/O. The extended two-phase method with dynamic partitioning, however, performed better than the direct method, since the I/O workload was better distributed. The worst case for the direct method was case VII, which spanned all columns of the array. The sections in case VIII were overlapping in both dimensions, and again the extended two-phase method with dynamic partitioning took the least time.

5.1.3 Reading Distinct Sections

Table 3 shows the time taken for reading distinct sections. Figure 8 illustrates the approximate location of these sections 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 $ov2$ must be large enough to ensure that the sections are distinct.

In case I, the requests of different processors were situated in separate locations in the file, because the sections requested were located along rows. As a result, I/O in the extended two-phase method with dynamic partitioning was identical to that in the direct method, and they took the same time. The extended two-phase method with static partitioning took longer than the direct method, because only a few processors performed I/O. The sections in cases II—IV were located along columns, and the requests of different processors were interleaved in the file. The extended two-phase method therefore performed considerably better for these cases. Static partitioning did not perform well for the sections in case II, because they spanned only a few columns. The best case for the extended two-phase method was case IV, since the sections spanned all columns. The sections in cases V and VI were partly interleaved in the file, and even for these cases, the extended two-phase method performed the best.

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

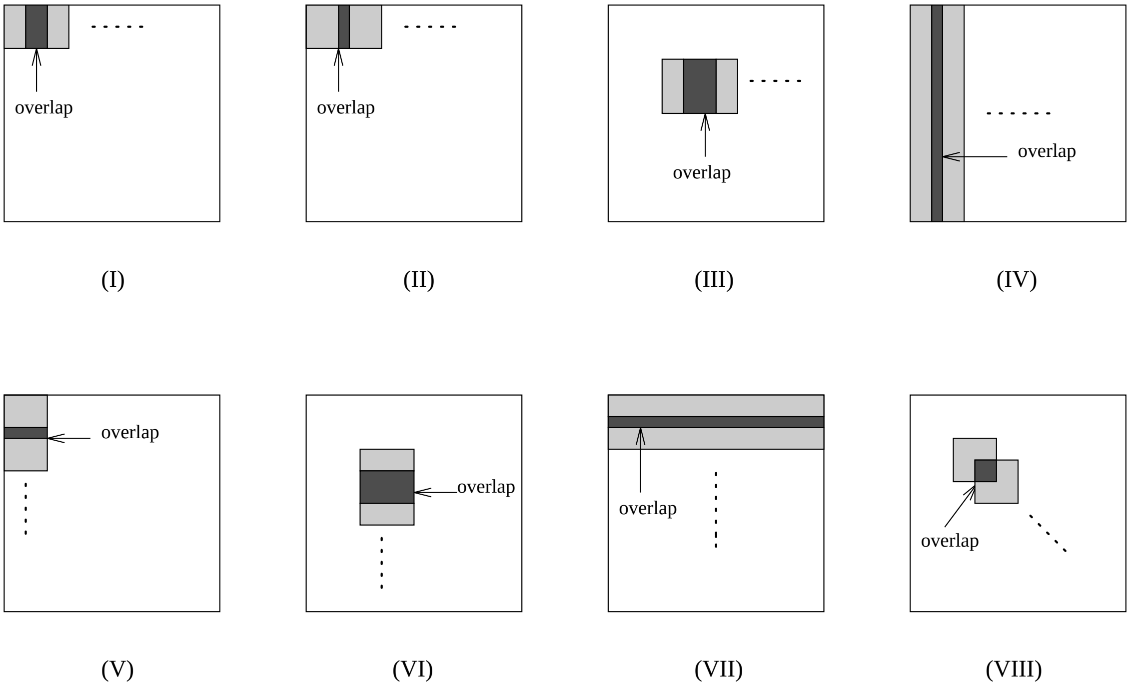

Figure 7: The overlapping sections listed in Table 2 (not to scale)

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	2.254	1.976
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

