ISSUES IN COMPILING I/O INTENSIVE PROBLEMS

Rajesh Bordawekar and Alok Choudhary

*ECE Dept., 121 Link Hall, Syracuse University,
Syracuse, NY 13244, 315-443-2372

1 INTRODUCTION

It has been widely acknowledged in high-performance computing circles that input/output (I/O) needs substantial improvement in order to make scalable computers truly usable. Many users see parallelism in I/O as the best way to improve the I/O performance. Recent surveys [12] of I/O needs of parallel applications have determined that their I/O requirements fall into following categories: (1) Initial/intermediate/final I/O, (2) Out-of-core Computations, (2) Checkpointing and Restart and (4) Real-time I/O. In order to improve the overall I/O performance, one or more of these requirements need to be addressed.

Although all the above mentioned requirements are important, in this paper, we focus on applications which perform extensive I/O during program execution for initial/intermediate/final data access and for out-of-core computations. We approach these problems using the compiler and runtime strategies that we have developed as part of the PASSION (PArallel and Scalable Software for I/O) project[1] at Syracuse University [4].

In this paper, we describe various issues connected with compiler and runtime support for I/O intensive applications. Sections 2 and 3 present architectural and programming models. These are also supported by the PASSION compiler and runtime framework. Section 4 analyzes different working spaces in I/O intensive applications. Section 5 uses these spaces to describe two execution models which can be used for developing I/O intensive applications. Section 6 presents a brief overview of the compilation strategies for out-of-core parallel programs. We summarize in Section 7.

[1] For further information, check the URL http://www.cat.syr.edu/passion.html
2 ARCHITECTURAL MODEL

Our architectural models assume the hierarchical memory model proposed by Vitter and Shriver [15, 14]. We consider an abstract machine consisting of a set of processors interconnected via a high-speed interconnection network. Each processor possesses three levels of memories, the first level includes registers/cache, the second level includes main memory and the third level is secondary memory which consists of individual disks or RAID drives.

Before describing the architectural models, let us define the term Parallel I/O. Parallel I/O refers to simultaneous access (read/write) by more than one processors of shared or distinct files. Note that parallel I/O on a shared file is possible only if the file is opened in parallel by the participating processors (also called parallel file). Similarly, each processor can access a different file simultaneously iff the file is opened by the processor accessing it (called local file).

We consider the following two basic models for performing parallel I/O on such a machine.

1. **High-level Processor-based I/O Parallelism:**
   Our models assume two different views of a file. One is the Global View, in which each processor views the entire file as a single object, whereas in the Local View each processor views a file to be consisting of several logical sub-files, each belonging to a separate processor. It should be noted that a parallel file can be viewed using either global or local view, but the local files can only be viewed using the local view. Several parallel machines like Intel Paragon provide this functionality in their file systems [9].

2. **Multi-level Controller-based I/O Parallelism:**
   In this model, a processor is connected to a local disk of its own. Any processor can access its own local disk or a remote disk on a remote processor. If a processor is connected with a local disk, it serves as the controller of the local disk (Figure 1-A).

   In a recent paper, Kotz et al. advocate the use of I/O nodes for computation (similar to our first model) [11].
Our models allow multiple processors accessing the same or distinct sections of a file simultaneously. In practice, different parallel machines have implemented this feature by providing different file access modes. For example, the Intel Paragon provides six different I/O modes, M.UNIX, M.LOG, M.GLOBAL, M.SYNC, M.RECORD, M.ASYNC. These modes allow multiple processors to access files individually and collectively, both in synchronous or asynchronous fashion [9]. Similar support is also provided in the CM-5 file system [13].

2. Middle-level File-based I/O Parallelism:

File-based parallelism is obtained by distributing a file over multiple disks (or RAID drives). This is termed file Striping. When a processor sends a request for reading or writing a data set, the I/O controller first computes the disks on which the requested data lies. Then the processor’s read/write request is split into several different disk requests which read/write the data in parallel. When used together with processor-based I/O parallelism, file-based I/O parallelism can obtain good I/O performance.

3. Lower-level Disk-based I/O Parallelism:

Recently, there has been a general trend to use RAID drives instead of using individual disks. As pointed out earlier, RAID drives increase the reliability of the secondary media using parity data and provide means of parallelism by striping data across the RAID array, called disk striping. The disk striping strategy depends on the RAID version (currently 1 to 5).

When a RAID drive receives a disk request to read/write data, it is split into different requests which are sent to individual disks in the RAID array. Each disk then reads/writes the necessary data by controlling its head. The resulting disk parallelism provides considerable performance improvement.

