Hiding I/O using SMT on the ARCHER2 HPE Cray EX system

Shrey Bhardwaj*, Paul Bartholomew† and Mark Parsons‡
EPCC, University of Edinburgh
Edinburgh
Email: *shrey.bhardwaj@ed.ac.uk, †p.bartholomew@epcc.ed.ac.uk, ‡m.parsons@epcc.ed.ac.uk

***Abstract*—This paper tests the hypothesis that hyper-threaded cores on the ARCHER2 Cray HPE machine can enhance the effective I/O bandwidth for applications conducting both I/O and computational operations. We present the I/O framework, iocomp, that supports both synchronous and asynchronous I/O approaches, while abstracting away complexities associated with multiple I/O back-ends such as MPIIO, HDF5, and ADIOS2. In the asynchronous case MPI is used to split the processes into dedicated compute and I/O servers, which are mapped to various core mappings using SLURM options to run the I/O processes on either oversubscribed, hyperthreaded, or dedicated cores. Data is then sent from the client processes to these I/O cores asynchronously, and the performance of different workloads writing their data using iocomp is compared when using such dedicated I/O servers against conducting synchronous I/O operations where the same process is responsible for compute and I/O operations. The STREAM and HPCG benchmarks are adapted as examples of client processes to perform computation, adding a step to write their data through the iocomp library. The present results demonstrate that adding a separate set of nodes to run a dedicated I/O server improves the performance for the STREAM and the HPCG benchmarks for all I/O back-ends tested. However, hyperthreads result in lower or at best comparable performance, depending on which I/O back-end is used.**

I. Introduction

With the widening gap between compute, memory, and Input/Output (I/O) rates in High Performance Computing (HPC) systems, achieving high performance can be limited by the I/O bottleneck. For example, ARCHER2 [1], an HPE Cray EX system with a total of 5860 compute nodes, each with 128 cores of dual AMD EPYC 7742 64-core 2.25GHz processors, has a peak theoretical DRAM bandwidth of up to 204.8GB/s [1] but only achieved an I/O bandwidth of 2.82GB/s as per the IO500 lists at SC22 [2]. Considering the disparity between the memory and I/O bandwidths it is clear that I/O may ultimately prove the limiting factor in performance.

An HPC system will typically employ (multiple) parallel file system(s), such as Lustre in the case of ARCHER2. These enable users to interact with the file system as though it were logically a single disk but are implemented upon parallel physical hardware for greater bandwidth. However, whereas a user may increase the scale of parallelism in their program execution by requesting more compute nodes, the degree of parallelism available in the I/O system is a relatively small, essentially fixed quantity. For example, each parallel file system on ARCHER2 has 12 Lustre stripes (roughly analogous to 12 disk servers); when compared to the nearly 6,000 compute nodes available the disparity in compute and I/O capabilities becomes apparent. It should also be noted that the I/O system is a shared resource on the machine and therefore subject to contention between users.

If I/O bandwidth creates a bottleneck in program execution, and users can't (significantly) increase the amount of I/O hardware then we should aim to hide the I/O step, performing it asynchronously and allowing useful work to proceed. One approach to this is to split the resources in a job between processes dedicated to compute and I/O tasks. This splitting means requesting more resources than would be dictated by compute requirements alone, or reducing the portion allocated to compute for a given job allocation size. A potential solution is to increase efficiency by employing idle resources within a core by using Simultaneous Multi-Threading (SMT) [3]. This technology allows a single physical core to simultaneously fetch and execute instructions from multiple threads. It is hypothesised that by using SMT, an unused hyper-thread can perform the I/O operations while the first thread is performing computation to exploit unused resources and increase efficiency. Thus, the cost of writing to disk could be

effectively "hidden" by sending the data asynchronously to the I/O threads while the compute processes finish their computational workload.

This work tests the idea of exploiting hyper-threads to perform asynchronous I/O tasks by comparing against several different approaches. As a reference approach, compute and I/O tasks are performed in sequence by a given process, reflecting a "typical" approach to I/O in applications. A similar setup as used for the hyper-threading experiments places the I/O processes on distinct cores, representing the case where additional resource are used. Finally, an "oversubscribed" setup is investigated with separate compute and I/O processes placed on a single hardware thread, this is not expected to achieve high performance but is included to explore the impact of process switching on shared resources without using hyper-threads. These mappings are further explained in section III-B. Each of these scenarios is tested using the STREAM [4] and High Performance Conjugate Gradient (HPCG) [5] benchmarks to simulate their use in HPC applications.

II. Related work

The benefits of using SMT in HPC machines have been more variable than a normal desktop machine. HPC clusters often have different softwares and variables such as math libraries which affect lots of scientific software. Celebioglu et. al. [6] have found that enabling the Intel implementation of SMT, Hyper-Threading Technology (HT) had a variable effect on the test applications simply by linking with different math libraries. They found that if they used mathematical libraries that were optimised for processor usage, enabling the HT had a negative effect on the performance due to resource sharing in virtual processors [6].

Curtis-Maury et. al. [7] demonstrate that scalable applications return sub-optimal performance on systems having more than 2 SMT cores due to a high volume of resource contention. They investigate other forms of multi threading techniques such as adaptive head throttling and speculative execution and their study shows that combining these techniques with thread parallelisation lead to significant performance improvements in parallel codes [7].

The resulting mixed view of highly variable performance benefits from SMT has been confirmed by various other researchers, including [8], [9], [10] and [11]. This has lead to the widely held belief that SMT is often not beneficial in HPC applications.

The potential for driving I/O workloads on SMT threads was studied by Jia et. al. [12] in the context of optimising machine utilisation. This work aimed to improve throughput by scheduling compute and I/O bound workloads on the same machine and making use of SMT with the goal of using resources not occupied by the neighbouring workload. Their work shows that the existing techniques used by CPU schedulers on SMT processors to improve I/O performance are inefficient on SMT processors. This is because of inefficient context switches when jobs are waiting for I/O tasks. They suggest a context retention technique which uses a hardware thread to store the context of an I/O workload. This can help reduce the overhead of context switches so that the workload can efficiently respond to I/O events.

More often a single workload will perform compute and I/O steps, in the simplest case running each sequentially. Several works have investigated running associated tasks asynchronously to improve overall performance of HPC applications. In the Met Office NERC Cloud model (MONC), the in-situ data analysis step prevented simulation progress until that step was completed. Brown et. al. [13] developed an I/O server framework which performs this task asynchronously. Progress of the compute process is enabled by sending data to an I/O server which receives the data and performs the analysis. This approach improved the scalability of the system, enabling simulations of over 2 billion points while its predecessor could not go beyond 16 million grid points. More broadly, a typical HPC application will preform regular checkpointing of data. To reduce the performance impact of this Heroult et. al. [14] have proposed cooperative I/O scheduling applications to reduce the contention for the I/O bandwidth such as reducing checkpointing volumes or by increasing checkpointing intervals. They have demonstrated that the I/O scheduler using non-blocking communication which they call the Least-Waste strategy was the most efficient out of different strategies which used blocking or non-blocking communication [14]. Furthermore they demonstrate that the non-blocking strategies are more resilient to system failures compared to the blocking strategies.

Tang et. al. [15] have developed an asynchronous I/O framework using back-end threads and implemented this approach for the HDF5 I/O library [15]. Their results show that the cost of the I/O was hidden when the application was performing non-I/O operations [15]. The asynchronous approach is handled using HDF5's VOL connector framework requiring additional code changes

to switch between the synchronous and the asynchronous I/O operations. Tseng et. al. [16] have performed a study of the causes and consequences of interference due to the asynchronous I/O on HPC systems. They implement dedicated resources such as multiple threads and seperate cores for I/O operations to reduce the impact of resource contention so that the application can continue without blocking for I/O operations [16]. For I/O operations they utilise POSIX threads.

This work uses asynchronous communication via MPI to send data to the I/O servers. By doing so the relative placement of the compute and I/O processes can be varied from hyperthreads on the same core to entirely separate nodes on an HPC machine. Additionally, iocomp provides access to several I/O layers ranging from low level interfaces such as MPIIO to the highly abstracted ADIOS2, allowing their relative strengths in these scenarios to be studied. This will provide a useful insight into the effects of the abstractions from the I/O layers when combined with non-blocking communication across different process placements.

III. Methodology

To evaluate the effects of using SMT cores for performing asynchronous I/O requests, we developed the iocomp library [17].

A. iocomp

This library is used to analyse the performance impact on the I/O bandwidth by using SMT on ARCHER2. The iocomp library implements a client-server model which interfaces with various I/O back-ends such as MPIIO, HDF5 and ADIOS2. While the client processes (aka compute processes) handle the computations in a typical HPC application, the server processes (aka I/O processes) complete the I/O requests from the client processes. This splitting of tasks enables the comparison between synchronous and asynchronous I/O with different mappings of available CPU cores used to investigate the impact of performing the asynchronous I/O process on SMT threads against separate physical cores.