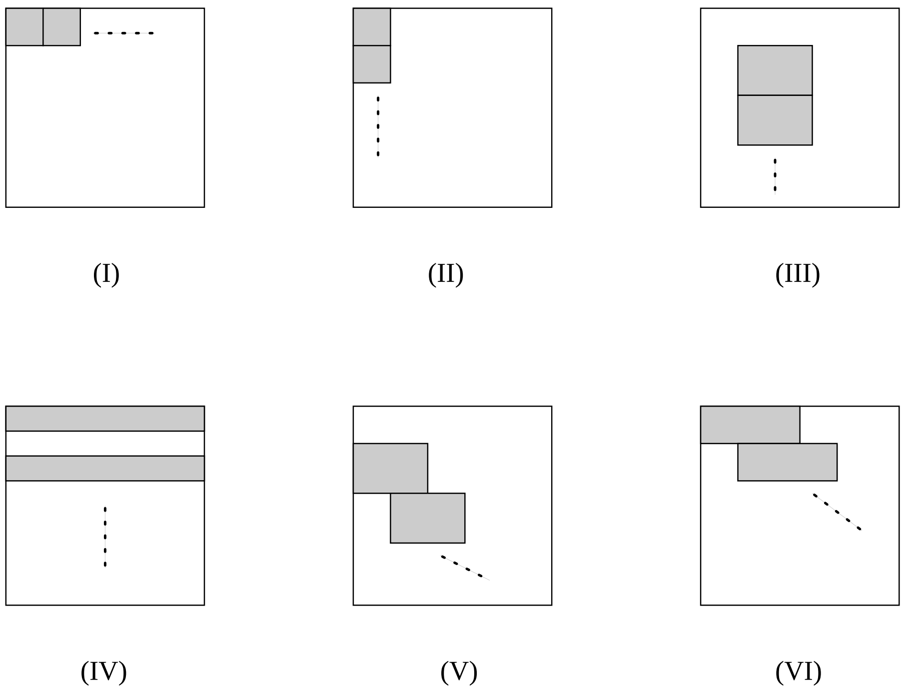

Figure 8: The distinct sections listed in Table 3 (not to scale)

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	2.166	1.944
II	(1+100p:100+100p:1, 1:100:1)	1.182	2.034	0.494
III	(200+200p:400+200p:1, 512:1024:1)	4.202	5.445	1.669
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

5.1.4 Writing Distinct Sections

We considered only the case where each processor writes a distinct section to the file, because other cases, such as writing overlapping or common sections, are unlikely to occur. Table 4 shows the time taken for writing distinct sections. The sections chosen were the same as those for reading (Table 3, Figure 8). As for reading distinct sections, the direct method and the extended two-phase method with dynamic partitioning took the same time for writing the sections in case I, whereas the extended two-phase method with static partitioning took longer. In the other cases, the extended two-phase method with dynamic partitioning performed considerably better than the direct method.

5.1.5 Accessing Sections with Non-Unit Strides

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

Table 5 shows the performance for reading sections with non-unit strides. The sections in case I spanned almost the entire array, with stride equal to the number of processors. As a result, static and dynamic partitioning took the same time. The sections in cases II and III were located diagonally across the out-of-core array. The sections in case IV were located along columns, and the sections in case V were located along rows. In all cases, the extended two-phase method was more than 20 times faster than the direct method. Table 6 shows the performance of the extended

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

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

two-phase method for writing sections with non-unit strides. The sections chosen were the same as in Table 5. Even for writing sections, the extended two-phase method improved I/O performance considerably.

5.1.6 Scalability

We also studied the scalability of the extended two-phase method for large number of processors, large array sections, and large out-of-core arrays. Since dynamic partitioning always performed better than, or at least as well as static partitioning, we considered 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 selected a few sections in each category—common, overlapping, distinct, and non-unit strides. Note that, as the number of processors was increased, the total amount of I/O performed also increased.

The extended two-phase method scaled well with the number of processors. In many cases,

Table 7: Scalability of the extended two-phase method. The number of processors accessing sections was 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 6(III)

II = (1:16:1, 1:4096:1), Figure 6(V)

III = (400:800:1, 400+25p:800+25p:1), Figure 7(III)

IV = (1+8p:16+8p:1, 1:4096:1), Figure 7(VII)

V = (1+25p:16+25p:1, 1:4096:1), Figure 8(IV)

VI = (1+32p:32+32p:1, 1+24p:1024+24p:1), Figure 8(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

the time taken increased only slightly as the number of processors was increased, indicating that we obtained higher I/O throughput by increasing the number of processors. For example, for the sections in case I, the time taken increased from 1.282 sec. to only 2.130 sec. when the number of processors was increased from 4 to 128. In some cases, such as case II, the time taken even decreased. The direct method performed quite poorly when the number of processors was increased, especially for cases II, IV, and VIII. The extended two-phase method also scaled well for writing sections. For small number of processors, the extended two-phase method took longer for writing, because of the extra read before each write. For large number of processors (≥ 16), however, the extended two-phase method performed better than the direct method in spite of the extra read. For sections with non-unit strides, the extended two-phase method performed considerably better than the direct method.