It should be observed that among these three levels, a user (or application) can control the degree of parallelism only in the highest level (Processor-based I/O). In this level, the degree of parallelism is proportional to the number of processors accessing the file(s) simultaneously. The higher the number of processors performing I/O, the greater the degree of parallelism. In the file-based I/O, parallelism can be improved by increasing the number of disks over which a file is distributed and in the disk-based I/O, parallelism can be increased by increasing the number of disks in the disk array. Most of the existing file systems do not provide any support for the user to explicitly control the file and disk distribution.

HPF provides several new data parallel constructs. The most important among them is the FORALL construct [8, 10]. The HPF FORALL[8] statement is a generalization of the Fortran 90 array assignment statement. The FORALL is a parallel construct with determinate semantics; i.e., the statement can execute in parallel for a variable number of processors and the results are identical. A FORALL statement is not a loop, nor it is a parallel loop[16]. The reason is that the FORALL does not iterate in any well-defined order. Therefore, it is appropriate to consider FORALL index values rather than FORALL iterations. Implementation of a FORALL statement consist of following steps: (1) Computing the active index set for the computation, (2) For the active index set, compute the values of the right-hand side of the FORALL body and
4 WORKING SPACES IN I/O INTENSIVE PARALLEL PROGRAMS

Let us consider an I/O intensive data parallel program (such as one written in HPF) running on a distributed memory machine. The primary data structures will be accessed from files stored on disks. We assume that the file will be striped across several disks (Section 2). Under such assumptions, we have to consider four working spaces in which an I/O intensive program operates: Program Space, Processor Space, File Space and Disk Space (Figure 3).

- **Program Space**
  The program space of an application is specified by the set of the executable statements and their ordering in the data parallel program. Let $S_i$ denote the $i^{th}$ executable statement in the program. Then the program space $P$ will be given by the set $\{S_1, \ldots, S_n\}$, where $n$ is the last executable statement in the program. $P$ can be used study the access patterns of the scalars and arrays used in the program. A variable (scalar or array) in the program is said to have spatial locality in program space if it is used (i.e., defined or referred) in statements $S_i$ and $S_j$, $j \geq i$. A program variable can also exhibit temporal locality. Temporal locality is observed if a program variable is referred to again after some time delay. Any scalar assignment in the FORALL construct results in temporal locality. Note that locality analysis in the program space is performed in the global name space (i.e., distribution characteristics of the variables are ignored).

  Consider the HPF program shown in Figure 3:A. This program contains a set of compiler directives and a set of assignment statements which operate on the distributed arrays. Let us consider the assignment statements. In the first assignment statement, element 1 of array $C$ is assigned to array $A$. This element will be used in each iteration of the assignment statement. Hence element $C(1,1)$ exhibits temporal locality. A section of the array $A$ is used in the statement 2. The section of array $A$ which is used in both statements 1 and 2 exhibits spatial locality. Similar locality is shown by the section of array $B$ which is used in statements 2 and 3.

- **Processor Space**
  The processor space of an application is determined by the distribution characteristics of the variables used in that application. Program variables are said to possess spatial locality in processor space iff they belongs to the same processor. In the HPF program (Figure 3:A), arrays $A, B, C$ are distributed over four processors in the BLOCK = BLOCK manner. The processors are arranged in a logical 2-D mesh. Figure 3:B shows array $A$ partitioned into four local arrays (corresponding...
### Issues in Compiling I/O Intensive Problems

Elements of a local array space since they belong to the same processor. Spatial locality in processor space but not disk space and vice versa.

- **File Space**

The file space of the application depends on the file. A set of variables is said to have spatial locality in the file if they lie in the same file. The definition of the spatial locality in the file follows: two elements are said to have sequential positions in the same file.

Consider figure 3:C. It shows the file correspondence distributed over four processors (Figure 3:E). The first sub-column of the file, followed by the first sub-column of the data belonging to a particular processor, may not be stored consecutively in the file space. But elements of each sub-column subset are stored in the same disk space and sequential file locality.

- **Disk Space**

The disk space of the application depends on the file. A set of variables is said to have spatial locality in the same disk. The definition of the locality in the disk space follows: two elements are said to have sequential positions in the same disk.

Consider figure 3:D. It shows how the global disk space is stored. Each disk contains elements of the first and third sub-columns of the file. For example, disk D0 contains first and third sub-columns of the file. Similarly, only the elements of the first sub-column disk, since they belong to the non-continuous data space, are stored at continuous positions in the disk. These elements also exhibit spatial file locality, since they will be stored at consecutive positions in the same file.
to four processors). Elements of a local array exhibit spatial locality in processor space since they belong to the same processor. Note that two elements can have spatial locality in processor space but not have spatial locality in program space and vice versa.

File Space

The file space of the application depends on how the distributed data is stored in files. A set of variables is said to have *spatial locality in file space* if it lies in the same file. The definition of the locality in file space can be extended to take into account the data storage order. We define *sequential* file locality as follows: two elements are said to have sequential file locality if they are stored at consecutive positions in the same file.

