Generalisation of Recursive Doubling for AllReduce

Martin Ruefenacht 
m.ruefenacht@ed.ac.uk
Mark Bull 
m.bull@ed.ac.uk
Stephen Booth 
s.booth@ed.ac.uk

Edinburgh Parallel Computing Centre
University of Edinburgh
Edinburgh, Scotland

ABSTRACT
The performance of AllReduce is crucial at scale. The recursive doubling with pairwise exchange algorithm theoretically achieves \(O(\log_2 N)\) scaling for short messages with \(N\) peers, but is limited by improvements in network latency. A multi-way exchange can be implemented using message pipelining, which is easier to improve than latency. Using our method, recursive multiplying, we show reductions in execution time of between 8% and 40% of AllReduce on a Cray XC30 over recursive doubling.

CCS Concepts

• General and reference → Performance; • Theory of computation → Massively parallel algorithms; Distributed computing models; • Networks → Network performance modeling; • Computer systems organization → Interconnection architectures;

Keywords
AllReduce, MPI, Scalability, Collective, Recursive Doubling, n-way, Message Pipelining

1. INTRODUCTION
As supercomputers are pushed further towards exascale, the scaling behaviour of the algorithms used on them becomes ever more important due to the increased levels of parallelism in these systems. Both application layer and network layer algorithms are required to scale, in order to make optimal use of the hardware present. In addition, future interconnect hardware may provide more flexibility and compute capability compared to current hardware.

The AllReduce operation is one of the most important operations used on supercomputers [15]. Improving AllReduce will result in improved scalability for many important applications. Aside from performance the operation must provide consistent (i.e. bit-wise identical) results across all processes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroMPI ’16, September 25 - 28, 2016, Edinburgh, United Kingdom
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4234-6/16/09...
 DOI: http://dx.doi.org/10.1145/2966884.2966913

and executions, thereby ensuring correct and repeatable results. Applications which perform simulations using floating point arithmetic require consistent ordering of operations to achieve this.

The state of the art algorithm used for AllReduce operations, recursive doubling with pairwise exchange, scales as \(O(\log_2 N)\). As ever larger computers are built this logarithmic behaviour will limit scaling behaviour and the only potential gain is by reducing network latency. Since network latency is a physically limited quantity and increasingly difficult to improve, another approach for performance gains must be found.

We seek an AllReduce algorithm which gives us consistency guarantees and performs better than the current recursive doubling with pairwise exchange algorithm. We show that using a model of message pipelining hardware, it is possible to extend the current recursive doubling with pairwise exchange algorithm (as used in MPICH[3]). Our method (recursive multiplying) reduces execution time, relative to that of recursive doubling, by between 8% and 40% on a Cray XC30.

The remainder of this work is organized as follows. In Section 2 the current state of the art algorithm for AllReduce is presented, including the specific details used in MPICH. Section 3 presents two models used to analyse the AllReduce operation in the context of message pipelining. The recursive multiplying algorithm is presented in Section 4. Experimental evidence for the theoretically better performance is demonstrated in Section 5. Section 6 presents related work for AllReduce. Finally, future work is discussed in Section 7.

2. BACKGROUND

MPICH[3] is one of the primary MPI[2] implementations which is used as a base for many commercial implementations. Cray MPI is an example of an implementation using MPICH, which re-implements only the low level network interface, the Netmod interface[14]. This design allows ease of development when porting to different network architectures without having to rewrite higher level algorithms.

One such operation which is abstracted from the actual network is AllReduce. AllReduce is heavily used by many simulation applications and therefore high performance is vital. In MPICH, recursive doubling with pairwise exchange is used for AllReduce when any of three conditions is met. The first is for small messages below 2 kilobytes. The second condition is when a user-defined operation is used to reduce...
Recursive doubling with pairwise exchange enables logarithmic scaling of $O(\log_2 N)$. Figure 1 illustrates the communication pattern which is separated into multiple stages. During each stage, every process in the group sends to its corresponding peer appropriate for the current stage. This results in an AllReduce operation being able to be performed, because previous stages will have communicated data across subgroups. Between each communication round, a reduce operation is executed locally on each process to combine the received partial result and local result.

