Asynchronous Collective Output With Non-Dedicated Cores

Phil Miller, Shen Li, Chao Mei
Department of Computer Science
University of Illinois at Urbana Champaign
Urbana, Illinois 61801
Email: {mille121,shenli3,chaomei2}@illinois.edu

***Abstract*—Parallel applications are evolving to place larger demands not just on computation and network capabilities, but on storage systems as well. Storage hardware has scaled to keep up, but the software to drive it must evolve alongside to service this increased potential. This paper presents an output forwarding middleware for message-driven parallel applications written in Charm++. This layer directs IO operations across the entire system to a designated subset of processors in order to minimize contention and overheads. Our implementation is distinctive in that these processors are not dedicated to this task, but can still contribute to the computational task. Other processors need not block while waiting for the designated IO processors to become ready or make progress. Using this new layer, we demonstrate speedups of $1.5 - 2.5\times$ in the popular scientific code NAMD over its previous parallel output implementation, along with reduced sensitivity to IO subsystem parameters.**

I. Introduction

Existing parallel software is evolving to place increasing demands on I/O performance of parallel computers, and new applications that deal in immense volumes of data are rapidly emerging. As software scales, application developers are forced to use progressively more sophisticated techniques to avoid I/O operations becoming a performance bottleneck [1]. Many libraries and frameworks have arisen to address these needs (e.g. [2], [3], [4], [5], [6]).

Because of the need for high-level information and wide-ranging coordination to achieve high performance, these implementations typically focus on optimizing *collective* I/O operations. By doing so, they set aside chunks of time in which all processors will focus on I/O, with computation to resume after some operation is complete. During this time, a subset of processors will actually interact with the underlying filesystem, while the others will wait to communicate the data involved. The time spent waiting is time in which those processors are not doing productive computation.

In large cluster and supercomputer installations, application code runs on dedicated nodes. There may be operational interference from the node OS or daemons, but other applications will not contend for computational resources once a job is running. On Bluegene systems (which isolate distinct job partitions) and full-bandwidth switched networks, applications are similarly not competing for shared communication resources. However, no such isolation is present in the storage subsystems of these computers. The potential contention can make application I/O performance unpredictable and highly variable. In a synchronized collective operation, the delay of a single participant can degrade the performance of the entire application, magnifying the impact of that variability.

The issues of wasted resources and unpredictable performance can be mitigated by performing I/O operations asynchronously with application logic. Instead of blocking while data makes its way to or from the storage system, a program can continue executing independent work, masking the unpredictable and difficult to control latency until the operation completes.

In this paper, we describe a fully asynchronous collective I/O library implemented in the Charm++ runtime system [7]. Our library designates a subset of cores to interact with the filesystem as each operation is started, but does not dedicate the time of those cores or any others to the I/O task. Data is transfered with minimal synchronization, but still in a fashion suited to avoiding contention, while application code continues to run.[1]

We evaluate our work by adapting the popular NAMD molecular dynamics code to use our library. NAMD's existing release carries its own implementation of parallel I/O, aimed at limiting per-node memory requirements and increasing throughput over a single-process-I/O implementation. However, its design experiences substantial contention, and implements avoidance mechanisms that are ultimately performance-limiting. Our library achieves effective bandwidth increases of $1.5 - 2.5\times$ over that implementation.

II. Mapping Data to Processors

According to measurements of Lustre by Cray and ORNL [8], up to a few thousand processes, writing to a separate file per processor can attain full bandwidth. However, beyond a cutoff point, performance in this approach falls off. On Lustre, there are many storage nodes, but only a single centralized meta-data server. As the number of processes performing IO increases, this creates contention at the metadata server, and eventually creates contention on the storage nodes as well.

Thus, scalability demands limiting the number of processes making filesystem calls. In order to accomplish this, data from

[1]This library is distributed as part of the Charm++ programming environment, available for download from http://charm.cs.illinois.edu/

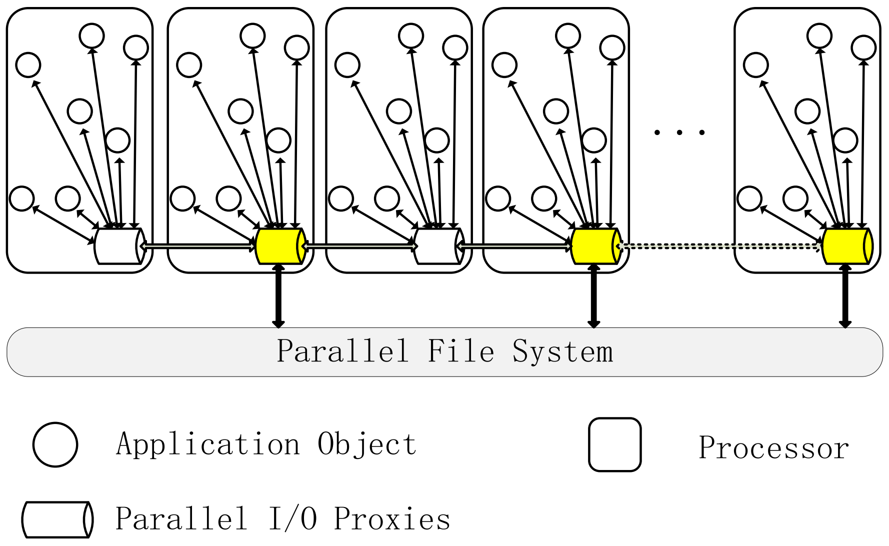


Fig. 1. Architecture: Application objects communicate with local IO proxy objects, which exchange data amongst themselves and interact with the filesystem.