Consider figure 3:C. It shows the file corresponding to the global array which is distributed over four processors (figure 3:B). The data is stored in the file in the column-major order. The first sub-column of processor 1 is stored at the beginning of the file, followed by the first sub-column of processor 3 and so on. Therefore, the data belonging to a particular processor (or having spatial processor locality) may not be stored consecutively in the file (or may not exhibit sequential file locality). But elements of each sub-column will exhibit both spatial processor locality and sequential file locality.

Disk Space

The disk space of the application depends on how the files are striped across disks. A set of variables is said to have *spatial locality in disk space* if it lies in the same disk. The definition of the locality in disk space can also be extended to take into account the data storage order. We can define sequential disk locality as follows: two elements are said to have sequential disk locality if they are stored at consecutive positions in the same disk.

Consider figure 3:D. It shows how the global files are striped across four disks in a round-robin fashion. Each disk contains sub-columns of different processors. For example, disk D0 contains first and third sub-columns of processors 1 and 2, disk D1 contains first and third sub-columns of processors 3 and 4 and so on. The elements of the first and third sub-columns of processor 1 possess both spatial processor and disk locality since they belong to same processor as well as same disk. These elements also exhibit spatial file locality because they belong to the same file. However, these elements do not have sequential locality in file space since these sub-columns are stored at non-consecutive positions in the global file. Similarly, only the elements of the first sub-column will exhibit sequential disk locality, since they will be stored at consecutive positions on the disk D0. This example shows that data may exhibit file locality but not disk locality and vice versa.
What is the significance of these spaces? During execution of I/O intensive programs, data needs to be fetched from external storage into memory. As a result, performance of such programs depends mainly on the time required to read and write data. In order to achieve reasonable speedups, the compiler or user needs to minimize the number of I/O accesses. This can only be achieved either by exploiting the inherent locality in the source program or by performing program transformations to maintain (or translate) localities between spaces.

In case of in-core programs, operations on local memory are analogous to the operations in file space in the I/O intensive programs. There are several memory optimizations that modify the computation order such that it corresponds to the data storage order in memory. The simplest memory optimization is interchanging the DO iterations so that memory accesses become conformal with the memory storage order (such as column-major in FORTRAN). Can we use these optimizations directly for I/O intensive problems accessing disks?

The answer is negative. In order to understand the interactions between different working spaces, let us analyze the operation of an I/O intensive program. Any I/O intensive program will contain a large number of read and write calls. Normally a read or write call takes following parameters as an input: an index to the underlying file (e.g., unit number or file pointer) and amount of data to be read or written. From the application level, the user can approximate the I/O cost of program by computing the number of I/O (read and write) calls. In reality, the I/O cost of an I/O intensive program will be also decided by the number of disk accesses generated in the program. The number of I/O calls will be decided by the amount of consecutive data read/written from/to a given file, whereas the number of disk accesses will be decided by the amount of consecutive data read/written from/to the disks. In order to improve the I/O performance, any application should access as much consecutive data as possible both from the file and disks. In other words, the user (or the compiler) should try to translate the program locality into spatial/sequential locality in file and disk space.

In order to reduce the number of I/O calls, the user should translate program locality into file locality and vice versa (by controlling file storage order or reordering program statements). As a result, every I/O call will access a larger amount of consecutive data. In order to reduce the number of disk accesses, the user (or the compiler) should modify her program so that computational ordering corresponds to the data ordering on disks. There are two problems in implementing such optimizations. First, the actual order in which the data is stored will be dependent on several factors such as number of disks, stripping policies etc. These factors vary from machine to machine, and it is very difficult for the user to get information about them. Secondly, it is very difficult to directly control disks from the application level. Therefore, at present, I/O performance can be only improved by reducing the number of I/O calls. Since the number of I/O calls does not give the true measure of the I/O cost of the program, this optimization is not very effective.

Even though the number of I/O calls does not truly measure the overall I/O cost, translation of locality to/from file space does provide some advantages. The most important advantages are transparency and portability. These characteristics are very important for compilers. Let us assume that a compiler performs certain code transformations so that locality in program space is translated into locality in disk space. However, the transformed program will give reasonable performance only for those disk configurations which conform with the generated code. But in all other cases, the performance of the application will be extremely poor. However, for a given application the number of I/O calls will remain constant irrespective of the corresponding number of disk accesses. As a result, a program optimized for file space will give reasonable and consistent performance across a variety of architectures.

5 EXECUTION MODELS

Execution models help the user or compiler to translate (maintain) locality between working spaces. Since we do not focus on disk space, we should consider the execution models which translate (maintain) locality between program, processor and file space. In this section, we present two execution models: the Local Placement Model and the Global Placement Model.