Each stage of the recursive doubling algorithm consists of pair-wise exchanges with combination of the results. These are implementations of the AllReduce operation over pairs of processors. The overall AllReduce operation is built upon recursively applying a series of smaller AllReduce operations over orthogonal sub-groups of processes, until the entire target group has been reduced. Pair-wise exchange and combination is the two processor version of a general algorithm for an AllReduce where each processor broadcasts its data to all other processors (the All-to-All communication pattern) and the results are reduced locally on every processor.

Recursive doubling requires powers of two pairs of processes, since each stage subdivides the group by two. With large numbers of processes, powers of two are sparse, so this is a strong limitation to the usefulness of the basic method. MPICH fixes the non-power-of-two problem by collapsing and expanding the group in two additional stages. Figure 2 illustrates the solution for six processes. To determine the correct rank with which to communicate, a virtual to real transform has to be performed for every send operation to avoid a deadlock.

First, the next lowest power of two to the size of the collective is found by $p = 2^{\lceil \log_2 N \rceil}$, and the remainder is given by $r = N - p$. This establishes a usable size for the recursive doubling algorithm. All ranks $i$ where $i$ is even and $i < 2r$ send to their peer of rank $i + 1$. Then the recursive doubling algorithm with pairwise exchange is executed on the remaining active processes. The pseudocode is shown in Algorithm 1. Finally, the expansion stage is performed, in which the original processes which received data from their neighbour return the final result.

This fix allows the non-power-of-two case to be handled elegantly with minimal additional stages. While recursive doubling scales well, it is important to maximize the performance, since AllReduce is an important operation. Fundamentally the algorithm is limited by the base of the logarithm which establishes the number of stages which are required. The latency of the network messages also limits recursive doubling since each stage requires each process to exchange messages with a peer process.

Rabenseifner et al. [16] also address the case of a non-power-of-two process count. By using a similar elimination they managed to reduce the rounds required to $\lceil \log_2 N \rceil + 1$ for small messages. By using a particular factorization of the process count they reduce the problem into sets of smaller AllReduce operations.

### 3. MODEL

The recursive doubling algorithm presented in Section 2 is based on a hypercube AllReduce algorithm. On a hypercube network, neighbours exchange messages in a pairwise fashion. A simple model such as the $\alpha/\beta/\gamma$ model by Chan et al. [7] can be used to model the entire AllReduce.

In the $\alpha/\beta/\gamma$ model the cost of communication is $\alpha + n \beta$, where $\alpha$ is the network latency and $\beta$ is the bandwidth. An additional $n \gamma$ term can be used to represent computation if required, where $n$ is the number of bytes which are transmitted and computed. The model assumes a fully connected topology, no network conflicts, and only single message transmissions (alongside other properties that are less important for an AllReduce). Using this model for the recursive doubling with pairwise exchange algorithm results in a total cost of:

$$(\alpha + n \beta + n \gamma) \times \log_2 N \quad | \quad N = 2^k, \quad k \in \mathbb{N}$$

The first term is the cost per stage of the AllReduce while the second term is the number of stages which will be performed. The MPICH fix simply adds two stages to the logarithmic term. Assuming $\beta \ll \alpha$ and $\gamma \ll \alpha$, then for small values of $n$, both terms tend to be much smaller than the latency. The cost of a small message AllReduce is therefore:

$$\alpha \left( \left\lfloor \log_2 N \right\rfloor + \begin{cases} 0 & \text{if } N = 2^k, \quad k \in \mathbb{N} \\ 2 & \text{otherwise} \end{cases} \right)$$

From this we can observe that recursive doubling requires improvements in network latency to further improve performance for small messages.