the overall set of processes must be funneled to a smaller operating set. Once data's position in output is determined, it is straightforward to map this to a subset of processes. Simple experiments contrasting 'round-robin' and 'long-stripe' arrangements have found that performance is generally higher with each output process writing long contiguous stripes. More detailed work [3], [9] exploits knowledge of how the underlying filesystem will map file data to storage nodes to minimize the number of interactions between compute and storage nodes, and the contention within those nodes.

The *stripe* is the smallest unit of data in which a storage subsystem happily deals. There is a tradeoff between amortizing overhead in transfers (favoring a larger stripe size), and exposing more parallelism and hence offering higher potential bandwidth (favoring a smaller stripe size). Keeping latency and interference low also leans toward smaller stripes. For any given parallel filesystem, the number of storage nodes is an upper bound on the available parallelism in data transfers to or from storage.

In this work, we do not directly consider how to optimize the number of processors to use for I/O, leaving that as a parameter to be set by application code. Given the number of working processors and an amount of data to be written, we can compute a reasonable chunk size to be assigned to each processor. These chunks should generally come out to multiples of the underlying storage system stripe size. Since only one processor will operate on the stripes that make up any given chunk, there is no risk of contention for locks from storage nodes.

III. Design

We have implemented an output forwarding layer for parallel applications written on top of the Charm++ parallel runtime system. As shown in Figure 1, the application's work is divided among several objects on each processor. The objects communicate with each other by asynchronous method invocation in an active-messaging scheme. Each processor can be executing work in one object while transmitting or receiving messages for others. This overlap of communication and computation is important for high performance.

Normal application development practices in Charm++ suggest the use of numbers of objects that correspond to a 'natural' decomposition of the problem being solved or the system being simulated, without direct regard for the number of processors in the system. The runtime can then map these objects to optimize for load balance and communication patterns. However, for an I/O library, the many considerations of process- and node-level buffering, connection and contention limiting, and others drive toward an implementation that explicitly considers how many processors are available and how work is distributed among them.

The central element of our implementation is a one-per-processor collection of objects (known as a *Chare Group* in Charm++) that we will interpose between application-level objects that own the data and the underlying parallel filesystem. Groups communicate by the same asynchronous mechanisms as other Charm++ objects, but are addressed by the processor on which they reside, rather than by an abstract index. The 'Parallel I/O Proxy' objects of Figure 1 are implemented as a chare group, instantiated at application startup. The interface to this group, including the message entry points and sequencing logic, can be seen in Figure 2. The corresponding implementation code can be seen in Figure 3.

When the application wants to perform output, it signals this collection to prepare for that process (`Manager::prepareOutout()`). The group signals its readiness to the application through a callback (`ready`), through which it delivers an opaque handle that the application should pass in along with the data. That handle acts as a 'parallel file descriptor,' allowing the proxy objects to look

```
group Manager {
  entry void prepareOutput_central(std::string name, size_t bytes,
                                   CkCallback ready, CkCallback complete,
                                   Options opts);
  entry void prepareOutput_distrib(int handle, std::string name,
                                   size_t bytes, Options opts);
  entry void prepareOutput_readied(CkReductionMsg *m);

  /// Serialize setting up each file, so that all PEs have the same sequence
  entry void run() {
    for (filesOpened = 0; true; filesOpened++) {
      if (CkMyPe() == 0)
        when prepareOutput_central(std::string name, size_t bytes,
                                   CkCallback ready, CkCallback complete,
                                   Options opts) atomic {
          // Default setting and error checking omitted

          nextReady = ready;
          thisProxy.prepareOutput_distrib(nextHandle, name, bytes, opts);
          files[nextHandle] = FileInfo(name, bytes, opts);
          files[nextHandle].complete = complete;
        }

      when prepareOutput_distrib[filesOpened](int handle, std::string name,
                                              size_t bytes, Options opts) atomic {
        if (CkMyPe() != 0) {
          files[handle] = FileInfo(name, bytes, opts);
        }

        // Open file if we're one of the active PEs
        if ((CkMyPe() - opts.basePE) % opts.skipPEs == 0 &&
            CkMyPe() < lastActivePE(opts)) {
          int fd = open(name.c_str(),
                        O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
          if (-1 == fd)
            CkAbort("Failed to open a file for parallel output");
          files[handle].fd = fd;
        }

        contribute(CkCallback(CkIndex_Manager::prepareOutput_readied(0),
                              thisProxy[0]), filesOpened);
      }

      if (CkMyPe() == 0)
        when prepareOutput_readied[filesOpened](CkReductionMsg *m) atomic {
          delete m;
          nextReady.send(nextHandle++);
        }
    }
  };

  entry void write_forwardData(int handle, const char data[bytes],
                               size_t bytes, size_t offset);
  entry void write_dataWritten(int handle, size_t bytes);
};
```