1) Initialisation: At initialisation the iocomp library accepts a flag to select the synchronous or the asynchronous I/O pipeline as shown in Fig. 1. If the asynchronous I/O method is chosen, the processes are split into compute and I/O processes. The I/O processes are initialised and given their own intra-communicator, and the initialisation function returns a separate intra-communicator for the compute processes to use. This design means that minimal changes are needed to any program that uses this library; beyond using iocomp for I/O, replacing MPI_COMM_WORLD with the communicator returned by the iocomp initialisation function should be the only required change.

2) Process splitting: A key step in our experiments is the process placement to ensure specific cores/threads run either compute or I/O processes as desired. In the single node case, the MPI processes are split based on lower and upper numbering within 1 node. i.e. if there are n MPI processes within 1 node, 0 to $n/2-1$ form one sub-group and $n/2$ to $n-1$ form another sub-group.

To split the processes across multiple nodes, the library uses a user defined value "NODESIZE" which is the number of cores in a node. This is then used to divide the cores within 1 node into I/O and compute servers. This enables pairing the resources within 1 node for the hyperthread and the oversubscribe mappings. These mappings are illustrated in table I and a more detailed pairing between processes is shown in table V which uses a "NODESIZE" value of 16 with 4 nodes.

MPI size	Division of MPI processes	
	Compute	**I/O**
128	0-63	64-127
256	0-127	128-255
384	0-127, 256-319	128-255, 320-383
512	0-127, 256-383	128-255, 384-511

TABLE I: Division of MPI processes under different MPI sizes with a "NODESIZE" value of 128.

These processes are then mapped to the physical cores by passing the list of CPUs to the SLURM scheduler. Once the global MPI processes are split, one half of the MPI processes are assigned to the "compute server" and the rest to the "I/O server".

The splittings are achieved by calling the MPI_Comm_split function [18], given in listing 1, using a "colour" to determine the splitting. Based on the assigned "colours", the global MPI communicator is split to give an intra communicator for the I/O processes and the compute processes, named "ioServerComm" and "computeComm" respectively. The communicator for the compute processes is returned to the client, the I/O processes, and their communicator, do not return, remaining hidden from the client.

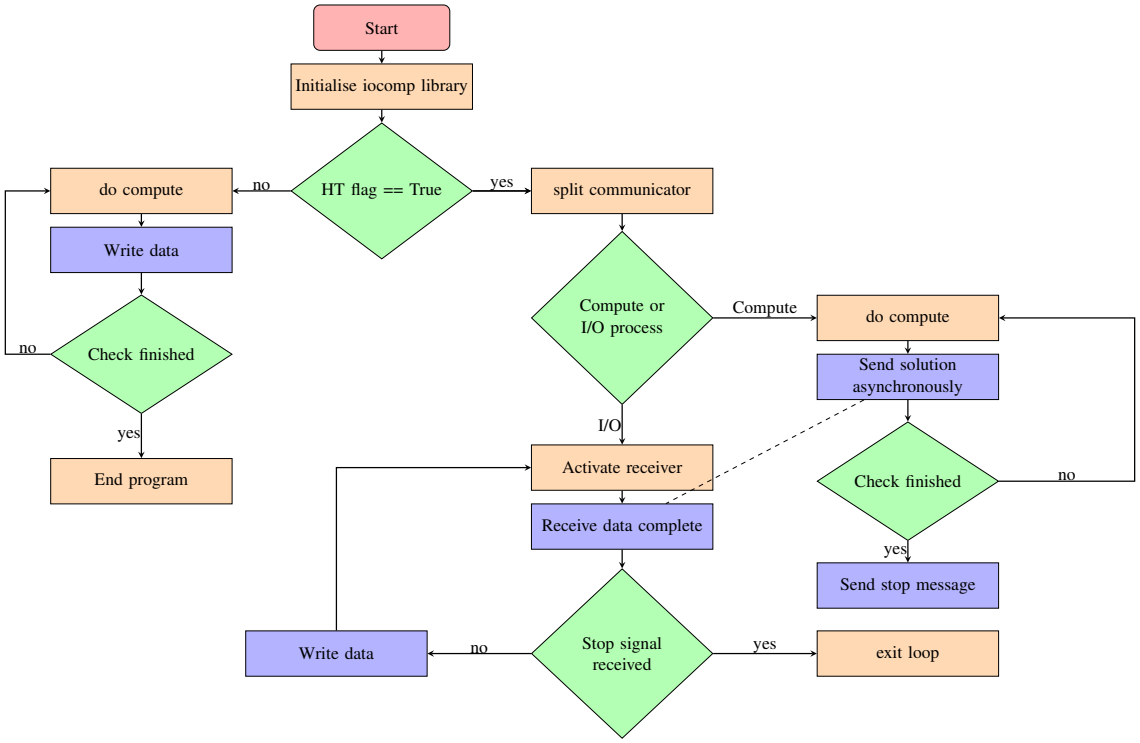

Fig. 1: Flowchart of `iocomp`.

Listing 1: Interface to split an MPI communicator.

```
MPI_Comm_split(MPI_Comm comm,
  int color, int key,
  MPI_Comm *newcomm)
```

3) Compute server: The client process performs the computational tasks, calling the "dataSend" function to send data to be written to the I/O server. If the synchronous I/O method is selected, then the function passes the data pointer directly to the I/O back-end. In this case, the communicators are not split based on colour and the global communicator is used throughout the program. However, if the asynchronous I/O method is selected, then the function sends the output data to the "I/O server" using the nonblocking "MPI_Isend" function. In this case, the library provides another function, "dataWait" which implements a "MPI_Wait" command to wait for the data stream to finish sending. In addition, the library also provides the dataTest function which issues MPI_Test functions to test the progress of the asynchronous requests.

4) I/O server: The "I/O server" processes are responsible for handling the I/O requests from the client processes. If the asynchronous I/O method is initialised, then the I/O server processes call the function "ioServer" which then enters an infinite for loop while it calls the "MPI_Probe" function. This returns the number of elements in the message from the "computer server" and then the message is received using the "MPI_Recv" function. After receiving the data it is written using the respective I/O functions. To signal the end of program execution the stopSend function is used to send a "ghost message" which is a variable of 0 length. When the ioServer detects a message of 0 length a break command is issued in the infinite for loop which stops the receiver from receiving any other messages. This logic is shown in pseudo-code given in code listing 2.

Listing 2: ioServer code with asynchronous flag.

```
for(;;) {
    MPI_Probe();
    len = MPI_Get_count();
    if (len > 0) {
        MPI_Recv();
        writeData();
    }
    else  {
    // Received ghost message, exit.
        break();
    }
}
```

When the "I/O server" receives the data stream and the elements to be written it defines the starting point of the local array within the global array, using the function "arrayParamsInit". Initially, dimensions of the global array were fixed as 1, and the offset point of the data stream within the global array was the product of the "I/O server" and the number of elements in the local data stream. However, with a larger number of ranks, it was found that global size value was bigger than the maximum size accommodated by an integer value and as a result the array parameters were negative numbers. This issue occurred because MPI only takes in arrays of int32 size. Thus, for higher process counts, only small sized arrays could be written out which created a limit to the maximum size of the array possible. This was solved by decomposing a given array into a user defined number of dimensions, which avoided the possibility of overflowing the integer value. At the time of writing this paper, the library version v1.1.3 decomposes the data into 2 dimensions as referenced in the code listing 3.

Listing 3: ioServer data decomposition.

```
// find closest sq root
root = sqrt(dataSize);
// if its a perfect square
if(root*root == dataSize) {
 dim[0] = root;
 dim[1] = root;
}
// if its a multiple
else if(dataSize%root == 0) {
 dim[0] = root;
 dim[1] = dataSize/root;
}
// else search for closest factorial
else if(dataSize%root != 0) {
 for(int i = 1; i < root; i++) {
  if(dataSize%(root-i) == 0)  {
   dim[0] = (root-i);
   dim[1] = dataSize/(root-i);
   break;
  }
 }
}
```

5) Testing: The library has a testing functionality enabled by including the preprocessor flag "READBACK". This enables the function "readBack" which uses rank 0 of the I/O server to read the elements written into the file by the chosen I/O library. This function is called after the "ioLibraries" function finishes its writing. The data stream is then printed out which can help in checking the outputs for verification purposes.