5.1 Local Placement Model

In the Local Placement Model, the out-of-core local array of each processor is stored in separate logical file called the Local Array File (LAF) of that processor. Each LAF can be stored separately as a different physical file or different LAFs can be appended to generate a single physical file. The processor which generates the LAF is said to own the LAF.

The node program explicitly reads from and writes to the LAF when required. A processor cannot explicitly access a file owned by some other processor. If a processor needs to access data from a file owned by a different processor, the required data first has to be read by the owner and then communicated to the requesting processor. Since each local array file contains the OCLA of the corresponding processor, the distributed (or user-specified) view of the out-of-core global array is preserved. In other words,
locality (spatial and sequential) in processor space is translated into locality in the file space.

One way to view the Local Placement Model is to think of each processor as having another level of memory which is much slower than main memory. The extra memory is in form of a virtual disk which consists of more than one physical disks. Each processor stores its LAF into its virtual disk. In other words, it is a straightforward extension of the distributed memory model. However, each processor lacks exclusive access to one or more physical disks that constitute the virtual disk. Physical disks may be shared by more than one processor. The mapping of physical to virtual disks is performed at runtime.

In order to store the data in the local array files based on the distribution pattern specified in the program, redistribution of the data may be needed in the beginning when the data is staged. This is because the way data arrives (e.g., from archival storage or over the network) may not conform to the distribution specified in the program. Redistribution requires reading the data from the external storage, shuffling the data over the processors and writing the data to the local virtual disks. This increases the overall cost of data access. This cost can be amortized if the out-of-core array is used repeatedly.

Figure 4 presents an out-of-core array distributed in BLOCK-BLOCK fashion over 4 processors. Each processor stores its OCLA in its LAF (shown using different shades). Each local array file is stored into a virtual disk. During computation, each processor brings a section of the OCLA into its memory (called an In-core Local Array or ICLA) and operates on it and (if necessary) stores it back.

5.2 Global Placement Model

In the Global Placement Model (GPM), the out-of-core global array is stored in a single file called the Global Array File (GAF). The global array is stored in the GAF in either row-major or column-major order, and the file is stored in a single virtual disk which is shared by all the processors accessing this array. As in case of the Local Placement Model, each virtual disk consists of one or more physical disks and the mapping of physical to virtual disks is performed at runtime.

Each processor can read and write parts of its local array from the GAF. Since the local arrays of the processors are interleaved in the GAF, there is no concept of a local array file. A processor can read data owned by other processors, however, it cannot write over the data owned by other processors. This is due to the fact that the
underlying programming model is SPMD in which computations are performed in the local name space. This property of GPM eliminates any consistency problems arising from data sharing. Since any processor can read any other processor's data, the I/O can be performed in global name space (using global coordinates). As the computation is still in the local name space, this model can be viewed as having shared memory on the files and distributed memory on the processors.

In the Global Placement Model, the global array view is preserved from the global HPF program. As a result, localities in program space (temporal and spatial) are translated into localities in file space.

Figure 5 shows columns of the local array of a processor interleaved with others in the global file (shown with different shades). The global file is distributed over four disks which are shared by all processors. Thus, parts of the global array, based on the array distribution, need to be brought into the memory for processing. For example, Figure 5 shows a sweep of the global array by columns (numbered 1 through 3), assuming that only one column per processor fits into memory. The corresponding file access pattern also shown. In general, the accesses may be viewed as loosely synchronous. That is, the I/O can be performed in a collective manner (similar to collective communication) using a global index space. Subsequently, the data may need to be re-shuffled in memory by mapping the global indices (of current data) into local index space. Since there are no local files in the GPM, data redistribution into files is not required.

5.3 Relative Comparisons of the Models

The program execution models translate localities between working spaces of the out-of-core computation. LPM translates temporal and spatial locality from data space to file space, whereas GPM translates locality from program space to file space. LPM uses the distributed memory approach in which I/O and computations are performed in local name space while GPM allows global name space I/O and local name space computation.

Programs using the Local Placement Model are required to generate local array files. If the source files are in a different format, local file generation requires data re-shuffling which results in extra overhead. If the files are temporary, the compiler (or user) can generate these files whenever needed. Since data belonging to each processor is stored in a separate LAF, the overall out-of-core computation becomes simplified. This model is suitable as an execution model for out-of-core parallel compilers because various hierarchical memory optimization techniques can be directly applied to improve performance.

The main advantage of the Global Placement Model is that the global program view is maintained while performing file I/O. Another advantage is that processors can read data owned by other processors. This property of GPM is very useful in the I/O optimizations like two-phase access [2]. Also since I/O is done in global name space, there is no need to generate local array files. This execution model is very suitable for application developers because the user can take advantage of global name-space I/O and a single data file. This model, however, is not suitable for parallel compilers for the following reasons: (1) it is very difficult to maintain data consistency and (2) global-to-local address computations become very tedious for non-trivial distributions (e.g., BLOCK-CYCLIC).

