High-Performance External Computations Using User-Controllable I/O

Jang Sun Lee
Database Section
ETRI
Daejon, Korea

Sunghoon Ko
Dept. of EECS
Syracuse University
Syracuse, NY

Sanjay Ranka
Dept. of CISE
University of Florida
Gainesville, FL

Byung Eui Min
AI Section
ETRI
Daejon, Korea

sunny@computer.etri.re.kr shko@top.cis.syr.edu
ranka@cise.ufl.edu bemin@computer.etri.re.kr

Abstract

The UPIO (User-controllable Parallel I/O) we proposed in [6] extends the abstraction of a linear file model into an n-dimensional file model, making it possible to control the layout of data blocks across disks and aggregating disk bandwidth through UPIO's interfaces. This enables users to produce high-performance external computation codes by planning I/O, computations, communication, and data reuse effectively in the codes. In this paper we show how well UPIO produces high-performance external computation codes by designing an I/O and communication-efficient external Laplace equation solver algorithm and exploring the effects of UPIO with the codes.

1. Introduction

We believe that users are better able than systems to reduce I/O time through explicit I/O management for the parallel computing environments. Thus, to achieve high performance, file systems or I/O mechanisms for parallel machines should be user-controllable in the sense that users should be able by considering the access patterns to determine the file structure, control the distribution of data blocks on physical disks, and easily present a variety of access patterns with a minimum number of I/O requests. However, the majority of current file systems for parallel computers are based on a linear file model and do not support user-controllable features [1, 7, 9].

The Vesta parallel file system [3] supports a two-dimensional file abstraction that provides a flexible way of data partitioning for parallel access, but it does not support the partitioning multidimensional data structures. The Galley [8] file system allows users to control declustering data across disks by defining *subfiles*. Each subfile resides on a single disk, and it is more structured by a collection of one or more independent *forks*. Each fork of a subfile is similar to a traditional Unix file. In Galley, however, there is no notion of file-level operations. Each operation is applied to only a single fork of a subfile.

The UPIO we proposed in [6] efficiently supports user controllability; an important part of UPIO is "logical disk," which can be defined by users based on data domain. Users have total control over the logical disk, making it possible to control the layout of data blocks across disks and aggregating disk bandwidth through UPIO's interfaces. This enables users to produce high-performance external computation codes by planning I/O, computations, communication, and data reuse effectively in the codes. In this paper we show how well UPIO produces high-performance external computation codes by designing an I/O and communication-efficient external Laplace equation solver algorithm and exploring the effects of UPIO with the codes.

The rest of this paper is organized as follows. In Section 2 we briefly introduce UPIO. I/O and communication-efficient Laplace equation solver algorithm using UPIO and the experimental results are presented in Sections 3 and 4. Finally, we make some concluding remarks.

2. User-Controllable Parallel I/O

2.1. Logical Disk and Data Access

We believe that users can represent the semantics of data domain on disks as it is by extending the abstraction of a one-dimensional file to an n-dimensional file. For example, if a two-dimensional matrix is in memory, application programs access an element of the matrix through the index of the element. This method can be applied to another two-dimensional matrix that is too huge to fit into the memory

all at one time. If we divide the matrix into submatrices that are small enough to fit into the memory, we can get another matrix of the submatrices and map each element of the newly created matrix into one of the submatrices.

If we define the newly created matrix and each element of the matrix as a *logical disk* (LD) and a *logical disk block* (LDB), respectively, we can have a two-dimensional LD. Users can access any submatrix through the index of the LD, because each element of the LD is mapped into one of the submatrices. Therefore, an LD and the index of the LD correspond to a file and its file pointer, respectively.

We can also define a one-dimensional array as an LD to map each element of the array with one of the submatrices. In other words, an LD is an n-dimensional array that is defined by users to map each element of the array into some data elements that correspond to an LDB, which is also defined by users. Hence, users can define a flexible file abstraction according to the needs of applications by defining the structure of an LD.

An LD is just an abstract block-index domain on which users control I/O explicitly. The LD does not allocate space for it, either in memory or on a physical disk. Users are therefore able to map the basic structure of computation into an LDB, represent the semantic of data domain by defining the shape of an LD, and control data distribution by distributing the abstract index across I/O or compute nodes.