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 9 shows the approximate location of these sections in the array. We considered common, overlapping, and distinct sections for reading and distinct sections for writing. The trend in the results was the same as for a 4K $\times$ 4K array (Table 7). The direct method performed much worse for accessing large sections than for small sections, whereas the extended two-phase method performed consistently well for sections of any size. Figures 10 and 11 compare the relative performance of the two methods for reading and writing the sections in case VI of Table 8.

5.2 Real Applications

We also studied the performance of the extended two-phase method with dynamic partitioning versus the direct method, for two real out-of-core parallel applications—matrix multiplication and a Laplace's equation solver.

5.2.1 Matrix Multiplication

Table 9 shows the I/O time for out-of-core matrix multiplication for different array sizes and number of processors. The I/O time was calculated as the maximum of the time taken by all processors, for all I/O (reading and writing) required in the out-of-core matrix multiplication algorithm described in Section 2. Note that in the extended two-phase method, the I/O time includes the time for data communication. In all cases, the extended two-phase method performed better than the direct method. Figure 12 shows that the percentage improvement in I/O time provided by the extended two-phase method over the direct method varied from 22% to 75%.

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 was 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

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	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

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	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

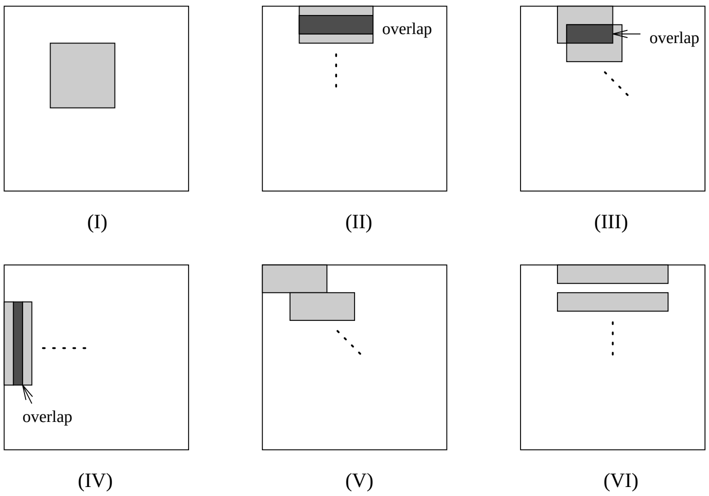

Figure 9: The sections listed in Table 8 (not to scale)

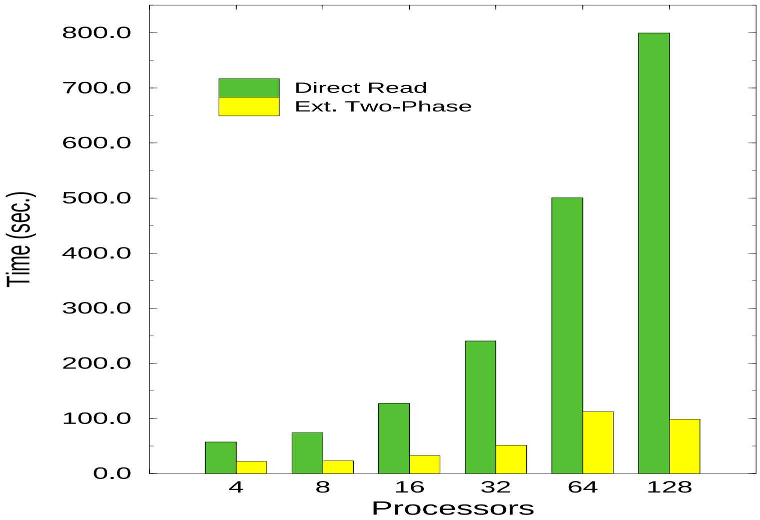

Figure 10: Scalability results, 16K $\times$ 16K array, time for reading sections in case VI of Table 8

