A First Implementation of Parallel IO in Chapel for Block Data Distribution[1]

Rafael LARROSA [a], Rafael ASENJO [a] Angeles NAVARRO [a] and
Bradford L. CHAMBERLAIN [b]

[a] *Dept. of Compt. Architect. Univ. of Malaga, Spain.*
{rafael, asenjo, angeles}@ac.uma.es

[b] *Cray Inc., Seattle WA, USA.*
bradc@cray.com

Abstract. This paper presents our preliminary implementations of parallel IO routines in Chapel, a high-productivity parallel language for large-scale systems. The IO functions are implemented using standard Chapel features, taking POSIX as the IO middleware layer and Lustre as the target parallel file system. In this work, we focus on the Chapel Block data distribution, for which we propose different algorithms to perform the file read and write operations. We pay particular attention to the collective writing operation, for which we evaluate different strategies. Our algorithms take into account some of the Lustre file system parameters that can have an impact on the write operation performance. Through our evaluation we find that a more careful selection of the stripe pattern (a feature of the Lustre system), as well as the appropriate selection of the number of IO processes that perform the IO calls, should be taken into account to better tune the performance. In addition, we compare our implementations with the ROMIO library, the de facto parallel IO standard, finding that we achieve comparable results with our higher level approach.

Keywords. Chapel, Parallel IO, Block data distribution, Object-based file system

1. Introduction

One of the challenges when dealing with large-scale computing, is that the bottleneck moves from data processing to data availability. The kind of applications targeted to large scale systems tend to be very data intensive, requiring adequate IO capabilities. For reasons of portability and convenient data representations, HPC systems are often deployed with a wide variety of software stacks to handle the IO. At the top end, scientific applications perform IO through middleware libraries such as Parallel NetCDF [7], HDF [9] or MPI IO [6]. However, these approaches still pose challenging low level details to the programmer, which we think is not appropriate for high productivity computing. On the other hand, towards the bottom of the stack, parallel file systems directly serve IO requests by striping file blocks across multiple storage devices. Obtaining a good collective-IO performance across many processes on top of these software layers

[1]This material is based upon work supported by Spanish projects: TIN2006-01078 from Ministerio de Ciencia e Innovación, and P08-TIC-3500 from Junta de Andalucía.

is a complex task. It requires not only awareness of the process data access patterns, but also an understanding of the entire software stack, especially the behavior of the underlying file system. We believe this is a daunting task from which the user should be insulated. In other words, the parallel language should provide the high level constructs to perform the IO operations, whereas the runtime, using locality hints provided by the user (e.g. data distribution across the nodes in the system), should be responsible for dealing with the middleware support for the corresponding file system.

In this paper we address the first implementation (to our knowledge) of parallel IO operations in the context of the Chapel high productivity parallel language. Currently, Chapel only supports basic sequential IO operations. Our new parallel IO functions are implemented using standard Chapel features, taking POSIX as the IO middleware layer and Lustre as the target parallel file system. First we present the Chapel IO interface and then we propose different algorithms to perform the read and write operations for the Chapel Block data distribution. In the paper, we pay particular attention to the collective writing operation, for which we evaluate different algorithms. Through our evaluation we find that the file stripe pattern (a feature of the Lustre system) and the number of IO processes that carry out the IO operations should be carefully configured in order to achieve optimal performance. A more detailed analysis of the impact of these parameters and how to automatically incorporate their tuning in our Chapel IO algorithms, is beyond the scope of this paper and it will be explored in a future work. Finally, after coding the equivalent IO write operation on top of the ROMIO library (an optimized MPI-IO framework) we found that not only is the resulting ROMIO code more complex, but it also performs slightly worse than the Chapel version.

2. Background

2.1. *Chapel programming issues*

Chapel [3] is a parallel language under development at Cray Inc. as part of the DARPA High Productivity Computing Systems (HPCS) program to improve the productivity of programmers. Chapel supports a Partitioned Global Address Space (PGAS) memory model permitting variables to be accessed where stored. This is, Chapel provides a global-view of data structures, as well as a global-view of control. In Chapel, *locales* are the abstract representation of the target architecture for the purpose of reasoning about locality. Accesses of a task to data are called *local* if the task and data are mapped to the same locale, or *remote* otherwise.