In UPIO all accesses to the data on physical disks are made through LDs defined by users. The mapping between LDBs and physical disk blocks is done automatically by simple mapping algorithms when data is accessed through the LD [6]. Specifying the indices of LDBs is the only thing that should be done by users in order to access data on physical disks.

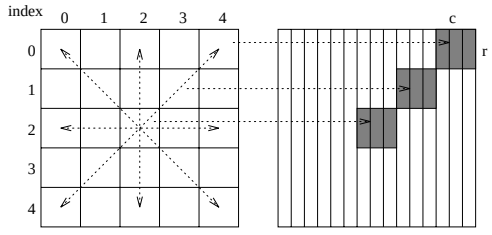

Figure 1. Example directions to read/write LDBs in a 2-dimensional LD

Suppose a two-dimensional matrix is mapped into a 5×5 LD as shown in Figure 1 and a user wants to access the data chunks represented as shaded rectangles in that figure. In UPIO users can access the data with one operation by identifying the LDBs that are mapped into the data chunks. All of the LDBs can be accessed at one time with the following statement:

LD_read(*ldd, mv_cnt, memv, cnt, news, index*)

This function reads three (= *cnt*) LDBs, beginning at the LDB *index*, (2, 2), toward the direction *news*, and stores data into the memory as specified by the memory vector *memv*, where *ldd* is a *logical disk descriptor*[1] [6].

The direction information, *news*, is represented by setting the values for the distance between LDBs in each dimension of an LD. Figure 1 shows some example directions that users can specify. The *memv* is a pointer to an array of (*mem*, *lcnt*); *lcnt* LDBs are stored beginning at *mem*; *mv_cnt* means the number of the array elements. More complex functions for presenting a variety of access patterns are described in [6].

2.2. Architecture and Storage Models for UPIO

Our architecture model assumes a MIMD multicomputer. In the model, rather than make specific assumptions about the underlying network, we assume that the cost for an access to a remote node is independent of the distance between the communicating nodes. Communication between nodes has a start-up overhead of τ, while the data transfer rate is $1/\mu$. The time taken to send a message from one node to another is modeled as $\tau + \mu m$, where m is the size of the message. For our complexity analysis we assume that τ and μ are constant, independent of the link congestion and distance between two nodes.

The UPIO operations can be applied to two different storage models for MIMD parallel machines: *local logical disk model* (LLDM) and *global logical disk model* (GLDM).[2] The models specify how the data is mapped into logical disk(s), how the LDBs are distributed across physical disks, and how the data is accessed through the LD by each compute node.

In LLDM, the (input) data is divided into local data belonging to each compute node. Each group of local data is mapped into a separate LD and stored in a separate file called *local file*, where each LD is the abstraction of each local file. In other words, each compute node has its own LD and local file. The compute node has sole control of this data and accesses the data through the LD. Any sharing with other compute nodes will be done through explicit message passing.

3. I/O and Communication Efficient Laplace Equation Solver

In the Jacobi iteration method the new values in each iteration are computed using the values from the previous iteration. We may represent the Jacobi update procedure

[1] This is similar to the file descriptor of the Unix file system.

[2] See [6] for GLDM.

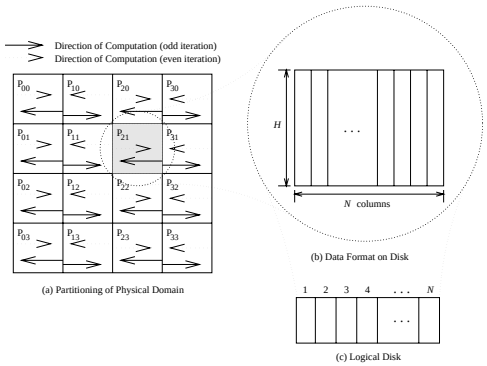

Figure 2. Data distribution for the Laplace equation solver

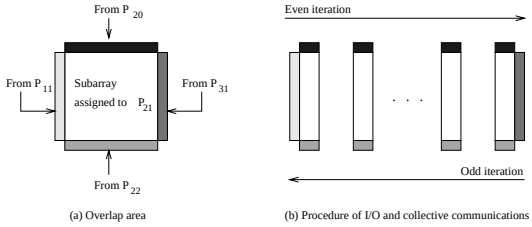

Figure 3. Procedure of I/O and collective communication with overlapped data

for any iteration using an array assignment statement as follows:

```
DO i = 2 : n-1
  DO j = 2 : n-1
    A(i, j) = 1/4(B(i, j-1) + B(i, j+1) + B(i+1, j) + B(i-1, j))
```