B. ARCHER2 cpu mappings

To analyse the impact on the I/O performance by using SMT on ARCHER2, different scenarios are compared using the same amount of data for computation and I/O:

- The "Sequential" scenario does not split the compute and I/O processes and is completed sequentially in the same process without using SMT threads. This scenario is probably the most common one found in HPC applications.
- The "Consecutive" scenario uses twice the number of cores without using SMT. The processes are split as before, but the compute and I/O processes are assigned to different cores.
- The "Hyperthread" scenario uses two threads per core on ARCHER2. One thread of the core is assigned to the compute process while the SMT thread of the same core is assigned to the I/O process.
- The "Oversubscribe" scenario places the compute and I/O processes on different MPI processes, but on the same core using oversubscribing when running the jobs. This case does not use SMT, but still splits the processes into compute and I/O as before.

The scenarios described above are illustrated in table II showing the total number of compute cores, MPI processes and their placement mapping for ARCHER2. As can be seen, on ARCHER2 [1], the hyperthreads of the cores start from a process id of 128. For example, the hyperthread of core ID 1 will be numbered 129 according to the lstopo output shown in Fig. 2.

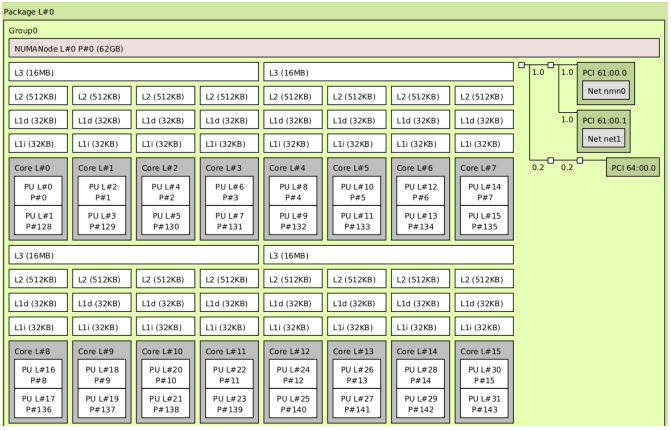

Fig. 2: lstopo output of NUMA node 0 on ARCHER2 showing the placement of Hyperthreads.

Case	Comp cores	MPI procs	cpu-map array
Sequential	4	4	0,1,2,3
Hyperthread	4	8	0,1,2,3 128,129, 130,131
Consecutive	4	8	0,1,2,3 4,5,6,7
Oversubscribe	4	8	0,1,2,3 0,1,2,3

TABLE II: `slurm` mappings for 4 compute processes distributed between the different settings, each run handles same global size data.

C. STREAM

The first test considered as an example computational kernel is the STREAM benchmark [4]. The STREAM benchmark is a simple synthetic benchmark problem, representing the extreme end of memory-bound problems, with many HPC applications being memory bound.

Each of the STREAM kernels, namely COPY, SCALE, ADD and TRIAD, are implemented in this test. The program takes in the following parameters from the command line; the size of the array, I/O library selection, and a flag to select between the direct or the asynchronous I/O pipelines. The iocomp library is initialised using the flag value and the I/O library selection. The initialised array generated using the size parameter value is passed to the STREAM kernels, and when the individual compute kernels finish their processing, the updated arrays are sent to the I/O server using the "dataSend" function. The individual kernels from the STREAM benchmark are timed to get the computation, wait and wall timings. These timings can then be used to analyse the different cases and compare their performance. When the computation and the data transfer are complete, the wall time is measured.

Typical HPC use cases are unlikely to write the solution at every time step. Thus, to represent a more realistic scenario, with multiple time steps performed between I/O operations, the STREAM implementation was run with a higher number of compute cycles per each I/O cycle. The timings per cycle were then summed and averaged by the frequency of compute per write cycle. For the results shown in this work, all the jobs were run with a total of 100 compute cycles and a total of 10 write cycles, setting the parameters "WRITE_FREQ" and "LOOPCOUNT" to 10 and 100, respectively. The program flow is illustrated in code listing 4.

Listing 4: STREAM implementation.

```
for(i=0;i<LOOPCOUNT;i++){
    copy(C) + test_triad(A)
    if ((i+1)%WRITE_FREQ == 0){
        wait_triad(A)
        send_copy(C)
    }

    scale(B) + test_copy(C)
    if (i%WRITE_FREQ == 0){
        wait_copy(C)
        send_scale(B)
    }

    add(C) + test_scale(B)
    if (i%WRITE_FREQ == 0){
        wait_scale(B)
        send_add(C)
    }

    triad(A) + test_add(C)
    if (i%WRITE_FREQ == 0){
        wait_add(C)
        send_triad(A)
    }
}
```

As the data is being sent asynchronously, the code must check this operation completes before the next computational kernel modifies the data of that array. Thus, "MPI_Wait" functions were placed so that spurious data was not sent for writing to iocomp. An illustration of this sequencing is shown in code listing 4

The program also issues multiple "MPI_Test" calls during kernel execution as "MPI_Test" can be used to complete request-based non-blocking operation [18]. The impact of calling "MPI_Test" on the wall time is also investigated.

D. HPCG

HPCG [5] is intended to better represent how today's HPC applications perform. It is of interest in this work, as the HPCG benchmark represents the memory bound edge in the spectrum of computational benchmarks with a Byte/Flop ratio of greater than 4 [19]. To use this benchmark as a client process, the HPCG benchmark repository was forked and updated to integrate the iocomp library [20] while keeping the computational kernels unchanged. To initialise the iocomp library, the HPCG_Init function was changed to accept the iocomp

parameter structure in its arguments. The command line parser was also modified to accept additional commands to provide run time information to the iocomp library such as the backend I/O library selection and the value of the flag to enable or disable asynchronous I/O and split the processes between compute and I/O as appropriate. After the initialisation of the iocomp library, a compute communicator is returned to the client process running HPCG, ensuring that there is a separation of the communicators between the compute processes and the I/O processes. After the Conjugate Gradient function, the solution vector pointer is passed to the dataSend function from the iocomp library along with the number of elements. Additional timers were implemented within the computational loop to measure the time taken to finish the computation, send this data, and wait for the data to be sent as shown in code listing 5 The parameter "numberOfCgSets" is set to 10 to obtain an average of the different values. These modifications enable the iocomp library to write the output data of the HPCG benchmark using various I/O libraries such as MPIIO, HDF5, ADIOS2 etc. As shown in [19], the per-rank memory consumption of HPCG is $\approx 1000\times$ the local problem size, meaning the size of the largest solution vector that could be written was limited by the available memory per core. By under-populating the nodes on ARCHER2, more memory is available per core so that a larger local problem size can be used to test the I/O handling capabilities of the iocomp library which would enable a large enough file size for a sufficient I/O benchmarking analysis. To implement this idea, each node was underpopulated with 16 tasks instead of the fully populated setting of 128 tasks. To obtain the correct mappings, iocomp was initialised with the variable "NODESIZE" set to be 16. The pairings for a sample job with 4 nodes are provided in table V using the preprocessor flag "PRINT_PAIRS". However, this was also insufficient to obtain a large size of data to be written. For 1 full node, only 2GB of data could be written out before reaching the memory limit per node. Using Linaro forge [21] for analysis, it was found that when testing with a problem size of $256\times256\times256$ using 16 cores and the hyperthread mapping, 26% of total thread time was spent on the computational element i.e. the CG function and 0.1% of time was spent on the I/O element. For this workload a peak memory consumption of 14GB per thread was observed, demonstrating the high memory requirements of the benchmark. To increase the volume of data written, it was decided to write the matrix coefficients as well. This would make each write step of the benchmark resemble a simulation checkpoint with the problem state and current solution being written to disk. However, the application reaches the memory limit per node when this problem size was scaled to more than 2 nodes. Hence, it was decided to use an array of dimensions $256\times256\times128$ with each core writing a 1.69 GiB block of data. The effective memory bandwidth of HPCG was analysed by the division of the memory traffic reported by HPCG and the wall time taken by the iocomp integration. The total memory traffic is obtained by taking the product of the 'Raw total bandwidth' and the 'total time of execution' from the output file generated.

Listing 5: HPCG computational loop.

```
initialise_matrix()
for(i=0;i<numberOfCgSets;i++)
{
 -- start loop timer

 -- start data send timer
 send(matrix)
 -- end data send timer

 -- start compute timer
 ZeroVector(x)
 test(matrix)
 CG(A,x,...)
 -- end compute timer

 testnorms_data.values;
 -- start wait timer
 wait(matrix)
 -- end wait timer
 testnorms()

 -- end loop timer
}
```

E. Experimental setup

The jobs were submitted at least 3 times to account for system noise. The same CPU cores are requested within 1 node for consistency. The timers for the different I/O phases were obtained by each rank and written by rank 0 after performing a reduction to get the maximum value of that timer across the ranks. The wall time is measured until the last I/O call by the program. For the lustre storage system on ARCHER2, the default stripe size and maximum stripe counts are used. The list of the

libraries and environments used for the results are given in table III An additional environment flag used was `FI_OFI_RXM_SAR_LIMIT=64k` as recommended in the user guide for ARCHER2 [22].