Fig. 2. The interface definition and coordination code for the I/O proxy group

up the parameters (which processors, stripe size, offsets, etc.) associated with each particular target file. Once the system is ready, the application objects will pass their portions of the data to the local element of the group (`Manager::write()`), which will redistribute the data according to the plan and perform write operations as whole stripes are assembled.

A. Control Flow

For each file, our IO forwarding layer takes as parameters a stripe size, a number of processors to use, a starting processor, and a numeric separation between those processors. It applies these parameters in a straightforward fashion to direct data provided by the application to the processor that will eventually pass it to the filesystem.

When application code is ready to write data to persistent storage, it calls the IO forwarding layer with a file name, size, and the parameters listed above. It also passes two callbacks, for signaling readiness and completion. These callbacks can represent a function to call or, more commonly, the target of a subsequent message send. The IO forwarding layer com-

```
struct Options {
  /// How much contiguous data (in bytes) should be assigned to each active PE
  size_t peStripe;
  /// How much contiguous data (in bytes) should a PE gather before writing it out
  size_t writeStripe;
  /// How many PEs should participate in this activity
  int activePEs;
  /// Which PE should be the first to participate in this activity
  int basePE;
  /// How should active PEs be spaced out?
  int skipPEs;
};

struct FileInfo {
  std::string name;
  Options opts;
  size_t bytes, total_written;
  int fd;
  CkCallback complete;
};

class Manager : public CBase_Manager {
  /// Application-facing methods, invoked locally on the calling PE
  void prepareOutput(const char *name, size_t bytes,
                     CkCallback ready, CkCallback complete,
                     Options opts = Options()) {
    thisProxy[0].prepareOutput_central(name, bytes, ready, complete, opts);
  }

  void write(int handle, const char *data, size_t bytes, size_t offset) {
    Options &opts = files[handle].opts;
    do {
      size_t stripe = offset / opts.peStripe;
      int pe = opts.basePE + stripe * opts.skipPEs;
      size_t bytesToSend = min(bytes, opts.peStripe - offset % opts.peStripe);
      thisProxy[pe].write_forwardData(handle, data, bytesToSend, offset);
      data += bytesToSend;
      offset += bytesToSend;
      bytes -= bytesToSend;
    } while (bytes > 0);
  }

  /// Internal methods, used for interaction among IO managers across the system
  void write_forwardData(int handle, const char *data, size_t bytes, size_t offset) {
    // Omitted error checking and interruption handle code for simplicity
    pwrite(files[handle].fd, data, bytes_left, offset);
    thisProxy[0].write_dataWritten(handle, bytes);
  }

  void write_dataWritten(int handle, size_t bytes) {
    files[handle].total_written += bytes;

    if (files[handle].total_written == files[handle].bytes)
      files[handle].complete.send();
  }

  int filesOpened;
  int nextHandle;
  std::map<int, FileInfo> files;
  CkCallback nextReady;

  int lastActivePE(const Options &opts) {
    return opts.basePE + (opts.activePEs-1)*opts.skipPEs;
  }
};
```

