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

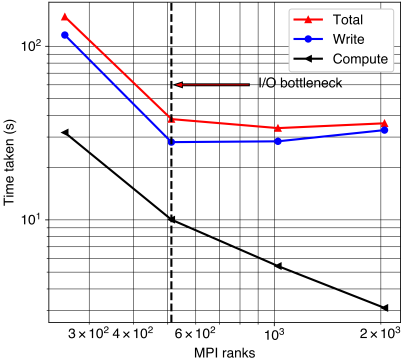
Shrey Bhardwaj
Paul Bartholomew
Mark Parsons

EPCC, University of Edinburgh

May 9, 2023

Outline

I/O bottleneck



xCompact-3D[1]

---

[1] https://doi.org/10.5281/zenodo.7898885

Introduction

- Hypothesis: Can hyperthreads be used as an I/O server to gain additional “effective” bandwidth?

Introduction

- Hypothesis: Can hyperthreads be used as an I/O server to gain additional "effective" bandwidth?
- Developed `iocomp` library[1]
- Compared performance of mapping dedicated I/O server on SMT cores against other configurations
- Benchmarks used: STREAM[2] and HPCG[3]

---

[1] `https://github.com/iocomp-org/iocomp.git`

[2] `https://github.com/jeffhammond/STREAM.git`

[3] `https://github.com/hpcg-benchmark/hpcg.git`

ARCHER2 hardware[1]

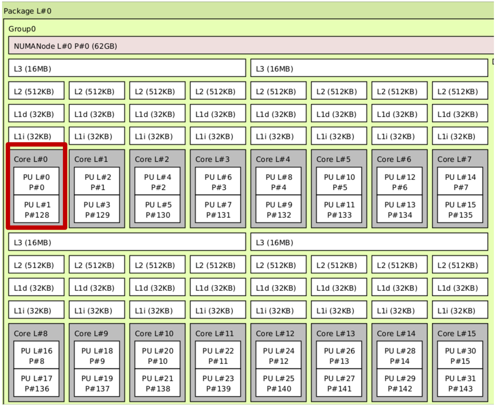
- HPE CRAY system
- 5860 nodes
- 2× AMD EPYC™ 7742, 2.25 GHz, 64-core
- Memory/node = 256GiB
- Memory/core = 2GiB
- 128 cores, 256 threads (including SMT) per node

---

[1] `https://docs.archer2.ac.uk`

ARCHER2 hardware[1]

- HPE CRAY system
- 5860 nodes
- 2× AMD EPYC™ 7742, 2.25 GHz, 64-core
- Memory/node = 256GiB
- Memory/core = 2GiB
- 128 cores, 256 threads (including SMT) per node
- Rank of SMT = 128 + rank of core *within* a node



```
Package L#0
Group0
NUMANode L#0 P#0 (62GB)
L3 (16MB)                                 L3 (16MB)
L2 (512KB)  L2 (512KB)  L2 (512KB)  L2 (512KB)  L2 (512KB)  L2 (512KB)  L2 (512KB)  L2 (512KB)
L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)
L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)
Core L#0    Core L#1    Core L#2    Core L#3    Core L#4    Core L#5    Core L#6    Core L#7
PU L#0      PU L#2      PU L#4      PU L#6      PU L#8      PU L#10     PU L#12     PU L#14
P#0         P#1         P#2         P#3         P#4         P#5         P#6         P#7
PU L#1      PU L#3      PU L#5      PU L#7      PU L#9      PU L#11     PU L#13     PU L#15
P#128       P#129       P#130       P#131       P#132       P#133       P#134       P#135
L3 (16MB)                                 L3 (16MB)
L2 (512KB)  L2 (512KB)  L2 (512KB)  L2 (512KB)  L2 (512KB)  L2 (512KB)  L2 (512KB)  L2 (512KB)
L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)  L1d (32KB)
L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)  L1i (32KB)
Core L#8    Core L#9    Core L#10   Core L#11   Core L#12   Core L#13   Core L#14   Core L#15
PU L#16     PU L#18     PU L#20     PU L#22     PU L#24     PU L#26     PU L#28     PU L#30
P#8         P#9         P#10        P#11        P#12        P#13        P#14        P#15
PU L#17     PU L#19     PU L#21     PU L#23     PU L#25     PU L#27     PU L#29     PU L#31
P#136       P#137       P#138       P#139       P#140       P#141       P#142       P#143
```

lstopo output clip for ARCHER2.