I/O library	Version
Prg-Env-gnu	8.0.0
craype	2.7.6
GCC	10.2.0
Cray MPICH	8.1.4
Cray HDF5 parallel	1.12.0
ADIOS2	2.8.0
GNU	9.1
iocomp	1.1.3

TABLE III: Environments used.

IV. Results

A. STREAM

1) Baseline comparison: A baseline comparison was established to compare the average compute time obtained using the different SLURM mappings against the STREAM benchmark code [4]. This benchmark was run as an array of 3 jobs on ARCHER2, and the values were averaged. The STREAM implementation using iocomp included the "MPI_Test" calls in the computational loop of the different kernels, thus the computation times include overhead due to the calls to "MPI_Test".

To measure a like-for-like comparison against the STREAM benchmark, the "MPI_Test" calls were disabled in the STREAM implementation by commenting out the "MPI_Test" preprocessor flag. When included in the compilation, this flag removes the "MPI_Test" calls from the program. The resulting compute times for a local array size of 0.125GiB are presented in Fig. 3. Both the consecutive and hyperthread configurations show similar trends to the sequential, with very minimal overhead for the consecutive whereas for hyperthread the overhead is slightly over 10%, reflecting the cost incurred by the switching between hardware threads running the compute and I/O processes. However, for the oversubscribe case, the overhead becomes significant with timings up to 36% higher in the COPY case, showing the inefficiency of the oversubscribe configuration, demonstrating the high cost of the context switching in a single core without the support of SMT threads.

2) Wall time comparison: The wall time is measured across the different I/O layers for a local size per core of 0.125GiB for 1,2,4,8,16 and 32 fully populated nodes with weak scaling across the different mappings. The plot for the MPIIO backend is shown in Fig. 4. It is consistently shown that the oversubscribe case has the

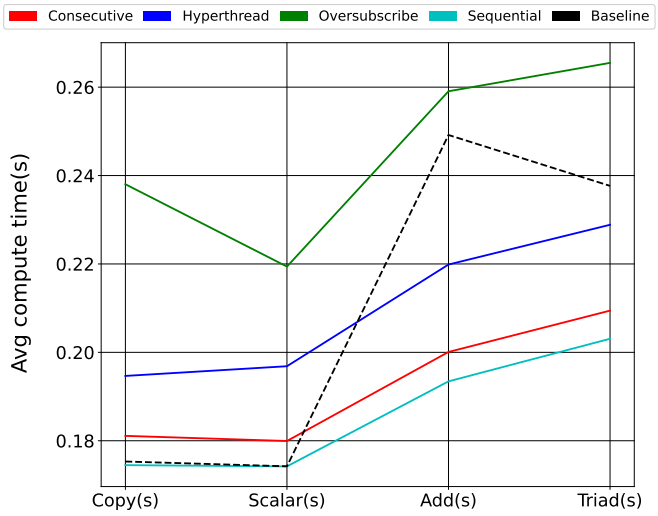

Fig. 3: Baseline comparison for STREAM benchmark local array size 0.125GiB.

	Copy(s)	Scalar(s)	Add(s)	Triad(s)
Consecutive	1.04	1.04	0.82	0.89
Hyperthread	1.12	1.13	0.9	0.97
Oversubscribe	1.36	1.26	1.06	1.12
Sequential	1.0	1.0	0.79	0.86

TABLE IV: Ratio of values obtained using iocomp divided by values obtained using STREAM benchmark for MPIIO, local size 0.125GiB.

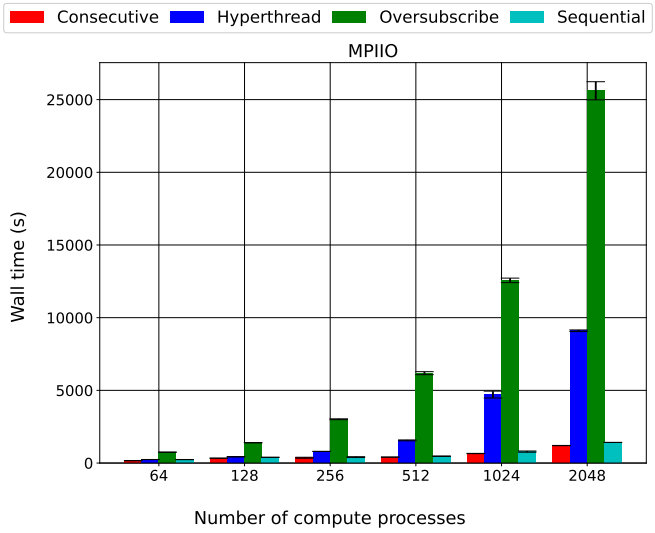

Fig. 4: STREAM wall time comparison using MPIIO comparing oversubscribe mapping to the other mappings with local size 0.125GiB.

highest wall time across all the mappings which is most likely due to the expensive context switching within the same hardware core. As expected, the cost of this increases over the number of ranks, which makes it unsuitable for the purposes of a dedicated I/O server. Thus, from this point, the oversubscribe mapping will be removed from the analysis and the rest of the mappings will be analysed. To observe the effects on wall time of

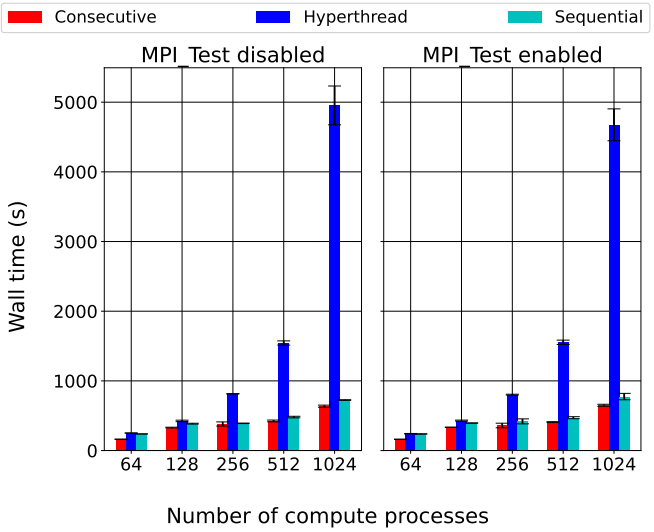

Fig. 5: Wall time comparison between STREAM runs using MPIIO I/O backend with "MPI_Test" enabled and disabled for a local size of 0.125GiB.

the STREAM implementation, the program was run with and without including the "MPI_Test" calls by using the preprocessor flag "MPI_TEST" when building the benchmark. The "MPI_Test" function is called through an abstracting function from iocomp, "dataSendTest", which tests the status of the corresponding user-provided MPI Request handle if the asynchronous I/O is selected. The results are plotted with a local size of 0.125GiB for each case and are presented in Fig. 5. It was observed that by including the "MPI_Test", the effects on the wall time of the STREAM implementation is seen from 256 compute processes. Further, the upper bound values of the hyperthread and the consecutive case have a higher wall time when the MPI tests are disabled, demonstrating the benefit of progressing the "dataSend" operation. There is no significant change in the wall time for the sequential case as the "dataSendTest" function takes no action when synchronous I/O is selected. Thus for a higher number of nodes, it demonstrates that enabling the "MPI_Test" calls is useful in improving the application wall time as they help in progressing the asynchronous MPI Requests [18]. The following results will have "MPI_Test" calls enabled.

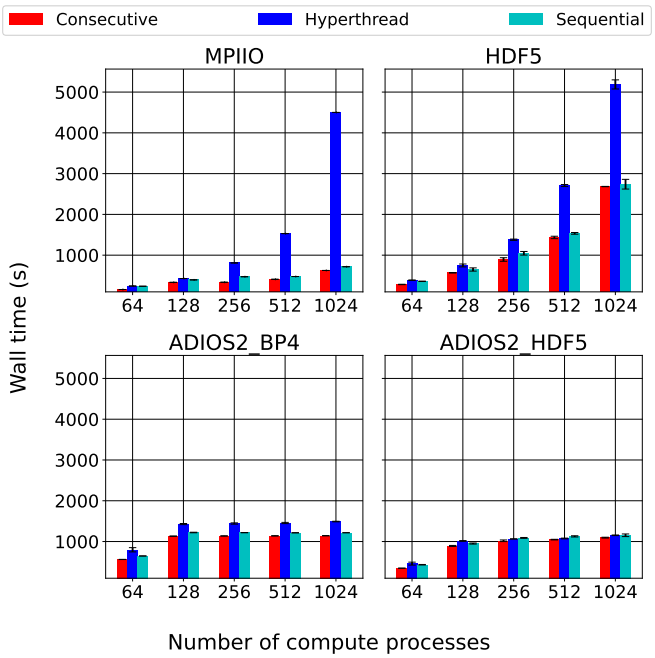

Fig. 6: STREAM wall time comparison using different I/O libraries for local size of 0.125GiB.