6 COMPILING OUT-OF-CORE PARALLEL PROGRAMS

In this section, we analyze various issues concerned with compilation of out-of-core parallel programs. We use HPF as a target language. We first describe the basic compilation strategy for in-core programs and then discuss the additional support needed for handling out-of-core programs. We explain the compilation strategy for both cases with the help of the HPF program fragment given in Figure 6. In this example, arrays A and B are distributed as two-dimensional blocks on a 4 x 4 grid.

```
1 REAL A(1024,1024), B(1024,1024)
       ............
2 !HPF$ PROCESSORS P(4,4)
3 !HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO P : A,B
       ............
4 FORALL (I=2:N-1, J=2:N-1)
5 A(I,J) =
6 END FORALL
```

Figure 6 HPF Program Fragment

The in-core HPF compiler uses distribution directives (Figure 6, lines 2–3) in the source program to find the distribution pattern of the arrays. Using the distribution information, arrays are partitioned into local arrays. After data distribution, the compiler analyzes the FORALL statement (Figure 6, Line 4). Since array B is distributed in a block-block fashion over 16 processors, the array expression requires fetching data from up to four neighboring processors. The HPF compiler generates a call to a specific collective communication routine (overlap shift), and the array expression is sequentialized into DO loops for local computation.
1. Data-parallel Phase
   (a) Analyze distribution patterns of each array.
   (b) Partition computation according to distribution patterns.
   (c) Determine communication required for array accesses.
   (d) Determine local space bounds.

2. Out-of-core Phase
   (a) Determine in-core bounds.
   (b) Partition computation in in-core local space.
   (c) Determine I/O and communication required for in-core computations.

3. Code Generation Phase
   (a) Sequentialize local code.
   (b) Insert communication and I/O.
   (c) Optimize communication and I/O.

In this paper, we only discuss the compilation process for the Local Placement Model. The compiler is conceptually divided into three phases as shown in Figure 7. The data-parallel phase distributes data to multiple processors. The arrays are first partitioned according to the parallelism directives and local lower and upper bounds for each local array are calculated. Statements such as FORALL are then analyzed to determine the required communication. In other words, the compilation in this phase proceeds in the same manner as for in-core computations. The out-of-core phase partitions out-of-core data into in-core local arrays. The resulting in-core arrays are analyzed to determine inter-array communication and the computation is partitioned to operate on one in-core array at a time. The code generation phase inserts and optimizes communication and I/O, generating an SPMD node program with explicit message passing and I/O operations. This process sequentializes the in-core local operations and inserts the necessary I/O and communication calls. I/O is performed using calls to runtime I/O library routines (e.g. PASSION [4]).

6.1 Compiling Communication in the Local Placement Model

In an out-of-core (OOC) application, computation is carried out in phases. Each phase reads a slab of data (or ICLA), performs computations using this slab and writes the slab back in the local array file. In this case processors may need to communicate because (1) computation of an in-core local array requires data which is not present in memory during the computation involving ICLA or, (2) ICLA contains data which is required by other processors for computation. The communication can be performed in two ways: (1) in a collective manner, using Out-of-core Communication and (2) in a demand basis, termed as In-core Communication.

We will now illustrate the two communication approaches using the example of the 2-D elliptic solver (using Jacobi Relaxation) (Figure 6). We now assume that array A is an out-of-core array which is distributed over 16 processors. Each processor stores its local array into its local array file.

![Figure 7 Compilation Phases in the Local Placement Model](image)

Out-of-core Communication

In the out-of-core communication method, the communication is performed collectively considering the entire OOC local array. All processors compute the element which are required for the computation of the OCLA but are not present in the OCLA. These elements are communicated either before or after the computation on the OCLA. The communication from node j to node i involves following steps...
1. Synchronize (if necessary).

2. Node $j$ checks if it needs to send data to other processors. If so, it checks if the required data is in memory. If not, node $j$ first sends a request to read data from disk and then receives the requested data from disk. If the required data is in memory then the processor does not perform file I/O.

3. Node $j$ sends data to node $i$.

4. Node $i$ either stores the data back in its local file or keeps it in memory (This would depend on whether the data required can be entirely used by the current slab in the memory; if not, the received data must be stored in local files).

To illustrate these steps, consider processors 5 and 6 from Figure 8 (A). Each processor performs operations on its OCLA in stages. Each OCLA computation involves repeated execution of three steps: (1) Fetching an ICLA, (2) Computing on the ICLA, (3) Storing the ICLA back in the local array file. Figure 8(B) shows the ICLA's using different shades. Figure 8(C) shows the data that needs to be fetched from other processors (called the ghost area). In the out-of-core method, all the processors communicate this data before the computation on the OCLA begins. To illustrate the point that out-of-core communication requires I/O, note that processor 5 needs to send the last column to processor 6. This column needs to be read from the local array file and communicated. Figure 9 shows the phases in the out-of-core communication method.