---

[1] `https://docs.archer2.ac.uk`

Outline

1. Overview
2. Methods
3. Results
4. Conclusions

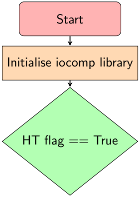
iocomp

- Framework created to enable splitting of processes into I/O and compute servers[1]
- After splitting processes, I/O server gathers data using MPI asynchronous sends from the client process
- Compute server is the client process
- Benchmarks used as client process include HPCG, HPL and STREAM

---

[1] `https://github.com/iocomp-org/iocomp.git`

iocomp

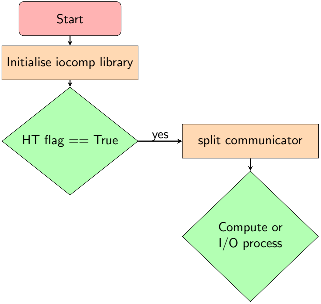


Start

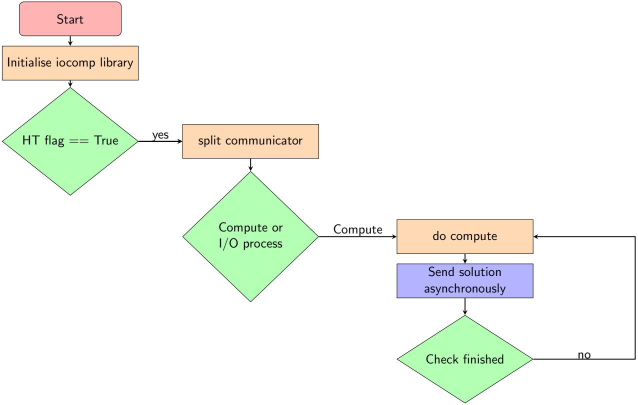
Initialise iocomp library

HT flag == True

iocomp


Start
Initialise iocomp library
HT flag == True
yes
split communicator
Compute or I/O process