Comparing the wall time achieved across all the I/O layers in Fig. 6, it can be seen that the consecutive mapping of the "I/O server" achieves less variable and in most cases improved wall time output. However, for the MPIIO and the HDF5 I/O back-ends, the hyperthread mapping performed slower than the other mappings and gets progressively worse as the number of ranks increases. In contrast, for the ADIOS2 HDF5 I/O back-ends, the hyperthread mapping returned comparable and less variable wall time output than the sequential mapping. The ADIOS2 back-ends showed improved scalability and more consistent results compared to the other I/O back-ends. To analyse these results, a breakdown of wall time into time doing computing and time spent doing I/O writing across all the kernels was plotted for each I/O back-end in Fig. 7. Each of the presented timers is reduced across processes to get the maximum value. The wall time is measured outside the computational loop and reduced across processes. As a result, when interpreting this data the compute I/O times may add up differently to the wall time.

The STREAM implementation was dominated by the time taken to complete the I/O operations, whereas the compute time was a fraction of the total time. The compute time under the sequential mapping stayed consistent

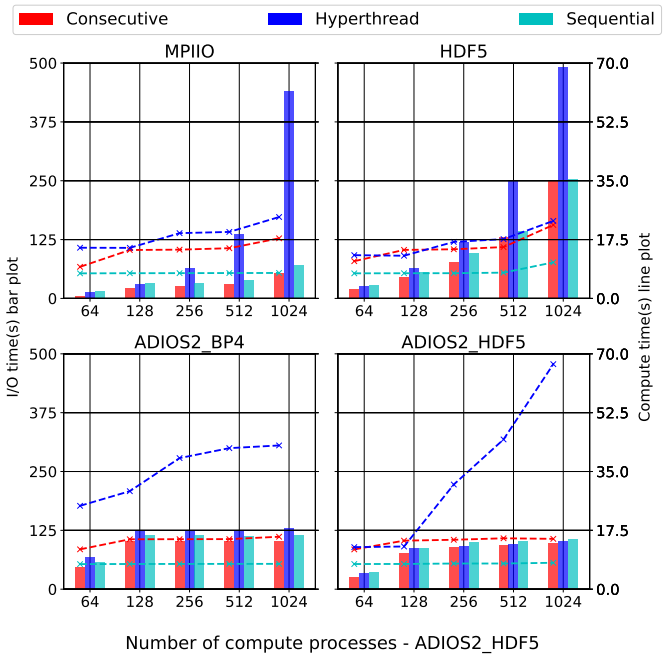

Fig. 7: Breakdown of compute time vs I/O time for local size 0.125GiB.

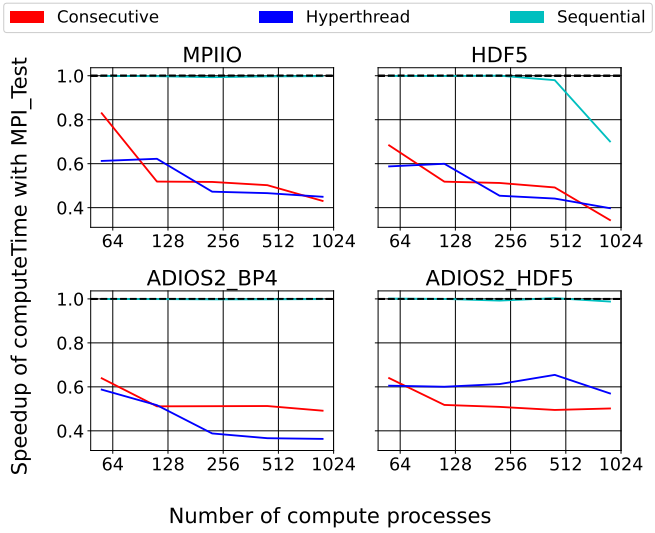

Fig. 8: Speedup of total compute time when MPI_Test is enabled w.r.t when it is disabled, for local size 0.125GiB.

with an increasing number of processes and across all the I/O back-ends, which was to be expected since the STREAM benchmark is weak scaling and is an embarrassingly parallel program. The compute time under the consecutive mapping was also similar across the different I/O back-ends, but it was higher than the sequential mapping, due to the cost of calling the "MPI_Test" calls, shown in code listing 4. This increase in compute time was offset by a reduction in the I/O time resulting in a lower wall time compared to the sequential mapping. The compute time by using the hyperthread mapping was higher than that by consecutive mapping, which is likely due to the added costs of context switching for the hyperthread cases when the I/O server cores are waiting for the I/O operations [12]. The performance of the hyperthread mapping also varied based on which I/O back-end was used. In the case of MPIIO and HDF5, the hyperthread mapping was conclusively poorer compared to the other mappings, with the I/O time increasing almost linearly with an increase in the number of processes. In contrast, for ADIOS2 BP4 and HDF5 the I/O times for the hyperthread mapping remained consistent similar to the other mappings. ADIOS2 HDF5 incurred the highest compute times under hyperthread mapping, suggesting that the MPI_Test commands introduced linearly increasing additional time costs as the number of processes increased. ADIOS2 BP4 shows a more consistent compute time under hyperthread mapping, which suggest that the MPI_Test commands are not increasing additional time costs with an increasing number of processes. The addition of MPI_Test calls is an important factor that could explain the changes in the compute times between the different mappings and the different I/O back-ends. To analyse this effect on compute time, a speedup plot is presented in Fig. 8 of runs with MPI_Test enabled wrt. the runs without the MPI_Test calls.

The speedup of compute times shows that in HDF5 and MPIIO, the addition of MPI_Test calls has almost similar costs for hyperthread and consecutive mappings. However, ADIOS2 BP4 clearly shows that addition of MPI_Test calls has a higher cost for the hyperthread mapping as compared to the consecutive mapping. In contrast, ADIOS2 HDF5 has a higher cost in the consecutive mapping when compared to the hyperthread mapping.

3) Strong scaling: The STREAM implementation was designed as a weak scaling application, however, the performance of iocomp under strong scaling is also of interest. For the strong scaling case a global size of 8 GiB is used with a writing frequency of 10 compute cycles per write. The strong scaling results presented in Fig. 9 show that the consecutive mapping scales better than the sequential mapping especially for the lower node counts which have a higher local size of data. It is observed that the hyperthread mapping still returns a higher wall time than the other mappings for almost

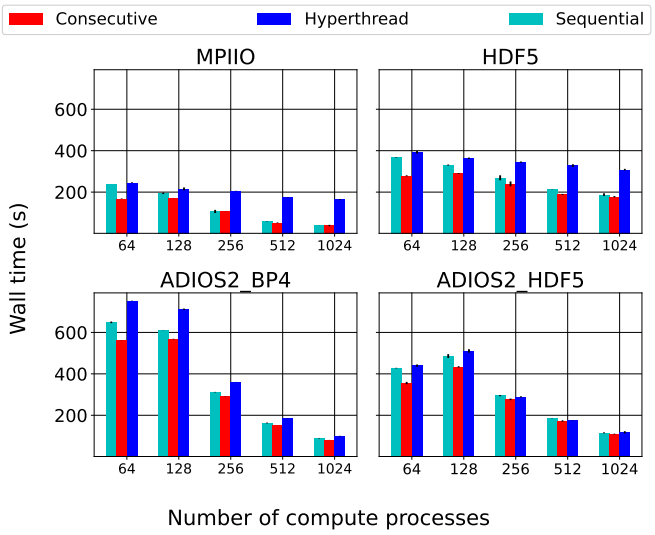

Fig. 9: STREAM wall time comparison using different I/O libraries in strong scaling with a global size of 8 GiB.