Fig. 3. The implementation of the I/O proxy group

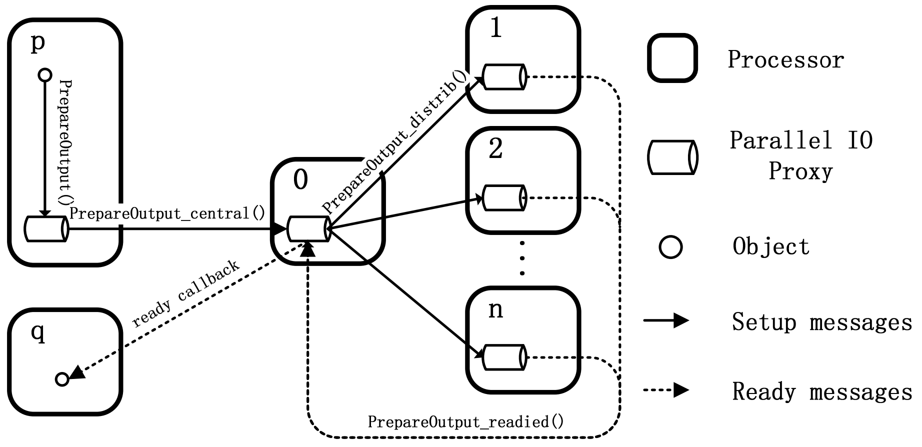


Fig. 4. Flow of execution to prepare for writes to a file

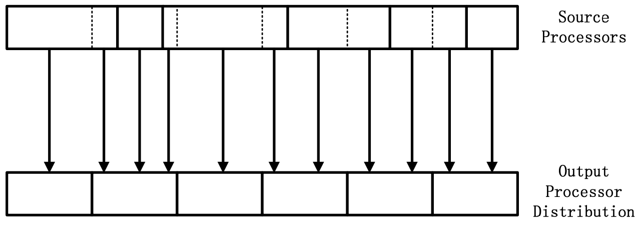


Fig. 6. Mapping chunks of a file from different processors to whole stripes

municates internally to ensure that all processors know how that file is to be handled. Every processor acknowledges these instructions by contributing to a parallel reduction operation. When the reduction reaches the root processor, it triggers the ready callback, passing a handle object used to look up the file. These steps are illustrated in Figure 4.

As shown in Figure 5, the application code sends this handle to any objects with data to be written. Each of these objects call the IO proxy object local to the processor on which they reside, passing the handle along with their data buffers. The local proxies forward data to proxies on other processors as needed, via the `Manager::write_forwardData()` method. From the application's point of view, the forwarding process completes immediately, and buffers can be reused or discarded without delay. When an IO proxy has written all the data it is responsible for, it notifies the master. When the master has heard from the IO proxies on every processor writing data, it signals the completion callback.

B. Striping

Given the substantial documented effect of matching application writes to filesystem striping, the optimization priority is to constrain each output processor's writes to distinct stripes. For each segment of data that the application wishes to write, we compute which stripes it intersects based on the stripe size parameter (Figure 3 line 33), as shown in Figure 6. Then, for each stripe, we compute which processor is responsible for writing that stripe to the filesystem (line 34). We send messages containing each stripe chunk to the IO proxy on its respective processor (line 36). When the IO proxy receives the data, it passes it to the filesystem (line 46), secure in the knowledge that it will not contend with other processors for access to that stripe, and notifies the root processor (line 47) so that we can tell when the process is complete (lines 50-55).

IV. Evaluation

In order to evaluate the effectiveness of our approach, we have adapted NAMD, an existing Charm++ application, to use our output framework. NAMD is a popular ($> 40,000$ users) code for classical (i.e. Newtonian) simulation of large (up to 100 million atoms) biomolecular systems at atomic scale. Its behavior of periodically dumping the state of the computation (i.e. the positions and velocities of all particles) to disk is typical of many scientific applications.

In its present version, NAMD contains mechanisms to do parallel output. Specifically, the particles' positions are collated on a subset of the processors, which coordinate to write their data to the filesystem. This coordination amounts to a control on how many of them will actually make `write()` calls simultaneously. This scheme was implemented to enable scalability to large target systems without exceeding the available memory on individual nodes. It makes some expedient choices to attain acceptable performance, and leaves several knobs for the user to set 'appropriately'. It takes no account of the type or parameters of the filesystem on which it runs.

NAMD's parallel output scheme [10] introduces a layer of indirection between the application objects and the IO processors, to balance the IO and memory load among the processors performing IO. Depending on how many processors are involved in output and when they perform their operation, performance can vary wildly. Figure 7 shows execution traces of NAMD for simulating a 2.8-million-atom virus molecule on