For an external computation let's assume that an $n \times n$ data array is decomposed on $P_X \times P_Y$ grid compute nodes in (BLOCK, BLOCK) manner as shown in Figure 2, and the subarray (say $H \times N = \frac{n}{P_Y} \times \frac{n}{P_X}$) assigned to each compute node cannot all be in the main memory at the same time due to the limits of the main memory and is stored in a local LD.[3]

We can define the LDB as block-wise or row-/column-wise for the Laplace equation solver, because four equidistant nearest neighbors are needed to calculate the new value of a point. A block-wise LDB can be defined as the basic unit of I/O, but in this case, in order to calculate the new values of the elements in an LDB, at least five LDBs are required.

However, if a column is defined as an LDB, reading only three LDBs is enough to calculate the new value of the data elements in one LDB.[4] Therefore, we may get a better performance in terms of I/O if we define an LDB as a column-wise block.

3.1. Algorithmic Modification

Let's suppose that Figure 2 (b) represents the data format assigned to P_{21}. Where N is the number of columns, H is the height of the data array assigned to each compute node.

[3]We use LLDM to extend the internal computation method to an external computation problem.

[4]We assumed that the main memory of each compute node can contain at least three LDBs.

Let's assume that each compute node has its own LD to store the data array and each column is defined as an LDB (see Figure 2 (c)).

Consider the subarray that is assigned to compute node P_{21}, shown in Figure 2. To compute the values in an LDB (say LDB l), LDBs $l-1$ and $l+1$ are required; to compute the values at the four boundaries of the subarray, P_{21} needs the values in compute nodes P_{20}, P_{11}, P_{31}, and P_{22} (see Figure 3).

For example, to compute the first LDB, compute node P_{21} requires some data elements from other compute nodes. If we assume that each compute node performs computations from the first LDB, in the SPMD programming model, the other compute nodes are also working on the first LDB. The bottom and top elements are in the memory of the bottom and top neighbor compute nodes, but LDB $N-1$, which will be computed in the last by compute node P_{11}, is not in memory. Thus compute node P_{11} needs to read the LDB and send it to compute node P_{21}. This is an extra I/O, because the LDB should be read again at the end of the iteration. At the end of the iteration, compute node P_{21} requires LDB 0 from compute node P_{31}, and thus P_{31} has to do an extra I/O to read the LDB and send it to P_{21}.

We can eliminate the extra I/Os by simply changing the direction of computation in every iteration. The arrows in Figure 2 represent the direction of computation. All compute nodes whose x coordinates are even numbers start computation from right (LDB $N-1$) to left (LDB 0), and other compute nodes whose x coordinates are odd numbers start computation from the opposite direction. The nodes contain the boundary values so that they can exchange boundary values before starting computation. In the next iteration, the direction of computation is reversed and will be reversed again after the iteration, and so on. Individual compute nodes do not have to write back the results on the disk at the end of each iteration, because the results are used in the next iteration.

While computing the data elements in LDB l, each compute node has to store the newly computed temporary results in the main memory and then write the results back

to the physical disk. Thus it seems that extra memory is required to store the temporary results shown in the code segments for the Laplace equation solver. However, if we use the memory space for LDB $l-1$ to store the computation results for LDB l and use the memory as a circular queue, no extra memory is required.

Therefore this pattern of computation requires no extra memory space and I/O operations caused by other compute nodes' requests to get boundary values, which optimizes the I/O operations.

3.2. Complexity

In this section we analyze the algorithm described above in terms of I/O and communications, because the complexity of computations is the same as that of an internal algorithm.