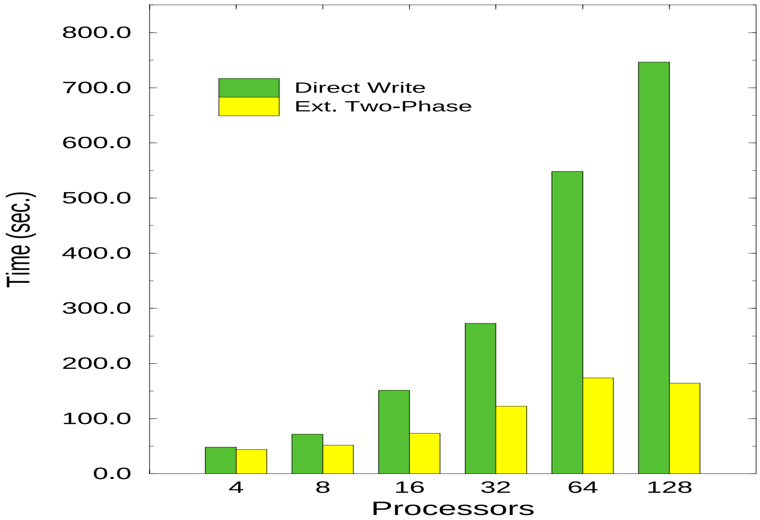

Figure 11: Scalability results, 16K $\times$ 16K array, time for writing sections in case VI of Table 8

Table 9: I/O time in seconds for out-of-core matrix multiplication using direct method and extended two-phase method with dynamic partitioning (ETP)

	1K × 1K array		2K × 2K array		4K × 4K array	
Procs.	Direct	ETP	Direct	ETP	Direct	ETP
8	44.65	34.77	103.4	80.43	589.0	416.8
16	39.88	24.78	94.37	69.87	465.9	326.8
32	37.80	18.88	108.6	76.36	536.4	354.5
64	50.65	17.66	168.8	122.8	814.2	501.1
128	161.0	24.76	377.1	218.1	1562	909.3

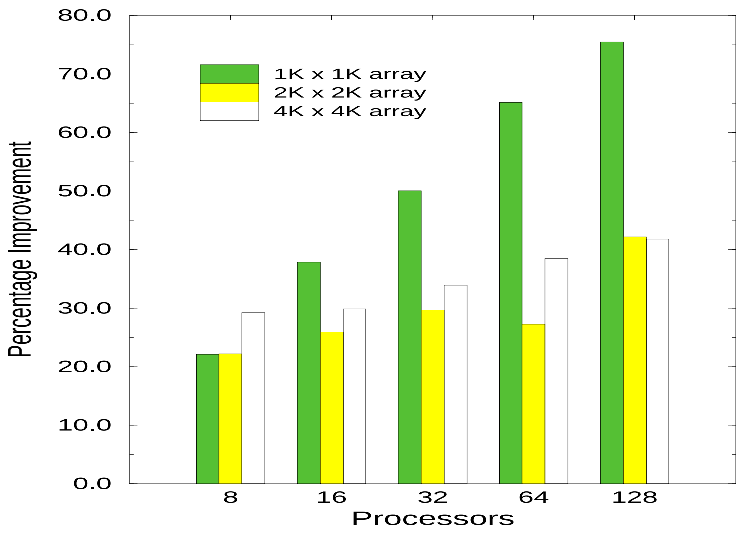

Figure 12: Percentage improvement in I/O time of out-of-core matrix multiplication by using extended two-phase method versus direct method

Table 10: I/O time in seconds for an out-of-core Laplace's equation solver using direct method and extended two-phase method with dynamic partitioning (ETP).

	1K × 1K array		2K × 2K array		4K × 4K array	
Procs.	Direct	ETP	Direct	ETP	Direct	ETP
8	27.15	25.03	72.34	68.00	387.1	356.7
16	17.06	15.27	61.96	54.65	434.0	294.3
32	18.59	13.29	50.27	43.63	448.3	273.3
64	19.20	14.80	49.15	42.06	383.6	280.0
128	31.40	18.16	64.67	53.10	508.5	334.4

5.2.2 Laplace's Equation Solver

Table 10 shows the I/O time for an out-of-core Laplace's equation solver for different array sizes and number of processors. The I/O time is the maximum of the time taken by all processors for all I/O (reading and writing) required in the out-of-core Laplace's equation solver algorithm described in Section 2. As in the case of matrix multiplication, the extended two-phase method performed better than the direct method. The percentage improvement in I/O time provided by the extended two-phase method over the direct method is shown in Figure 13. The percentage improvement was lower than in the case of matrix multiplication, possibly because of the difference in the I/O access patterns of the two applications. Recall that in out-of-core matrix multiplication, matrix B is accessed in blocks along columns. The results with synthetic access patterns in Section 5.1 indicate that the extended two-phase method performs very well for such accesses.

6 Conclusions

The extended two-phase method is clearly superior to a direct method for accessing sections of out-of-core arrays. In our experiments with real applications as well as several synthetic access patterns, the extended two-phase method outperformed the direct method significantly.