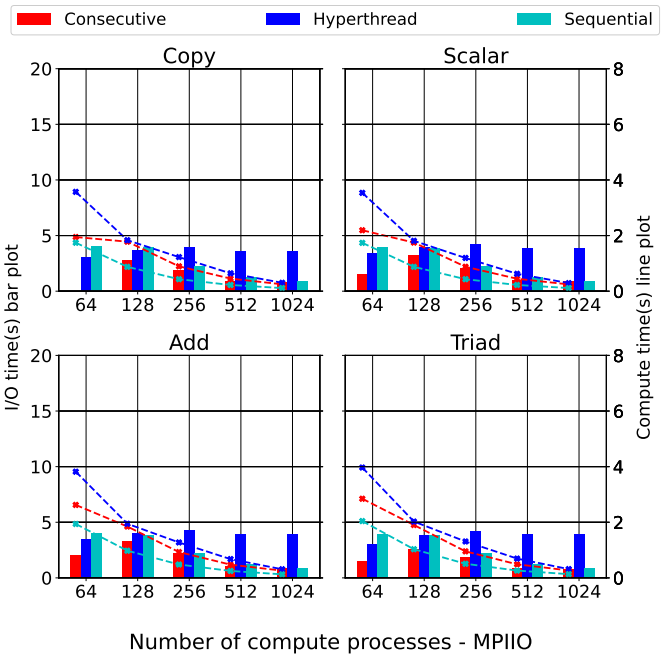

Fig. 10: STREAM breakdown comparison using MPIIO in strong scaling with a global size of 8 GiB.

all the I/O back-ends. The ADIOS2 BP4 I/O back-end appears to scale at a better rate than the other back-ends for all the mappings, this is particularly clear for the hyperthreaded setup. This was expected as ADIOS2 uses one aggregator per compute node as default, increasing the available communication bandwidth between the

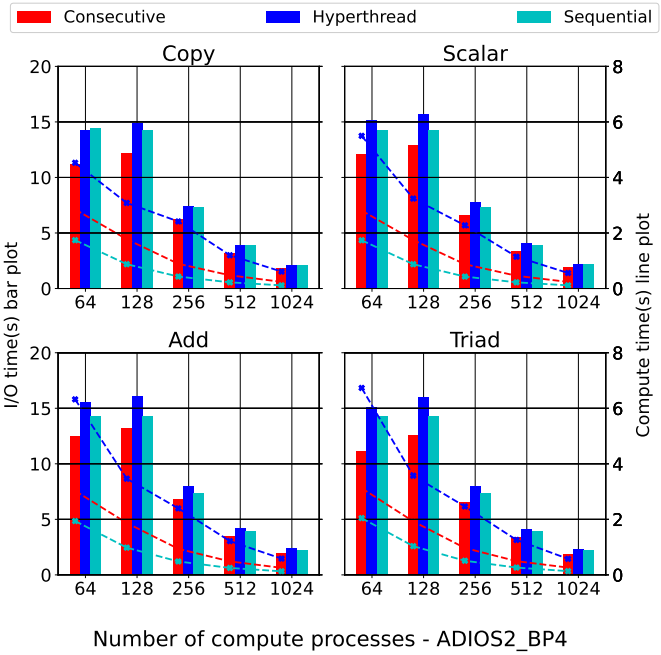

Fig. 11: STREAM breakdown comparison using ADIOS2 BP4 in strong scaling with a global size of 8GiB.

compute nodes and Lustre system proportionally. Similar to the weak scaling results from Fig. 6, the ADIOS2 engines have a comparable performance between the hyperthread mapping and the other mappings.

B. HPCG

As discussed in section III-D, the effective bandwidth of HPCG was analysed by using the total amount of memory used by the benchmark and dividing by the wall time taken by the iocomp integration. The results are shown in Fig. 12.

It can be seen that the consecutive mapping shows the highest effective bandwidth out of all the mappings, which holds true for all the I/O back-ends. This matches the observations from the previous results. Likewise, using the ADIOS2 engines the hyperthread and sequential mappings show comparable effective bandwidths to consecutive as well as higher effective bandwidths than the other I/O back-ends. As before, ADIOS2 HDF5 outperforms the other I/O back-ends, especially when considering the hyperthread mapping. As per this result, the ADIOS2 HDF5 abstraction returns better performance, even when compared against the original HDF5 I/O back-end, with comparable effective bandwidths from using the consecutive mapping. However, when using the hyperthread and the sequential mappings the ADIOS2

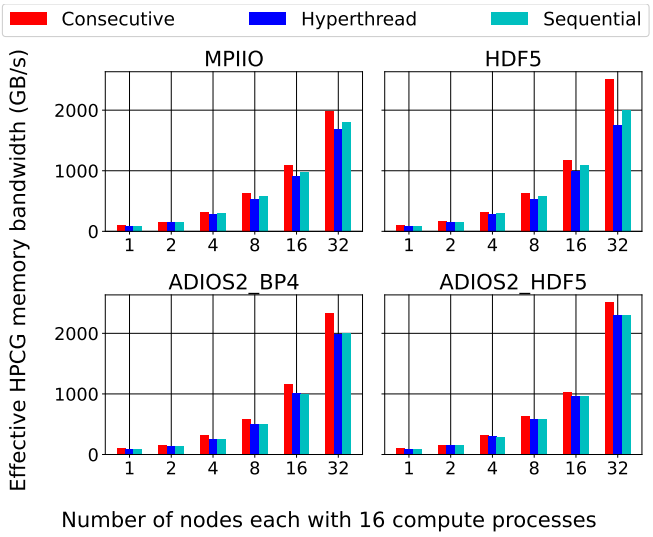

Fig. 12: HPCG effective bandwidth using 1.69GiB per core of memory.

HDF5 I/O backend returns better performance. These results match what was observed in STREAM ad are analysed further in Fig. 13, showing the compute times and I/O times which are a sum of the wait times and the sending times.

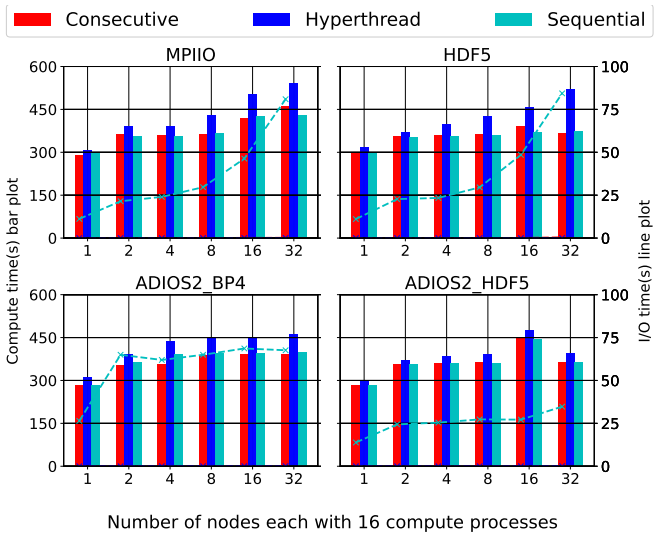

Fig. 13: HPCG timers breakdown using local size of1.69GiB.

Comparing the times across the mappings, it was found that the I/O times for the consecutive and the hyperthread mappings are negligible. In comparison, the sequential mapping shows a much higher I/O time than the other mappings, while the compute time is comparable to that of the consecutive mapping in most cases. This implies that the wait times at the end of the loop as per code block 5 are very small for the hyperthread and consecutive mappings. Thus, demonstrating that the cost of sending data to a dedicated I/O server was absorbed into the compute time for these mappings.

From comparing the times across the I/O back-ends, it was found that as the number of nodes increase, the I/O times also increase for the MPIIO and the HDF5 I/O back-ends. In comparison, the I/O times for the ADIOS2 back-ends stay roughly consistent after 2 nodes, thus showing the improved scalability performance of ADIOS2 compared to the other I/O back-ends. Interestingly, the ADIOS2 HDF5 performs better than the ADIOS2 BP4 I/O back-end with improved compute times and I/O times thus explaining the higher effective bandwidth achieved by the ADIOS2 HDF5 I/O back-end from Fig. 12.

To explain the difference in the performance results obtained by using the different I/O back-ends, STREAM was instrumented with the DARSHAN I/O tool [23]. Using a small job run with 10 compute cycles and 1 I/O cycle for the same file size as the production runs, it was observed that the different I/O back-ends present different behaviour when writing to disk. These results are obtained by using 64 compute processes in the hyperthread mapping. It was found that the ADIOS2 BP4 back-end was writing using mainly 1 process, and much larger file sizes in one aggregation. This was in contrast to the other back-ends which were writing with more processes. These would explain the improved relative performance of the hyperthread case shown in Fig. 7, where it is observed that the ADIOS2 back-ends use less time completing their I/O operations compared to the HDF5 and MPIIO back-ends.

V. Conclusions and Future Work

The iocomp library [17] was developed to provide a comparison between asynchronous I/O and direct synchronous I/O to test the hypothesis that SMT cores on ARCHER2 could perform the I/O requests from the main core asynchronously, potentially resulting in higher application performance. The iocomp provided the "I/O server" implementation and the "client server" tested against the STREAM [4] and HPCG benchmarks to replicate the behaviour of typical HPC applications.

The STREAM benchmark was implemented based on the repository by Jeff Hammond [4]. The data output by the kernels were sent through iocomp to several I/O back-ends and the different timings were recorded for analysis.