One of the most distinguishing features of Chapel is its support for standard and user-defined data distributions that permit advanced users to specify their own global-view array implementations. To this end, Chapel incorporates the novel concept of *domain map* [4], which is a recipe that instruct the compiler how to map the global view of data to node's memory. Chapel is designed to include a library of standard domain maps to support common distributions. Currently, Chapel's standard domain maps library supports Block and Cyclic distributions. Block-cyclic and Replicated distributions are under development. In this paper, we have focused on the 1D and 2D Block distribution for which we have developed a new interface to perform parallel file read/write operations.

2.2. File system issues

When performing IO operations, several components in the system software stack, particularly in the file system layer, have to be considered to meet the demands of parallel applications, such as those coded in Chapel. Parallel file systems such as PVFS [2], GPFS [10] and Lustre [1] distribute portions of a file across different servers. When multiple clients read and write a file, coordination between the activities becomes essential to enforce a consistent view of the file system state. In particular, the enforcement of such consistency can conflict with performance and scalability goals. For instance, PVFS provides high-bandwidth access to IO servers without enforcing overlapping-write atomicity, leaving it entirely to the applications (or runtime libraries, such as MPI IO) to handle such consistency requirements, at the cost of being less productive from the programmer point of view. On the other hand, GPFS and Lustre enforce byte-range POSIX consistency, hiding this consistency problem from the application. Typically protocols based on distributed locking are used in these latter cases to ensure consistency.

In this paper we use Lustre [1] as the target parallel file system for which we perform the evaluation of our parallel file read/write Chapel operations. We select Lustre as the target parallel file system because is an open source project which is used in half the top 30 supercomputers in the world (including the first, second and third one as of the June 2011 top500.org list). Lustre presents an object-based storage architecture. A typical Lustre system consists of a *Metadata Server* (MDS) which manages the names, directories and metadata in the file system; and of *Object Storage Servers* (OSS) which provide file IO services. The metadata is stored in the *Metadata Target* (MDT). Each OSS can be associated with several *Object Storage Targets* (OST) which are the interface to each exported volume or LUN (Logical Unit Number). The sources of parallelism in the Lustre file system are due to: i) once metadata is obtained from the MDS, subsequent IO operations are directly served by the OSTs to the client; and ii) files are usually striped across several OSTs [5].

The Lustre file system incorporates a consistency protocol to arbitrate accesses to data and meta-data objects: it must ensure that the state of the system is consistent in the presence of multiple updates and prevent stale data from being read when the system implements client-side caching. In Lustre, each OST acts as a lock server for the objects it controls [1]. The locking protocol requires that a lock be obtained before any file data can be modified or written into the client-side cache.

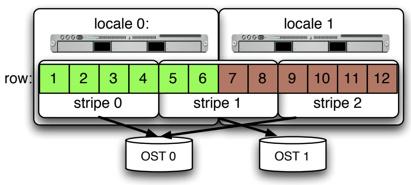
Another issue that has to be considered when developing the parallel IO interface, is that we have assumed that data will be stored in a single file. We have adopted a distribution-agnostic view of the file storage, so that reads can be done for any number of locales and for any kind of data distribution in Chapel, independently of the original data distribution for the data originally written to the file. This approach also permits a file's contents to be accessed by other external programs that process or visualize them. In our Chapel IO implementation, all of these issues are not exposed to the programmer. In other words, the parallel IO operations in Chapel are going to be described just in terms of the array that needs to be read or written, and all the decisions about stripe size, stripe count, stripe offset and other optimizations are left to the internal Chapel runtime.