The extended two-phase method also provides much flexibility in partitioning the I/O workload among processors. We have described one dynamic partitioning scheme that performed significantly better than a static partitioning scheme, but it may be possible to do even better. For example, instead of dividing the bounding section among processors in a column-block fashion, 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 in such a way that the I/O requests from different processors go to different disks or I/O nodes. Furthermore,

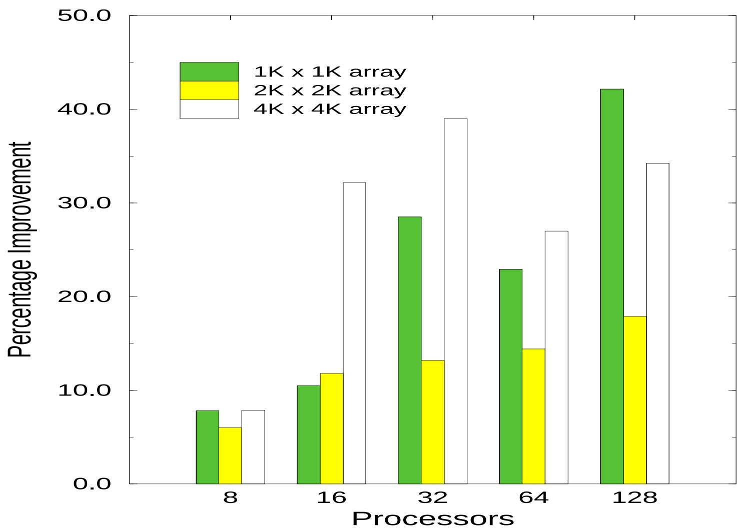

Figure 13: Percentage improvement in I/O time of out-of-core Laplace's equation solver by using extended two-phase method versus direct method

if the ratio of processors to disks on the machine is very high, it is possible to have only a few processors perform I/O, thereby reducing contention for the I/O system.

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, and then dividing it among processors such that the file domain of each processor is located contiguously in the file.

Array sections other than those that 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 by using the extended two-phase method. This requires a more general notation for representing such sections. The data structures, such as FAD and FDAT, must be modified to handle such requests, but the basic idea remains the same.

It is not necessary that all processors running the application must call the extended two-phase read/write routine. Even a subset of processors may call the routine and participate in the two-phase process. The I/O workload can be divided among the processors in this subset.

The extended two-phase method is not specific to any particular machine, file system, or architecture; it can be easily implemented by using any file-system interface, or by using portable interfaces, such as MPI-IO [16], resulting in portable implementations. It can also be easily modified and tuned for any particular system—by defining file domains appropriately and possibly using a different algorithm for interprocessor communication.

The best way to use the extended two-phase method is to implement it as a library routine that can be called from an application program. We have implemented it in the PASSION runtime library [15], which is available on the World-Wide Web at `http://www.cat.syr.edu/passion.html`.

References

[1] 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 the World-Wide Web at `http://www.ccsf.caltech.edu/SIO/SIO_apps.ps`, 1994.

[2] R. Bordawekar, A. Choudhary, and J. del Rosario. An Experimental Performance Evaluation of Touchstone Delta Concurrent File System. In *Proceedings of the 7th ACM International Conference on Supercomputing*, pages 367–376, July 1993.

[3] R. Bordawekar, A. Choudhary, and R. Thakur. Data Access Reorganizations in Compiling Out-of-Core Data Parallel Programs on Distributed Memory Machines. Technical Report SCCS–622, NPAC, Syracuse University, September 1994. On the World-Wide Web at `ftp://erc.cat.syr.edu/ece/choudhary/PASSION/access_reorg.ps.Z`.

[4] 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.

[5] 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.

[6] 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.

[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*, pages 56–70, April 1993.

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

[9] IBM Corp. IBM AIX Parallel I/O File System: Installation, Administration, and Use. Document Number SH34-6065-01, August 1995.

[10] K. Klimkowski and R. van de Geijn. Anatomy of an Out-of-Core Dense Linear Solver. In *Proceedings of the 1995 International Conference on Parallel Processing*, pages III–29—III–33, August 1995.

[11] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994. Updated as Technical Report PCS–TR94–226, Dept. of Computer Science, Dartmouth College.

[12] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing '95*, December 1995.

[13] 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.

[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*, pages 119–128, October 1994.

[15] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. Passion: Optimized I/O for Parallel Applications. *IEEE Computer*, 29(6):70–78, June 1996.

[16] The MPI-IO Committee. MPI-IO: A Parallel File I/O Interface for MPI, Version 0.5. World-Wide Web `http://lovelace.nas.nasa.gov/MPI-IO`, April 1996.