Using I/O Servers to Improve Application Performance on Cray XT™ Technology

Thomas Edwards, Kevin Roy
Cray Centre of Excellence for HECToR

Amdhal's Law proposes that parallel codes are combinations of parallel and serial tasks. In many cases these task are inherently parallel and can be decomposed and performed asynchronously. Each task operates on a dedicated subset of processors with highly scalable tasks operating on very large numbers of processors and less scalable tasks (like IO) operating on a smaller number. By moving to a Multiple Instruction Multiple Data paradigm codes can achieve greater parallel efficiency and scale further. This paper specifically addresses the implementation and experiences of optimising an application important to HECToR by asynchronously writing output data via a set of dedicated I/O server processors.

Keywords: I/O Server, Lustre, SHMEM, HELIUM, Asynchronous

Introduction

Jaguar, a Cray XT5™, was the first system to sustain a petaflop on a real world application and has continued to demonstrate high levels of performance on real applications. This comes about from the ever increasing number of cores that successive generations of MPP architectures offer and the most successful applications have at their heart algorithms that will scale to very large numbers of cores.

The I/O systems of these architectures are not built with the same high levels of parallelism as the computational components. While the computational section may have hundreds of thousands of cores, the I/O system will typically bep constructed from hundreds of service threads.

Larkin (1), Crosby (2), Shan and Shalf (3) and have demonstrated that the Lustre filesystem used by Cray XT™ achieves near peak performance when using only a fraction of the total computational cores available, with performance increasingly only gradually as more processors are added.

Previous papers have concentrated on methods and techniques that improve the peak performance of applications writing data to disk, however the issue still remains that in the majority of applications many thousands of cores are involved in I/O with no improvement in performance. Instead this paper outlines a technique for running I/O operations in parallel by using a subset of dedicated processors. Though not necessarily universally applicable to all forms of I/O in all types of application, it proves most suitable to applications performing “checkpoint” operations where a large amount of data is written to disk in a short period.

Many applications adopt the Single Instruction Multiple Data (SIMD) paradigm to solve problems in parallel. Each processor performs the same actions but on different sections of the dataset, occasionally the master process may perform some tasks on their own. This tends to favour a phased approach, where sections of computation are followed by sections of communication and/or I/O operations, with all processors involved in all aspects.

This paper advocates adopting a Multiple Instruction Multiple Data (MIMD) paradigm where processors are divided into subsets that are dedicated to discrete sets of tasks, specifically splitting computation from I/O. This allows computation to continue simultaneously with I/O, improving the overall performance of the application.

This paper first considers HELIUM which is heavily used on the HECToR Cray XT5h™ service. The code simulates the interaction of high intensity laser pulses with the electrons of a helium atom by solving the time-dependent Schrödinger equation. It is a classic boundary decomposition problem, except that the symmetry of the problem allows users only to solve the upper triangle of the two-dimensional domain. The code scales exceptionally well and exhibits weak scaling up to hundreds of thousands of processors, this type of scaling generates large amounts of data, especially during the regular checkpoints. Efforts have been undertaken by the Cray Centre of Excellence for HECToR to optimise HELIUM’s I/O performance on very large numbers of processors.

Overlapping Computation and I/O

HELIUM, like many HPC applications, is an iterative code that performs regular number of time steps and then writes the results to disk. The code performs regular file-per-process checkpoints to disk as part of the scientific output and as a point to restore the data from. During output operations the computation is suspended until all data is written to disk in the form of individual files for each processor. As all writing occurs almost simultaneously on each processor, it can place significant demand on the lustre file system.

The I/O in Helium is implemented efficiently with large writes to individual files, this requires little buffering, file locking or management. It achieves a high percentage of peak I/O bandwidth which means traditional approaches to I/O optimization will not make any worthwhile improvements. However as it is a weak scaling code, as the process count increases the checkpoint volume increases as well but after only a few hundred processors the I/O system reaches its peak performance. This results in checkpoint operations becoming an increasingly large component of the overall job time as the application scales.