3. Parallel Chapel IO Interface

Our new Chapel interface for performing collective write operation is shown in Figure 1(a). After the first three lines with the declaration of the Block distributed $n \times n$ matrix `A` in lines 1–3, we write matrix `A` by: a file object (outf) creation, file open, write and close (lines 4–7). A file read operation is similarly coded.

```
1 var Space = [1..n, 1..n];
2 var Dom: Space dmapped Block(Space);
3 var A: [Dom] real;
4 var outf = new file(fname, ...);
5 outf.open();
6 outf.write(A);
7 outf.close();
```

(a)



(b)

Figure 1. File write interface in Chapel (similar for reading) and write serialization in Lustre for non-aligned stripe boundaries

As a first step, and due to the read operation having no consistency problems, our read operation is collectively done in parallel. That way, data will be served from the OSTs to all the locales concurrently to exploit the maximum parallelism available at the IO path. However, the parallel writing operation is more challenging than the read due to performance degradations that arises when two different tasks end up writing to the same stripe at the same time. For example, this can happen if two locales are writing concurrently to two neighbor regions of the same row as shown in Figure 1(b). In this figure, half a row is stored in locale 0 while the other half is in locale 1. Assuming a stripe size of 4 array elements, locales 0 and 1 share the stripe 1, so write accesses are not aligned on stripe boundaries. Because of Lustre's consistency protocol, the writes to stripe 1 will be serialized leading to performance degradation [8]. An additional source of performance degradation is due to contention in the access to the OSTs. For instance, in Figure 1 in which the file is striped over two OSTs (stripe factor of 2), stripe 0 and 2 contend for the same OST 0.

In this work we evaluate different alternatives to alleviate these two penalizing factors. On the first hand, to avoid concurrent access to the same stripe, the stripe size can be configured to achieve stripe-aligned access to the data. However, this is not always possible since Lustre limits the stripe size to a multiple of 64KB. A second possibility is to implement a two-phase collective write in order to minimize the number of concurrent accesses to neighboring sections of data, as we do in our algorithms that will be explained in the next subsections.

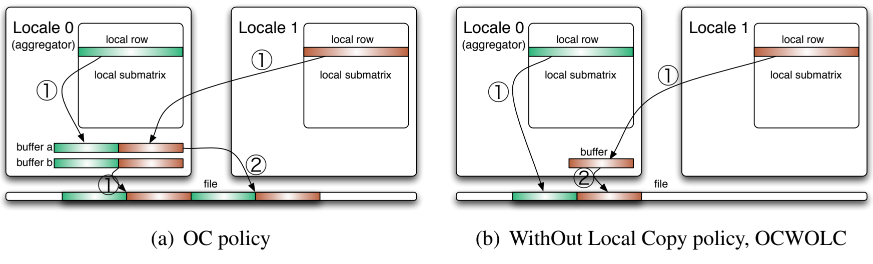
On the other hand, to reduce the contention in the OSTs, there are several alternatives. As a baseline, we can just allow one locale to write its local block at a time. We call that a "serial IO" (SIO) implementation, although there will still be parallel writes if the file is striped across several OSTs. A better way to exploit the parallel file system without stressing the OSTs too much is by performing the parallel write only from a subset of the locales (the *aggregators*). Depending on the number of aggregators, the contention in the OSTs can be tuned. We call this approach "*Partially Parallel IO*" (PPIO) to distinguish it from the extreme case: the "fully *Parallel IO*" (PIO) in which all the locales write to all the OSTs (which represents the worst case regarding OST contention).

For writing a 1D BlockDist Chapel matrix to disk, each locale can write its own data. In this case in which aggregation is not considered, we have implemented two possible alternatives: SIO (*Serial IO*), and PIO (*Parallel IO*). The first one is the most basic algorithm which lets each locale write its data to disk sequentially (SIO). With this approach we only take advantage of the parallelism offered by the file system. On the other hand, in the Parallel IO approach, PIO, all the locales write to the file at the same time. This alternative tries to further stress the OSS service nodes and OSTs.