To explore further options, a model that better represents modern hardware is required. Hoefler et al. [13] introduced the LoP model, which modified the popular LogP [8] model to include message pipelining. This functionality was present on the Infiniband network at the time. The LoP
We modified the $a/b/\gamma$ model to accommodate message pipelining by removing the original property requiring single message transmission. In addition we decomposed the $a$ term into $a_p$ and $a_r$, where $a_p$ is the part of the latency cost of issuing a send which can be overlapped with subsequent sends, and $a_r$ is the remainder of the latency cost of the sends (which cannot be overlapped). For example, $a_r$ may include critical sections in the MPI library or processing time in the network interface, whereas $a_p$ includes propagation delay time in the network itself. If the number of potentially overlapping messages sent is $b$, the cost of sending multiple messages in a pipelining fashion using the $a_p/a_r/\beta/\gamma$ model is given by:

$$a_p + b \times (a_r + n\beta + n\gamma)$$

With this improvement, a model for a recursive multiplying method with an additional requirement for message pipelining capabilities in the underlying network can be constructed. At each stage, instead of exchanging a message with one peer, as in the recursive doubling algorithm, each rank exchanges messages with $b$ peers. If we assume small messages such that the terms involving $n$ are zero, the cost of an AllReduce operation with recursive multiplying is:

$$(a_p + b\alpha_r) \times \log_{b+1} N \mid N = (b+1)^k, k \in \mathbb{N}, b \geq 1$$

Note that this includes the recursive doubling case when $b = 1$.

Figure 4 shows the cost of the recursive multiplying AllReduce plotted for a range of $b$ and $N$ values with an $a_p$ to $a_r$ ratio of 4. The minimal curve is only dependent on the ratio $\alpha_r/\alpha_p$, and can be expressed using the Lambert W function which is an inverse relation of $ze^z$. The value of $b$ which results in the minimum cost for a given hardware capability is given by:

$$b_{opt} = e^{W\left(\frac{a_p - a_r}{a_r}\right) + 1 - 1}$$

A ratio equal to one means each message costs the same amount, while a ratio higher means sending more messages is cheaper. Figure 3 shows $b_{opt}$ for a range of ratios. An overlap ratio of less than one is not reasonable.
The number of stages \( s \) is determined by the number of factors in the decomposition, while the base of a particular stage is the factor of that stage. Using this decomposition of \( N \) we can express the total cost of all stages as a summation of our model cost shown in Equation 1. The total cost within the model for an AllReduce operation is:

\[
N = \prod_{i=1}^{s} (b_i + 1)
\]

The specific amount of memory required is given by the schedule. In comparison with recursive doubling which requires one buffer per stage, recursive multiplying requires \( b_i - 1 \) buffers per stage. As with the memory requirement, the total amount of data communicated at each stage is also increased. Due to this, the network needs to provide enough bandwidth to handle multiple messages, and be able to handle potential congestion.

The prime factorisation of \( N \) can be used to conveniently find an acceptable schedule, which can be performed through recursive multiplying. However, the prime factorisation will likely provide many small factors which are well below the multicast ability. To reduce the number of stages and maximise the usage of multicast, these smaller stages are combined into larger combinations. This can be done exhaustively to find all possible schedules which would result in a correct result. To find the minimal time schedule it is possible to measure the \( \alpha_p \) and \( \alpha_r \) values and then use the models discussed in Section 3 to predict the required time.

Figure 5 shows a schedule performed with the same setup as Figure 2. By utilising a base other than two it is possible to over two stages perform an AllReduce of six processes, compared to the recursive doubling four stages. Equally for an AllReduce size of ten a potential schedule would be \((2, 5)\) compared to the five stages required by recursive doubling.

In the case of large primes when it is inefficient to multicast a generalisation of the fix used previously is available. Utilising the message pipelining ability to perform cheap multicast we collapse groups of processes in a subgroup. Compared to the previous fix this allows a larger reduction in group size for the remaining execution of the AllReduce. This generalisation still requires an additional stage before and after the execution of the internal AllReduce to allow for the processes which did not participate directly to receive the final result.