In the external Laplace equation solver each compute node reads $W-1 (\geq 2)$ LDBs at the beginning, exchanges H elements with the right or left neighbor compute node and $W-1$ elements with the top and bottom neighbor compute nodes, and computes $W-2$ LDBs. The computation results ($W-2$ LDBs) are written back to the physical disk. Thus the minimum required amount of memory for each compute node will be

$$M = WH + W - 1 \geq 3H + 2 = 3\frac{n}{P_Y} + 2, (W \geq 3).$$

Reading, communication, computation, and writing LDBs of $W-2$ will be repeated until the last LDB in each compute node is read.[5] At the end of each iteration, individual compute nodes do not store the computation results for the last $W-1$ LDBs and use them in the next iteration. Instead, the results should be stored when the computations are completed, and the total number of I/O operations will be

$$\begin{aligned} & 2\left(1 + I\left\lceil\frac{N-W+1}{W-2}\right\rceil\right) \\ = \; & 2\left(1 + I\left\lceil\frac{(H+1)(N+1)-(M+1)}{M-1-2H}\right\rceil\right) \\ = \; & 2\left(1 + I\left\lceil\frac{(n+P_X)(n+P_Y)-(M+1)P}{(M-1)P-2nP_X}\right\rceil\right), \end{aligned}$$

where I is the number of iterations and $P = P_X \times P_Y$. The algorithm is optimal in terms of the number of I/O operations, because each LDB is accessed only once for read and write.

Let's consider the communication cost. In every iteration to compute the first and last LDB, each compute node exchanges H elements with the right or left neighbor compute node. While computing whole LDBs (N LDBs) assigned to each compute node, each compute node exchanges N elements with the top and bottom neighbor compute nodes. Exchanging N elements with the top or bottom neighbor compute node requires $1 + \lceil\frac{N-(W-1)}{W-2}\rceil$. Thus the communication cost will be

$$\begin{aligned} & 2I\left((\tau + \mu H) + \left(\tau\left(1 + \left\lceil\frac{N-(W-1)}{W-2}\right\rceil\right) + \mu N\right)\right) \\ = \; & 2I\left(\tau\left(2 + \left\lceil\frac{N-(W-1)}{W-2}\right\rceil\right) + \mu(H+N)\right) \\ = \; & 2I\left(\tau\left(2 + \left\lceil\frac{n-P_X(W-1)}{P_X(W-2)}\right\rceil\right) + \mu(\frac{n}{P_Y} + \frac{n}{P_X})\right), \end{aligned}$$

where I is the number of iterations and $W \geq 3$.

4. Experimental Results

We implemented UPIO on top of the Proteus [2] parallel-architecture simulator which runs on a Sun workstation. Proteus is an execution-driven simulator for parallel architectures; it has been validated against real message-passing machines.

For I/O nodes in our architecture model we added IOPs (I/O Processor) and Ruemmler and Wilkes' HP97560 disk [10] that is attached to an SCSI bus whose peak bandwidth is 10 MB/s. The disk capacity and peak transfer rate are 1.3 GB and 2.34 MB/s, respectively.[6] The file system block size is 4 KB. In our experiments we configured the simulator with 16 compute nodes (4×4 grid) and 16 I/O nodes. Each node has a 25 MHz CPU and a direct-memory access (DMA) capability.

In order to evaluate the performance of the external Laplace equation solver based on the algorithm described in the previous section, we implemented the algorithm with an LD and a traditional file system using file pointers on our simulator. (Hereafter, we call the programs the LD and FP versions, respectively.) In our experiments a $4K \times 4K$ array is decomposed on 4×4 grid compute nodes in (BLOCK, BLOCK) manner. Each compute node stores a $1K \times 1K$ subarray using a 1-D local LD in the LD version; the subarray is stored in a local file in the FP version. In the FP version the subarray is stored in column-major order and a column of the subarray is defined as an LDB in the LD version, where data is stored contiguously and randomly on the disk.

Figure 4 compares the performances of the external Laplace equation solver using a file pointer and an LD, where the memory contains $W \times 1024 + W - 1$ elements and W is varied 3, 6, 9, and 12. In this figure we can see that the FP version performs the worst in contiguous and random disk-block allocations. This result can be explained as follows.

In the algorithm for the external Laplace equation solver, a submatix is forwardly and backwardly accessed in every other iteration repeatedly. Under a contiguous disk block allocation, disk blocks are accessed contiguously in both

[5] LD_read and LD_write are used to read and write the LDBs.