There is sufficient time spent in computation between checkpoints that writing data to disk could be overlapped with the continuing computation. This would offer a performance advantage to the majority of checkpoints where there is further computation, but not the final checkpoint. It would also require data to be cached in memory and written out asynchronously, without stalling the computation.

There are several potential methods of doing this, the Fortran 2003 standard provides a method for asynchronous I/O however it is not universally implemented across compilers and may revert to synchronous operation. The MPI-IO interface supports asynchronously write operations, however the simulation does not currently use MPI-IO and the output format does not fit well with the I/O model. Alternatively it is possible to spawn a small number of additional nodes as I/O servers to perform the writes to disk. Though this adds an additional degree of complexity to many applications it ensures the code will work with every Fortran compiler and allows for more sophisticated serial processing of data as it is written out. To maintain the efficiency of the code only a small number of I/O servers should be spawned with a high ratio of compute processes to I/O processes.

Implementation in MPI

Data to be written to disk has to be cached in a separate memory location to prevent it being overwritten during by subsequent computation. This has to be memory from the compute node, not only because the data volume is larger than the buffer capacity of an individual node, but also because it is counter-productive to direct large volumes of data through a single link in a short time as introduces a bottleneck. Instead, once each compute processor's data has been safely cached locally it can be individually transferred to its I/O server to be written to disk. This could be done by initiating an asynchronous `MPI_ISend` call from each compute node to its I/O server and having the I/O Server process the receives in turn, however with potentially large numbers of compute nodes to each I/O server this could generate large numbers of simultaneous outstanding point-to-point messages that could potential degrade performance. Instead the I/O Server must be able to limit the number of messages it receives.

Instead of the compute nodes pre-emptively sending messages to the I/O server, they must wait for a "ready" message from the I/O server before sending the data. By controlling the number of "ready" messages it sends, the I/O server limits the number of simultaneous incoming messages. However, this requires the compute server to acknowledge and act upon the "ready" message. This could be achieved by blocking the compute node, which is no better than the original scheme, or by regularly polling for the ready message. This introduces further complexity to the application as regular checks have to be introduced which potentially introduce load imbalance, however the poll has to be frequent enough that the delay between the I/O server sending a "ready" message and the compute node acting upon it is small enough that it does not delay the overall output of data.

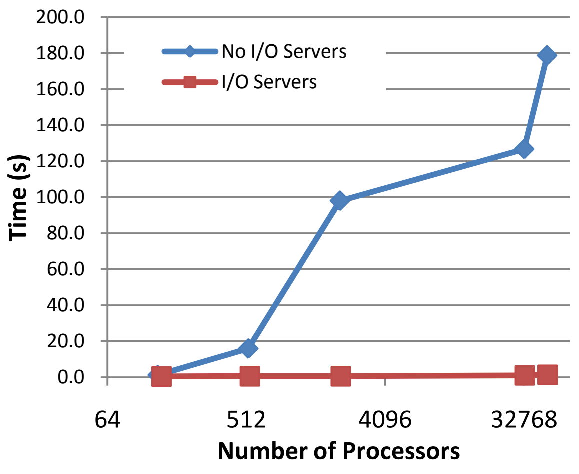

Figure 1: Waiting time on compute processes per checkpoint.

MPI communicators are very useful for co-ordinating the subsets of processors. `MPI_COMM_WORLD` can be split into a compute communicator which replaces its parent on the compute side and an I/O communicator. Separate communicators that group each I/O Server as rank 0 and its compute clients also proved useful. I/O servers were selected at even intervals across the ranks of the Global Communicator to prevent clustering of I/O servers on single multi-core nodes.

Performance on the XT5™

# Compute (+ I/O Servers)	CP time Compute only	CP time + Servers
136(+8)	1.2	0.4
528(+12)	15.9	0.6
2080(+32)	98.0	0.6
32896(+332)	126.7	1.0
46360(+464)	178.7	1.4

Table 1: Wallclock time for compute nodes to perform a checkpoint with and without I/O servers

Experiments were run on a Cray XT5™ system to evaluate the performance of this scheme. Each run used an additional 1% of processors acting as I/O servers (rounded up to a whole number of nodes). Experiments ran in an intensive I/O mode, running for 250 computational timesteps and producing a checkpoint every 50 timesteps.