In the 2D BlockDist Chapel case, concurrent access to the same file stripe can occur. Therefore the "Parallel IO" (PIO) alternative in which all the locales write their local matrices at the same time performs very poorly if the local rows are not aligned to the stripe size. To avoid this problem, we propose alternatives based on aggregating the local rows so as to remap data from the 2D distribution to a 1D one. Therefore, in a two dimensional mesh of locales, the locales in the first column of the mesh will be the aggregators, and therefore they will take care of aggregation and IO duties. Since Chapel follows a row-wise ordering for array storage, this selection of aggregators supports the accumulation of large block of rows that can later be written to disk in a single IO operation. This way not only is the number of IO operations reduced, but also the stripe-boundary misalignment problem is minimized due to a single locale being responsible for writing a large block of consecutive stripes.

The first aggregation policy we propose is identified as "Non-Overlapping Communications" (NOC). The idea is that each aggregator first allocates a buffer capable of storing nr matrix rows. Then, the aggregators gather the non-local row sections from the other locales and arrange the data in the buffer with all the nr rows as a sequential block. This matrix block is later written to disk in a single IO operation.

An optimization of the previous alternative is based on a double buffer approach that can overlap the collective communication with the IO operation. We call this approach the "Overlapping Communication" (OC) policy. As in NOC, buffers can accomodate nr matrix rows. In Figure 2(a) this strategy is sketched for nr=1 and a 2×2 mesh for which we only show locales 0 and 1, since locales 2 and 3 behave similarly. In ① "buffer a" is being arranged while "buffer b", which already contains the previous row, is being written to the file. After finishing the IO write which is usually slower, in ② "buffer a" should be ready for the next IO, and now "buffer b" will simultaneously be gathering the next row. In the experimental section we have evaluated the bandwidth achieved by this strategy for different values of nr.



(a) OC policy (b) WithOut Local Copy policy, OCWOLC

Figure 2. Overlapping Communication policies.

In both previously described policies, aggregators are moving local rows from their local matrices to the buffer. We also wanted to check whether it is worthwhile to save this local movement. For that reason we have also tested an “Overlapping Communication WithOut Local Copy” (OCWOLC) alternative that is illustrated in Figure 2(b). Basically, aggregators overlap the IO write of the local row with the gather of the remaining data of the row in a buffer, ①. Later this buffer is written to disk, ②. Thus, not only is local data movement avoided, but also the buffer size is reduced (it only needs to hold a single non-local row). Note that two different IO calls per matrix row are always necessary, since accumulating more than one non-local row makes no sense. Hence nr is always 1.

These aggregating policies can be combined with a serial or parallel invocation of the aggregators. That is, if aggregators write to both halves of the file at the same time, then we have a “Partially Parallel IO” (PPIO) implementation, and depending on the aggregation policy we have PPIO_NOC, PPIO_OC and PPIO_OCWOLC. In the “Serial IO” SIO approach, the aggregators write in turns, starting from locale 0, and so we will consider NOC and OC aggregation policies leading to SIO_NOC and SIO_OC algorithms.

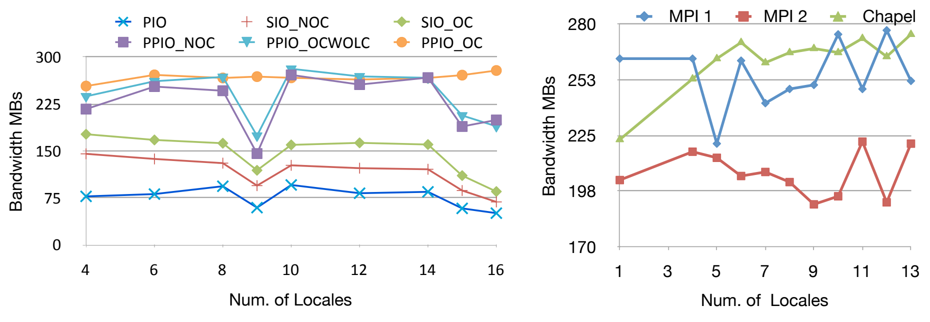
4. Evaluation

We have conducted our experiments on “Crow”, a system that can be considered representative of small and medium size supercomputers optimized for memory-intensive and/or compute-biased HPC workloads. This system is a Cray XT5 with 20 8-core compute nodes clocked at 2.8 GHz. The operating system is SLES 11 modified by Cray and called CLE (Cray Linux Environment). The Lustre file system is Lustre-Cray/1.8.2 and there is a MDT (MetaData Target) and three OSTs (Object Storage Targets).