iocomp



Start

Initialise iocomp library

HT flag == True

yes

split communicator

Compute or I/O process

Compute

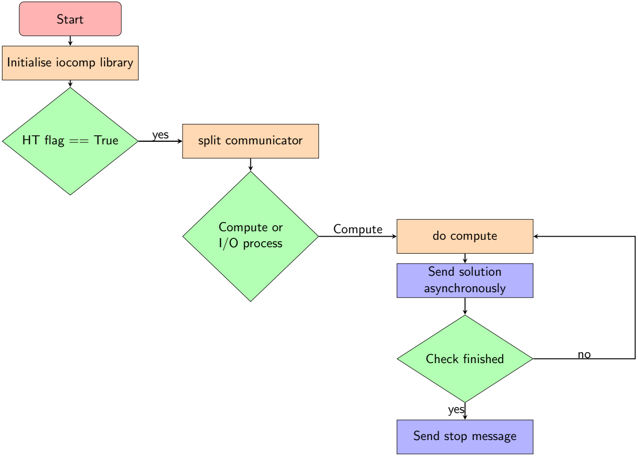
do compute

Send solution asynchronously

Check finished

no

iocomp



Start

Initialise iocomp library

HT flag == True

yes

split communicator

Compute or I/O process

Compute

do compute

Send solution asynchronously

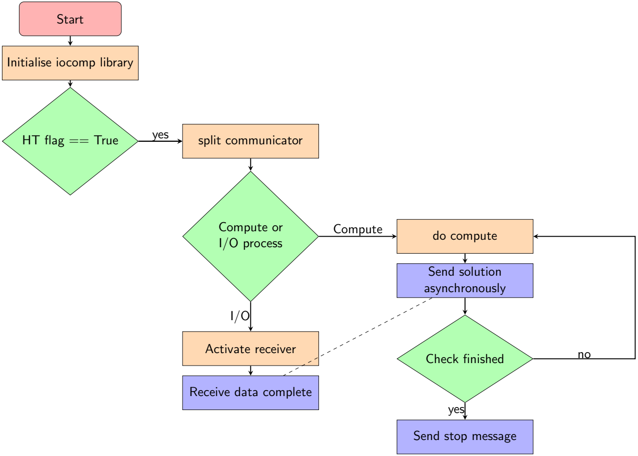
Check finished

no

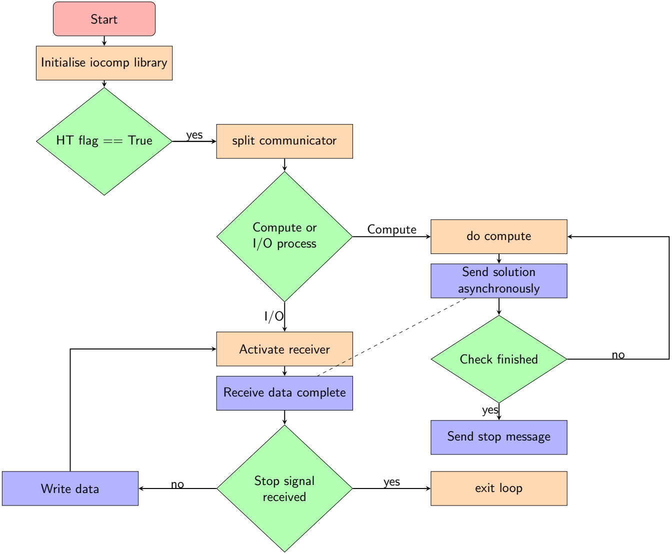
yes

Send stop message

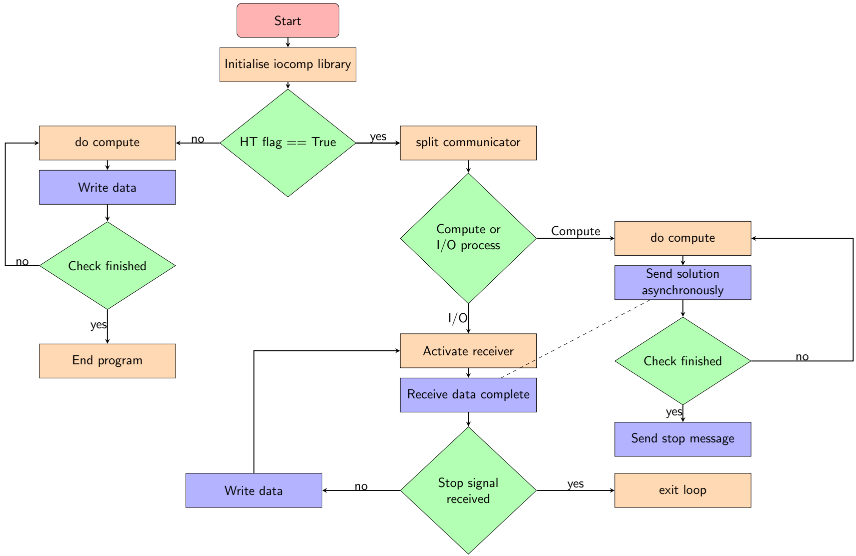
iocomp


Start
Initialise iocomp library
HT flag == True
yes
split communicator
Compute or
I/O process
Compute
do compute
Send solution
asynchronously
I/O
Activate receiver
Receive data complete
Check finished
no
yes
Send stop message

iocomp


Start
Initialise iocomp library
HT flag == True
yes
split communicator
Compute or I/O process
Compute
do compute
Send solution asynchronously
I/O
Activate receiver
Check finished
no
Receive data complete
yes
Send stop message
Write data
no
Stop signal received
yes
exit loop

iocomp


Start
Initialise iocomp library
HT flag == True
no
yes
do compute
Write data
Check finished
no
yes
End program
split communicator
Compute or I/O process
Compute
I/O
do compute
Send solution asynchronously
Check finished
no
yes
Send stop message
Activate receiver
Receive data complete
Stop signal received
no
yes
Write data
exit loop

iocomp initialisation

```
MPI_Comm iocompInit(struct iocomp_params *iocompParams, MPI_Comm comm, bool FLAG,
            int ioLibNum, int fullNode)
// iocompParams is structure for the library
// comm is the global communicator
// FLAG is used to switch between the direct synchronous and the asynchronous I/O
// ioLibNum is used to select the I/O library
// fullNode is used to specify number of ranks placed in 1 node
```