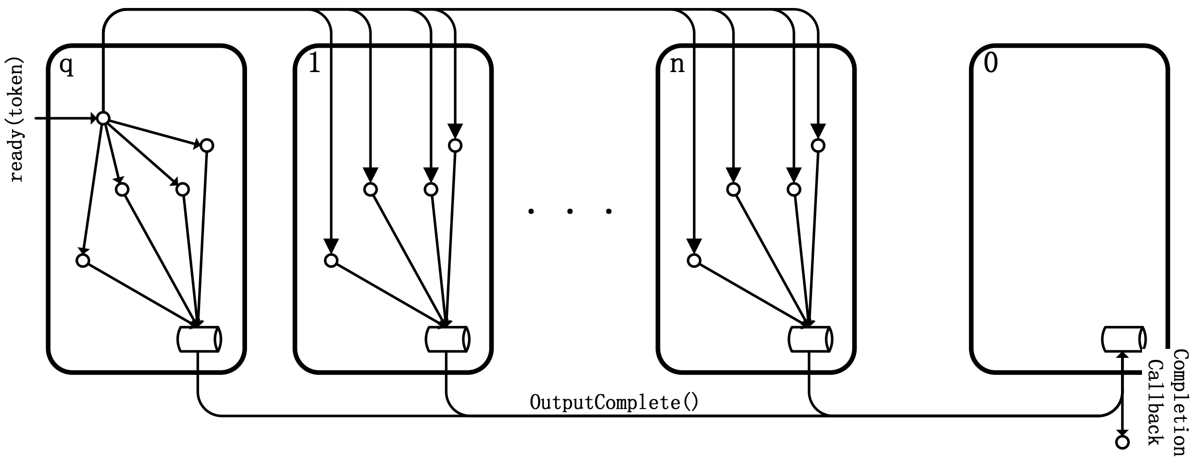


Fig. 5. Flow of execution once a file is ready to be written

32 nodes of Jaguar PF (using 10 cores/node and one output processor per node) before our modifications. In the figure, the dark yellow bars represent the time spent on output, while bars in other colors represent the time of different types of force computation. Figure 7(a) shows each processor writing to a separate file, a scheme not supported by surrounding tools or NAMD's own input reader, as a point of reference; Figure 7(b) shows all writes going to one file one after another while figure 7(c) shows all writes going to one file but simultaneously. Note that these traces are presented on approximately the same timescale, illustrating that an incautious 'all-at-once' mode of output can be disastrous.

The disappointing performance in NAMD's working output code can be explained by the various kinds of contention that it creates in the storage subsystem. Among the processors performing output, data is divided among them evenly, for the sake of load balance. Since the simulation data set does not precisely divide into neat power-of-two size chunks, this means that there is no alignment of each processor's writes to filesystem or storage hardware boundaries. Additionally, because each processor's responsibility crosses stripe boundaries, there are substantially more connections to the storage nodes than are necessary.

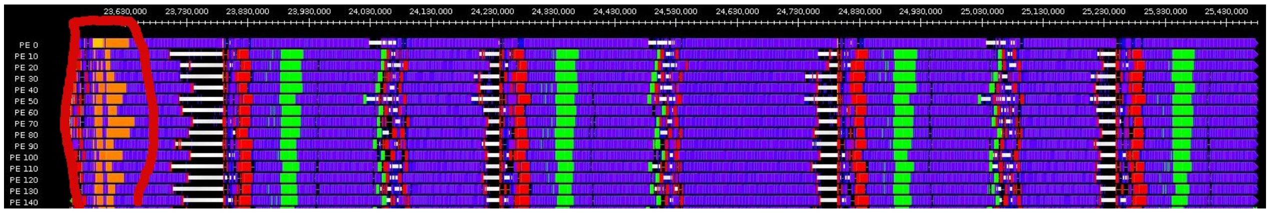
In Figure 8, we present measurements of the average bandwidth obtained by coordinate outputs from a 10-million atom simulation on Jaguar, a Cray XT5 with a Lustre-based storage system. Each output step wrote 283MiB of data, and the data presented are the average of 8 output steps each. We can see that NAMD's existing implementation peaks around 220MiB/s using 36 out of 480 processors for output, and falls off rapidly. In contrast, when NAMD is adapted to use our library, it sees substantially better performance on both 240 and 480 processors. Measured bandwidth ranges from 305MiB/s to 580MiB/s. Moreover, our attained bandwidth does not decay as rapidly as one leaves the 'sweet spot' of output processor count. Thus, it is less reliant on the user to choose a good value for the number of output processors. Finally, because it successfully uses a larger number of cores, we are able to run much larger simulations (with a correspondingly larger memory footprint) without output times increasing sharply.

Measurements by the machine's operators [8] suggest that the underlying Lustre filesystem can offer bandwidth of at least 1.5 GiB/s to this many processors. Thus, there is still a long ways to go in minimizing time spent performing output.

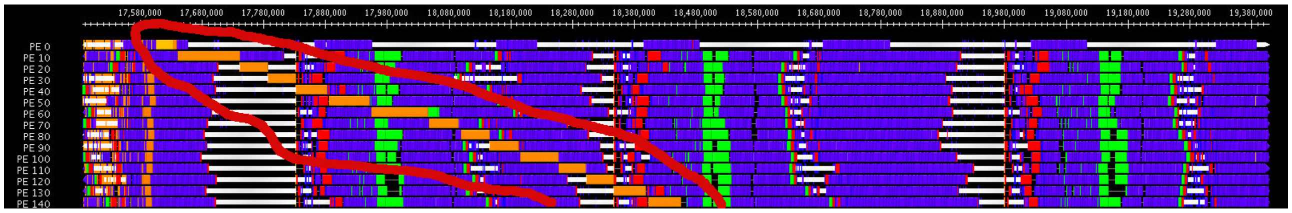
V. Related Work

Collective I/O and asychronous I/O are necessarily popular techniques used in large parallel applications to prevent I/O becoming a performance bottleneck [1]. There is plentiful effort to provide efficient implementations of the standard MPI-IO API (e.g., [2], [3]). Additional efforts are directed at optimizing the use of lower-level APIs by data formatting libraries such as the Hierarchical Data Format (HDF) [11] and the Common Data Format (CDF) [12]. Following Cormen's dicta about storage transparency [13], these various efforts are slowly exploiting the various bits of knowledge that are latent in the filesystem and data formatting implementations.