In the out-of-core communication method, communication and computation are performed in two separate phases. As a result, the OCLA computation becomes atomic, i.e., once started it goes to completion without interruption. This method is attractive from the compiler point of view since it allows the compiler to easily identify and optimize collective communication patterns. Since the communication will be carried out before the computation, this strategy is suitable for HPF FORALL-type of computations which have copy-in-copy-out semantics. In the above example, four shifts are required which result in disk accesses, data transfer and data storage (in that order).

**In-core Communication**

For OOC computations, the communication may be performed in an entirely different way by just considering the communication requirements of the ICLA (or slab in memory) individually. In other words, the communication set for each ICLA is generated individually. The basic premise behind this strategy is that if the data present in the memory can be used for communication while it is resident in memory, it may reduce the number of file I/O steps.
The In-core communication method differs from the out-of-core communication method in two aspects, (1) in the In-core communication method, communication is not performed collectively. The two phases, computation on the ICLA and communication are interleaved. However, the computation on the ICLAs is still carried out in an SPMD fashion. The data to be communicated is the data which is required for the computation of the ICLA but is not present in the memory (but it may be present in remote memory or another processor’s file). The In-core communication can be further divided into two types, (1) Demand-driven Communication and (2) Producer-driven Communication.

**Demand-driven In-core Communication (Consumer decides when to fetch)**

In this strategy, the communication is performed when a processor requires off-processor data during the computation of the ICLA. Figure 10 illustrates the demand-driven communication method. Node 2 requires off-processor data at point 2 (Figure 10). Let us assume that the required data is computed by node 1 at point 1 and stored back on disk. When node 2 requires this data, it sends a request to node 1 to get this data. Node 1 checks if the data is in memory else it reads the data (point 3). After reading the data from disk, node 1 sends this data to node 2. Node 2 receives this data (point 5) and uses it during the computation of the ICLA.

This method can be illustrated using the example of the elliptic solver (Figure 8). Consider again processor 5. Figure 8(B) shows the different ICLAs for the processor 5. Let us consider slab 1 (shown by the darkest shade). The ghost area of this slab is shown in Figure 8(D). When this ICLA is in processor’s memory it requires data from processors 1, 4 and 9. Hence, processor 5 sends requests to processors 1, 4 and 9 check whether the requested data is present in the ICLA or it has to be fetched from the local array file. Since processors 1 and 9 have also fetched the first slab, the requested data lies in the main memory. Hence processors 1 and 9 can send the requested data without doing file I/O. However, since processor 4 has also fetched the first slab, the requested data does not lie in the main memory. Therefore, processor 4 has to read the data (last column) from its local array file and send it to processor 5. It is important to note that the shift collective communication pattern in the original OOC communication is broken into different patterns when in-core communication is considered.

**Producer-driven In-core Communication (Producer decides when to send)**

Figure 11 Producer-driven In-core Communication.
The basic premise of this communication strategy is that when a node computes on an ICLA and can determine that a part of this ICLA will be required by another node later on, this node sends data while it is in its present memory. Note that in the demand-driven communication, if the required data is stored on disk (as shown in Figure 10), the data needs to be fetched from disk which requires extra I/O accesses. This extra I/O overhead can be reduced if the data can be sent to the processor either when it is computed or when it is fetched by its owner processor.

This approach is shown in Figure 11. Node 2 requires some data which is computed by node 1 at point 1. If node 1 knows that data computed at point 1 is required by node 2 later, then it can send this data to node 2 immediately. Node 2 can store the data in memory and use it when required (point 3). This method is called the Producer-driven communication since in this method the producer (owner) decides when to send the data. Communication in this method is performed before the data is used. This method requires knowledge of the data dependencies so that the processor can know beforehand what to send, where to send and when to send. It should be observed that this approach saves extra disk accesses at the sending node if the data used for communication is present in its memory.

In the example of the elliptic solver, assume that processor 5 is operating on the last slab (slab 4 in Figure 8(D)). This slab requires the first column from processor 6. Since processor 6 is also operating on the last slab, the first column is not present in the main memory. Hence, in the demand-driven communication method, processor 6 needs to fetch the column from its local array file and send it to processor 5. In the producer-driven communication method, processor 6 will send the first column to processor 1 during the computation of the first slab. Processor 5 will store the column in its local array file. This column will be then fetched along with the last slab thus reducing the I/O cost.