We have used one simple synthetic benchmark, coded in Chapel, that performs a file write operation of a square $n \times n$ Block-distributed matrix. The size of the written file was 8 Gbytes (32,768 $\times$ 32,768 64 bit entries). We chose this large file size to facilitate the transfer of big data blocks, and to avoid (to some extent) the influence of the cache system. We have set the stripe size to 64KBytes, the stripe count to 3 and the stripe offset to 0. Regarding the performance metric, we have measured the effective bandwidth (MB/sec) of the file write operation. We performed each test at least 7 times, and computed the average effective bandwidth and the standard deviation.

In a first set of experiments we evaluate the performance of the writing algorithms proposed in Section 3. We start by comparing the performance for the basic algorithms proposed for the 1D BlockDist Chapel case. In this case, the PIO version systematically achieves better performance: nearly 20% improvement on average, but we observed more disparity in the measured values. For instance, the standard deviation for the PIO bandwidth was around 5.7% while for the SIO algorithm it was less than 2%.

Next, we evaluate the algorithms proposed for the 2D BlockDist Chapel case: SIO_NOC, SIO_OC, PPIO_NOC, PPIO_OC, PPIO_OCWOLC and PIO. Figure 3 (a) shows the average bandwidth for different numbers of locales. In these experiments, the number of rows written by each IO call is $nr = 1$. The PPIO versions systematically achieve better performance than the SIO ones. However, the fully parallel version PIO, in which all the locales concurrently perform the IO call, is the one that obtains the worst results. This is due to the non-aligned accesses and OST contention problems discussed in Section 3. Also, for the NOC and OCWOLC versions, the bandwidth is smaller as each row of the array is divided among more locales, which happens for 9, 15 and 16 locales ($3\times$ 3, 5 $\times$ 3, $4\times$ 4 meshes respectively). This is because these aggregation polices are



(a) Writing Bandwidth for Chapel versions

(b) PPIO_OC Chapel vs. MPI

Figure 3. Writing bandwidth for 8GB matrix and comparison with MPI IO

not as effective as the OC one. In fact, on average, SIO_OC improves the performance by 26% when compared with SIO_NOC while PPIO_OC improves the bandwidth by 11% and 17% when compared with PPIO_OCWOLC and PPIO_NOC, respectively.

By increasing the size of the buffer where we aggregate the rows ($nr > 1$), we can further improve the performance: i) because we reduce the number of IO calls to the file system, and therefore we reduce some overhead; and ii) because by transferring bigger blocks we can take advantage of the aggregate bandwidth in the network. We found that although the optimal number of aggregated rows depends upon the exact array size, number of locales, and number of OSTs used, the aggregation of $nr = 16$ rows worked reasonably well in our experiments.

In the next experiment we compare the performance of the Chapel PPIO_OC algorithm (with $nr = 16$) vs. the bandwidth we obtain with MPI IO (*de facto* standard for doing parallel IO) when writing a two dimensional Block distributed matrix of 32,768 $\times$ 32,768 elements (8 Gbytes). In both cases, the stripe size was 64KBytes and the stripe count 3, so we were using all OSTs available. It is noteworthy that the MPI-IO code equivalent to the Chapel one shown in Figure 1(a) requires nearly 100 code lines. The bandwidth results are shown in Figure 3 (b). In particular, the MPI IO code is based in ROMIO [11], a high performance implementation of MPI IO that has been developed and maintained at Argonne National Laboratory (ANL). We chose the Cray optimized version of ROMIO as the parallel IO implementation with which we compare because it is highly optimized for the computer system used. In order to consider the different heuristics that MPI IO can use in our target system we show two results for MPI IO, one setting MPICH_MPIIO_CB_ALIGN to 1 (MPI 1), and another setting it to 2 (MPI 2). In particular, the two-phase redistribution strategy [12] (MPICH_MPIIO_CB_ALIGN = 2) is the default system choice, yet it is slower in our tests than the rest of the algorithms.