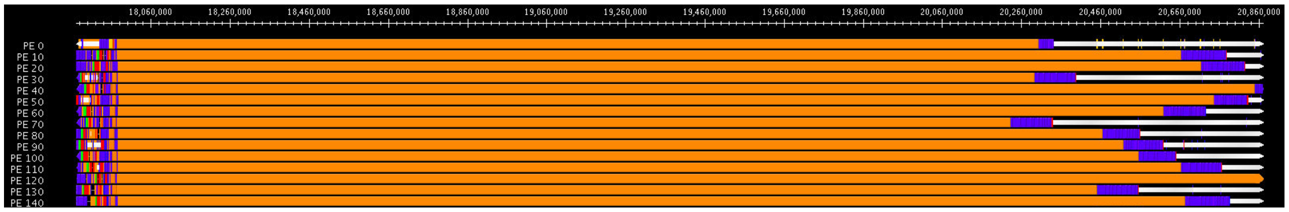
Lang et al. [14] conduct several experiments to show how the changes on configurations influence the aggregated bandwidth of Intrepid (the Argonne Leadership Computing Facility BG/P system) under different application I/O benchmarks. Their work convinces us that, besides the applications' I/O pattern, the system performance also heavily rely on tuning parameters. Thus, researchers have to expend extra effort to tune them when these data formats are employed in demanding applications. For instance, Howison et al. [5] proposed several detailed modifications on HDF5 to make it perform better above the Lustre [15] file system as configured on a trio of Cray supercomputers. For example, they align all objects in a file over a particular size threshold since most parallel file systems performs best when data access falls on chunk



(a) Multiple files, dummy scheme



(b) Single file, one at a time



(c) Single file, simultaneously

Fig. 7. Execution timelines of unmodified NAMD performing an output step generated by the PROJECTIONS tool (with comparable time scales).

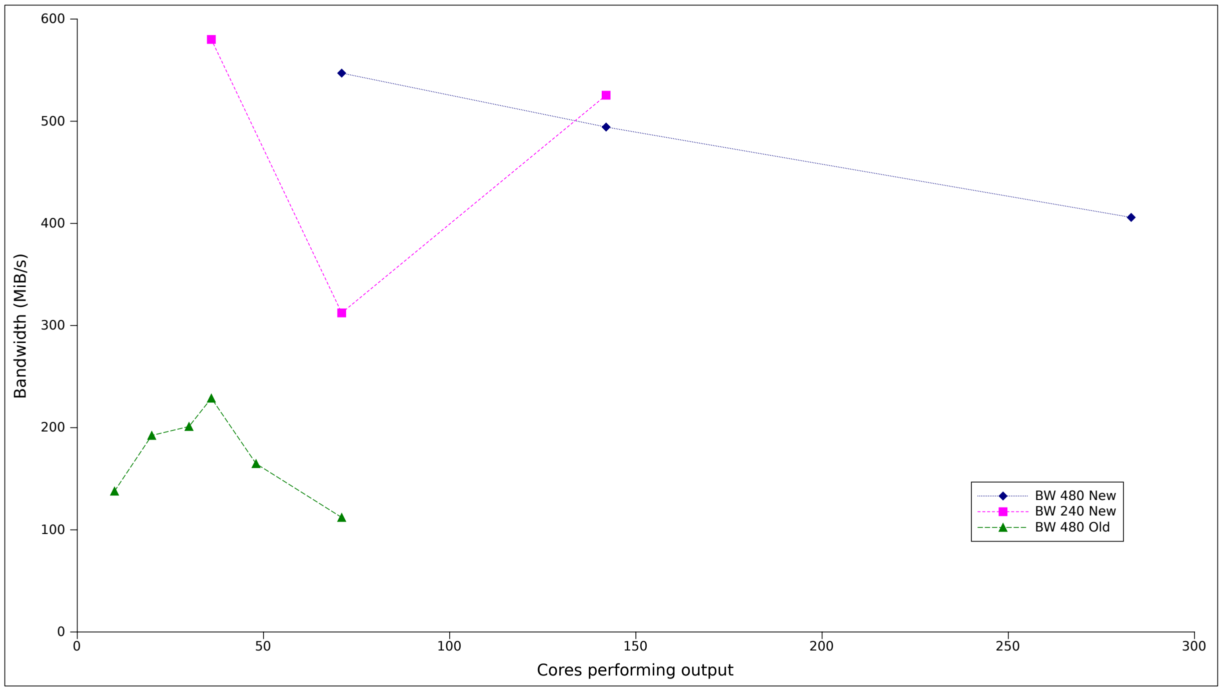


Fig. 8. NAMD's output bandwidth on 240 and 480 processors of Jaguar and Kraken, with varying numbers of processors touching a single output file.

boundaries. This benefit of alignment motivates the present work.