iocomp initialisation

```c
    MPI_Comm iocompInit(struct iocomp_params *iocompParams, MPI_Comm comm, bool FLAG,
            int ioLibNum, int fullNode)
  // MPI_Comm will be the new "global" communicator for the client process
```

ioServer implementation

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

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 square root is a factor
else if(dataSize%root == 0) {
 dim[0] = root;
 dim[1] = dataSize/root;
}
// else search for closest factors
else {
 for(int i = 1; i < root; i++) {
  if(dataSize%(root-i) == 0)  {
   dim[0] = (root-i);
   dim[1] = dataSize/(root-i);
   break;
  }
 }
}
```

SLURM mappings

HT flag disabled

- Sequential
    - Default case, with sequential compute and I/O processing

SLURM mappings

HT flag disabled

- Sequential
    - Default case, with sequential compute and I/O processing

HT flag enabled

- Consecutive
    - Uses 2x number of cores as sequential, **without** SMT
    - Compute and I/O processes are placed on separate cores

SLURM mappings

HT flag disabled

- Sequential
    - Default case, with sequential compute and I/O processing

HT flag enabled

- Consecutive
    - Uses 2x number of cores as sequential, **without** SMT
    - Compute and I/O processes are placed on separate cores
- Hyperthread
    - Uses the same number of physical cores as sequential, **with** SMT
    - Corresponding SMT cores as I/O processes

SLURM mappings

HT flag disabled

- Sequential
  - Default case, with sequential compute and I/O processing

HT flag enabled

- Consecutive
  - Uses 2x number of cores as sequential, **without** SMT
  - Compute and I/O processes are placed on separate cores
- Hyperthread
  - Uses the same number of physical cores as sequential, **with** SMT
  - Corresponding SMT cores as I/O processes
- Oversubscribe
  - Uses the same number of cores as sequential **without** SMT
  - Compute and I/O processes are placed on the same cores

iocomp ranks division

Number of nodes	**MPI size**	**Division of MPI processes**	
		Compute	**I/O**
1	128	0-63	64-127
2	256	0-127	128-255
3	384	0-127, 256-319	128-255, 320-383
4	512	0-127, 256-383	128-255, 384-511

Table: Division of MPI processes under different MPI sizes with a "fullNode" value of 128.

STREAM implementation[1]

- Representing an extreme end of memory-bound computational kernel
- Contiguous memory access runs at limit of largest level in memory hierarchy used
- Implemented as a test case for the iocomp library under stream directory

---

[1] `https://github.com/iocomp-org/iocomp.git`

STREAM code block

```
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)
    }
}
```

STREAM overview.

```
add(c, a, b ...)
{
    for(int i = 0; i<size; i++)
    {
        c[i] = a[i] + b[i];
        if(i%WRITE_FREQ == 0)
        {
            dataSendTest(...);
        }
    }
}
```

STREAM add kernel.

HPCG integration[1]

- More representative of typical HPC applications
  - combining multigrid preconditioner and Conjugate Gradient solver
  - adds indirection and communication
- Another example of a memory-bound program

---

[1] `https://github.com/iocomp-org/iocomp-hpcg.git`

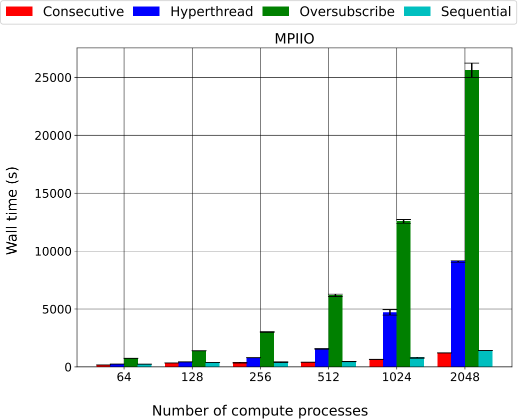
HPCG integration