Discussion

The main difference between the in-core and out-of-core communication methods is that in the latter, communication and computation phases are separated. Since the communication is performed before the computation, an out-of-core computation consists of three main phases: Local I/O, Out-of-Core Communication and Computation. The local I/O phase reads and writes the data slabs from the local array files. The computation phase performs computations on in-core data slabs. The out-of-core communication phase performs communication of the out-of-core data. This phase redistributes the data among the local array files. The communication phase involves both inter-processor communication and file I/O. Since the required data may be present either on disk or in on-processor memory, three distinct access patterns are observed.

1. Read(write) from this processor’s logical disk.
   This access pattern is generated in the in-core communication method. Even though data resides on the logical disk owned by a processor, since the data is not present in the main memory it has to be fetched from the local array file.

2. Read from some other processor’s memory.
   In this case the required data lies in the memory of some other processor. In this case only memory-to-memory copy is required.

3. Read(write) from some other processor’s logical disk.
   When the required data lies on some other processor’s disk, communication has to be done in two stages. In case of data read, in the first stage the data has to read from the logical disk and then communicated to the requesting processor. In case of data write, the first phase involves communicating data to the processor that owns the data and then writing it back to the disk.

The overall time required for an out-of-core program can be computed as a sum of times for local I/O $T_{i/o}$, in-core computation $T_{comp}$ and communication $T_{comm}$.

\[ T = T_{i/o} + T_{comm} + T_{comp} \]

$T_{i/o}$ depends on (1) Number of slabs to be fetched into memory and, (2) I/O access pattern. The number of slabs to be fetched is dependent on the size of the local array and the size of the available in-core memory. The I/O access pattern is determined by the computation and the data storage patterns. The I/O access pattern determines the number of disk accesses. $T_{comm}$ can be computed as a sum of I/O time and inter-processor communication time. The I/O time depends on (1) whether the disk to be accessed is local (owned by the processor) or it is owned by some other processor, (2) the number of data slabs to be fetched into memory and, (3) the number of disk accesses which is determined by the I/O access patterns. The inter-processor communication time depends on the size of data to be communicated and the speed of the communication network. Finally the computation time depends on the size of the data slabs (or size of available memory). Hence, the overall time for an out-of-core program depends on the communication pattern, available memory and I/O access pattern.
6.2 Experimental Performance Results

The applications were implemented on the Intel Touchstone Delta machine at Caltech. The Touchstone Delta has 512 compute nodes arranged as a 16×32 mesh and 32 I/O nodes connected to 64 disks. It supports a parallel file system called the Concurrent File System (CFS).

Table 1 presents performance of 2D out-of-core elliptic solver using the three communication strategies. The problem size is 4K×4K array of real numbers, representing 64 MBytes of data. The data distribution is (BLOCK-BLOCK) in two dimensions. The number of processors is 64 (with a 8×8 logical mapping). The size of the ICLA was varied from 1/2 of the OCLA to 1/16 of the OCLA.

Table 1 and 2 show three components of the total execution time; namely, Local I/O (LIO) time, Computation time (COMP) and the Out-of-core Communication time (COMM) for the Out-of-core Communication Method, the Demand-driven In-core Communication Method and the Producer-driven In-core Communication Method. The experiment was performed for four values of the memory ratio (ICLA/OCLA). From these results we make the following observations

1. COMP remains constant in all the three communication methods. This is expected as the amount of computation is the same for all cases.

2. COMM is largest in the out-of-core Communication method. This is because, each processor needs to read boundary data from a file and write the received boundary data into a file. Since the boundary data is not always consecutive, reading and writing of data results in many small I/O accesses. This results in an overall poor I/O performance. However, in this example, for the out-of-core communication method, COMM does not vary significantly as the size of the available memory is varied. As the amount of data to be communicated is relatively small, it can fit in the on-processor memory. As a result, communication does not require stripmining (i.e. becomes independent of the available memory size). If the amount of data to be communicated is greater than the size of the available memory, then COMM will vary as the size of the available memory changes.

3. Producer-driven in-core communication, even though it performs the best, does not provide significant performance improvement over the Demand-driven in-core communication method. The main reason that is due to lack of on-processor memory, the receiver processor stores that received data on disk and reads it when needed. This results in extra I/O accesses.

4. In both Demand and Producer-driven communication methods, COMM does not vary significantly as the amount of available memory is changed. In the 2-D Jacobi method, the inter-processor communication forms a major part of in-core communication. Since the in-core communication requires small I/O, the in-core communication cost is almost independent of the available memory.

5. As the amount of memory is decreased, more I/O accesses are needed to read and store the data. This leads to an increase in the cost of LIO. It should be noted that the local I/O and the I/O during communication are the dominant factors in the overall performance.

6. In in-core communication methods, the structured communication pattern (shift) gets distributed into several unstructured patterns (for each in-core data slab). In order to optimize these communication patterns, we need to use Producer/Demand-driven communication methods.