IBM's GPFS [16] relies heavily on its distributed lock manager and byte-range locking policies to achieve good performance. The distributed lock manager hands out lock tokens to particular nodes, and those nodes do not need to request new locks for successive accesses to the same disk object. The byte-range locking allows multiple nodes writing to different parts of the same file simultaneously by negotiating for the token of required range. As long as the access patterns are predicable, their negotiation protocol will be able to minimize conflicts. The access pattern has significant dependence on how files are organized which GPFS itself cannot control. Thus, there is no guarantee for the performance of prediction algorithm.

The striping technique is also employed by Lustre [15] which, however, focuses on other benefits presented by striping, e.g., maximum file size is not limited by the size of a single target and I/O bandwidth to a single file can aggregate the bandwidth of all the objects that make up that file. PVFS [17] allows users to specify stripe size of each file. But their minimum granularity is one file, i.e., all chunks of the same file share identical size.

Much software that uses or coordinates parallel I/O uses a small subset of compute nodes as *I/O delegates* that combine data and perform interactions with low level APIs. Asynchronous I/O allows I/O workload to overlap with computational workload instead of impeding computing. Besides the above two techniques, Ohta et al. [6] employ pipeline techniques to further overlap I/O requests at the network and filesystem layers. The present work is distinct from these other systems in that the processors performing I/O need not be dedicated to that task, but can also contribute to the overall computational effort of the application. By reducing the necessary number of distinct types of components, this sharing can also potentially simplify system management and reduce overall costs.

The Damaris project [4] describes the use of dedicated cores in multicore nodes to offload and accelerate I/O. Given dedicated cores that were spending substantial time idling, they explore various ways to use the 'loose' processing power. By compressing the data to be stored, they increased the effective bandwidth available to the application, and reduced load on the storage system. By performing some analysis and visualization calculations while a simulation is running, they shorten time to result and enable earlier detection of failing runs that might otherwise consume substantial computer resources. In our setting, the Charm++ programming system, these can be executed as additional asynchronous tasks that get slotted in with the rest of an ongoing computation, sharing the system's computational resources in a flexible fashion to achieve an application's goals. For instance, the LiveViz library assembles visualization data from a running program, and the Projections performance tracing framework can be configured to compress logs as they're written when the user expects that to be beneficial.

The layout-aware collective I/O system [3] matches the delegated compute nodes directly to particular storage nodes, in order to minimize contention. This is straightforward on filesystems like Lustre, that simply round-robin stripes of each file across a set of storage nodes, because the mapping requires only simple arithmetic. On more variable distributed filesystems, such as Ceph [18], determining which delegate is responsible for transferring which bits may become more complicated.

VI. Future Work

Beyond the simple stripe-separation optimization described here, there are a few other techniques implemented by other systems that would be suitable here.

A. Stripe Buffering

Storage systems at many levels gain performance when data is written in whole blocks, lines, or stripes at a time. One frequent source of this constraint is the potential need to read some data in order to write other data nearby. For instance, RAID5 and RAID6 arrays need to keep parity blocks up to date when storing data blocks. If a store is presented as an entire stripe, spanning as many disks as make up the array, the parity can be calculated wholly from the input data, with the old bits on disk making no contribution.

Once data is forwarded to a designated processor, which knows how large a chunk it will write, there is a decision to be made: dispose of the data immediately, or buffer it in order to present fewer, neater requests to the storage system? The tradeoffs here are not entirely clear. By writing immediately, the system achieves some degree of pipelining, and storage systems with sufficient write cache can still avoid the read-modify-write slowdown. Buffering presents fewer requests to the filesystem, and hence incurs lower overhead. If the buffer is sized and aligned correctly to write as a unit, the storage system may not need to cache it at all. However, it also increases memory usage on processors servicing I/O requests, which may be undesirable or potentially unacceptable.

B. Pipelining

As mentioned above, servicing write requests immediately has the potential effect of pipelining a processor's total I/O load. However, this is a very crude way to do so. In particular, if a processor receives a message carrying a large quantity of data to be written, it might try to write that entire blob at once. The latency incurred by doing so could be substantial, blocking the processor from doing other work or receiving other messages in the meantime.

Given a willingness to buffer at least some data, each processor could write some fixed fraction of its data as that portion is available. For instance, if a processor were responsible for 64MiB of data, it could write it in 4 16MiB chunks. This is large enough to attain good performance from lower-level systems, without introducing needless delays.

C. Mapping

As discussed in section II, there are substantial considerations involved in deciding which processor should be responsible for any given byte or block of data. Liao and Choudhary [9] describe a set of algorithms that are tuned to the caching and synchronization structure of the popular Lustre and GPFS systems. These algorithms could be adapted to benefit from an asynchronous communication structure.

Past work on parallel I/O has treated the processors in a job uniformly, when that may not be the case. Particularly in the torus networks of Cray and Bluegene systems, some compute nodes may be much closer to the I/O service nodes than others. In concert with choosing a suitable number of I/O processors and the distribution of data among them, we can choose to select which processors should fill that role based on their proximity to the system's connection to storage. The Charm++ runtime system embeds support for these optimizations, and previous work has demonstrated that they can be applied to great effect [19], [20], [21], [22].