It is possible to visualise the recursive multiplying algorithm as a recursive AllReduce algorithm as shown in Figure 6. The recursive doubling algorithm represents each stage as an exchange on a single dimension, compared to the recursive multiplying algorithm which has multiple processes exchange partial results within a single stage. Then all groups in a stage communicate with their respective peers in further stages. This visualisation uses \( n \)-dimensional hyper-cuboids compared to binary hypercubes for the recursive doubling case. An \( n \)-dimensional hypercuboid can be used with a large AllReduce operation.

Another approach to dealing with large primes is to do merging, which does not require two additional stages. This allows a composite number to be used instead of a multiple of a base. The first stage is executed performing the first stage of the schedule while the exposed remainder process broadcasts its own value to all processes as required. During the final stage all processes of a group send their final value.
to the remaining processes which then reduce these themselves. By decomposing the size of the AllReduce operation into two numbers, one of which is well factorisable, we can make efficient use of multicast.

In the given example shown in Figure 7 only two stages exist to perform an AllReduce across seven processes. The first stage consists of the first six processes performing an AllReduce within two groups, while the seventh process broadcasts its value to an entire group at the same time. This allows all members of that group to calculate the reduction with the contribution of the seventh process. During the second, and final, stage three groups of two processes perform a pairwise exchange, while a single group also sends its partial results to the seventh process to reduce them by itself. By using prime merging we enabled the seventh process to receive all required information within the two stages instead of the fourth required by the generalised fix.

4.2 Implementation

The pseudocode for recursive multiplying is presented in Algorithm 2. The pseudocode shows several required generalisations, compared to the recursive doubling in which simplifications of these were enough. The transformation from virtual ranks to real ranks is done similarly with a branching statement, though the transformation is more complex than the power-of-two-case. In addition, the masking to find the relevant peers for a stage has changed from an exclusive-or operation, to be a group and offset calculation. Finally, the mask incrementing has changed from a left shift operation, to a multiply by the stage base.

The schedule is passed directly to the AllReduce operation globally and not computed on the fly. This enables library implementers to have control over which schedules are used when. The schedules consist of instructions which some stages will execute depending on previous stages. The general factor type is a simple \( "aB" \) stages where \( B \) is the base for that stage. The collapse and expand instructions are encoded as either \( "c\hat{f}mB" \) or \( "c\hat{f}mB" \). The threshold \( T \) is defined to be divisible by \( B \) and is used as the delin-

| Algorithm 3 Recursive Multiplying Collapse |
|--------------------------------------------|
| 1: procedure ALLREDUCE(rank, com, schedule) |
| 2:  value \leftarrow\ com \quad \triangleright \text{initialize variables} |
| 3:  pthres \leftarrow\ 0 |
| 4:  phase \leftarrow\ 1 |
| 5:  wid \leftarrow\ rank |
| 6:  \text{if} \ pthres < phase \text{then} |
| 7:    \text{peer} \leftarrow\ rank \odot phase + phase + phase - 1 |
| 8:    \text{Send non-blocking value to peer} |
| 9:  \text{else} |
| 10: \text{Recv value from peer} |
| 11: \text{Reduce value rbuf} |
| 12: \text{if} \ rank \mod phase \neq (base - 1) \text{then} |
| 13: \text{Send value to peer} |
| 14: \text{Do} |
| 15: wait on sends |
| 16: \text{else} |
| 17: \text{Recv rbuf from peer} |
| 18: \text{Reduce value rbuf} |
| 19: \text{if} \ rank \mod phase \neq (base - 1) \text{then} |
| 20: \text{Rwi} \leftarrow\ rank - phase \times base - 1 |
| 21: \text{else} |
| 22: \text{end if} |
| 23: wait on sends |
| 24: \text{end if} |
| 25: \text{end if} |
| 26: \text{end if} |
| 27: \text{end for} |
| 28: \text{complete stage} |
| 29: \text{end for} |
| 30: \text{end for} |
| 31: \text{end if} |
| 32: \text{end if} |
| 33: \text{end for} |
| 34: \text{end if} |
| 35: \text{end if} |
| 36: \text{end if} |
| 37: \text{end if} |
| 38: \text{end if} |
| 39: \text{end if} |
| 40: \text{end if} |
| 41: \text{end if} |
| 42: \text{end if} |
| 43: \text{end if} |
| 44: \text{end if} |
| 45: \text{end if} |
| 46: \text{end if} |
Algorithm 4 Recursive Multiplying Expansion