```
initialise_matrix();
for(i=0;i<numberOfCgSets ;i++) // numberOfCgSets=10
{

 dataSend(matrix...);            Send time


 ZeroVector(x)
 dataSendTest(matrix...);
 CG(A,x,...);                    Compute time
 testnorms_data.values;


 dataWait(matrix...);            Wait time

 testnorms();
}
```

Outline

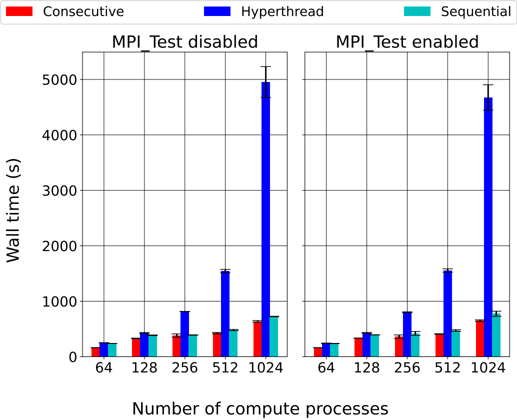
1. Overview
2. Methods
3. Results
4. Conclusions

STREAM



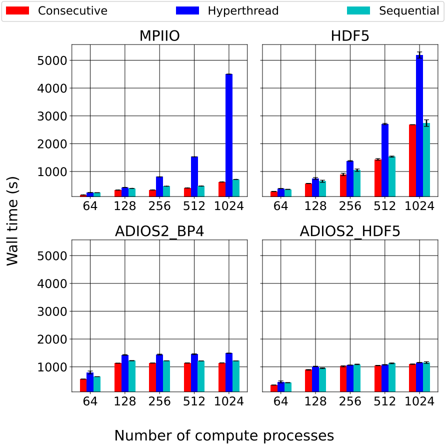
STREAM wall time comparison using MPIIO comparing oversubscribe mapping to the other mappings with local size 0.125GiB.

STREAM



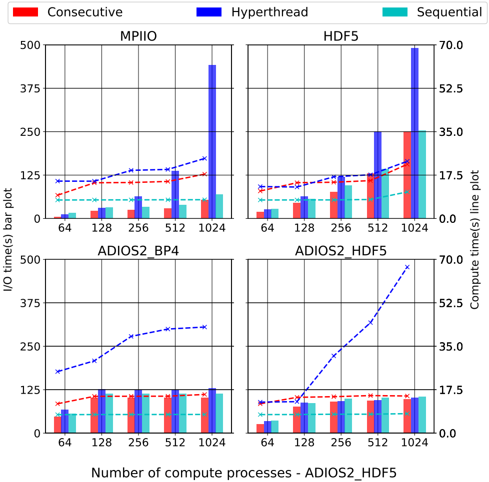
Wall time comparison between STREAM runs using MPIIO I/O backend with "MPI_Test" enabled and disabled for a local size of 0.125GiB.

STREAM



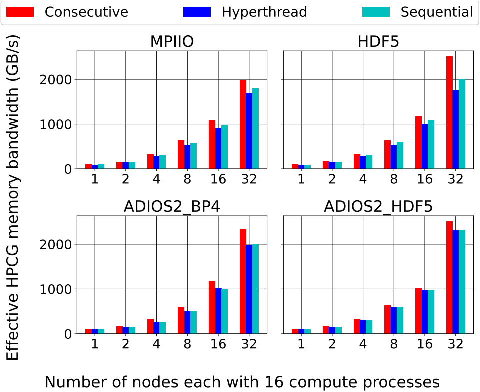
STREAM wall time comparison using different I/O libraries for local size of 0.125GiB.

STREAM



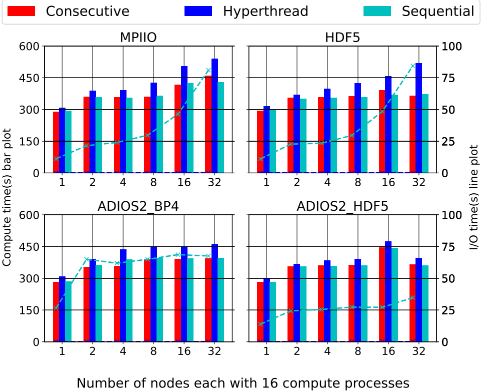
Breakdown of compute time vs I/O time for local size 0.125GiB.

HPCG



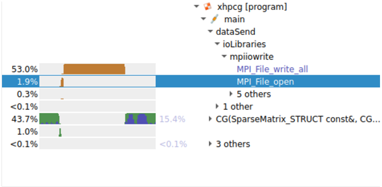
HPCG effective bandwidth 1.69GiB local size.

HPCG times breakdown



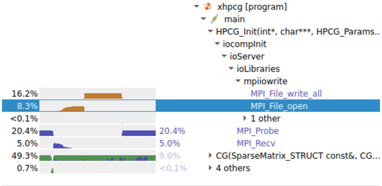
Breakdown of compute time vs I/O time for HPCG with 1.69GiB local size.

ARM MAP breakdown