The compute times of the relatively simple STREAM kernels with contiguous memory accesses were dwarfed by the I/O times and as a result, the wall time was mainly due to the I/O times. This skewed the results and the wall time was then dependent on the I/O bandwidth per kernel. To better represent typical HPC applications, the number of compute cycles was increased for a given number of I/O cycles. For these results, a frequency of 10 compute cycles per write was set. From Fig. 4, it was found that the oversubscribe mapping was returning sub-optimal performance. As compared to the other mappings, the oversubscribe mapping suffers with additional costs of context switching within the same set of resources. Due to the demonstrated poor performance of this mapping, it was removed from the later analyses. The effects of calling MPI_Test to progress asynchronous operations were tested by comparing against the case with the calls disabled, shown in Fig. 5. It was found that enabling the MPI_Test calls reduced the variability in the times, thus returning a more reproducible result and also results in a reduced time for the hyperthread mappings at higher process counts. Comparing the wall times across the different I/O back-ends shown in Fig. 6, it was observed that the consecutive mapping requires a lower wall time compared to the hyperthread and the sequential mapping. It was also found that the ADIOS2 I/O engines returned lower wall times for the hyperthread mappings, with results comparable to the sequential mapping. This was unexpected, because the hyperthread mappings returned a much higher wall time compared to the sequential mappings when using the MPIIO and HDF5 I/O back-ends. Analysing the breakdown in the wall times, it was observed that the hyperthread mapping showed an increase in the compute times as compared to the other mappings, implying that it spent more time in the MPI_Test calls. This resulted in the lowered performance of the hyperthread mapping as compared to the sequential mapping.

When testing the iocomp library with the HPCG benchmark [24], the opposite problem was faced to the STREAM benchmark. In the initial test, the output vector was sent to the iocomp library for writing, however the memory consumption per solution element limited the local size per node of ARCHER2. To test the library with a more significant I/O load, the matrix coefficients were also output through the iocomp library, simulating a checkpoint step. This increased the data size of arrays to be written by iocomp and the I/O back-ends, providing a more realistic test of our hypothesis. From analysing the effective bandwidth of HPCG in Fig. 12, it was observed that once again the sequential mapping returns better performance than the other mappings. The ADIOS2 engines show a comparable performance between the sequential and the hyperthread mapping with the ADIOS2 HDF5 outperforming the ADIOS2 BP4 back-ends. This is then analysed further in Fig. 13, where it is observed that the ADIOS2 HDF5 engine has the least compute time and the least I/O times out of all the back-ends, and this difference is more stark for the higher node counts. From a MAP analysis comparing the behaviours of the 2 back-ends, it was found that the ADIOS2 BP4 engine has a higher I/O time due to the cost of flushing data from the BP4 buffers in the current setup of iocomp.

From these results, it was found that the consecutive mapping shows the better performance compared to the hyperthread and the sequential mappings. However, the hyperthread mapping shows a reduced performance compared to the sequential mapping for the MPIIO and HDF5 back-ends, whereas it shows comparable performance when using the ADIOS2 BP4 and HDF5 back-ends.

Future investigations using iocomp will investigate the compute bound case using the HPL benchmark, and its application to real world problems using the FEniCSx finite element solver [25], [26], [27]. It would also be interesting to implement this server on machines with more than 2 SMT cores to further test the effectiveness of SMT cores as dedicated I/O servers. Fulhame [28], an ARM based supercomputer having 4 SMT cores in a CPU would be an ideal test bed for this experiment. A range of different ratios between the number of compute and I/O servers could be investigated to find an ideal ratio of compute servers to I/O servers. Finally, the iocomp library could be tested against real-life HPC applications with checkpointing workloads, which would be a strong use case for this library.

ACKNOWLEDGMENTS

This work used the ARCHER2 UK National Supercomputing Service (https://www.archer2.ac.uk). This work was supported by an EPCC funded studentship as part of the ASiMoV project (EP/S005072/1). Funding and guidance from EPCC is gratefully acknowledged.

REFERENCES

[1] Hardware - ARCHER2 user documentation. [Online]. Available: https://docs.archer2.ac.uk/user-guide/hardware/

[2] IO500 - SC22 - IO500 list. [Online]. Available: https://io500.org/

[3] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism."

[4] "GitHub - jeffhammond/STREAM: STREAM benchmark." [Online]. Available: https://github.com/jeffhammond/STREAM

[5] J. Dongarra, M. A. Heroux, and P. Luszczek, "A new metric for ranking high-performance computing systems," *National Science Review*, vol. 3, no. 1, pp. 30–35. [Online]. Available: https://academic.oup.com/nsr/article/3/1/30/2460324

[6] O. Celebioglu, A. Saify, Tau Leng, Jenwei Hsieh, V. Mashayekhi, and R. Rooholamini, "The performance impact of computational efficiency on HPC clusters with hyper-threading technology," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE, pp. 250–255. [Online]. Available: http://ieeexplore.ieee.org/document/1303311/

[7] M. Curtis-Maury, T. Wang, C. Antonopoulos, and D. Nikolopoulos, "Integrating multiple forms of multithreaded execution on multi-SMT systems: a study with scientific applications," in *Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*, pp. 199–208.

[8] R. Grant and A. Afsahi, "Power-performance efficiency of asymmetric multiprocessors for multi-threaded scientific applications," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, p. 8 pp. [Online]. Available: http://ieeexplore.ieee.org/document/1639601/

[9] S. Saini, H. Jin, R. Hood, D. Barker, P. Mehrotra, and R. Biswas, "The impact of hyper-threading on processor resource utilization in production applications," in *2011 18th International Conference on High Performance Computing*. IEEE, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/document/6152743/

[10] A. Vega, A. Buyuktosunoglu, and P. Bose, "SMT-centric power-aware thread placement in chip multiprocessors."

[11] L. Porter, M. A. Laurenzano, A. Tiwari, A. Jundt, W. A. Ward, Jr., R. Campbell, and L. Carrington, "Making the most of SMT in HPC: System- and application-level perspectives," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 4, pp. 1–26. [Online]. Available: https://dl.acm.org/doi/10.1145/2687651

[12] W. Jia, J. Shan, T. O. Li, X. Shang, H. Cui, and X. Ding, "VSMT-IO: Improving i/o performance and efficiency on SMT processors in virtualized clouds."

[13] N. Brown, M. Weiland, A. Hill, and B. Shipway, "In-situ data analytics for highly scalable cloud modelling on cray machines," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 1, p. e4331. [Online]. Available: http://arxiv.org/abs/2010.14127

[14] T. Herault, Y. Robert, A. Bouteiller, D. Arnold, K. Ferreira, G. Bosilca, and J. Dongarra, "Optimal cooperative checkpointing for shared high-performance computing platforms," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, pp. 803–812. [Online]. Available: https://ieeexplore.ieee.org/document/8425494/

[15] H. Tang, Q. Koziol, J. Ravi, and S. Byna, "Transparent asynchronous parallel i/o using background threads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 891–902. [Online]. Available: https://ieeexplore.ieee.org/document/9459479/

[16] S.-M. Tseng, B. Nicolae, F. Cappello, and A. Chandramowlishwaran, "Demystifying asynchronous i/o interference in HPC applications," *The International Journal of High Performance Computing Applications*, vol. 35, no. 4, pp. 391–412. [Online]. Available: http://journals.sagepub.com/doi/10.1177/10943420211016511

[17] S. Bhardwaj, "iocomp." [Online]. Available: https://github.com/iocomp-org/iocomp.git

[18] MPI forum. [Online]. Available: https://www.mpi-forum.org/

[19] V. Marjanović, J. Gracia, and C. W. Glass, "Performance modeling of the HPCG benchmark," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds. Springer International Publishing, vol. 8966, pp. 172–192, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-319-17248-4_9

[20] Shrey Bhardwaj, "iocomp-hpcg." [Online]. Available: https://github.com/iocomp-org/iocomp-hpcg.git

[21] linaroforge. [Online]. Available: https://www.linaroforge.com/

[22] I/o and file systems - ARCHER2 user documentation. [Online]. Available: https://docs.

archer2.ac.uk/user-guide/io/
[23] Darshan – HPC i/o characterization tool. [Online]. Available: https://www.mcs.anl.gov/research/projects/darshan/
[24] HPCG benchmark. [Online]. Available: https://hpcg-benchmark.org/
[25] M. W. Scroggs, I. A. Baratta, C. N. Richardson, and G. N. Wells, "Basix: a runtime finite element basis evaluation library," *Journal of Open Source Software*, vol. 7, no. 73, p. 3982. [Online]. Available: https://joss.theoj.org/papers/10.21105/joss.03982
[26] M. W. Scroggs, J. S. Dokken, C. N. Richardson, and G. N. Wells, "Construction of arbitrary order finite element degree-of-freedom maps on polygonal and polyhedral cell meshes," *ACM Transactions on Mathematical Software*, vol. 48, no. 2, pp. 18:1–18:23. [Online]. Available: https://dl.acm.org/doi/10.1145/3524456
[27] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, "Unified form language: A domain-specific language for weak formulations of partial differential equations," *ACM Transactions on Mathematical Software*, vol. 40, no. 2, pp. 9:1–9:37. [Online]. Available: https://dl.acm.org/doi/10.1145/2566630
[28] Fulhame | EPCC. [Online]. Available: https://www.epcc.ed.ac.uk/hpc-services/fulhame