VII. Conclusion

In this paper, we have described the implementation and evaluation of an IO optimization framework for applications built for the Charm++ parallel runtime environment. This framework reduces contention for filesystem resources by limiting the number of processors that interact directly with the filesystem. It then avoids contention among those active processors by making processors responsible for stripes of a file, which can be sized according to the filesystem's parameters. Unlike existing IO frameworks, the processors assigned IO tasks are not set aside to perform just those tasks, but can also participate in the computation occurring on the rest of the system. In adapting NAMD, an application that already implemented a parallel output scheme of its own, to the new library, we saw speedups of $1.5 - 2.5\times$, along with reduced sensitivity to the exact parameters guiding the IO process and and increase in the number of processors that can feasibly participate in IO.

VIII. Acknowledgments

Development and performance measurement for the work presented in this paper was done with the use of Jaguar at Oak Ridge National Laboratory, under the DOE INCITE allocation program, and of Kraken under NSF TeraGrid allocation TG-ASC050039N.

References

[1] A. Shoshani, S. Klasky, and R. Ross, *Scientific Data Management: Challenges and Approaches in the Extreme Scale Era*. Proceedings of SciDAC, 2010.

[2] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," *Frontiers of Massively Parallel Processing, Symposium on the*, vol. 0, p. 182, 1999.

[3] Y. Chen, H. Song, R. Thakur, and X.-H. Sun, "A layout-aware optimization strategy for collective i/o," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, (New York, NY, USA), pp. 360–363, ACM, 2010.

[4] M. Dorier, "Src: Damaris - using dedicated i/o cores for scalable post-petascale hpc simulations," in *Proceedings of the international conference on Supercomputing*, ICS '11, (New York, NY, USA), pp. 370–370, ACM, 2011.

[5] M. Howison, Q. Koziol, D. Knaak, J. Mainzer, and J. Shalf, "Tuning hdf5 for lustre file systems," *Workshop on Interfaces and Abstractions for Scientific Data Storage September*, September 2010.

[6] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa, *Optimization Techniques at the I/O Forwarding Layer*. IEEE International Conference on Cluster Computing, 2010.

[7] L. Kalé and S. Krishnan, "Charm++ : A portable concurrent object oriented system based on C++," in *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993.

[8] "I/o tips – lustre striping and parallel i/o." http://www.nics.tennessee.edu/io-tips, retrieved 2011-03-30.

[9] W. keng Liao and A. Choudhary, "Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC08), Austin, Texas*, November 2008.

[10] O. Sarood, "Benfits of parallelizing io of large data-intensive applications with a case study of namd," Master's thesis, Computer Science, University of Illinois at Urbana-Champaign, 2009.

[11] T. H. Group, *Hierarchical data format version 5*. 2000-2010. http://www.hdfgroup.org/HDF5.

[12] "Common data format (cdf)." http://cdf.gsfc.nasa.gov/.

[13] T. H. Cormen and D. Kotz, *Integrating Theory and Practice in Parallel File Systems*. DAGS/PC Symposium on Parallel I/O and Databases, 1993.

[14] S. Lang, P. Carns, K. Harms, and W. Allcock, *I/O performance Challenges at Leadership Scale*. Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing), 2009.

[15] "Lustre." http://www.lustre.org/.

[16] F. Schmuck and R. Haskin, *GPFS: A Shared-Disk File System for Large Computing Clusters*. Proceeding of the Conference on File and Storage Technologies, Berkeley, CA: USENIX, 2002.

[17] P. H. Carns, R. B. W. B. L. III, and R. Thakur, *PVFS: A Parallel File System For Linux Clusters*. Proceedings of the 4th Annual Linux Showcase and Conference, 2000.

[18] S. A. Weil, S. A. Brandt, E. L. Miller, and D. D. E. Long, *Ceph: A Scalable, High-Performance Distributed File System*. Symposium on Operating Systems Design and Implementation (OSDI), USENIX, 2006.

[19] A. Bhatelé and L. V. Kalé, "Benefits of Topology Aware Mapping for Mesh Interconnects," *Parallel Processing Letters (Special issue on Large-Scale Parallel Processing)*, vol. 18, no. 4, pp. 549–566, 2008.

[20] A. Bhatelé, L. V. Kalé, and S. Kumar, "Dynamic topology aware load balancing algorithms for molecular dynamics applications," in *23rd ACM International Conference on Supercomputing*, 2009.

[21] A. Bhatele, G. Gupta, L. V. Kale, and I.-H. Chung, "Automated Mapping of Regular Communication Graphs on Mesh Interconnects," in *Proceedings of International Conference on High Performance Computing (HiPC)*, 2010.

[22] A. Bhatele and L. V. Kale, "Heuristic-based techniques for mapping irregular communication graphs to mesh topologies," in *Proceedings of Workshop on Extreme Scale Computing APplication Enablement - Modeling and Tools*, September 2011.