Figure 1 shows the average wallclock time that was spent by compute nodes waiting for checkpoints to complete with and without I/O servers. The figures are the averaged over the 5 checkpoints and the volume of data written by each processor is constant at 44.25 MB per processor per checkpoint. The figures clearly show that the measured time in writing data to disk is significantly reduced using the I/O servers.

Figure 2 plots the average time between completed checkpoints, so includes the time of computation and check pointing. Though with smaller numbers of processors (below 4096), using I/O servers is slight slower than using larger numbers, as the volume of data increases with the increased number of processors using I/O servers offers an ~11% in runtime performance for a ~1% cost in processors.

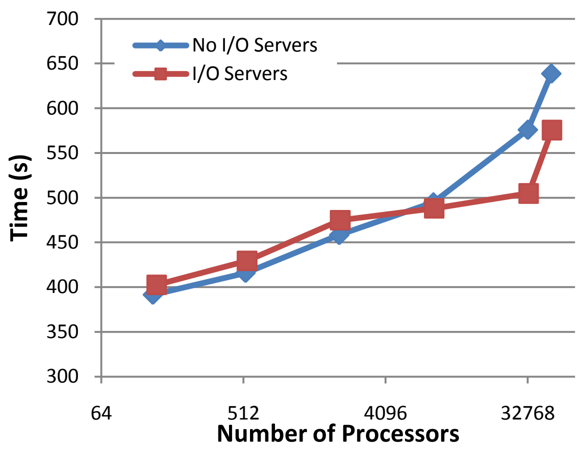

Figure 2 Average measured wall clock time between completed checkpoints.

Efficiency of the I/O Servers

The I/O servers report the percentage of time they were busy processing compute node data compared to sitting wait for data to be received. Table 2 presents these figures for the Cray XT5™ runs and shows that the I/O servers are busiest when operating on large numbers of processors, but still spend almost three quarters of their time idle.

# Compute (+ I/O Servers)	Busy
136(+8)	1.3%
528(+12)	3.3%
2080(+32)	7.6%
32896(+332)	26.1%
46360(+464)	23.3%

Table 2: Measured load on the I/O Server during operation

If ultimate efficiency is required, then the number of I/O servers could be reduced to keep the server as close to 100% busy as possible, however this increases the risk that the checkpoint data is incomplete should the job stop unexpectedly. To achieve the greatest I/O performance a larger number of I/O servers could be used, but would be sitting idle once the data had been written to disk.

Using SHMEM For I/O Server Communication

For the I/O server to efficiently transfer data the number of polls the compute node makes, N_{poll}, which is the product of the frequency of the poll, ω_{poll}, and the compute time, $t_{compute}$, must be greater than the number of compute nodes per IO server $\frac{N_{compute}}{N_{IO}}$. i.e

$$\omega_{poll} \times t_{compute} = N_{polls} > \frac{N_{compute}}{N_{IO}}$$

The initial I/O server implementation places the user is in control of the frequency of polls for new messages which reduces the potential number. An alternative approach is to use the SHMEM API's remote push and get functionality.

Rather than pushing the data from the compute node to the I/O server, instead the compute processor copies the output data to a symmetrically allocated buffer and sets a local tag declaring the data is ready to collect. The I/O server constantly polls the tag on the compute node watching for a state change, when it occurs it pulls the data from the compute node to the I/O server and saves it to disk. Once this is complete it changes the notification area on the compute processor to confirm receipt of the data.