47:   else if type(stage) is expand then
48:     if rank < phibases then
49:       if rank \mod phibase = (base-1) then
50:         for b do
51:           peer ← wid \times phibase + b
52:           Send non-blocking value to peer
53:         end for
54:       else
55:         Recv value from peer
56:       end if
57:       Wait on sends
58:     end if
59:   end if
60: end for
61: Return value
62: end procedure

5. RESULTS

All the experiments reported in this Section were run on the ARCHER[1] supercomputer, a Cray XC30 machine with 4920 compute nodes, each with two 12-core Intel E5-2697 v2 CPUs. The interconnect is the Cray Aries in a Dragonfly topology. The environment used was:

- “PrgEnv-cr/5.2.56”
- “dmapp/7.0.1-1.0502.10246.8.47.ari”
- “cray-mpich/7.2.6”
- “pmi/5.0.7-1.0000.10678.155.25.ari”
- “ugni/6.0-1.0502.10245.9.9.ari”

In all cases we used one rank per node, so that all communication is over the network and not in shared memory on the node. All measurements are performed using an AllReduce summation operation of a single 8 byte integer.

5.1 Multicast Performance

The $\alpha_p/\alpha_r/\beta/\gamma$ model presented in Section 3 is interesting in the context of AllReduce since no round trip occurs within the algorithm. Only the local overhead of issuing a message and the latency until a message from a different peer arrives is important.

The benchmark used to measure both the $\alpha_p$ and $\alpha_r$ values is very similar to the PingPong benchmark used by Hoefler et al[13]. We replicate the function of a multicast operation with multiple pipelining messages. The timing routines are issued directly before and after the message initiation. This explicitly measures the overhead in the LogP/LoP/LogP\textsuperscript* model.

Figure 8 shows the distribution of timing results of a multicast. The number of peers shown is the number of messages which were sent in a single operation. The centre of the notch is the median value, while the width of the notch represents the confidence interval of the median. The box extents are the 25th and 75th percentile of the measured results. The error bars are shown as either a minimum or maximum of the measured values if the value is within 1.5 times the interquartile range, otherwise they are 1.5 times the interquartile range. Outliers are shown above the whiskers.

The benchmark to evaluate the algorithm presented in Section 4 is implemented using the Cray DMAPP library [4], which supports a PGAS-based approach to communication. Although the algorithm presented does not explicitly require single-sided communication, using Cray DMAPP allows the least amount of time between message issues without a large software stack, which enables us to maximize the message pipelining.

The memory consumption is less efficient than a point-to-point channel implementation. Both approaches are difficult to quantify; point-to-point channels consume $O(1)$ memory, but $O(\log_2 N)$ channels exist in memory. The PGAS-based implementation utilises a memory array allocated in the data segment of the application which stores the addresses to write to for each peer.

The live ARCHER system was used for measurements,

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
 & Minimum ($\mu$s) & Median ($\mu$s) \\
\hline
$\alpha_p$ & 1.34 & 1.51 \\
$\alpha_r$ & 0.34 & 0.38 \\
\hline
\end{tabular}
\caption{\small $\alpha_p$ and $\alpha_r$ values from multicast experiment (non-blocking sends).}
\end{table}

by dots. Since these results were gathered on a live system much noise is encountered, but the underlying distribution is clearly positive skewed. A fixed minimum is expected since the message transmission is limited by the speed of light. The tail of the distribution is not bounded and could be very large.