From the results, we find that, on the target platform, the ROMIO version for collective writing generally performs slightly worse than our PPIO_OC Chapel version. Since ROMIO is designed for a general data distribution, more communications are necessary for the aggregation phase, compared with our more restricted approach (Block distribution only). On the other hand, with Chapel it is possible to implement runtime optimizations to better exploit the available IO resources. We also plan to extend the parallel IO implementation for Chapel's other standard data distributions, such as Cyclic and Block-Cyclic.

5. Conclusions

In this paper, we have presented a new Chapel IO interface to perform parallel read and write file operations. Our first implementation is based on POSIX IO and takes Lustre as the target parallel file system. We have proposed different algorithms to perform the IO operations, taking the Chapel Block data distribution as a case study. In particular, through experimental evaluation on Crow, a Cray XT5 computer, we have investigated the Lustre file system parameters that can have an impact in the performance of the different Chapel write implementations that we have proposed, finding that: i) the stripe pattern chosen when creating a file (i.e. the stripe size, the stripe factor and the stripe offset), as well as ii) the number of IO processes that perform the IO calls, should be taken into account in our algorithms to carefully tune the performance.

In addition, we have compared our Chapel implementation with the current state-of the art MPI IO implementation (ROMIO), finding that our PPIO_OC writing algorithm performs slightly better in the target architecture. Moreover, the nearly 100 code lines needed in MPI to write an array have been replaced by merely 7 lines of Chapel code. We believe that Chapel not only eases the specification of parallel IO, but can also provide a better runtime optimization of the IO implementation. The reason for the latter is that the Chapel compiler can produce a more specialized runtime for the IO operations depending on the used data distributions. As a future work we will explore how to extend user defined domain map interface to permit author to leverage work without replicating too much code. We will also study how to tune the different IO pattern alternatives (the assignment of OSTs to locales) to reduce IO contention.

References

[1] P.J. Bramm. The Lustre storage architecture. http://www.lustre.org.

[2] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Linux Showcase and Conference*, pages 391–430. MIT Press, 2000.

[3] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *Intl. Journal of High Performance Computing Applications*, 3(21):291–312, August 2007.

[4] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, David Iten, and Vassily Litvinov. Authoring user-defined domain maps in chapel. In *CUG conference*, June 2011.

[5] Phillip M. Dickens and Jeremy Logan. A high performance implementation of MPI-IO for a Lustre file system environment. *Conc. and Computation: Practice and Experience*, 22(11):1433–1449, 2010.

[6] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*. High Performance Computing Center Stuttgart (HLRS), September 2009.

[7] P. Kevin, B. Ron, and W. Joshua. Cplant: Runtime system support for multi-processor and heterogeneous compute nodes. In *IEEE Intl. Conference on Cluster Computing*. IEEE Computer Society, 2002.

[8] W.-K. Liao, A. Ching, K. Coloma, Alok Choudhary, and L. Ward. An implementation and evaluation of client-side file caching for MPI-IO. In *IPDPS 2007*, pages 1 –10, march 2007.

[9] D.B. Michelson and A. Henja. A high level interface to the HDF5 file format. Technical report, Swedish Meteorological and Hydrological Institute (SMHI), 2002.

[10] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 231–244, 2002.

[11] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, 1999.

[12] Rajeev Thakur and Ewing Lusk. An abstract-device interface for implementing portable Parallel-I/O interfaces. In *Symp. on the Frontiers of Massively Parallel Computation*, pages 180–187, 1996.