APPENDIX

Consecutive						Hyperthread						Oversubscribe					
TYPE	**MPI RANK**	**MPI SIZE**	**CPU-ID**	**NODE-ID**	**PAIR**	**TYPE**	**MPI RANK**	**MPI SIZE**	**CPU-ID**	**NODE-ID**	**PAIR**	**TYPE**	**MPI RANK**	**MPI SIZE**	**CPU-ID**	**NODE-ID**	**PAIR**
COMP	0	64	0	nid004204	16	COMP	0	64	0	nid004204	16	COMP	0	64	0	nid004204	16
COMP	1	64	1	nid004204	17	COMP	1	64	1	nid004204	17	COMP	1	64	1	nid004204	17
COMP	2	64	2	nid004204	18	COMP	2	64	2	nid004204	18	COMP	2	64	2	nid004204	18
COMP	3	64	3	nid004204	19	COMP	3	64	3	nid004204	19	COMP	3	64	3	nid004204	19
COMP	4	64	4	nid004204	20	COMP	4	64	4	nid004204	20	COMP	4	64	4	nid004204	20
COMP	5	64	5	nid004204	21	COMP	5	64	5	nid004204	21	COMP	5	64	5	nid004204	21
COMP	6	64	6	nid004204	22	COMP	6	64	6	nid004204	22	COMP	6	64	6	nid004204	22
COMP	7	64	7	nid004204	23	COMP	7	64	7	nid004204	23	COMP	7	64	7	nid004204	23
COMP	8	64	8	nid004204	24	COMP	8	64	8	nid004204	24	COMP	8	64	8	nid004204	24
COMP	9	64	9	nid004204	25	COMP	9	64	9	nid004204	25	COMP	9	64	9	nid004204	25
COMP	10	64	10	nid004204	26	COMP	10	64	10	nid004204	26	COMP	10	64	10	nid004204	26
COMP	11	64	11	nid004204	27	COMP	11	64	11	nid004204	27	COMP	11	64	11	nid004204	27
COMP	12	64	12	nid004204	28	COMP	12	64	12	nid004204	28	COMP	12	64	12	nid004204	28
COMP	13	64	13	nid004204	29	COMP	13	64	13	nid004204	29	COMP	13	64	13	nid004204	29
COMP	14	64	14	nid004204	30	COMP	14	64	14	nid004204	30	COMP	14	64	14	nid004204	30
COMP	15	64	15	nid004204	31	COMP	15	64	15	nid004204	31	COMP	15	64	15	nid004204	31
IO	16	64	0	nid004206	0	IO	16	64	128	nid004204	0	IO	16	64	0	nid004204	0
IO	17	64	1	nid004206	1	IO	17	64	129	nid004204	1	IO	17	64	1	nid004204	1
IO	18	64	2	nid004206	2	IO	18	64	130	nid004204	2	IO	18	64	2	nid004204	2
IO	19	64	3	nid004206	3	IO	19	64	131	nid004204	3	IO	19	64	3	nid004204	3
IO	20	64	4	nid004206	4	IO	20	64	132	nid004204	4	IO	20	64	4	nid004204	4
IO	21	64	5	nid004206	5	IO	21	64	133	nid004204	5	IO	21	64	5	nid004204	5
IO	22	64	6	nid004206	6	IO	22	64	134	nid004204	6	IO	22	64	6	nid004204	6
IO	23	64	7	nid004206	7	IO	23	64	135	nid004204	7	IO	23	64	7	nid004204	7
IO	24	64	8	nid004206	8	IO	24	64	136	nid004204	8	IO	24	64	8	nid004204	8
IO	25	64	9	nid004206	9	IO	25	64	137	nid004204	9	IO	25	64	9	nid004204	9
IO	26	64	10	nid004206	10	IO	26	64	138	nid004204	10	IO	26	64	10	nid004204	10
IO	27	64	11	nid004206	11	IO	27	64	139	nid004204	11	IO	27	64	11	nid004204	11
IO	28	64	12	nid004206	12	IO	28	64	140	nid004204	12	IO	28	64	12	nid004204	12
IO	29	64	13	nid004206	13	IO	29	64	141	nid004204	13	IO	29	64	13	nid004204	13
IO	30	64	14	nid004206	14	IO	30	64	142	nid004204	14	IO	30	64	14	nid004204	14
IO	31	64	15	nid004206	15	IO	31	64	143	nid004204	15	IO	31	64	15	nid004204	15
COMP	32	64	0	nid004207	48	COMP	32	64	0	nid004206	48	COMP	32	64	0	nid004206	48
COMP	33	64	1	nid004207	49	COMP	33	64	1	nid004206	49	COMP	33	64	1	nid004206	49
COMP	34	64	2	nid004207	50	COMP	34	64	2	nid004206	50	COMP	34	64	2	nid004206	50
COMP	35	64	3	nid004207	51	COMP	35	64	3	nid004206	51	COMP	35	64	3	nid004206	51
COMP	36	64	4	nid004207	52	COMP	36	64	4	nid004206	52	COMP	36	64	4	nid004206	52
COMP	37	64	5	nid004207	53	COMP	37	64	5	nid004206	53	COMP	37	64	5	nid004206	53
COMP	38	64	6	nid004207	54	COMP	38	64	6	nid004206	54	COMP	38	64	6	nid004206	54
COMP	39	64	7	nid004207	55	COMP	39	64	7	nid004206	55	COMP	39	64	7	nid004206	55
COMP	40	64	8	nid004207	56	COMP	40	64	8	nid004206	56	COMP	40	64	8	nid004206	56
COMP	41	64	9	nid004207	57	COMP	41	64	9	nid004206	57	COMP	41	64	9	nid004206	57
COMP	42	64	10	nid004207	58	COMP	42	64	10	nid004206	58	COMP	42	64	10	nid004206	58
COMP	43	64	11	nid004207	59	COMP	43	64	11	nid004206	59	COMP	43	64	11	nid004206	59
COMP	44	64	12	nid004207	60	COMP	44	64	12	nid004206	60	COMP	44	64	12	nid004206	60
COMP	45	64	13	nid004207	61	COMP	45	64	13	nid004206	61	COMP	45	64	13	nid004206	61
COMP	46	64	14	nid004207	62	COMP	46	64	14	nid004206	62	COMP	46	64	14	nid004206	62
COMP	47	64	15	nid004207	63	COMP	47	64	15	nid004206	63	COMP	47	64	15	nid004206	63
IO	48	64	0	nid004248	32	IO	48	64	128	nid004206	32	IO	48	64	0	nid004206	32
IO	49	64	1	nid004248	33	IO	49	64	129	nid004206	33	IO	49	64	1	nid004206	33
IO	50	64	2	nid004248	34	IO	50	64	130	nid004206	34	IO	50	64	2	nid004206	34
IO	51	64	3	nid004248	35	IO	51	64	131	nid004206	35	IO	51	64	3	nid004206	35
IO	52	64	4	nid004248	36	IO	52	64	132	nid004206	36	IO	52	64	4	nid004206	36
IO	53	64	5	nid004248	37	IO	53	64	133	nid004206	37	IO	53	64	5	nid004206	37
IO	54	64	6	nid004248	38	IO	54	64	134	nid004206	38	IO	54	64	6	nid004206	38
IO	55	64	7	nid004248	39	IO	55	64	135	nid004206	39	IO	55	64	7	nid004206	39
IO	56	64	8	nid004248	40	IO	56	64	136	nid004206	40	IO	56	64	8	nid004206	40
IO	57	64	9	nid004248	41	IO	57	64	137	nid004206	41	IO	57	64	9	nid004206	41
IO	58	64	10	nid004248	42	IO	58	64	138	nid004206	42	IO	58	64	10	nid004206	42
IO	59	64	11	nid004248	43	IO	59	64	139	nid004206	43	IO	59	64	11	nid004206	43
IO	60	64	12	nid004248	44	IO	60	64	140	nid004206	44	IO	60	64	12	nid004206	44
IO	61	64	13	nid004248	45	IO	61	64	141	nid004206	45	IO	61	64	13	nid004206	45
IO	62	64	14	nid004248	46	IO	62	64	142	nid004206	46	IO	62	64	14	nid004206	46
IO	63	64	15	nid004248	47	IO	63	64	143	nid004206	47	IO	63	64	15	nid004206	47

TABLE V: Example of division of processes using iocomp with NODESIZE 16 and 4 Nodes.