The multicast was implemented in two different ways, using blocking and non-blocking sends. A multicast operation constructed with non-blocking sends clearly outperforms blocking sends. An interesting artefact occurs at eight peers when both blocking and non-blocking have a slight discontinuity which does not occur with any higher peer count. This could be due to packet combining, since the payload size is 8 bytes and the packet size is 64 bytes.

A linear regression is performed using minimum and median values to approximate the $\alpha_p$ and $\alpha_r$ values seen experimentally. The values are presented in Table 1 using only the experimental results from one to eight peers.

5.2 AllReduce Benchmark

The benchmark to evaluate the algorithm presented in Section 4 is implemented using the Cray DMAPP library [4], which supports a PGAS-based approach to communication.

Figure 8: Distribution of times with varying multicast size with blocking and non-blocking sends.
5.3 AllReduce Schedule Comparison

The performance results of the benchmark are presented in Figure 9. The errors are shown as described for Figure 8. As can be seen, the minimum values are significantly less than the median values, with considerable spread of all measurements. Neither the minimum nor median results follow a logarithmic curve when going to large scales. Comparing recursive doubling to the best recursive multiplying schedule there is a significant advantage by using message pipelining: the median value for recursive multiplying is near the 25th percentile value for recursive doubling. Important to note is the reduction in improvement as the scale at which the AllReduce is performed grows.

Comparing the results in Figure 9 using the recursive doubling schedule directly to MPICH shows a minor MPI stack overhead. Therefore the performance results using recursive multiplying are outperforming MPICH as well.

5.4 AllReduce Model Comparison

The model presented in Section 3 can be used to predict the time required for an AllReduce operation. To measure the accuracy of this approach we used the prediction of the model compared to the experimental results given in Section 5.3. Using the values for $\alpha_p$ and $\alpha_r$ evaluated previously for the median and minimum we can use the model to predict the minimum and median values for AllReduce operations.

Figure 10 shows the relative difference of the experimental results to the predictions by the model. As can be seen, the minimum values follow the model well until 24 nodes is reached. An upwards trend is clear from there onwards similarly to the median values. The median values show a consistent increase in the difference. The difference between the theoretical and actual values is likely due to skew of process arrival times. This increases as the size of the AllReduce increases, since it is more likely that a process is delayed.

6. RELATED WORK

Motivated by the LoP and LogfP model, Hoefler et al[11] introduced a barrier operation based on the n-way dissemination pattern which allows for higher performance. The n-way dissemination pattern is an extension of the original dissemination pattern which allows for higher performance. The n-way dissemination pattern also used for a barrier operation[6, 10]. By allowing a process to send multiple messages the scaling of the dissemination barrier is improved from $O(\log_2 N)$ with the n-way dissemination barrier.

End et al[9] introduced the n-way dissemination AllReduce which allows for a large improvement on InfiniBand. When $N \neq (n + 1)^k$ there is potential for duplication. An adaption is presented which performs a post process when duplication occurs, based on the data boundary from the previous stage. This allows for the correct result to be computed. Since butterfly-like patterns require an associative
operator, this algorithm is only suitable for a subset of use cases.

7. FUTURE WORK

The recursive multiplying algorithm is currently only implemented in the benchmark presented in Section 5. An implementation for the Edinburgh MPI for Research Library (EMPI4Re) is planned, which will include prime merging alongside the generalized fix. Since larger messages are expected in the context of MPI, an exploration will be performed to evaluate the capabilities of the Cray Aries[5] network such that a appropriate threshold can be used for the recursive multiplying method. Finally, the EMPI4Re implementation will be compared directly to MPICH.