```
                          xhpcg [program]
                           main
                            dataSend
                             ioLibraries
                              mpiiowrite
53.0%                          MPI_File_write_all
1.9%                           MPI_File_open
0.3%                          5 others
<0.1%                        1 other
43.7%             15.4%     CG(SparseMatrix_STRUCT const&, CG...
1.0%
<0.1%             <0.1%     3 others
```

MAP breakdown of Sequential run using HPCG with local size 0.125GiB.



```
                          xhpcg [program]
                           main
                            HPCG_Init(int*, char***, HPCG_Params..
                             iocompInit
                              ioServer
                               ioLibraries
                                mpiiowrite
16.2%                            MPI_File_write_all
8.3%                             MPI_File_open
<0.1%                           1 other
20.4%             20.4%         MPI_Probe
5.0%              5.0%          MPI_Recv
49.3%             9.0%       CG(SparseMatrix_STRUCT const&, CG...
0.7%              <0.1%      4 others
```

MAP breakdown of Hyperthread run using HPCG with local size 0.125GiB[1].

---

[1] To note: these timing proportions should be multiplied by 2 to get the total time of writing due to the averaging being used per node by MAP

Outline

1. Overview
2. Methods
3. Results
4. Conclusions

Conclusions and Future Work

- iocomp was created to compare different mappings for an I/O server
    - Different cases considered; Hyperthread, Consecutive, Oversubscribe and Sequential
    - Different I/O backends were also tested with these mappings
    - HPCG and STREAM benchmarks were tested
    - Consecutive was the best performer, and hyperthreads performance was dependent on the I/O backend used

Conclusions and Future Work

- iocomp was created to compare different mappings for an I/O server
    - Different cases considered; Hyperthread, Consecutive, Oversubscribe and Sequential
    - Different I/O backends were also tested with these mappings
    - HPCG and STREAM benchmarks were tested
    - Consecutive was the best performer, and hyperthreads performance was dependent on the I/O backend used
- Future work
    - Hyperthread implementation would be more optimised with a shared memory access
    - FEniCSx, a PDE solver will be integrated and tested using iocomp
    - Checkpointing simulations will be tested with the library

Acknowledgements

I would like to thank Dr. Paul Bartholomew, Prof. Mark Parsons and EPCC for guidance and suggestions.

This work used the ARCHER2 UK National Supercomputing Service [1] and was supported by an EPCC funded studentship as part of the ASiMoV project (EP/S005072/1).

---

[1] `https://www.archer2.ac.uk`

Thank you!

QR code for iocomp-org

Thank you!

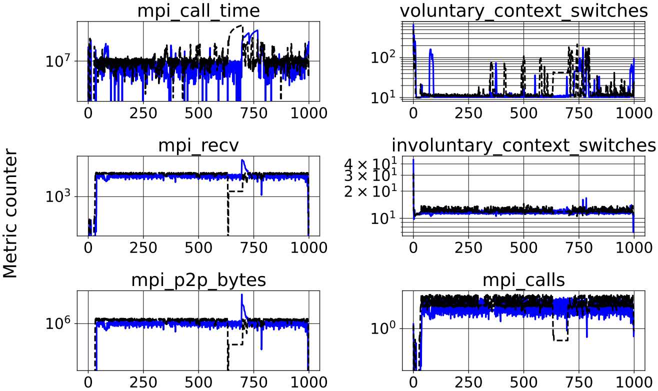
QR code for iocomp-org

- → email: shrey.bhardwaj@ed.ac.uk
- → iocomp: `https://github.com/iocomp-org/iocomp.git`

Outline

5 Appendix

ARM MAP Analysis



Time steps, dashed line for sequential and solid line for hyperthread