[6] The disk model is reimplemented by Kotz [4] and has been validated against disk traces provided by HP.

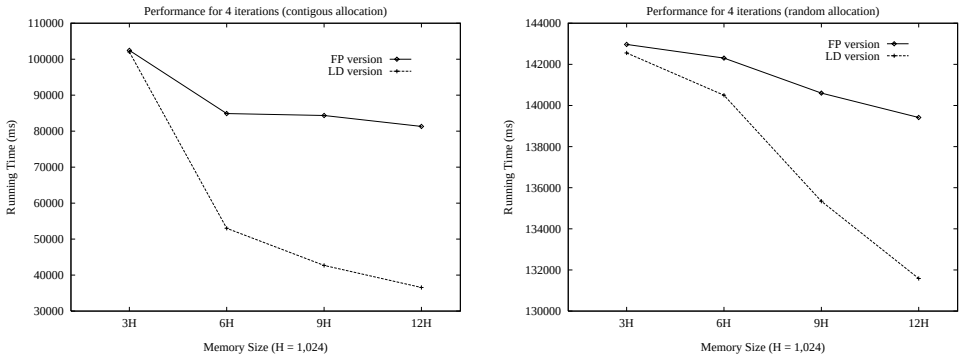

Figure 4. Performance of Laplace equation solver

data access patterns. The direction of data access is the only difference between the data access patterns. However, from our experiments described in [5], we observed that the backwardly accessed performance is worse than that for forwardly accessed.

When a subarray is forwardly accessed, multiple columns of the array are accessed at the same time in all program versions of the algorithm. If the subarray is backwardly accessed, only a single column of the array is accessed at one time in the FP version. Thus memory size does not much affect the performance of the FP version. However, in UPIO it is possible to access multiple LDBs in any direction and optimize disk-head movements in an I/O node. Thus, as the memory size increases, more columns can be accessed and the effect of backwardly accessing data decreases in the program using an LD.

5. Conclusions

In this paper we showed how well UPIO produces high-performance external computation codes by designing I/O and communication-efficient external Laplace equation solver algorithm and exploring the effects of UPIO with the codes. In this algorithm we reduced the number of I/O operations by reusing data already brought into memory at the end of each iteration and by using a memory vector of UPIO, which is an array of (memory offset, number of LDBs) and makes it possible to read or store data into or from non-contiguous memory space. By explicitly scheduling the direction of computations, it is possible for compute nodes to communicate each other collectively, thereby eliminating extra I/Os caused by communications. This algorithm is also memory efficient, because no extra memory is required to store temporary computation results.

References

[1] M. L. Best, A. Greenberg, C. Stanfill, and L. W. Tucker. CMMD I/O: A parallel Unix I/O. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 489–495, 1993.

[2] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A high-performance parallel architecture simulator. Technical Report MIT/LCS/TR-516, MIT, September 1991.

[3] P. F. Corbett and D. G. Feitelson. Vesta file system programmer's reference. Technical Report Research Report RC 19898 (88058), IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, October 1994. Version 1.01.

[4] D. Kotz, S. B. Toh, and S. Radhakrishnan. A detailed simulation model of the hp 97560 disk drive. Technical Report Technical Report PCS-TR94-220, Dept. of Computer Science, Dartmouth College, July 1994.

[5] J. S. Lee, J. Kim, P. B. Berra, and S. Ranka. Logical disks: User-controllable I/O for scientific applications. In *Proceedings of the 1996 IEEE Symposium on Parallel and Distributed Processing*, pages 340–347, 1996.

[6] J. S. Lee, S.-G. Oh, P. B. Berra, and S. Ranka. User-controllable I/O for parallel computers. In *Proceedings of Parallel and Distributed Processing Techniques and Applications*, pages 442–453, 1996.

[7] S. J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E. D. Milne, and R. Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Conference*, pages 291–305, 1993.

[8] N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[9] P. Pierce. A concurrent file system for a highly parallel mass storage system. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160. Golden Gate Enterprises, Los Altos, CA, March 1989.

[10] C. Ruemmler and J. Wilkes. Modelling disks. Technical Report HPL-93-68, revision 1, HP Lab., July 1993.