Currently the schedule which is experimentally determined or analytically found is assumed to be order invariant except for the collapse and expand stages. While this is reasonable on a network such as the Cray Aries[5], which is very close to an all-to-all network, on different networks such as fat-trees the ordering could be important. By performing stages in a specific order dependent on the network, traffic can potentially be reduced. This is an obvious extension to allow for shared memory, on-node operations. On ARCHER[1] this would imply a 24-way all-to-all, since it can be performed efficiently, which reduces the costs associated with this operation. In addition, exploring the behaviour of message pipelining on the Infiniband network would be interesting with recursive multiplying. Finally, evaluating a schedules using an efficient heuristic would be required for large-scale computers to determine the best schedule to use, since an exhaustive search would be prohibitively expensive.

8. CONCLUSIONS

We presented the recursive multiplying algorithm, which is a generalisation of the state of the art recursive doubling with pairwise exchange algorithm used in MPICH for small message AllReduce operations. We showed experimental results of reductions in execution times of 8% to 40% with recursive multiplying compared to recursive doubling on ARCHER, a Cray XC30.

Using message pipelining we replaced the original pairwise exchange with a multi-way exchange allowing small message AllReduce operations to be performed faster. With this extension we were able to construct AllReduce operations with variable b values to accelerate large-scale operations with special cases only for large prime factors, instead of for non-power-of-two cases. If future hardware designs support a higher degree of message pipelining, the time for AllReduce operations could be further improved.

9. ACKNOWLEDGMENTS

We would like to thank Daniel Holmes for the help regarding the evaluation strategy and Justs Zarins for various coffee discussions.

This work was supported by grant EP/L01503X/1 for the University of Edinburgh School of Informatics Centre for Doctoral Training in Pervasive Parallelism (pervasiveparallelism.inf.ed.ac.uk) from the UK Engineering and Physical Sciences Research Council (EPSRC) and by the European Commission through the EPIGRAM project (Grant Agreement no.610598, www.epigram-project.eu). The work has made use of resources provided by the Edinburgh Parallel Computing Centre (EPCC, www.epcc.ed.ac.uk).

10. REFERENCES

[1] ARCHER. http://www.archer.ac.uk/. Accessed: 2016-04-18.
[2] MPI: A Message-Passing Interface Standard. https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. Accessed: 2016-05-09.
[3] MPICH. http://www.mpich.org/. Accessed: 2016-04-18.
[4] Using the GNI and DMAPP APIs. http://docs.cray.com/books/S-2446-5202/S-2446-5202.pdf. Accessed: 2016-04-18.
[5] B. Alverson, E. Froese, L. Kaplan, and D. Roweth. Cray @ XC Series Network. pages 1–28.
[6] E. D. Brooks. The butterfly barrier. International Journal of Parallel Programming, 15(4):295–307, 1986.
[7] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn. Collective communication: theory, practice, and experience. Concurrency and Computation: Practice and Experience, 19(13):1749–1783, 2007.
[8] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, 28(7):1–12, 1993.
[9] V. End, R. Yahyapour, C. Simmendinger, and T. Alrutz. Adaption of the n-way Dissemination Algorithm for GASPI Split-Phase Allreduce. (c):13–19, 2015.
[10] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. International Journal of Parallel Programming, 17(1):1–17, 1988.
[11] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. Fast barrier synchronization for InfiniBand. 29th International Parallel and Distributed Processing Symposium, IPDPS, 2006.
[12] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. LogIP - a model for small Messages in InfiniBand. Parallel and Distributed Processing Symposium, 2006.
[13] T. Hoefler and W. Rehm. A Communication Model for Small Messages with InfiniBand. In *PARS Mitteilungen*, pages 32–41, 2005.

[14] H. Pritchard, I. Gorodetsky, and D. Buntinas. A uGNI-based MPICH2 nemesis network module for the Cray XE. *Recent Advances in the Message Passing Interface*, pages 110–119, 2011.

[15] R. Rabenseifner. Automatic MPI counter profiling. *42nd CUG Conference*, 2000.

[16] R. Rabenseifner and J. L. Träff. More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 3241:36–46, 2004.

[17] R. Thakur and W. Gropp. Improving the performance of collective operations in MPICH. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, (2840):257–267, 2003.

[18] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.