This approach offers numerous advantages over the MPI message based approach. The requirement for the compute server to check for messages from the I/O server is removed from the user and placed on the underlying SHMEM implementation. This means there can be a much higher frequency of polling and on upcoming network architectures, hardware support for the transfer. The number of messages going to the I/O server is still limited which prevents any performance problems associated with overloading. The disadvantage is that code becomes a hybrid communications code, and while integration between the two libraries is excellent on the Cray XT™, users are required to understand two different paradigms.

Shmem vs MPI Performance

Comms	Avg T/S	Cost in I/O	% in I/O	Real Cost
MPI	9.31s	2.33%	14%	0.32%
Shmem	9.72s	0.19%	25%	0.05%

Table 3 Performance Comparison MPI vs Shmem

Using SHMEM to provide I/O servers requires fewer changes to the source code of the compute nodes compared to using MPI. However, it will potentially increase the number and frequency of messages between the I/O server and the compute nodes during idle periods. In the initial SHMEM implementation the I/O server continually pings the compute nodes, looking for a change of state. By timing in more detail the lengths of individual time steps and boundary exchanges both when the I/O server is transferring data and when it has finished it is possible to quantify the cost the I/O server implementation has on the compute stages.

Table 3 shows the results of high resolution runs that were performed on 4056 Cray XT6™ compute nodes and with 48 I/O servers on a file system with far fewer OSTs. These results show that straight off the MPI implementation outperforms SHMEM, which is to be expected on this architecture. They also show that it does take additional time to perform computation while the I/O server is active, the increased length of timestep is labelled the "Cost in I/O". This value is quite small, but is measurable, however because the majority of time steps are not during the I/O the total increase in run length, the "Real Cost", is also quiet small, alleviating concerns that the I/O servers are displacing time spent doing I/O rather than masking it.

The SHMEM implementation places a more uniform load on the compute nodes as it constantly polls them for status updates. It may be beneficial on the current architecture to introduce less frequent polling during idle periods to prevent performance interference. It is also expected with the move to the Gemini network architecture, which more efficiently supports many concurrent small messages, that the SHMEM implementation would see a significant improvement.

Conclusions

Writing to disk can constitute a significant proportion of runtime in applications that generate large volumes of data. While the number of parallel cores available to an application for computation is very large, the I/O performance will reach its peak when driven by far fewer processors. Asynchronous I/O offers the opportunity to overlap computation and communication with writing data to disk. I/O servers are a technique for implementing asynchronous I/O using standard networking constructs and offer additional flexibility for data structures that do not fit the standard paradigms. It also offers opportunities to perform additional serial processing of data that might otherwise require secondary post-processing.

The technique has been demonstrated with two communication libraries, MPI and SHMEM on the Cray XT5™ and Cray XT6™ architectures and demonstrates near complete masking of I/O from computation without significantly affecting the performance of the computational sections. This significantly improves the effective performance of codes with significant output requirements and allows them to scale to the largest numbers of processors.

Using MPI to implement the I/O servers requires greater modification of existing code, but in the current implementation offers slightly improved performance over using SHMEM. Further performance improvements are expected on the next generation of Cray architectures which offers increased bandwidth, reduced latency and native support for remote memory transfers.

About the Authors

Kevin Roy is a member of the Cray Centre of Excellence for HECToR. He joined the Centre of Excellence in October 2007 from the Cray benchmarking team. Prior to this he was part of the UK National Supercomputing facility, CSAR.

Thomas Edwards joined the Cray Centre of Excellence for HECToR in September 2009 after completing a PhD in HPC Optimisation of Fusion Applications at EPCC, University of Edinburgh. Prior to this he worked on the Ported Unified Model for the UK Met Office.

References

1. *Practical Examples for Efficient I/O on Cray XT Systems.* **Larkin, Jeff.** ORNL : s.n., 2009. CUG 2009 Proceedings. p. 12.

2. *Perforance Characteristics of the Lustre File System on the Cray XT5 with Respect to Application I/O Patterns.* **Crosby, Lonnie D.** ORNL : s.n., 2009.

3. *Using IOR to Analyze the I/O performance for HPC Platforms.* **Hongzhang Shan, John Shalf.** Seattle : CUG, 2007.