Table 2 illustrates the performance for the same problem with the same level of scaling for the problem size and the number of processors. This example solves a problem of 8K×8K on 64 processors. Clearly, for both problem sizes, the out-of-core communication strategy performs the worst in terms of the communication time due to the fact that communication requires many small I/O accesses.
Table 2 Out-of-core 2D-Jacobi (8Kx8K) on 64 Processors, time in sec.

<table>
<thead>
<tr>
<th></th>
<th>Demand-driven In-core Comm.</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Ratio=$\frac{1}{2}$</td>
<td>Ratio=$\frac{1}{4}$</td>
<td>Ratio=$\frac{1}{8}$</td>
<td>Ratio=$\frac{1}{16}$</td>
</tr>
<tr>
<td>COM</td>
<td>1.92</td>
<td>1.78</td>
<td>2.10</td>
<td>2.34</td>
<td></td>
</tr>
<tr>
<td>COMP</td>
<td>5.8</td>
<td>5.9</td>
<td>5.49</td>
<td>5.29</td>
<td></td>
</tr>
<tr>
<td>LIO</td>
<td>215.53</td>
<td>218.51</td>
<td>224.41</td>
<td>246.72</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Producer-driven In-core Comm.</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Ratio=$\frac{1}{2}$</td>
<td>Ratio=$\frac{1}{4}$</td>
<td>Ratio=$\frac{1}{8}$</td>
<td>Ratio=$\frac{1}{16}$</td>
</tr>
<tr>
<td>COM</td>
<td>1.88</td>
<td>1.98</td>
<td>2.03</td>
<td>2.61</td>
<td></td>
</tr>
<tr>
<td>COMP</td>
<td>5.50</td>
<td>5.41</td>
<td>5.32</td>
<td>5.60</td>
<td></td>
</tr>
<tr>
<td>LIO</td>
<td>218.70</td>
<td>220.79</td>
<td>235.33</td>
<td>244.04</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Out-of-core Comm.</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Ratio=$\frac{1}{2}$</td>
<td>Ratio=$\frac{1}{4}$</td>
<td>Ratio=$\frac{1}{8}$</td>
<td>Ratio=$\frac{1}{16}$</td>
</tr>
<tr>
<td>COM</td>
<td>250.76</td>
<td>261.73</td>
<td>288.97</td>
<td>279.37</td>
<td></td>
</tr>
<tr>
<td>COMP</td>
<td>5.90</td>
<td>5.88</td>
<td>5.6</td>
<td>5.8</td>
<td></td>
</tr>
<tr>
<td>LIO</td>
<td>251.54</td>
<td>268.39</td>
<td>312.75</td>
<td>348.19</td>
<td></td>
</tr>
</tbody>
</table>

7 SUMMARY

Lack of efficient I/O prevents many applications from achieving high performance, especially those involving the use/manipulation of large datasets. For this reason, many users have to be content with working on small problem sizes. This problem, commonly known as the I/O bottleneck needs to be attacked from many directions. In this paper, we considered the I/O problem from a compiler and runtime point of view. We introduced different working spaces in I/O intensive problems and presented execution models which could be used for optimizing I/O intensive applications. We described how an I/O intensive parallel program is compiled and presented different strategies for performing communication in out-of-core programs. We supplemented the analysis by presenting experimental results of a 2-Dimensional Jacobi Relaxation application.

REFERENCES

INTRODUCTION TO MULTIPROCESSOR I/O ARCHITECTURE

David Kotz
dfk@cs.dartmouth.edu
Department of Computer Science
Dartmouth College, Hanover, NH 03755-3510

ABSTRACT

The computational performance of multiprocessors continues to improve by leaps and bounds, fueled in part by rapid improvements in processor and interconnection technology. I/O performance thus becomes ever more critical, to avoid becoming the bottleneck of system performance. In this paper we provide an introduction to I/O architectural issues in multiprocessors, with a focus on disk subsystems. While we discuss examples from actual architectures and provide pointers to interesting research in the literature, we do not attempt to provide a comprehensive survey. We concentrate on a study of the architectural design issues, and the effects of different design alternatives.

1 INTRODUCTION

As high-performance computers continue their stunning increases in computational performance, fueled in part by rapid improvements in processor and interconnection technology, I/O becomes an increasingly important component of overall system performance. This fact is especially true for parallel computers, where the combination of numerous processors boosts computational performance, leaving I/O as the serial bottleneck that limits scalability [2]. Indeed, many scientific and commercial applications have tremendous I/O requirements [20], both for moving data in and out of the parallel computer, as well as for manipulating datasets too large to fit in primary memory. Thus, it is imperative that a parallel I/O architecture is provided to support the parallel computational architecture.

The author is funded by NSF under grant number CCR-9404919, and by NASA Ames under agreement number NCC 2-849. The author speaks for himself and not for NSF or NASA.